

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Tangible language for educational programming of robots and other targets

Ângela Cardoso



**FEUP** FACULDADE DE ENGENHARIA  
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisors: Armando Sousa and Hugo Ferreira

February 24, 2019



# **Tangible language for educational programming of robots and other targets**

**Ângela Cardoso**

Mestrado Integrado em Engenharia Informática e Computação

February 24, 2019



# Abstract

The demand for Science, Technology, Engineering and Mathematics (STEM) graduates has been increasing and is expected to continue growing. In accordance, educational programs are being adapted. Schools are introducing programming at a younger age, but there are some difficulties. Robots can help children to learn, as they make coding notions concrete, by showing the effects of each block of code.

For schools, one of the main issues is the cost, since both robots and programming stations tend to be expensive. A way to manage this is to group the students into teams and to make several teams share the same robot and programming station. While sharing a robot for programming tasks is quite feasible, the same cannot be said for programming stations, since they are in use most of the time. An idea is to use smartphones or tablets, because they are less expensive. But the smaller size of the screens makes creating computer programs harder.

In this project, the tangible programming language `Tactode` was created. It can be executed in multiple commercial robots, namely `Ozobot`, `Cozmo`, `Sphero` and `Robobo`, but also in the non-robotic platforms `Scratch` and `Python`. The programs are captured through a photograph, whose code is compiled and sent to the desired target. This makes the use of hand-held devices practical, while allowing them to be shared across teams. To keep the cost low, Ethylene Vinyl Acetate (EVA) was used to make puzzle like blocks, that can be fitted together to create programs.

This solution was tested from its early stages, by conducting small experiments with students of different age groups. A variety of robotic and non robotic targets was used, as well as different smartphones, tablets and computers to compile the code. Earlier experiments helped to guide the development, while the later ones showed that the system is viable and ready to be used.

Expansion of `Tactode` into new targets and capacities is simple, given the architecture of the compiler. Development using the `Ionic` framework makes the `Tactode` application run in multiple operating systems and platforms. The competitive manufacturing price of the blocks makes it a real contender for schools. Plus, it is interesting, because it is flexible and multi-target, while tapping into the proven advantages of tangible systems, particularly for children.

**Keywords:** Robotics. Programming Education. Children. Tactile. Tangible. Visual. STEM Education.

## ACM Computing Classification System:

- Computer systems organization → Robotics
- Software and its engineering → Domain specific languages
- Software and its engineering → Visual languages
- Hardware → Tactile and hand-based interfaces



# Resumo

A procura por graduados em Ciência, Tecnologia, Engenharia e Matemática (STEM) tem vindo a aumentar ao longo dos anos. Em concordância, os currículos educacionais estão a ser adaptados. As escolas estão a introduzir a programação mais cedo, mas há algumas dificuldades. Os robôs podem ajudar as crianças a aprender, dado que tornam as noções de programação concretas, mostrando os efeitos de cada bloco de código.

Para as escolas, uma das maiores dificuldades é o custo, dado que os robôs e as estações de programação tendem a ser caros. Uma forma de gerir o custo é organizar os alunos em equipas e fazer várias equipas partilharem o mesmo robô e estação de programação. Enquanto que partilhar um só robô em tarefas de programação é plausível, o mesmo não pode ser dito sobre as estações de programação. Uma ideia é usar smartphones ou tablets em vez de computadores, uma vez que são mais baratos. Mas o tamanho mais pequeno dos ecrãs nestes aparelhos dificulta a programação.

Neste projeto foi criada a linguagem de programação tangível *Tactode*. Pode ser executada em múltiplos robôs, nomeadamente *Ozobot*, *Cozmo*, *Sphero* e *Robobo*, mas também nas plataformas não robóticas *Scratch* e *Python*. Os programas são captados através de uma fotografia, cujo código é compilado e enviado para a plataforma desejada. Isto torna prático usar smartphones e tablets, permitindo ainda que estes aparelhos sejam partilhados entre equipas. Para manter o custo reduzido, usou-se acetato-vinilo de etileno (EVA) para fazer peças tipo puzzle, que podem ser encaixadas para criar os programas.

Esta solução foi testada desde o início, conduzindo pequenas experiências com alunos de diferentes idades. Foram usadas várias plataformas de execução, assim como smartphones, tablets e computadores para compilar o código. As primeiras experiências ajudaram a guiar o desenvolvimento e as últimas mostraram que o sistema é viável e está pronto para ser usado.

A expansão do *Tactode* a novas plataformas de execução e capacidades é simples, dada a arquitetura do compilador. O desenvolvimento em *Ionic* faz com que a aplicação *Tactode* corra em vários sistemas operativos e aparelhos. O preço competitivo de fabrico das peças torna-o uma possibilidade real para as escolas. Além disso, é interessante, porque é flexível e multi-plataforma e faz uso das vantagens dos sistemas tangíveis, especialmente para crianças.

**Palavras-chave:** Robótica. Ensino Programação. Crianças. Táctil. Tangível. Visual. Educação STEM.

## Sistema de Classificação Computacional ACM:

- Organização de Sistemas computacionais → Robótica
- Software e a sua engenharia → Linguagens de domínio específico
- Software e a sua engenharia → Linguagens visuais
- Hardware → Interfaces tácteis e baseados em mão





*“Early childhood education is the key to the betterment of society”*

Maria Montessori



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context . . . . .	1
1.2	Motivation and Goals . . . . .	2
1.3	Dissertation Structure . . . . .	4
<b>2</b>	<b>STEM Education, Programming and Robots</b>	<b>5</b>
2.1	STEM education . . . . .	5
2.2	Educational Programming Languages . . . . .	6
2.2.1	Block Based Coding Languages . . . . .	7
2.2.2	Visual Programming Languages . . . . .	11
2.2.3	Tangible Programming Languages . . . . .	14
2.3	Educational Robots . . . . .	20
2.4	Conclusion . . . . .	27
<b>3</b>	<b>Problem Statement</b>	<b>29</b>
3.1	Introduction . . . . .	29
3.2	Ideal . . . . .	30
3.3	Reality . . . . .	30
3.4	Consequences . . . . .	31
3.5	Proposal . . . . .	32
3.6	Research Questions . . . . .	33
3.7	Conclusion . . . . .	34
<b>4</b>	<b>Tactode- A Tangible Programming Language</b>	<b>35</b>
4.1	Introduction . . . . .	35
4.2	Alternatives . . . . .	36
4.2.1	Block Shape . . . . .	36
4.2.2	Materials and Manufacturing . . . . .	37
4.2.3	Programming Paradigms . . . . .	38
4.2.4	Piece Recognition . . . . .	39
4.2.5	Target Platforms . . . . .	39
4.3	Tactode Programming Language . . . . .	40
4.4	Challenges . . . . .	45
4.4.1	Regular Polygon . . . . .	45
4.4.2	Obstacle Reaction . . . . .	46
4.4.3	Simple Maze . . . . .	46
4.4.4	Prime Number Generator . . . . .	48
4.5	Evaluation . . . . .	50

# CONTENTS

4.6	Conclusion . . . . .	52
<b>5</b>	<b>Tactode Application</b>	<b>53</b>
5.1	Introduction . . . . .	53
5.2	Requirements . . . . .	54
5.2.1	Functional Requirements . . . . .	54
5.2.2	Non-Functional Requirements . . . . .	55
5.3	Architecture . . . . .	56
5.4	Transpiler . . . . .	60
5.4.1	Parser . . . . .	60
5.4.2	Code Generators . . . . .	62
5.5	Application . . . . .	63
5.6	Testing . . . . .	64
5.6.1	Platform Tests . . . . .	64
5.6.2	Visibility Tests . . . . .	67
5.7	Evaluation . . . . .	70
5.8	Conclusion . . . . .	71
<b>6</b>	<b>User Testing</b>	<b>73</b>
6.1	Introduction . . . . .	73
6.2	Objectives . . . . .	74
6.3	Methodology . . . . .	74
6.3.1	First Experiment . . . . .	74
6.3.2	Second Experiment . . . . .	76
6.3.3	Third Experiment . . . . .	78
6.4	Results . . . . .	81
6.4.1	First Experiment . . . . .	81
6.4.2	Second Experiment . . . . .	82
6.4.3	Third Experiment . . . . .	83
6.5	Discussion . . . . .	85
6.6	Conclusion . . . . .	88
<b>7</b>	<b>Conclusion</b>	<b>89</b>
7.1	Overview . . . . .	89
7.2	Contributions . . . . .	90
7.3	Future Work . . . . .	91
7.3.1	Language . . . . .	91
7.3.2	Manufacturing . . . . .	91
7.3.3	Application . . . . .	92
7.3.4	User Testing . . . . .	92
	<b>References</b>	<b>93</b>
<b>A</b>	<b>Tactode Pieces</b>	<b>99</b>
A.1	Numbers . . . . .	99
A.2	Colors . . . . .	99
A.3	Letters . . . . .	100
A.4	Operators . . . . .	100
A.5	Variables . . . . .	102

## CONTENTS

A.6	Events . . . . .	102
A.7	Control . . . . .	102
A.8	Sensors . . . . .	103
A.9	Movement . . . . .	104
A.10	Sound . . . . .	106
A.11	Visual . . . . .	107
<b>B</b>	<b>Questionnaires for the experiments</b>	<b>109</b>
B.1	First and Second Experiment . . . . .	109
B.2	Student Questionnaire for the Third Experiment . . . . .	110
B.3	Monitor Questionnaire for the Third Experiment . . . . .	110

## CONTENTS

# List of Figures

2.1	An example of a Scratch program. . . . .	8
2.2	A Snap! program defining a for block. . . . .	8
2.3	An example of a Stencyl game. . . . .	9
2.4	A simple Blockly program. . . . .	9
2.5	A MakeCode chicken generation program for Minecraft. . . . .	9
2.6	The Alice IDE with some code. . . . .	10
2.7	An Etoys example program. . . . .	10
2.8	An example of a ScratchJr program. . . . .	11
2.9	Selecting a code block in Kodu. . . . .	12
2.10	Programming the Lego Mindstorms EV3 robot. . . . .	12
2.11	A level of the game Lightbot. . . . .	13
2.12	One of the Code-a-Pillar application challenges . . . . .	13
2.13	The T_ProRob blocks. . . . .	14
2.14	The V_ProRob blocks. . . . .	14
2.15	Children using the AlgoBlock language. . . . .	15
2.16	All the different components of Electronic Blocks. . . . .	16
2.17	A Cubelets robot and some other blocks. . . . .	16
2.18	A Code-a-Pillar robot with some of its component blocks. . . . .	17
2.19	The TagTile puzzle pieces. . . . .	17
2.20	A program written in Quetzal. . . . .	18
2.21	A program written in Tern. . . . .	18
2.22	The T-Maze blocks, a maze and the sensors. . . . .	19
2.23	The Osmo Coding Family base and some of its blocks. . . . .	19
2.24	The Code Bits puzzle pieces, one of the mazes and the mobile application. . . . .	20
2.25	The Lego Mindstorms EV3 robot in two of its many possible forms. . . . .	22
2.26	Cozmo and its cubes. . . . .	22
2.27	The robot KUBO going over a program in TagTile. . . . .	22
2.28	The robot KIBO and some of its tangible programming blocks. . . . .	23
2.29	The mBot robot. . . . .	23
2.30	The Lego Boost robot in three of its possible shapes. . . . .	24
2.31	The robot Thymio. . . . .	24
2.32	The robot Dash and its little companion Dot. . . . .	25
2.33	The Ozobot and its color coded programming language. . . . .	25
2.34	The robot Sphero in its SPRK+ edition. . . . .	26
2.35	A Robobo robot. . . . .	26
4.1	The basic shape of a Tactode piece. . . . .	36
4.2	The usual shape of a Tactode command block. . . . .	40

## LIST OF FIGURES

4.3	The usual shape of a Tactode argument block. . . . .	40
4.4	The contents of the Tactode if piece: color, text, icon, Aruco tag and indentation. . . . .	41
4.5	A Tactode program with flow control blocks. . . . .	43
4.6	A Tactode if piece, with yellow EVA, printed paper and clear plastic on top. . . . .	45
4.7	The regular polygon generation program in Tactode for Ozobot, Cozmo, Sphero and Robobo. . . . .	46
4.8	The regular polygon generation program in Tactode for Scratch. . . . .	47
4.9	An obstacle reaction program in Tactode: run away from objects in the back. . . . .	47
4.10	An obstacle reaction program in Tactode: follow objects in front. . . . .	47
4.11	Example of a finite simple maze with the entry in red and the exit in green. . . . .	48
4.12	The finite simple maze solution program in Tactode. . . . .	49
4.13	The prime number generation program in Tactode. . . . .	51
5.1	The use cases diagram for the Tactode application. . . . .	55
5.2	The activity diagram for importing a new program in the Tactode application. . . . .	57
5.3	Simplified class diagram for the Tactode application. . . . .	59
5.4	The AST for the boolean expression $1 + 2 = 3$ . . . . .	62
5.5	The home page of the Tactode application, ready to import a new program. . . . .	63
5.6	The home page of the Tactode application, after importing a program. . . . .	63
5.7	The previous programs page of the Tactode application. . . . .	64
5.8	The settings page of the Tactode application. . . . .	64
5.9	The Tactode error test after being parsed . . . . .	66
5.10	Prime generator, top view, <b>yes</b> . . . . .	67
5.11	Prime generator, rotation, <b>no</b> . . . . .	67
5.12	Prime generator, 180° rotation, <b>yes</b> . . . . .	68
5.13	Prime generator, bottom perspective, <b>no</b> . . . . .	68
5.14	Prime generator, shadow, <b>yes</b> . . . . .	68
5.15	Prime generator, reflection, <b>no</b> . . . . .	68
5.16	Regular polygon generator, top view, <b>yes</b> . . . . .	69
5.17	Regular polygon generator, rotation, <b>yes</b> . . . . .	69
5.18	Regular polygon generator, left perspective, <b>no</b> . . . . .	69
5.19	Regular polygon generator, right perspective, <b>yes</b> . . . . .	69
5.20	Regular polygon generator, bottom perspective, <b>yes</b> . . . . .	70
5.21	Regular polygon generator, shadow <b>yes</b> . . . . .	70
6.1	Teenagers using old Tactode pieces to program Ozobot. . . . .	75
6.2	Teenagers using Ozoblockly to program Ozobot. . . . .	76
6.3	Children using the Tactode application to take a picture of their program to draw a regular polygon in Scratch. . . . .	77
6.4	Children using the Tactode application to take a picture of their program to solve a simple maze with Ozobot. . . . .	77
6.5	Children programming in Tactode to make Ozobot draw a regular polygon. . . . .	79
6.6	Children loading their program onto Ozobot. . . . .	80
6.7	Children placing Ozobot in the corner of a square to execute their program. . . . .	80
6.8	Completion times (in minutes) of each task by group. . . . .	84
6.9	Number of photos taken for each task by group. . . . .	84



# List of Tables

2.1	Educational Programming Languages Comparison . . . . .	21
2.3	Educational Robots Comparison . . . . .	28
4.1	A sample of Tactode pieces . . . . .	41
6.1	Material used in the third experiment. . . . .	78
6.2	Answers of the students to the classification questions, in the first questionnaire. . . . .	81
6.3	Answers of the students to questions comparing Tactode and Ozoblockly, in the first experiment. . . . .	82
6.4	Answers of the students to the classification questions, in the regular polygon challenge of the second experiment. . . . .	82
6.5	Answers of the students to questions comparing Tactode and Scratch, in the regular polygon challenge of the second experiment. . . . .	83
6.6	Answers of the students to the classification questions, in the maze challenge of the second experiment. . . . .	83
6.7	Answers of the students to questions comparing Tactode and Ozoblockly, in the maze challenge of the second experiment. . . . .	83
6.8	Children participating in the third experiment. . . . .	83
6.9	Answers of the students to the classification questions, in the third experiment. . . . .	86
6.10	Answers of the students to the robot related questions, separated by the robot used, in the third experiment. . . . .	86
A.1	Tactode number pieces . . . . .	99
A.2	Tactode color pieces . . . . .	99
A.3	Tactode letter pieces . . . . .	100
A.4	Tactode operator pieces . . . . .	101
A.5	Tactode variable pieces . . . . .	102
A.6	Tactode event pieces . . . . .	102
A.7	Tactode control pieces . . . . .	103
A.8	Tactode sensor pieces . . . . .	103
A.9	Tactode movement pieces . . . . .	104
A.10	Tactode sound pieces . . . . .	107
A.11	Tactode visual pieces . . . . .	107

## LIST OF TABLES

# Abbreviations

ABS	Acrylonitrile Butadiene Styrene
AST	Abstract Syntax Tree
EVA	Ethylene Vinyl Acetate
GPS	Global Positioning System
IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
IR	Infrared
LED	Light Emitting Diode
LCD	Liquid Crystal Display
PLA	Polylactic Acid
RFID	Radio Frequency Identification
RGB	Red Green Blue
ROS	Robotic Operating System
SDK	Software Development Kit
STEM	Science, Technology, Engineering and Mathematics
USB	Universal Serial Bus
VGA	Video Graphics Array



# Chapter 1

## Introduction

This chapter contains an overview of this dissertation, presenting the context of our work. It also describes the structure of this document.

### 1.1 Context

Science, Technology, Engineering and Mathematics (STEM) are at the forefront of the current industry demands. In fact, according to the United States Department of Commerce [1], in the 2005-2015 decade employment in STEM occupations grew at a rate of 24.4%, while for non-STEM employment the growth was only of 4%. According to the same report, between 2014 and 2024, the expected increments in STEM vs. non-STEM jobs are 8.9% to 6.4%. This means that, although at a slower rate, the portion of total employments devoted to STEM areas will continue to increase.

The industry needs alone justify an investment in better STEM education, but there are other factors. STEM workers have higher salaries, earning 29% more in 2015; they have lower unemployment rates, less than half in 2015; higher educational attainment, nearly 75% STEM vs. 33% non-STEM hold at least a college degree. Moreover, STEM graduates earn 12% more than other graduates even if they work in non-STEM jobs.

In spite of the clear attractiveness of the STEM job market, students are not typically successful and interested in these subjects in school. For example, according to the United States Department of Education [2], only 16% of American high school seniors simultaneously are competent at math and show interest in a STEM career.

Amongst STEM occupations in the United States, 49% are in computer and math fields [1] and these are projected to grow 13.1% in 2014-2024, more than the average 8.9% of all STEM jobs. When it comes to math, it is considered the hardest subject in school by the large majority of students. As for computers, although current generations have grown with all sorts of digital devices, the large majority of them is still only comfortable with them on a generic user level.

They can play games, watch videos, use social media or chat applications, but that only makes them content consumers, not content creators. For a lot of what is involved in creating such contents, one needs to learn programming, which is still a difficult and avoided subject, even in the current digital age.

Teaching robotics is a steady trend in education, with several studies showing its positive effects. Karim et al. [3] showed that in K-12 education robots are being successfully used to teach mathematics and physics, as well as creative thinking and problem solving, with students showing more interest and a better attitude, although they also perceived a limited presence of this kind of technology in the curricula. Chetty [4] focusses on strategies for teaching young kids to program using robots, based on evidence that learning at an earlier age leads to future interest in Computer Science. Avanzato [5] showed how robot design competitions can foster the interest of K-12 students in STEM careers. Preliminary results from a summer program described in Tewolde et al. [6] show how building and programming robots can attract students for Engineering careers. Benitti et al. [7] used a systematic literature review to conclude that robots are a flexible learning tool in numerous STEM subjects, fostering team work and problem solving, and giving students the opportunity to experience the Engineering Design Process.

There is also evidence of the advantages of teaching children to program, even more so due to the ubiquity of technology. For example, Chetty [4] found evidence that learning to program at an earlier age counteracts the usual aversion people have to this subject and leads to future interest in Computer Science. Resnick et al. [8] pointed out that learning to program expands what children can do with a computer, the range of what they can learn and their problem solving and design capabilities. Furthermore, several European countries, including Portugal [9], have already added computer programming fundamentals as a mandatory subject in their primary schools [10]. So there is a clear need to invest in ways to make this subject more tractable for young children.

When introducing novices to programming, evidence shows that robots can be of great help. In their systematic literature review, Major et al. [11] noted that 75% of the included literature concluded that the using robots in introductory programming courses is an effective teaching tool. In their study based on a college course of introduction to programming, Özüorçun et al. [12] showed that the use of robots improved the performance of the students, noting that the robots were helpful in understanding the effects of each algorithm. Huei [13] used a robot in mini projects with the `Python` language and concluded that it lead to higher creativity, problem solving and collaboration capabilities amongst the students, while helping them to understand the programming concepts.

## 1.2 Motivation and Goals

In order to make it easier for groups of students to collaborate and, specially in the case of younger children, to remove the distractions and technicalities of using a computer, we decided to explore a tangible programming language, that is, one where the interface is physically manipulated. An immediate benefit of this kind of language is that it complies with the recommendation of the

## Introduction

American Academy of Pediatrics [14], that young children should spend a limited amount of time with screens each day. Also, studies have shown that tangible interfaces have several advantages: they provide a better collaborative environment and students tend to be more involved than with their graphical counterparts.

In their 2007 article, Horn et al. [15] present two tangible programming languages as well as the practical advantages of this approach, with initial studies revealing that children between the ages of six and seven years old were capable of designing chains of actions, with some even finding bugs. In a subsequent 2009 study [16], the authors conducted a museum experiment, where the visitors classified the tangible and graphical interfaces as equally easy to use, but were significantly more likely to interact and collaborate with the tangible interface. Later, in 2012 [17] they proposed a hybrid approach between tangible and graphical interfaces in which teachers and students chose the best fit for each situation, given that both have strengths and limitations.

In 2012, Sapounidis et al. [18] made use of a kit with both tangible and graphical interfaces to measure children preferences, concluding that the tangible interface was simultaneously easier to use and deemed as more enjoyable. Later, in 2015 [19], they conducted a formal study with 109 children between the ages of 6 and 12, where they compared the same programming language with two different interfaces: tangible and graphical. The results showed that children using the tangible interface made less errors, were more likely to effectively debug their errors and, in the case of the younger children, needed less time to accomplish robot programming tasks. Also, when interacting freely, the older children were more engaged, designed programs with higher complexity and used a wider variety of commands, with the tangible interface.

In addition to tangible, we also opted for a visual approach to programming, instead of the more typical text based languages. The advantages of such an approach are threefold. First, by removing the language barrier, younger students can participate in the simpler programming activities even before they can read. Second, visual clues are easier to learn, even for those who can already read. Third, by removing words, our programming language becomes universal and independent from the mother language of each particular student.

Building a tangible and visual language is the first main objective of this dissertation. But the second is getting it to run on several educational robots as well as non robotic platforms. This part is no less important in the learning process, its where the abstract becomes concrete and a real execution environment helps the students to understand what each part of their program does. But in order for it to work and be simple, we have to build a new application that will translate our tangible programming language into something that the robot can actually execute. That is the second main challenge we face.

The idea is to use the camera of the smartphone to capture the tangible code. Then, using image processing, the application will detect each individual component of the program. At this point, it will generate an abstract syntax tree of the program, whilst detecting eventual errors and reporting them. If there are no errors, the application is in conditions to generate code that can be executed in the desired destination platform, after which the translated program can be sent to the robot and executed.

Although we have several preselected robots and non robotic platforms that will be used to test our concept, our objective is to create a language and an application that are as open and free from platform restrictions as possible, in order to ensure that future addition of other platforms is completely straightforward. Overall, this work strives for a tangible, visual, concrete, real, multi-target, open and extendible programming solution, that will capture the imagination of young students and foster their development into the software creators of tomorrow.

### **1.3 Dissertation Structure**

In addition to the current introductory chapter, there are six chapters in this dissertation. Chapter 2 contains the state of the art of STEM education and presents educational programming languages and robots. In Chapter 3, the problem we intend to solve with this project is stated, presenting our goals. Chapter 4 presents the tangible programming language developed in this project. In Chapter 5, we describe the implementation of the application that transpiles our tangible code into code that is ready to be executed in each destination platform. Chapter 6 exposes the experiments we conducted to test our solution with its intended users, as well as their results. Finally, Chapter 7 is devoted to some conclusions and future work regarding this project.



## Chapter 2

# STEM Education, Programming and Robots

This chapter describes the state of the art and presents related works in the fields of STEM education, as well as educational programming languages and robots. We start with a general survey of recent works in STEM education, but focusing on the use of robots and programming. Then we present a comprehensive review of current educational programming languages. We also perform a revision of the available educational robots.

### 2.1 STEM education

Improving the education of STEM subjects is as important as it is challenging. Even if, against predictions [1], the job market demands for STEM jobs do not increase in the next couple of decades, the success of technological and scientific evolution relies on the quality of the future STEM graduates.

Research [20] shows that the large majority of neurons in the human brain develop up to the age of three. Also, in the formation of active neural pathways, the absorption of information is crucial. This means that early childhood learning shapes our future learning abilities. There are various well known examples, in areas such as music, dance or sports, where the greatest started at a very young age. But even though it may not be common knowledge, the current scientific understanding of our brain and how people learn points to this being a universal characteristic of all subjects, including those related to STEM.

The fact that starting young increases the probability of higher achievement, does not mean that children should be introduced to knowledge that they are not prepared to acquire. In fact, that is one of the common factors of under performance in school. A simple experiment where this can be clearly observed is when a new kind of toy is first introduced to a child. Even if they show initial interest in the toy, if they are not mature enough to play with it, for example because they

have not mastered the motor skills to manipulate it, they will quickly lose interest. More, this rejection of the toy can have permanent effects, as the child associates the unpleasant feelings of failure with the toy.

Determining the age at which a child is prepared to grasp a concept or perform an activity is hard, especially because it varies a lot. Although the age indications at most children aimed products might lead us to believe otherwise, there are many different speeds of development, all within the healthy range. Even the starting age of formal school is a reason for debate: Dee et al. [21] present some evidence about the effect of school starting age on the students' outcomes, based on data from a Danish survey; Durand [22] shows the divergent opinions on this matter with some governments looking to increase school starting age, while others are attempting the opposite. This age issue coupled with the increased difficulty most students show towards some STEM subjects are enough to make educators reconsider their strategies and search for better methods. A possible path is to choose adaptable approaches. Much like there are different ways to play with the same toy, there are different ways to introduce the same concept.

Among the semi-recent educational methods is the incorporation of technology in schools. It makes sense, given that technology has infiltrated nearly every aspect of modern life. But it is also a way to capture the interest of the younger generations. Of course, it is important to use technology that actually teaches, it should not be just a gimmick that grabs the attention of kids without expanding their knowledge. In this respect, robots are a particularly good fit: they can be successfully used to teach mathematics, physics, creative thinking and problem solving [3]; they are very adept to teach young kids to program [4]; they lead to future interest in STEM careers [5], particularly those in Engineering [6]; they foster team work, whilst giving students a first experience of the Engineering Design Process [7].

Let us take all of these ideas together and focus on educational programming. There are benefits in teaching programming at a younger age [4, 8] with several countries already taking concrete measures and introducing it in their primary school curricula [10]. But it is also important to evaluate when each child is ready to learn the abstract and complex notions of this field. Robots provide a concretization of these notions and help to introduce them earlier on. But the language with which the robots are programmed is fundamental in determining how early one can teach to code. Ideally, this language would be simultaneously easy to start learning and capable of expanding into more complex projects over time [23].

## 2.2 Educational Programming Languages

A lot of computer programmers started their learning path with what one may call a real programming language, that is, a language designed for writing professional programs. This is certainly the case for those that learn to code with a concrete and pressing objective in mind. Regardless, if one simply searches the Internet for the best language to start learning, the usual suggestions revolve around Python, C/C++, Java, JavaScript or Ruby.

Higher education institutions also seem to prefer wide use programming languages as an introduction to the discipline, as evidenced in [24]. Although some institutions do use educational languages, such as `Scheme` or `Scratch`. Perhaps this preference is partly due to the need to compress courses into a limited time period. But the expected maturity of students at this level more than likely plays an important role as well.

In any case, when dealing with younger students, such as those from primary or middle school, there are clear advantages in introducing an educational language first [8], as these languages typically provide simpler syntax, activities designed to capture the interest of the young and low entry level requirements. Even for students in secondary school, depending on their previous experiences, this may be the case as well. Hence, given the context of our project, we have chosen to introduce in the following subsections some educational programming languages aimed at children and young adolescents.

At the end of this section, Table 2.1 presents a summary comparison of the educational programming languages listed in the following subsections. For each language, the following features are listed:

- Blocks - yes if the language is block based, no otherwise;
- Visual - yes if the language does not depend on text, but on visual clues, no otherwise;
- Tangible - yes if the language interface is tangible, no otherwise;
- Paradigms - programming paradigms supported by the language;
- Constructs - programming constructs that the language implements;
- Availability - whether the language is available for use online, download or purchase;
- Price - price of the language in case it is being sold;
- Number Blocks - number of blocks the language is shipped with, in case it is sold and tangible.

### 2.2.1 Block Based Coding Languages

Block based coding languages are programming languages where one uses predefined code blocks to compose a program. Eliminating the need to write the code, as happens in typical programming languages, makes the process much simpler. The code blocks can be dragged into the programming area or otherwise selected and linked to each other.

Developed by the MIT Media Lab's Lifelong Kindergarten group, `Scratch` [8, 25] is probably the best known example of a block based language. It is also an event driven programming language, that is, a language in which the program advances when certain events occur. Aimed at children ages eight and up, it is a success story, as it is currently used by a variety of robots (including `Robobo`) and games, introducing more and more children to coding. Its online community has millions of registered users and shared projects [26]. As one can see in the `Scratch`

program example shown in Figure 2.1, although the user has to input some text, such as numbers or variable names, most of the code is already written in the code blocks that one can drag to the coding window. The distinct format of the blocks according to their type, is also quite helpful, because it points the user towards the correct block to complete the code. It also avoids syntax errors, because users can only fit blocks in ways that are syntactically correct.

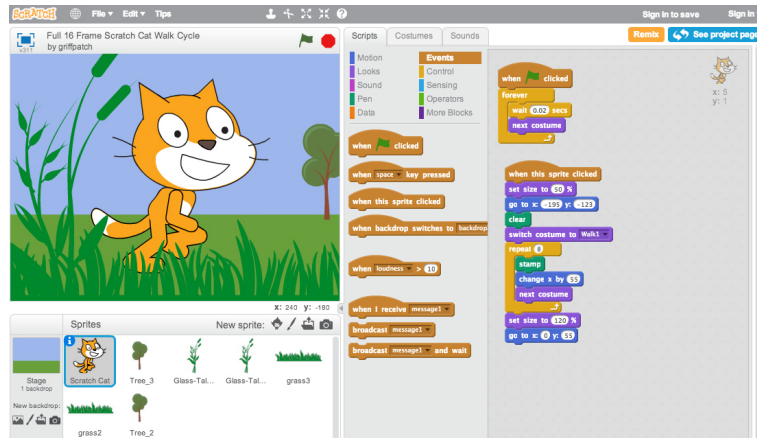


Figure 2.1: An example of a Scratch program.

Snap! [27] was developed by Jens Mönig in the University of California at Berkeley. In the words of the author it is an “extended reimplementaion of Scratch”. The added features are the ability to build customized blocks, as well as the implementation of first class lists, procedures, objects and continuations, in which it was inspired by the Scheme programming language. The idea is to use these added features to allow for a more serious introduction to computer science. Figure 2.2 shows the creation of a `for` block, which is not predefined in the Snap! blocks.

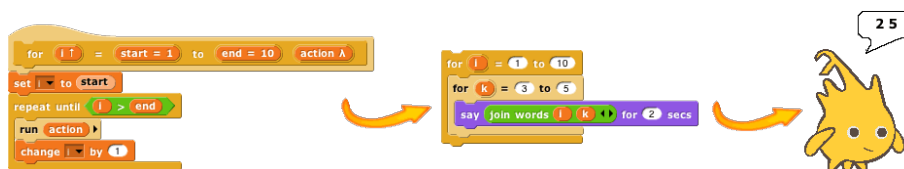


Figure 2.2: A Snap! program defining a `for` block.

Also an extension of Scratch is the game development tool Stencyl [28]. Developed by Jonathan Chung, it allows users to create their own Flash games for computers or smartphones, with the advantage of portability, that is, the same project can be exported into many different platforms. The IDE includes three components, the Scene Designer, that can be used to create the worlds for the game, the Actor Editor, that is used to create the game actors, and the Code Editor, where the game can be programmed by dragging code blocks. Figure 2.3 shows an example of a Stencyl game being developed.

Developed by Google, Blockly [29, 30, 31] is typically seen as another example of a block based language, quite similar in its format to Scratch. However, it is actually a library and an editor that can be used by developers to create their own block based languages. As shown in

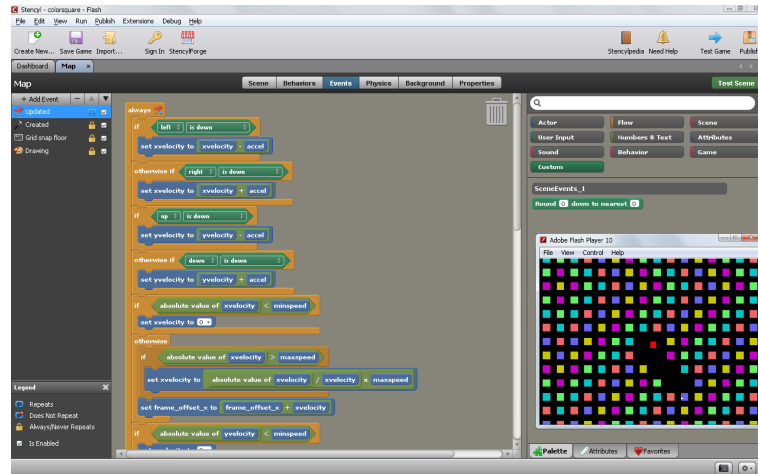


Figure 2.3: An example of a Stencyl game.

Figure 2.4, Blockly based languages usually have a text language generation feature, which can be used to create JavaScript, Python, PHP or Dart code. This ability can be particularly helpful for students wishing to transition from block based to text based languages. Even before they are ready to fully use text based languages, students can generate most of their code with blocks and then make small changes in the translated version of the code.

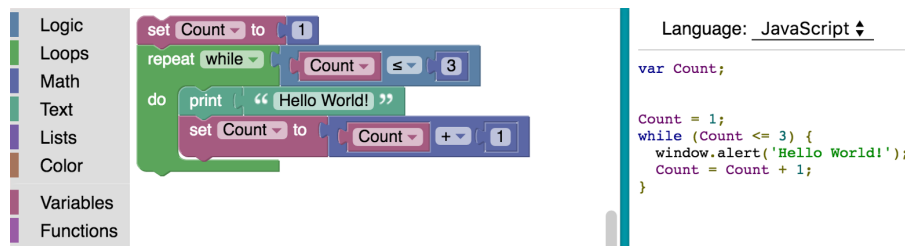


Figure 2.4: A simple Blockly program.

Microsoft's contribution to the block based coding languages comes with MakeCode [32]. Like Blockly, it can be translated to JavaScript. Also, users may chose to write their programs directly in JavaScript. It comes with a simulator that can help users to immediately see what their program does. Another interesting feature is that it can be used to program the very popular game Minecraft, as evidenced in Figure 2.5.



Figure 2.5: A MakeCode chicken generation program for Minecraft.

Although it is an event driven block based programming language, Alice [33, 34, 35] also belongs to the object based paradigm, which is based on the concept of objects with their attributes and procedures. Other distinguishing features of Alice are that it comes with its own IDE, shown in Figure 2.6, and it is typically used to create computer animations with 3D models. Also, used together with the Netbeans IDE, Alice code can be converted to Java. This is very helpful for introducing students to object based languages as they can move on to the staple of object oriented languages once they have matured enough.

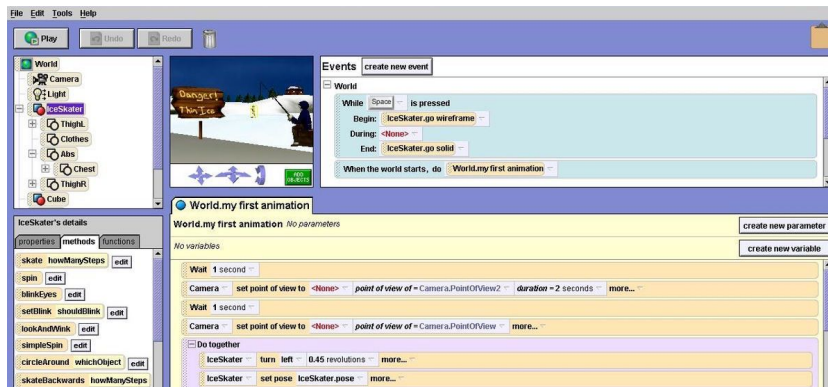


Figure 2.6: The Alice IDE with some code.

Another possible introduction language to the object oriented paradigm is Etoys [36]. The first step to create an Etoys program is to draw the objects one wishes to manipulate. Then, using the code blocks available, one can dictate the behavior of our drawn objects. For example, the program shown in Figure 2.7 tells the car to follow the green line.

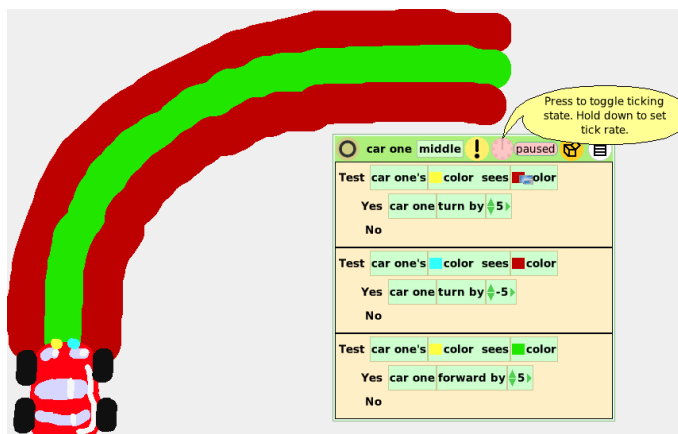


Figure 2.7: An Etoys example program.

The developers of Scratch were inspired by Alice and Etoys and intended to make the entry level of their language even lower while also making it appropriate for a wider variety of projects. It is fair to say that they have achieved these objectives. However, there is still room for improvement, as evidenced by the Stencyl and Snap! extensions of Scratch. Also, this was possibly part of the reason Google and Microsoft provided their own languages clearly

resembling scratch, but with further additions. For example, by providing the ability to translate to JavaScript, they provide a bridge towards text based programming languages. Recognizing this need for improvement as well as some of the better features of Blockly is probably one of the reasons why the development of recently released Scratch 3.0 is based on Blockly, which as the big plus of allowing it to run directly on tablets and smartphones.

### 2.2.2 Visual Programming Languages

In the context of this project, a visual programming language is one where there is no significant amount of text, that is, a programming language where reading or writing are not requirements. The programmer may use a simple word they know to refer to a variable or object in their program, but the syntax of the language itself contains no words, its made up of symbols or pictures.

Depending on the source, different definitions may be found. For example, according to the Wikipedia page on the subject [37] a visual programming language is one where the user does not need to write the code and can manipulate the elements of the language graphically. This includes all the languages in the previous section, while our definition does not. The reason for this is that we feel the word visual points towards images or icons and not text. Also, although eliminating the need to write greatly simplifies the process of coding, as there are no syntax errors, if the programming language elements only have text, then the user still needs to know how to read. So these languages are still text dependent, going against the frequent use of the expression ‘visual programming language’ in opposition to ‘text based programming language’.

As the name implies, ScratchJr [38] is a version of Scratch aimed at younger children, from five years of age. It comes in the form of a mobile application for tablet devices. The main difference is the substitution of the text with symbols, as can be seen in Figure 2.8. This is the primary reason for the drop in age, since reading is no longer a requirement. Unfortunately, with its simplification, ScratchJr also lost a lot of its power. The absence of conditional blocks, for example, highly constricts what the programmer can do. We do recognize a possible reasoning behind this, since as the child matures enough to be introduced to new coding constructs, they are probably ready to transition into Scratch. It is still a language change though.



Figure 2.8: An example of a ScratchJr program.

Kodu [39] was developed by Microsoft’s FUSE Labs for their operating system and their console. It is an IDE where, through element selection, the user can program games in 3D revolving around an eponymous robot. As shown in Figure 2.9, text is used, but it is made up of single words and they are accompanied by illustrative images. One of the key characteristics of this language is the fact that the behavior of the robot is programmed by rules. In practice this means that one needs to specify a list of events and corresponding actions. For example, to make the robot move according to a keyboard, the rule would be ‘when keyboard do move’, and the user simply needs to add the keyboard and the move blocks in their corresponding slots.



Figure 2.9: Selecting a code block in Kodu.

Lego started their Mindstorms (named after the Seymour Papert [23] book) robots line back in 1998 and they have always been programmable. In addition to the officially supported languages there have been numerous others. Together with the current Mindstorms generation, EV3 [40], released in 2013, Lego also released mobile and computer applications, with the design shown in Figure 2.10. The interface is similar to that of Scratch and other block dragging languages, but the symbols and images used are particular to this language. In Figure 2.10 we can see a sensor block and some motor blocks, but also blocks to create loops and to test conditions, which are very emblematic of programming languages.

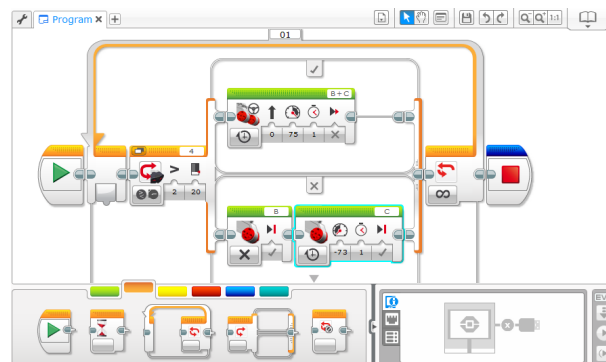


Figure 2.10: Programming the Lego Mindstorms EV3 robot.

Developed by Danny Yaroslavski, Lightbot [41] is a puzzle game that teaches programming while the user completes each level. Although it is not exactly a programming language it does



come with its own programming blocks, hence the inclusion in this section. As shown in Figure 2.11, the objective is to move a robot through a maze, lighting the blue squares along the way. In order to do this, the user can use basic blocks that turn on a light, or make the robot move or turn. More interestingly, there are programming constructs such as procedures and recursion, so that the same result can be achieved with less blocks. The small increments in difficulty at each new level, help the user learn how to use these coding blocks in a constructive way.

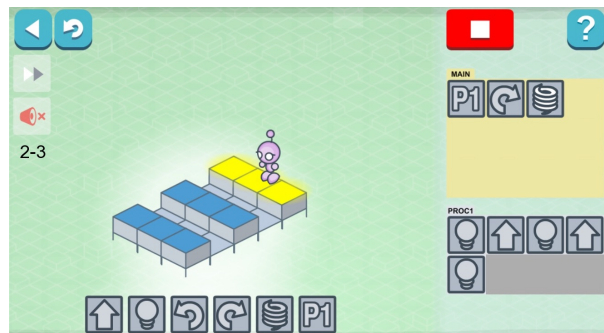


Figure 2.11: A level of the game Lightbot.

Like Lightbot, the Fisher Price Think & Learn Code-a-Pillar [42] is also a puzzle game that teaches very basic programming concepts. As can be observed in Figure 2.12 the objective is to guide a caterpillar to its destination, using visual blocks that can be dragged. However, it is aimed at children from four years of age, and for that reason the programming concepts it teaches are simple. The main take away is the effect of each of the thirteen different blocks (distributed between moving, overcoming obstacles and animations) and when it should be used to form the correct sequence. This a good basis to build on and some of the puzzles are already quite challenging for the intended age group. Still, it would be interesting to see the introduction of concepts such as repetition for the more advanced children.



Figure 2.12: One of the Code-a-Pillar application challenges

There are other applications with puzzle like levels aimed at teaching children and your adolescents to program, such as Coding Safari [43], SpriteBox [44] or codeSpark [45], to

name a few. New applications seem to be appearing all the time, which goes to show how much and how early programming is entering the lives of children.

Most of the languages exemplified in this subsection, in addition to removing text, also simplified the programming constructs available, in comparison to languages of the previous section. They also seem to be more dedicated, less project independent. But the EV3 language, although entirely dedicated to the corresponding robot, is quite capable. Unfortunately, some of its blocks, such as the motor controlling ones, are somewhat complex.

### 2.2.3 Tangible Programming Languages

A tangible programming language is one where the screen has been removed and replaced with something that can be physically manipulated by the programmer. The most frequent examples are made up of blocks or pieces that the user places together to obtain the program. In this sense, they are block based languages that instead of being graphically dragged or selected are physically grabbed and placed in the desired spot.

A formal comparative study of tangible and graphical languages aimed at children [19] revealed that there are several advantages of using a tangible interface, especially for younger children up to ten years of age. Two robot programming languages were used: the tangible T\_ProRob, see Figure 2.13, made up of cubes with symbols that can be combined, and its graphical isomorphic equivalent V\_ProRob, see Figure 2.14, that uses blocks with the same symbols.



Figure 2.13: The T\_ProRob blocks.



Figure 2.14: The V\_ProRob blocks.

A total of 109 children randomly split into pairs according to five different age groups performed two tasks, and a third task was given only to the older children. All groups used both interfaces, but half started with the tangible one and the other half with the graphical one. The measures observed were task completion time, error percentages, and debugging, that is, the percentage of error correction once an error was found. Performance was better across all measures for the tangible interface:

- Except for the oldest group, the completion time was better;
- There were less errors across all tasks and all groups;
- There was more partial and full debug.

But better performance is not all, the tangible interface revealed other advantages:

- Both students collaborated instead of one programming and the other observing;
- In free interaction with the languages, students interacted for longer periods of time, used a wider language vocabulary and, in the case of the older ones, achieved higher complexity;
- Students considered it more attractive, more enjoyable and, for younger children, easier to use.

Typical disadvantages of tangible languages are the higher cost and lack of portability. This is mostly related to the materials used, which very often include electronic components, and to the fact that the language needs to be physically replicated across multiple classrooms or schools.

Among the most influential tangible programming languages is `AlgoBlock` [46]. As shown in Figure 2.15, it is made up of cubes that can be connected together to form the program. The cubes are connected to a computer where the program is executed in a graphical interface. The cubes contain several commands, like move or turn, and there are also control blocks, such as condition testing or loop forming. Another interesting feature of these blocks is the presence of a parameter switch in some of them.



Figure 2.15: Children using the `AlgoBlock` language.

Also very significant is the language `Electronic Blocks` [47]. Shown in Figure 2.16, this language is composed of Lego Duplo Primo blocks with electronics on the inside. There are three sensor blocks in the left, four logic blocks in the center and three action blocks on the right. The code is created by attaching the blocks. For example, if one attaches a sound action block to a touch sensor block, then a sound will play when the sensor is touched. Compared to `AlgoBlock` this language is more limited, because it contains no control blocks. But it is self contained and completely eliminates the need for a computer. Plus, the blocks simultaneously make up the language and a robot.



Figure 2.16: All the different components of `Electronic Blocks`.

The `Cubelets` [48, 49, 50] robot seems to have been inspired by `Electronic Blocks`. Shown in Figure 2.17, these electronic cubes can be attached to form a variety of robots. The behavior of the robot depends on the cubes attached and their positions. There are four black sensor cubes: brightness, distance, temperature and knob. The five transparent ones correspond to actions: move, flash light, rotate, speaker and bar graph. The others are what they call `think Cubelets`: inverse, passive, maximum, minimum, threshold and blocker. There are also cubes for battery and Bluetooth. The main disadvantages of this kit are its price and the lack of typical programming constructs such as loops or conditional blocks. On the other hand, it is self contained and it makes use of the fun idea of building the robot and programming it at the same time.



Figure 2.17: A `Cubelets` robot and some other blocks.

In the same genre of programming while building the robot, but aimed at children between three and six years old, the Fisher Price `Think & Learn Code-a-Pillar` [51] can be seen in

Figure 2.18. It is very similar to the digital edition mentioned in the previous section. Each block has a specific function, illustrated by the symbol on top; two USB connectors (one male and one female); and some kind of electronic components on the inside. The blocks connect linearly to each other and to the head of the caterpillar, which has a female USB connector and a power button, that is used to begin executing the sequence represented by the blocks. As part of the more advanced set of blocks, there is a repeat block, where one selects the number of repetitions (between 1 and 5) using a knob. Unfortunately, only one block can be repeated, the one that is placed immediately before the repetition block. This is very limited, as the children cannot program the robot to repeat a sequence of several movements, that is they cannot create loops. However, given the target age group, it is still an interesting tangible programming language/robot.



Figure 2.18: A Code-a-Pillar robot with some of its component blocks.

The language TagTile [52], which was created to program the robot KUBO, uses only RFID emitters in its pieces. The puzzle pieces shown in Figure 2.19 are arranged in a linear sequence with specific pieces denoting the beginning and the end. Afterwards the robot goes over each piece in succession, recognizing the corresponding instruction. Once this is done, if the robot is placed on top of a run piece, it will execute the program previously read. In addition to the move, turn, sound and light pieces, there are also loops. Plus it is possible to use one procedure.

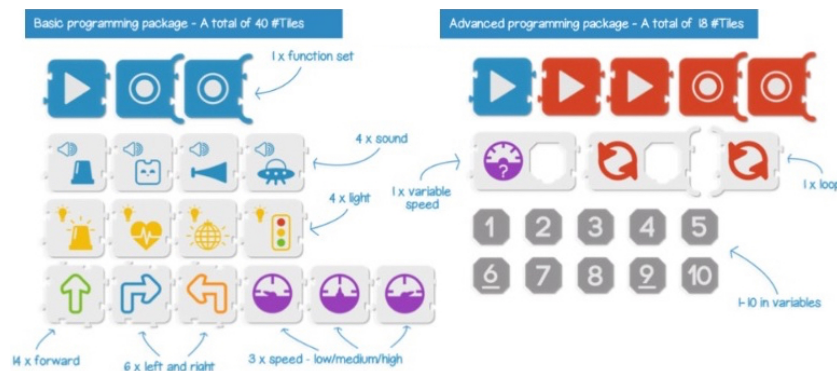


Figure 2.19: The TagTile puzzle pieces.

Quetzal and Tern [15] are the first languages in this list whose blocks have no electronic components. Quetzal, shown in Figure 2.20 is a language used to control the Lego Mindstorms RCX edition robot. Its interlocking plastic pieces include conditional blocks, infinite loops formed

through a merge statement, concurrent tasks and parameters. Shown in Figure 2.21, the puzzle like wooden blocks of Tern contain conditional statements, loops, subroutines and parameters. There are also coiled wires that form the goto statements of text based languages. With Tern the users can program virtual robots running on a computer.



Figure 2.20: A program written in Quetzal.

Developed after Quetzal, one of the objectives of Tern was to make the users focus more on programming and less on building robots, for which the authors made a case defending an hybrid approach in which the computer runs the tangible code, similarly to AlgoBlock. This is an interesting approach since it is the opposite of most robot programming solutions, in which the execution environment is tangible and the coding one is graphical.

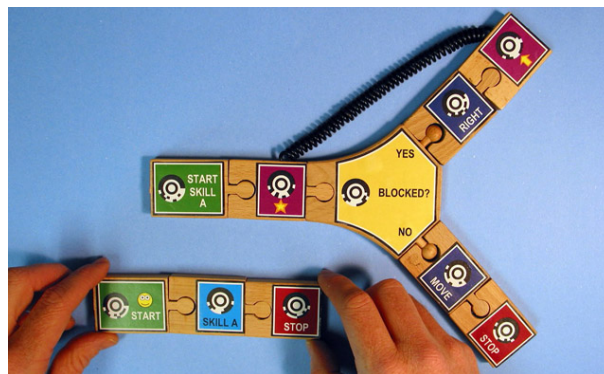


Figure 2.21: A program written in Tern.

T-Maze [53, 54] also employs no electronics in the language elements. As can be seen in Figure 2.22, there are three electronic sensors (button, temperature and light), and they have corresponding cubes which are used together with other cubes to program. The cubes have images illustrating their type, like the start cube and the end cube with their doors. There is also a ‘normal’ cube, loop cubes and movement cubes. To program with T-Maze, one builds a maze using the cubes. The maze is captured in a camera and the computer vision system TopCodes (circular markings on the cubes) is used to translate it into the computer. The idea is that the code should lead a character in the computer from the start to the end of the maze. Whenever this character

hits a part of the maze with a sensor cube, the user must manipulate the corresponding sensor for the character to proceed.

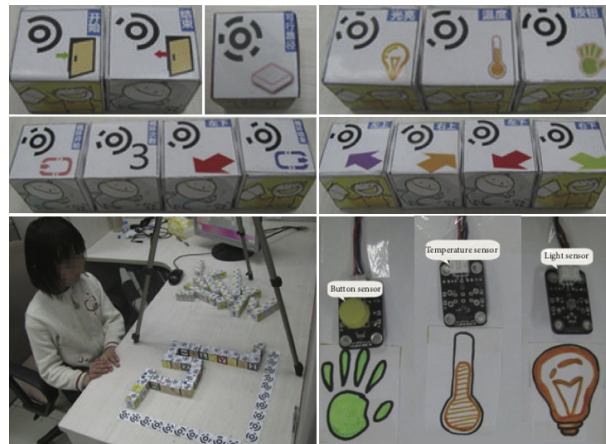


Figure 2.22: The T-Maze blocks, a maze and the sensors.

Osmo is a brand of tangible meets digital educational toys. The general idea is to use the camera on a tablet to capture the tangible pieces, so that the children can interact with what is on the screen by manipulating the pieces. Three of their toys are dedicated to teaching children the basics of programming and can be acquired in a set called *Coding Family* [55], part of which can be seen in Figure 2.23. The first toy of the set, in addition to the typical sequencing, makes use of parameters, loops and conditionals. The second toy introduces subroutines and nested repeats. Finally, the third toy teaches synchronization and makes use of all the previous games constructs to create more complex programs. This is an impressively complete set of programming constructs as well as an interesting use of the graphical interface with tangible tiles, especially considering that the pieces have no electronic components.



Figure 2.23: The Osmo Coding Family base and some of its blocks.

Code Bits [56] is also made up of puzzle pieces without any electronics. In fact, as shown in Figure 2.24, they are just pieces of paper. In addition to these pieces there are also mazes printed in sheets of paper and an accompanying mobile application. The user connects the pieces in succession in such a way that will lead a character from the beginning to the end of the puzzle, much like T-Maze. The camera of the phone is used to capture parts of the code, which it then executes in augmented reality over the maze.



Figure 2.24: The Code Bits puzzle pieces, one of the mazes and the mobile application.

The current tendencies in tangible programming languages are mostly moving away from electronics in pieces, thus reducing the cost, and into image processing and computer vision techniques, to capture the tangible code so that robots or applications can run it. This seems to be a good idea, since it helps to eliminate the main disadvantages of tangible code, while keeping all its positive aspects.

## 2.3 Educational Robots

Educational robot kits are very common these days and many of them are being sold directly to the families, instead of just to schools. Together with mobile applications or other ways to program the robots, they usually have tutorials so that children can learn to code while playing. In order to attract more buyers, cost reduction is a common concern. With lower costs, schools can buy packs of robots and families can also join the fun without risking bankruptcy. The widespread use of mobile devices also contributes to the success of many of these robots.

Perhaps the best known educational robot kit is the Lego Mindstorms. As mentioned before, the current EV3 [57] edition, shown in Figure 2.25, comes with a mobile or computer application that can be used to program it. But the wide community of users have contributed to many other ways to program these Lego robots. They come with color, touch, ultrasonic and gyroscope sensors. Actuators are comprised of motors, lights, a speaker and a display. For communications it comes with Bluetooth and infrared remote control. Together with the fact that it is compatible with regular Lego pieces and can be built into an infinity of models, this is a very powerful robotic kit. The selling price is 400€.

Developed by Anki, Cozmo [58] is a robot that appears to have a personality, like when it gets bored if ignored, or when it is a sore loser. Pictured in Figure 2.26, it comes with three cubes that it can pick up and use to play games. Equipped with a gyroscope, an accelerometer and a VGA camera, one of the key features of Cozmo is its facial recognition capabilities. In fact, much of the users interaction with this robot is similar to owning a pet, which you have to feed and exercise.



## STEM Education, Programming and Robots

Table 2.1: Educational Programming Languages Comparison

Language	Blocks	Visual	Tangible	Paradigms	Constructs	Availability	Price	Quantity
Scratch	yes	no	no	procedural, event driven	sequences, events, flow control, lists, variables, procedures	online	0€	∞
Snap!	yes	no	no	procedural, event driven	sequences, events, flow control, lists, variables, procedures	online	0€	∞
Stencyl	yes	no	no	procedural, event driven	sequences, events, flow control, lists, variables, procedures	download	0€	∞
Blockly	yes	no	no	procedural	sequences, flow control, procedures, variables, lists	online	0€	∞
MakeCode	yes	no	no	procedural, event driven	sequences, events, flow control, procedures, variables, lists, arrays	online	0€	∞
Alice	yes	no	no	object oriented, procedural	sequences, flow control, procedures, functions, objects, variables	download	0€	∞
Etoys	yes	no	no	procedural	sequences, flow control, procedures, variables	download	0€	∞
ScratchJr	yes	yes	no	event driven	sequences, events, loops	tablet app	0€	∞
Kodu	yes	yes	no	rule based, event driven	sequences, events, flow control	download	0€	∞
Mindstorms	yes	yes	no	imperative	sequences, flow control, variables	download	0€	∞
Lightbot	yes	yes	no	procedural	sequences, procedures	mobile app, online	0€	∞
Code-a-Pillar app	yes	yes	no	imperative	sequences	tablet app	0€	∞
V_ProRob	yes	yes	no	imperative	sequences, flow control	none	-	∞
T_ProRob	yes	yes	yes	imperative	sequences, flow control	none	-	?
AlgoBlock	yes	yes	yes	imperative	sequences, flow control	none	-	?
Electronic Blocks	yes	yes	yes	imperative	sequences	none	-	?
Cubelets	yes	yes	yes	imperative	sequences	purchase	250€	12
Code-a-Pillar	yes	yes	yes	imperative	sequences	purchase	60€	8
TagTile	yes	yes	yes	procedural	sequences, procedures, loops	purchase with robot	256€	46
Quetzal	yes	yes	yes	imperative	sequences, flow control	none	-	?
Tern	yes	yes	yes	imperative	sequences, flow control	none	-	?
T-Maze	yes	yes	yes	imperative	sequences	none	-	?
Coding Family	yes	yes	yes	procedural	sequences, flow control, procedures	purchase	159€	42
Code Bits	yes	yes	yes	imperative	sequences	none	-	?



Figure 2.25: The Lego Mindstorms EV3 robot in two of its many possible forms.

It can move around, uses a display that shows his eyes, with several different expressions, and it can also speak. But more importantly for us, it can be programmed, either using its own mobile application with a version of ScratchJr or Scratch, or using its SDK. It is sold at 150 €.



Figure 2.26: Cozmo and its cubes.

The robot KUBO [52] was already mentioned in the previous section, due to its TagTile programming language. Shown in Figure 2.27, this little robot can detect and recognize the TagTile puzzle pieces using RFID technology. It can also move while balancing on its two wheels, make sounds and change the color of the LEDs on its neck. Its price tag is 324 €.



Figure 2.27: The robot KUBO going over a program in TagTile.

KIBO [59, 60] is a funny looking robot, as can be seen in Figure 2.28. It can be programmed with a set of wooden blocks, has sensors for sound, light and distance, and in addition to the motors that control its wheels, it also has LEDs as actuators. The robot also has a scanner to detect the barcodes in the wooden programming blocks, which is how it recognizes the language. It comes in four robotic kits with prices between 190 € and 420 €, depending on the number of coding blocks included.



Figure 2.28: The robot KIBO and some of its tangible programming blocks.

Developed by Makeblock, mBot [61] comes in a construction kit made up of 40 parts and intended to stimulate invention skills in children. The constructed robot is pictured in Figure 2.29. It is equipped with light, ultrasonic and line following sensors. For actuators, it has a buzzer, a LED and two motors for its wheels. It also has four ports that can be used to connect additional sensors. It can be programmed with the language mBlock, which is based on Scratch. The selling price is 80 €.



Figure 2.29: The mBot robot.

The Lego Boost [62] robotic kit, is aimed at younger children than Mindstorms. It is equipped with color and distance sensors, plus some motors. It can be programmed with a mobile application specific for the effect, whose language is block based and visual. It is sold for 160 €. Its main advantages are the fact that it can be built into multiple robots, some shown in Figure 2.30, and that it is compatible with Lego bricks. While the limited availability of sensors and actuators are the principal disadvantages.

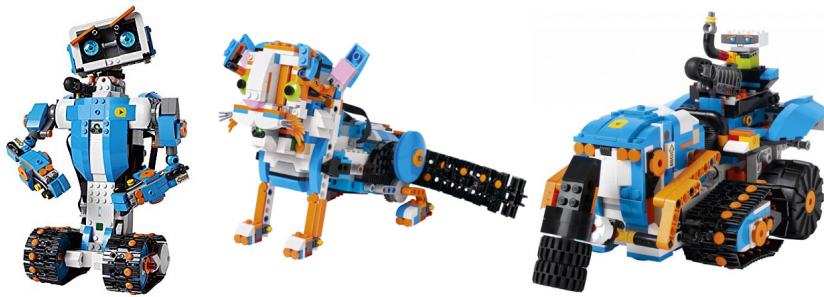


Figure 2.30: The Lego Boost robot in three of its possible shapes.

Thymio [63, 64, 65] was built to be a fully featured, reasonably priced educational robot. Shown in Figure 2.31, it has nine infrared proximity sensors, five capacitive touch buttons, an accelerometer, a thermometer and a microphone for sensors. It also has 39 LEDs, two motors and a speaker for actuators. For communication it is equipped with an infrared receiver and a wireless internet card. It is also able to hold a pencil and pull a trailer. Finally, it has a microSD card reader. To program it one can use its own visual programming language, Blockly or Scratch. It is sold for 155 €. The main advantages are its many features and the fact that it can be programmed with multiple languages.



Figure 2.31: The robot Thymio.

Dash [66] is an educational robot that comes with several accessories and even a little friend named Dot, many of which are displayed in Figure 2.32. It has sound and distance sensors, motors, speakers and several LEDs. It is also equipped with infrared receivers and transmitters for communication. There are quite a few mobile applications to interact with Dash and it can be programmed using Blockly or the Swift Playgrounds application from Apple. It also comes with accessories that make it Lego compatible, which is a nice feature. Starting at 130 €, the basic kit does not include Dot and comes with a single accessory. To get the complete pack, one needs to spend 230 €.

Ozobot evo [67] is a tiny robot with some interesting features. As shown in Figure 2.33 it is about the size and shape of a ping pong ball, flattened at the bottom. It has proximity sensors and an optical sensor at the bottom with which it detects the color of the lines beneath it. For actuators, it has motors, LEDs and a speaker. It can be remote controlled with a mobile application and



Figure 2.32: The robot Dash and its little companion Dot.

programmed with two languages: Ozoblockly, a language created using Blockly to program Ozobot; or Ozocodes, which are color codes that change the behavior of the robot, like making it go faster, slow down or rotate. This color coded language is actually the most interesting feature of the robot. The Ozobot *evo* is sold for 100€, while the less featured *Bit* version, which does not have Bluetooth or proximity sensors, costs 60€. There are also discounts for schools buying packs with several robots.



Figure 2.33: The Ozobot and its color coded programming language.

The educational robot Sphero [68] is made by the eponymous company. Pictured in Figure 2.34 in its *SPRK+* version, it is about the size and shape of a tennis ball. It is equipped with an accelerometer and a gyroscope as sensors. For actuators, it has LEDs and motors. It can be remote controlled by a mobile application and communication is done via Bluetooth. There is also a mobile educational application to program the robot, which can be done in three different languages: drawing, block based and JavaScript. The drawing language simply tells the robot how to move and which color to display. The other languages are fully featured. There are tutorials with increasing difficulty levels for all languages and there is also a lot of community shared programs. The best qualities of the robot come precisely with its educational application. It is priced at 110€.

The last robot in our list is the smartphone robot Robobo [69, 70], pictured in Figure 2.35. The robotic base is equipped with two motors for the wheels and, for the phone holder, tilt and pan



Figure 2.34: The robot Sphero in its SPRK+ edition.

motors. In terms of sensors, the base has nine infrared distance sensors and four odometric sensors in the motors. The smartphone, depending on the model, has two cameras (front and back), proximity, light and temperature sensors, gyroscope, accelerometer, magnetometer, GPS, microphone and touch screen. Plus, it also has a speaker and an high resolution display for actuators.



Figure 2.35: A Robobo robot.

Being a smartphone robot, despite the low power processor on the base, equipped with a mid-range phone, Robobo has very high processing capacity. It also has a Bluetooth connection on the base, and 3G/4G, WiFi and USB on the phone.

In terms of programming, Robobo can be programmed with block based languages, using a Scratch extension or their own IDE based on Blockly. Advanced users can program it with the Robotic Operating System (ROS) in Python or C++. They also have a Java native framework that allows users to build applications for the Android smartphone that can interact with Robobo. The developers of Robobo already made two applications, one for programming with Scratch and the other for ROS. They also made a series of interactive lessons that can be used in schools to teach students to program Robobo. The price tag is 363 €.

Table 2.3 presents a summary of all the robots we have listed. There are many options for educational robots, and we have only covered the most relevant. Prices vary and so do the features,

in most cases one increases with the other. Most programming languages depend on mobile applications, but there are some tangible or screen free options as well. With all the robots available and the increasing interest from parents and schools, many kids are having early access to these devices.

### **2.4 Conclusion**

Nowadays there are plenty of options to support STEM teaching. Even stores started having their STEM corners, with experiments, robots and games dedicated to these areas. For programming and robotics, the number and versatility of the current offer is already very interesting. Which is good, because parents and schools can make their choices from a wider variety and give their children toys that will amuse them while teaching.

When it comes to educational programming languages, experiments show that the tangible, visual, block based combination is the best overall. It provides a better collaboration environment, engages kids for longer periods of time, lowers the entry barrier and leads them to achieve the best results.

As for the robots, having more features without increasing the price seems to be the current challenge, since the inexpensive models usually have less sensors and actuators.

## STEM Education, Programming and Robots

Table 2.3: Educational Robots Comparison

Robot	Age	Price	Coding	Processor	Sensors	Actuators	Comms	Others
EV3	10+	400 €	EV3, RobotC, Scratch, C, C++, C#, Java, Python,...	ARM926EJ-S core @300MHz	touch, color, infrared	178×128 Monochrome LCD, 3 motors	USB, WiFi, Bluetooth	remote control, microSDHC, multiple robots
Cozmo	8+	150 €	Scratch, ScratchJr, Python	no info	camera VGA 30fps, cliff, encoders and IMU	speaker, 128×64 OLED display, motors for: head, lift, threaded wheels	Bluetooth, IR receiver, IR transmitter	facial recognition, programmed like pet, plays games
KUBO	4-10	256 €	TagTile	no info	RFID	motors, RGB LEDs	WiFi	includes tangible language
KIBO	4-7	190 €-420 €	wooden blocks	no info	sound, light, distance, barcodes	motors, LEDs	—	includes tangible language
mBot	8+	80 €	mBlock	based on Arduino Uno	light, ultrasonic, line follower	buzzer, RGB LED, 2 motors	—	4 ports for more sensors
Boost	7-10	160 €	visual	ARM Cortex M0 @48MHz	color, distance	motors, RGB LED	Bluetooth	multiple robots
Dash	5-8	130 €-230 €	Blockly, Swift	ARM Cortex M0	3 microphones, 3 distance sensors	speaker, motors, RGB LEDs	Bluetooth, 2 IR receivers	multiple accessories
Thymio	6+	155 €	visual, Blockly, Scratch, text	PIC24 @32MHz	5 capacitive touch buttons, accelerometer, 5 proximity, 2 ground, microphone, temperature	2 motors, 39 LEDs, speaker,	USB, WiFi, IR receiver	pencil holder, trailer hook, memory card
Ozobot	8+	75 €	Blockly, Ozocodes	no info	4 proximity, 1 optical (color)	2 motors, LEDs, speaker	micro USB, Bluetooth	remote control mobile application
Sphero	8+	110 €	drawing, block based, JavaScript	ARM Cortex	accelerometer, gyroscope	2 motors, 2 RGB LEDs, 1 blue LED	Bluetooth	remote control mobile application
Robobo	10+	363 €	Blockly, Scratch, ROS, Java	base: low capacity phone: depends on model	base: 9 IR proximity, 4 odometric encoders phone: 2 high-res cameras, proximity, light, temperature, gyroscope, accelerometer, magnetometer, GPS, microphone, touch screen	base: 4 motors, 9 RGB LEDs phone: speaker, high-res LCD screen	base: Bluetooth phone: 3G/4G, WiFi, USB	includes interactive lessons, accessories



## Chapter 3

# Problem Statement

This chapter formalizes the problem we wish to solve. We begin with a summary of the gaps in the existing solutions, which we intend to explore, and end with an in-depth description of our goals.

### 3.1 Introduction

According to Seymour Papert [23] an ideal educational programming language should have “low floors”, in the sense that new users can quickly pick up its simplest examples, and “high ceilings”, in the sense that experienced users can still use it in more complex projects. The creators of *Scratch*, Resnick et al. [8], added to this room metaphor the concept of “wide walls”, meaning that there should be a multitude of different paths for students to go from the low floor to the high ceiling whilst they are learning.

Evidence shows that tangible programming languages, that is, languages with blocks that can be physically manipulated, have many educational advantages, particularly for younger children. According to [16], people are more likely to interact and collaborate with tangible interfaces. Also, formal comparative studies [18, 19] between a graphical and a physical interface for the same programming language, revealed the following advantages of the tangible version: children find it easier and more enjoyable; they complete programming tasks faster, make fewer mistakes and are more likely to debug the errors they do make; in free interaction, the children spent more time with the tangible interface, created more complex programs and used a wider variety of commands.

Unfortunately, tangible languages are harder to execute, because they need to be captured into a device with computer like capabilities. They are also more expensive and less portable than their graphical or text based counterparts. More than that, with the notable exception of *Osmo Coding Family*, tangible languages fail to implement the variety of programming concepts necessary for the students to be able to reach higher levels of complexity. As for having many different paths of learning, the currently available tangible languages are extremely lacking, since they were all

## Problem Statement

created for a single purpose, such as programming a robot or getting a game character to follow a certain path.

In the wordings of Papert and his pupil Resnick, we want to create a tangible programming language whose floor is even lower than that of the current educational languages. But we also want to keep the high ceilings and wide walls that languages like `Scratch` have accustomed us to.

### 3.2 Ideal

The importance of having a low entry level in any educational tool is that early success is one of the best motivations any student can have. People do not like to fail and the frustration of not being able to complete a task or understand a concept often leads children to give up. On the opposite side, succeeding makes people feel good and they associate the positive feelings with what they were trying to learn.

Programming is an inherently difficult task, it is abstract and implies molding our thought processes into specific patterns hardly ever used in other tasks. It is easy to make mistakes and many of those are hard to detect and fix, it requires perseverance. But this is also what makes learning to program so important, it teaches children how to approach a problem, how to divide a complex task into smaller steps that they can solve, how to deal with frustration when their solution does not work as intended, how to spot what went wrong and how to repair it.

If we want to make children enjoy a pursue of something as demanding as programming, we need to focus on making it as easy as possible for them to take their first steps in that field. However, this is not all, they also need to be able to grow in their knowledge. For starters, if they can only do the simple things, they will quickly get bored. But also, they will never realize their potential and truly learn the discipline. That is why a good educational programming language must try to implement the programming concepts and constructs that the other programming languages do, so that children can keep learning with the same language and build increasingly involved projects.

Finally, a good educational programming language must capture the interest of as many children as possible. But children have diverse interests and things that they prefer doing. Thus, the language should be capable of realizing a variety of different types of projects, as well as, accommodate distinct ways of thinking, because the same problem can have many solutions and each child should be able to implement their particular solution.

### 3.3 Reality

As we saw in Section 2.2, there are many educational programming languages and the field has been getting increasingly more attention. Researchers and software companies have invested a lot into creating the ideal educational language. Some, such as `Scratch`, have mostly accomplished

## Problem Statement

this. But their recommended starting age is eight years old. Plus, it requires reading at a good level, in order to understand the blocks.

On the other side, we have the tangible languages, whose concern with lowering the entry level is clear, since they rely on visual clues instead of text and their physical interface is both more appealing and easier for the younger children. But these have other issues, they are all single purpose languages, so they fail to reach the wide walls ideal. Plus, almost all of them also fail at the high ceilings ideal.

Then, there are the issues of price and lack of portability. A great deal of tangible languages use electronic components, but is reflected on their price. Which explains why the recent examples that use this strategy are those in which the language is also part of the machine that executes it, namely, they are simultaneously languages and robots.

An alternative that completely avoids the electronics is to use computer vision. This is the technique of *Quetzal* and *Tern*, as well as *T-Maze* and *Code Bits*, that use fiducial markers to identify the pieces. There is also *Osmo Coding Family*, where the blocks are camera captured and identified by their visual elements, without any fiducial markers.

To the best of our knowledge *Quetzal*, *Tern* and *T-Maze* are not commercially available. *Code Bits* seems extremely inexpensive, since one just needs to print and cut paper pieces. But we could not find it available for printing nor could we find the *Android* application necessary to run it. Also, it is probably the most limited of these languages, because it only allows sequences of forward and turn movements.

*Coding Family* costs 159€ and ships with 42 coding blocks. It is quite ambitious in terms of the programming concepts it teaches. But one can only use the blocks with their three game like applications, which are exclusively available on some *Apple* or *Amazon* tablet devices.

All these tangible languages are either expensive, limited in their capabilities or have a small number of blocks. Many have all these issues. None of them can be used in the creative ways that children have been using *Scratch*, which has nearly 38 million projects shared, from programming a (virtual or real) robot to follow a line, to making interactive postcards or creating games.

### 3.4 Consequences

Currently, schools and parents have to choose between free graphical solutions, that can only be used by older children, or limited and expensive tangible solutions, that cannot accompany the children in their learning path for long enough and that will quickly bore them.

It is hard to justify the steep investment in something that children will quickly outgrow. So most will opt to wait, meaning that many children will miss the opportunity to acquire their first programming concepts early on, as well as all the advantages that could bring, such as formation of specific neural pathways.

Those that begin with something like *Scratch* before they are prepared to will associate negative feelings with the discipline and cast it aside. That is a potential for learning, enjoyment

and creative expression that might not be realized. One will never know how far a child could reach if one keeps missing the best opportunities to teach them.

### 3.5 Proposal

We propose to create *Tactode*, an inexpensive tangible language whose entry level allows children to start programming earlier and whose capabilities allow complex and varied projects in a multiplicity of platforms.

Due to our focus on learning and the constricted budgets of most schools, particularly the public ones, as well as a belief that education is a fundamental right of everyone, independently from their financial status, a big concern for *Tactode* is to keep its cost as low as possible. This preoccupation is mainly revealed in the materials used for the blocks, but is also reflected on the compilation and execution platforms.

Tangible programming languages are, by design, block based, that is, each language command is represented by a block. This because it would not be feasible to manipulate several physical characters to compose a single command, as one does with text based languages. With this in mind, with *Tactode* we aim for not only a block based language, but one where the number of different blocks is as small as possible, without compromising the range of possibilities.

Among the block based languages, there is an important distinction between the visual ones and the ones that rely mostly on textual elements. They are both easier to learn than the traditional languages, because the problem of knowing which commands exist and how exactly to write them does not exist. But the visual languages, due to their independence from text, are even more user friendly. They require no reading skills and can be used without translation by children from any native tongue. That is the reason why *ScratchJr*, which is directed at children younger than those targeted by *Scratch*, is visual in addition to block based.

The problem with visual languages is that they are either limited in the different notions they express or they quickly become very complex and hard to understand. Of course some notions, such as movement can be easily expressed with an image or a symbol. But others, such as the flow control structures that most programming languages employ, are harder to capture that way.

With these conflicts in mind, we plan to use both text and visual elements in *Tactode*. The idea is that those that can read can make better sense of the images and symbols, but that is not strictly necessary. Also, children can start with simpler blocks and as their reading skills improve learn to use the more complex ones. Plus, given that we mean to use English for the text, children can begin to learn the keywords used in most programming languages.

One of issues of tangible programming languages is that they are typically for a single purpose. For example, there are some robots, like *KUBO*, *Cubelets* and *Code-a-Pillar* that can be programmed with tangible languages, but each robot has its own. The same thing is not true for the block based languages with a graphical interface, there are plenty of robots that use a *Scratch* based language, such as *Cozmo*, *mBot*, *Thymio*, *Robobo* and *Sphero*, which made the change when *Scratch 3.0* implemented some of ideas and the technology used in *Blockly*.

## Problem Statement

When designing `Tactode` we aim at a multi-platform experience, where the language could be a constant for a multitude of targets. The first focus, due to their engaging nature and educational value was on robots, but we also consider non-robotic platforms, because they allow schools to keep the cost low, are easier to maintain and in some cases more practical to use. Plus, they bring forward a wider variety of different kinds of projects that can be implemented. This objective is one of the main distinctions between `Tactode` and the other available tangible languages.

Our final main goal for `Tactode` is to make it capable of growing with the users. When first introduced to programming, children mostly learn some basic commands and sequencing. But as their familiarity with the language increases they can gradually and progressively be introduced to more complex programming concepts, such as choices, loops, procedures and recursion. We want to create a language that will be simple for first attempts, but also able to accompany children through their advancement.

### 3.6 Research Questions

From the goals we want to fulfil with this project, some research questions arise, that will be answered throughout the remaining of this document.

**RQ 1.** *Given that we mean obtain a safe and interesting product with a good price, what materials should we use and how should we manufacture the `Tactode` blocks?*

**RQ 2.** *Considering that we want young children to find programming in `Tactode` easy, what shape should the blocks have and how should they be arranged to create programs?*

**RQ 3.** *What visual and textual elements should we use to make the purpose of each `Tactode` piece evident for young children?*

**RQ 4.** *In order to support multiple paths of learning, which programming paradigms should the `Tactode` programming language support?*

**RQ 5.** *Considering that it should be capable of accompanying children in increasingly complex projects, which programming constructs should `Tactode` implement?*

**RQ 6.** *Having in mind that we aim to appeal to children and allow varied types of projects, which target execution platforms should we use?*

**RQ 7.** *How should we capture a tangible program into a computer and identify each piece and its position in the program?*

**RQ 8.** *How can we compile the captured `Tactode` programs so that they can be executed in each target platform?*

**RQ 9.** *How can we get the compiled `Tactode` programs into each target platform to be executed?*

Chapter 4 is meant to answer the Research Questions 1 to 6, while Chapter 5 should answer the Research Questions 7, 8 and 9.

### 3.7 Conclusion

In order to lower the entry level of educational programming languages we decided to create the tangible programming language `Tactode`. We also want this language to be able to grow into the more demanding projects that children will create as they reach higher depths in their learning. Finally, we want each child to be able to chose from a variety of execution platforms and types of projects, according to what interests them and captures their attention.

To achieve our goals for `Tactode` we must choose the right materials and manufacturing processes, design the block shapes, visual and text elements, select the programming paradigms and constructs to implement and define the execution targets. Plus we have create an application that is capable of capturing the `Tactode` programs, compiling and exporting them to be executed in each target platform.

## Chapter 4

# Tactode- A Tangible Programming Language

This chapter exposes the development of our tangible programming language. We present the alternatives we considered for its development and justify our final choice. After this we show the language itself and explore its elements, giving examples of typical constructions. We also present an evaluation of our language, exposing its features and limitations.

### 4.1 Introduction

A programming language is considered tangible when its components have to be physically manipulated to create the programs. This is not the case of most programming languages, in which the interaction is accomplished through a machine, typically a computer, and the interface is virtual. In fact, the large majority of programming interfaces are text based, with only a few graphical examples and even less tangible ones.

Given the purpose of programming languages, which is to define the behavior of machines, their distribution amongst tangible and non-tangible ones makes perfect sense. Indeed, any language will eventually have to be compiled and interpreted by a machine and the difficulty of that process is increased when the elements of the language are physical, because one has to first capture them to the machine. This can be done using electronic components in the tactile language blocks, or using computer vision to identify each piece and its location.

Independently of how the tangible code is captured into the machine that will execute it, this is only the first of the added difficulties of this kind of interface. Even when no electronic components are involved, the fact that the blocks have to be made of some kind of material adds to the cost. That is why many of the commercially available tangible languages are shipped with a very limited number of blocks. There is also the distribution issue, because unlike their virtual

counterparts, tangible languages have to be physically transported or manufactured in each place where they will be used.

With all their added complications, it is natural to wonder why one would use or create a tangible programming language. The reason is education. These languages are clearly not well suited for commercial software development or for experienced programmers, but they are very adapt for those that are taking their first steps in the world of programming, particularly the younger ones. Indeed, studies have shown the advantages of tangible programming languages in education environments

Given the educational advantages in the difficult endeavour of introducing people, particularly children, to programming, investment in tangible languages is justified. Of course one still needs to mitigate the disadvantages, in order to make this kind of interface competitive and increase its adoption. This is especially critical for schools, where the costs of supplying entire classrooms can mount very fast. Our tangible programming language, Tactode, was developed with these ideas in mind. We aimed at an affordable, reliable and resilient product, that can be easily obtained and used by schools and families to teach children to program, without compromising its capabilities.

## 4.2 Alternatives

In order to design Tactode we had to make some choices regarding: the design of the blocks, such as their shape and how they would fit together; the materials and manufacturing processes that would be used to create the physical pieces; the programming paradigms that Tactode should adhere to; which tools to use for piece recognition; and the target platforms, that is, the robots and non-robotic platforms in which Tactode would be executed. This section presents the alternatives we considered and the decisions we made.

### 4.2.1 Block Shape

The current basic shape for the Tactode blocks, and answer to RQ 2 is shown in Figure 4.1. In essence, each piece is a rectangle with possible trapeze shaped slots on the left and top, and tabs on the right and bottom. The idea is that all pieces fit easily into each other. Whether a specific piece has a certain slot or tab, depends on its function.

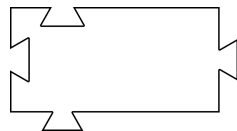


Figure 4.1: The basic shape of a Tactode piece.

An alternative considered was using star shaped blocks which one would be able to connect directly to several other blocks. Another idea was to have pieces with several slots, but not tabs. Then, pairs of these pieces would be put together with the help of connectors (possibly using magnets). The star shaped pieces could be used this way.



On the opposite end, we also considered an approach where the tabs and slots would have varying shapes, so that only the pieces that are supposed to go together to form a syntactically correct program would fit. This would minimize the number of possible connections instead of maximizing it, like the star shaped blocks.

In a way, our solution is the middle ground between these two ideas. There are many possible ways to connect the pieces, considering that all tabs fit into all slots. But we have limited this, by making sure each piece only has the necessary tabs and slots, given its purpose. We believe this forces the programmers to use critical thinking and to debug programs in which the wrong connections were made.

#### 4.2.2 Materials and Manufacturing

The main goals for the materials and the manufacturing process were to keep the cost low and to obtain a durable and high quality product. Also, we wanted to create pieces with different colors for the different types of pieces. We considered the following options:

- 3D printed Acrylonitrile Butadiene Styrene (ABS) or Polylactic Acid (PLA) plastic pieces with colored printed paper glued on top;
- laser cut plywood, acrylic or traffolyte pieces with color printed paper glued on top for the the design;
- laser cut and color printed cardboard pieces (like the traditional puzzle pieces);
- 3D printed ABS cutters to mold pieces from colored polymer clay, where colored printed paper would be glued;
- Ethylene Vinyl Acetate (EVA) pieces cut with laser, metal cutting dies or manually with an x-acto knife, with either printed paper on top for the design or laser printing on the EVA pieces directly.

Although 3D printing has become quite common, it is still a relatively expensive medium, especially for large quantities. Also, plastic pieces are not the best at fitting together due to their very low malleability. Thus, we quickly eliminated this option from our considerations.

We inquired for budgets on laser cutting and the lowest price was plywood, which was 80€ for 150 pieces. At the time, this discouraged us from pursuing more laser cutting options. There were also some technical difficulties with using laser to cut EVA, due to the high probability of burning and the fumes released. However, we believe other companies provide that service.

The current Tactode pieces are built using two A4 EVA sheets with an adhesive side and 2 mm of thickness. These two sheets are glued together using the adhesive of one of them. The adhesive of the other is used to glue regular color printed A4 paper. Finally, in order to increase durability we added a layer of clear self-adhesive plastic on top of the paper. But this has the effect of creating reflections on some lighting conditions, impacting the code capturing process in a very negative way. Thus, we do not recommend this plastic layer for all situations. Indeed, in answer

to RQ 1, the recommended materials and manufacturing process for Tactode is to use printed paper glued to 4 mm EVA sheets, that should be manually cut.

We printed, glued and cut nearly 2000 Tactode pieces with a very attractive budget. The A4 2 mm adhesive EVA sheets cost 0,25 € each. We estimate the cost of color printing at 0,10 € per A4 sheet. Since we use two EVA sheets to obtain 4 mm thickness, this yields a cost of 0,60 € for each A4 sheet of Tactode pieces. In order to make a varied set of pieces we can fit an average slightly above 30 pieces in each A4 sheet. This translates into 0,02 € for materials cost per piece, which is quite inexpensive. Indeed, the materials for a set of 2000 pieces, which is enough for ten or more groups to program simultaneously, would cost only 40 €.

Of course it is still necessary to cut the pieces and given their shape, particularly, the tabs and slots where the pieces fit together, this is not an easy task. But with adult supervision, for example in some arts and crafts class, children around ten years of age should be able to accomplish this task. This means that a school that wishes to use Tactode could do this with as little as 40 €, involving the children in the process from the beginning, spanning the work across multiple curricular units and assuming the pieces are shared across multiple classrooms.

### 4.2.3 Programming Paradigms

When it comes to programming paradigms, although several options were discussed, we focussed early on an imperative approach. The reason for this is that it introduces important and common programming concepts such as sequencing and flow control.

Within the imperative paradigm, we decided to aim at procedural programming. In its current state, Tactode is not a procedural language, simply because we have yet to materialize that functionality. In any case, we highly value the ability to split a large program into smaller procedures, that can be invoked (eventually several times) from a main set of instructions.

We also designed Tactode as an event-driven language, because this is a useful feature in robotic applications. Indeed, with the exception of Ozobot, all the robots we chose as targets have the possibility of detecting certain events and choosing their actions accordingly. This makes sense, the robot may be behaving in a predefined way, such as randomly moving around, and upon a certain event, like running into a wall, change its behavior, for example, stopping or turning around. Unfortunately, there is a single event currently implemented in Tactode, which is the event that indicates when the program should start running, so we cannot really claim our language as being event-driven, but that is one of the directions in which it is heading.

Another paradigm discussed for Tactode was functional programming. This is a subset of the declarative paradigm, which is in opposition to the imperative paradigm we have chosen for Tactode. However, a language can implement multiple paradigms, even highly contrasting ones. Python and C++ are examples of multi-paradigm languages with which one can write both procedural and functional programs [71, 72]. In fact, functions and procedures are both subroutines, which is what we want to realize in Tactode. The difference is that functions return a value, while procedures simply execute subsets of instructions. Also, in pure functional programming there is no state, such as setting or changing the value of a global variable. This absence of state, amongst

other things, compels the use of recursion, which is an interesting and important programming concept. Another good reason for introducing functions is their relevance in mathematics, which would make `Tactode` more relevant in a multidisciplinary setting.

Dataflow was also thought of as a possible programming paradigm. In robotics this paradigm is very useful and can make some programs quite simple. The idea is to connect the data from the sensors to the actuators, so that the behavior of the robot is a direct consequence of the input from its sensors. For example, given a robot with wheels and proximity sensors, if we connect the proximity sensor in the back of the robot to the wheels, the robot will move away from obstacles in its tail. Notwithstanding its usefulness, there are currently no plans to implement this paradigm, mainly because we do not consider it very compatible with the physical format that was chosen for the pieces of `Tactode`.

In sum, the answer to RQ 4 is that `Tactode` aims to be a procedural and event-driven programming language that also supports the functional paradigm.

### 4.2.4 Piece Recognition

In order to capture `Tactode` programs from their tangible form into one that can be executed, we need to use some image processing technique. At first we considered recognizing the pieces from their different colors and icons. That would be the cleanest solution, in the sense that the pieces would contain only the elements necessary for the children to identify them. But it would also be the hardest solution to implement and the most likely to have situations in which the software would be unable to recognize all the pieces and their relative positions in an image.

Considering our limited development time, the need to identify many pieces and their positions in the same image and the desire for a robust solution that could easily be operated by children without a lot of frustration, we veered into the fiducial marker territory. We looked at `ARTag` [73], `AprilTag` [74] and `Aruco` [75, 76]. One possible answer to RQ 7 is to use `Aruco`, although there does not seem to be a significant difference between it and `AprilTag`, they are both state of the art. But `Aruco` used the knowledge of `AprilTag` and built on top of it, plus it is implemented in `OpenCV` [77], a very popular open source computer vision library.

### 4.2.5 Target Platforms

Looking for an answer to RQ 6, we wanted to design `Tactode` and its application in a way that facilitated future support of multiple platforms. However, it quickly became clear that the multiple platform support was an important requirement, which we decided to embrace from the first versions of `Tactode`. Currently, the platforms supported are:

- Ozobot;
- Cozmo;
- Sphero;

- Robobo;
- Scratch;
- Python.

The decision of which targets to implement first was based, for the robots, mostly on their feature/price relation, but also on the information available, namely regarding their natively supported programming languages. As for the non-robotic platforms, `Scratch` was an evident choice due to its ubiquity as an educational programming language, `Python` because we consider it a good text based language for educational (as well as work) purposes.

### 4.3 Tactode Programming Language

As we saw in the previous section, the shape of a typical `Tactode` block is akin to the one shown in Figure 4.1. A single block will not have both the left and top slots, but only one of them. Also, the tabs might both be present or only one of them. This depends on what that block does and how we want it to fit the rest of the program puzzle.

There are two main kinds of blocks: command and argument. The command (or operation) blocks tell the target platform what to do, such as `say something`. These blocks are arranged from top to bottom in order of execution. Thus, they usually contain a top slot and a bottom tab. They also might have the right tab, so that arguments can be attached, as shown in Figure 4.2.

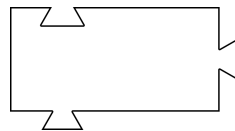


Figure 4.2: The usual shape of a `Tactode` command block.

The argument blocks are fed on the right of command blocks. For example, the `if` command needs a boolean expression as argument. These blocks have the left slot, so that they can be attached to command blocks, and the right tab, so that further argument blocks can be attached to them, as depicted in Figure 4.3.

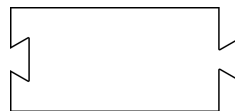


Figure 4.3: The usual shape of a `Tactode` argument block.

This top to bottom and left to right organization of the blocks in a `Tactode` program is the natural flow of text in the text based programming languages. Operations are sequenced vertically, whilst their arguments are placed to the right. Another feature that is common in those languages is indentation or nesting, which is implemented by `Tactode` as well. When multiple events are implemented in `Tactode`, each event used will start an independent column of blocks.

In terms of contents, a Tactode piece has five elements, which can be seen in Figure 4.4:

- color - shows what kind of piece it is:
  - black for numbers,
  - grey for colors,
  - white for letters,
  - green for mathematical operators,
  - red for variables,
  - orange for events,
  - yellow for flow control,
  - cyan for sensors,
  - blue for movements,
  - pink for sounds,
  - purple for visual effects;
- text - says in few words what that block does;
- icon - illustrates what the block does;
- Aruco tag - allows the piece and its position to be identified in a photograph;
- indentation - allows nesting of the commands to be executed inside the body of flow control blocks, is also used for each argument in commands that require multiple arguments.



Figure 4.4: The contents of the Tactode if piece: color, text, icon, Aruco tag and indentation.

Tables A.1– A.11 in Appendix A show all the currently implemented Tactode pieces and answers RQs 3 and 5. Table 4.1 shows a sample of Tactode pieces for each category.

Table 4.1: A sample of Tactode pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	4	number 4	yes	yes	yes	yes	yes	yes
	56	color red	yes	yes	yes	yes	yes	no
	100	letter A	yes	yes	yes	yes	yes	yes

## Tactode- A Tangible Programming Language

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	133	letter c	yes	yes	yes	yes	yes	yes
	202	multiplication op.	yes	yes	yes	yes	yes	yes
	300	variable 300	yes	yes	yes	yes	yes	yes
	399	create variable						
	398	a variable, 300-394	yes	yes	yes	yes	yes	yes
	397	variable name, string						
	400	when flag clicked event	no	yes	yes	yes	yes	no
	502	repeat a number of times	yes	yes	yes	yes	yes	yes
	503	end of repeat	yes	yes	yes	yes	yes	yes
	600	front left prox. sensor	yes	no	no	yes	no	no
	704	move forward						
	708	distance, num. exp.	yes	yes	yes	yes	no	no
	710	speed, num. exp.						
	800	say a phrase, string	yes	yes	yes	yes	yes	yes
	901	put pen down to write	no	no	no	no	yes	no

Figure 4.5 shows an example of a Tactode program meant for a robot with a distance sensor in the front. When the green flag is clicked (or the program starts), the robot will enter a potentially infinite loop, where it keeps testing if there is something in front of it. When the answer to

that question is affirmative, the robot will say ‘hi’ and exit the infinite loop, thus terminating the program. Otherwise, that is when there is no obstacle in front of the robot, it will move forward a distance of 16 with a speed of 16 (units depending on the robot in question).

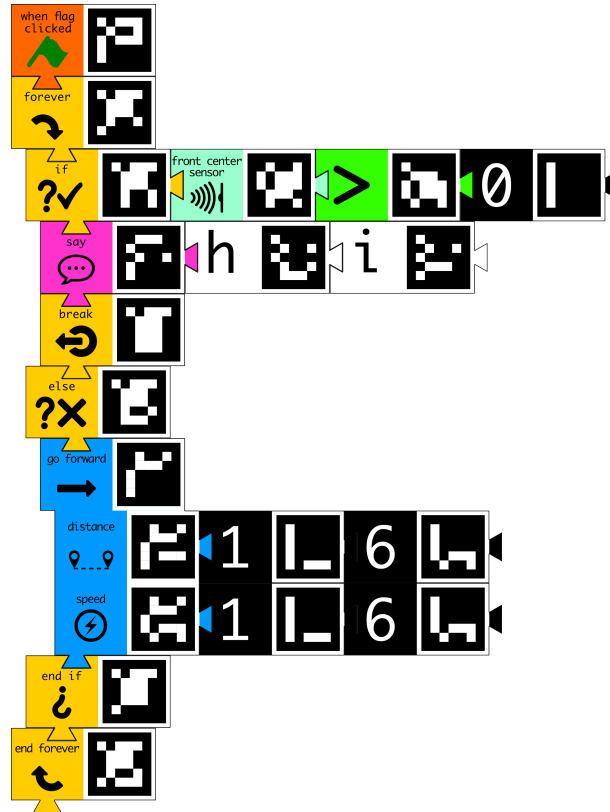


Figure 4.5: A Tactode program with flow control blocks.

Most command blocks take numeric expressions as arguments. There are however a few exceptions. The `if` and `while` blocks take boolean expressions as arguments. The `think` and `question` blocks take either numeric expressions or strings as their arguments. The `say` block argument type varies according to the platform: in Ozobot it is either a numeric expressions or a color block, in the others it is either numeric expressions or strings.

The `create variable` block takes a `variable` block (the blocks whose tags vary between 300 and 394) as first argument and a string formed by `letter` blocks as second argument. One may also use `number` blocks in the names of variables, but the first block must always be a letter one. The `set variable` block takes a `variable` block as a first argument and a numeric expression as the second argument. This means that Tactode variables always have the same type, they are integers. The reason they are integers is that the numbers one can build are integers and the mathematical operators applied to integer numbers produce integer numbers, in particular the division corresponds to integer division of integers. Future versions of Tactode might allow variables of different types, such as strings and booleans, we might also allow floating point numbers.

The event blocks are the only command blocks without the top slot. That is because they are meant to be placed only on top of a sequence (or column) of command blocks.

Careful observation of the flow control Tactode blocks (the yellow ones) will show that their top tab or bottom slot might be indented. This is done so that the blocks inside a control block body are indented (in relation to the ones outside) and can be more clearly identified. Also, it helps to teach children about an important practice in most text based programming languages. Another feature of most flow control blocks is that they have a corresponding end block. This blocks has two functions, closing the body of their beginning control block, to separate it from the rest of the program, and removing the indent. All of this can be seen in Figure 4.5.

We believe the use of most Tactode blocks is self explanatory, with the possible exception of the variable blocks. The variables (with Aruco tags from 300 to 394) do not have text or icons. The idea is that on use, a sticker (post-it) for example should be placed on the empty left side of the block, that says what it is supposed to represent. The name of the variable is defined in the create variable block. This means that when Tactode is compiled to the language of the desired target, the variable will have that name. Alternatively, variables can be used without being created. In that case, the variable name in the destination language will be given according to its tag, such as var306 for the variable with the tag 306. What one should always do before using a variable is setting its initial value using the set variable block. Otherwise, the behavior might be unexpected. In Python an error will be thrown on execution due to the unknown value. In the other target languages, variables are given the default value of 0 on creation.

Another distinguishing feature of the create and set variable blocks, as well as some of the movement blocks is their shape, which is the same shape of three (or two) command blocks grouped together, with an indentation from the first to the second. This is done so that one can provide multiple arguments to a single block, like distance and speed in case of the move forward block. Naturally, in the specific case of the movement blocks, one might wonder about the units. They depend on the target platform, but are mostly the same, the distance is measured in mm in Ozobot, Cozmo, Sphero and Robobo, and in steps in Scratch. Speed is measured in mm/s for move forward and backward blocks and in degrees/s for turn blocks. Angles are measured in degrees.

Currently, the color blocks are used to compare with the line color block in Ozobot. In the future they will also be used to set the color of LEDs (Ozobot, Cozmo, Sphero and Robobo), to detect the color of objects (Robobo) and to detect colors on the screen (Scratch).

About the say block, it is placed under the sounds color category, because that is what happens in all robotic platforms, they say using speakers the argument of this block. But for Scratch and Python the say block does not produce any sound. Indeed, in Scratch a balloon pops up from the cat with the argument and in Python the argument is printed out into the console where the program is being executed. The think block is placed together with the say block, due to their relation, but it is only defined in Scratch, where it produces no sound.

An example of a material Tactode block, in this case an if block, with two sheets of 4 mm



yellow EVA, printed paper and self-adhesive clear plastic on top, is shown in Figure 4.6. The rectangular body of the piece is 50 mm wide and 25 mm long, and each tab adds 4.4 mm to these dimensions. The indentation of the bottom tab relatively to the top slot measures 5 mm.

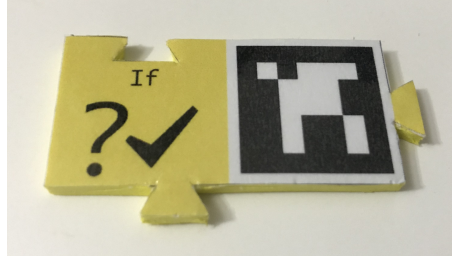


Figure 4.6: A Tactode if piece, with yellow EVA, printed paper and clear plastic on top.

## 4.4 Challenges

For each target platform we decided that a set of challenges would be designed for experiments with that platform. Those challenges are detailed in this section. In spite of the fact that many of the features of each platform are yet to be implemented, there are plenty of possibilities for programming each target. Our challenges were chosen due to their educational value, from the use of different programming constructs to the need of mathematical concepts, as well as their exploration of different types of features in each platform. There are four main challenges: regular polygon construction, obstacle reaction, finite simple maze solution and prime number generation. Depending on the platform, we have some variations of these challenges.

### 4.4.1 Regular Polygon

The regular polygon construction challenge is available for all platforms except Python. However, depending on the target, there is a variation, with the Scratch version being more complex. The idea is to make the target draw a regular polygon as it moves. To do this, the children must know a few things about regular polygons: their sides all have the same length, their internal (and external) angles all have the same amplitude, and for a polygon with  $n$  sides the amplitude of each external angle is  $360^\circ/n$ . Figure 4.7 shows the Tactode program that should be built to move in the shape of regular polygon in Ozobot, Cozmo, Sphero and Robobo. In this case we are building an hexagon. Since Ozobot does not implement events, the flag block on top on the program should be removed for this target.

For Scratch, see Figure 4.8. one can also use the question block to ask the user how many sides the polygon should have, the answer to this question will be given on runtime and placed in the answer block, which is then used like a variable block. The other difference of the Scratch version of this challenge is that it actually draws the polygon as the Scratch cat moves in the screen, using the erase, pen down and pen up blocks. Note that the question and answer blocks could also be used in Robobo.

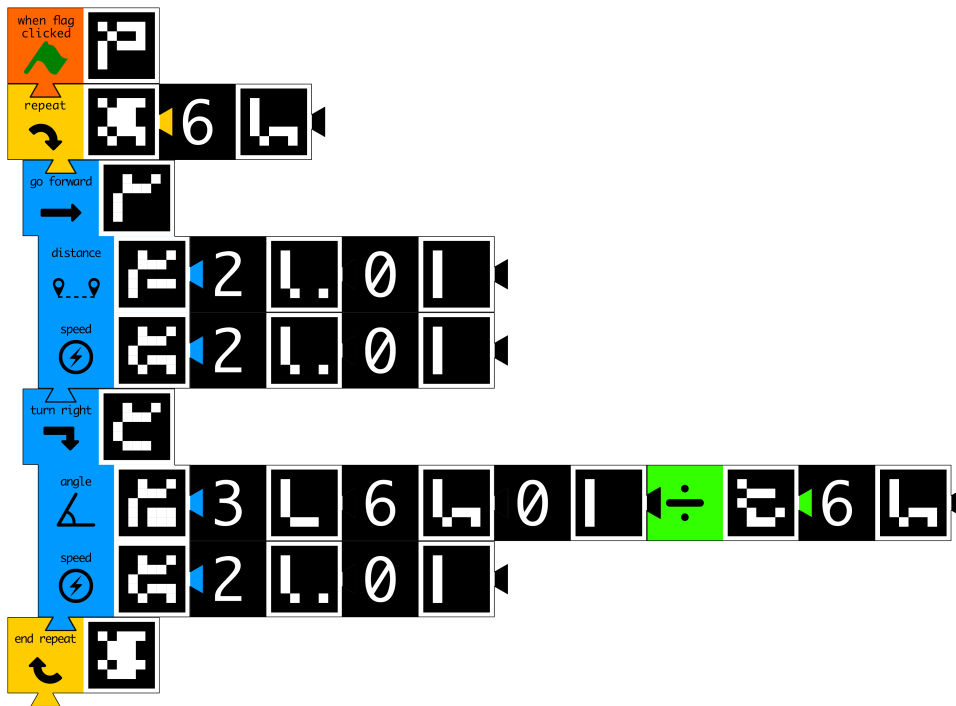


Figure 4.7: The regular polygon generation program in Tactode for Ozobot, Cozmo, Sphero and Robobo.

#### 4.4.2 Obstacle Reaction

The obstacle reaction programs are for Ozobot, but they can also be executed in Robobo, if we add the `flag` block on top of each. In these examples we are running away from obstacles on the back of the robot (Figure 4.9) and following obstacles in front of the robot (Figure 4.10). But there are other examples of this kind, such as running away from obstacles on the front, moving forward and turning when an obstacle is found in front, and following walls. An interesting aspect of these programs is that the code inside the `forever` loop is actually an example of the dataflow programming paradigm.

#### 4.4.3 Simple Maze

The Tactode program for solving a simple maze works only in Ozobot, because that is the only robot on our list with line detection capabilities. A maze is called simple when it is connected and contains no loops, such as the example shown in Figure 4.11.

One can solve any finite simple maze by using the following strategy:

1. Pick a preferred direction between left and right (in this case we chose left);
2. Follow the line until an intersection is reached;
3. If the intersection has a line on the left, pick the left direction;

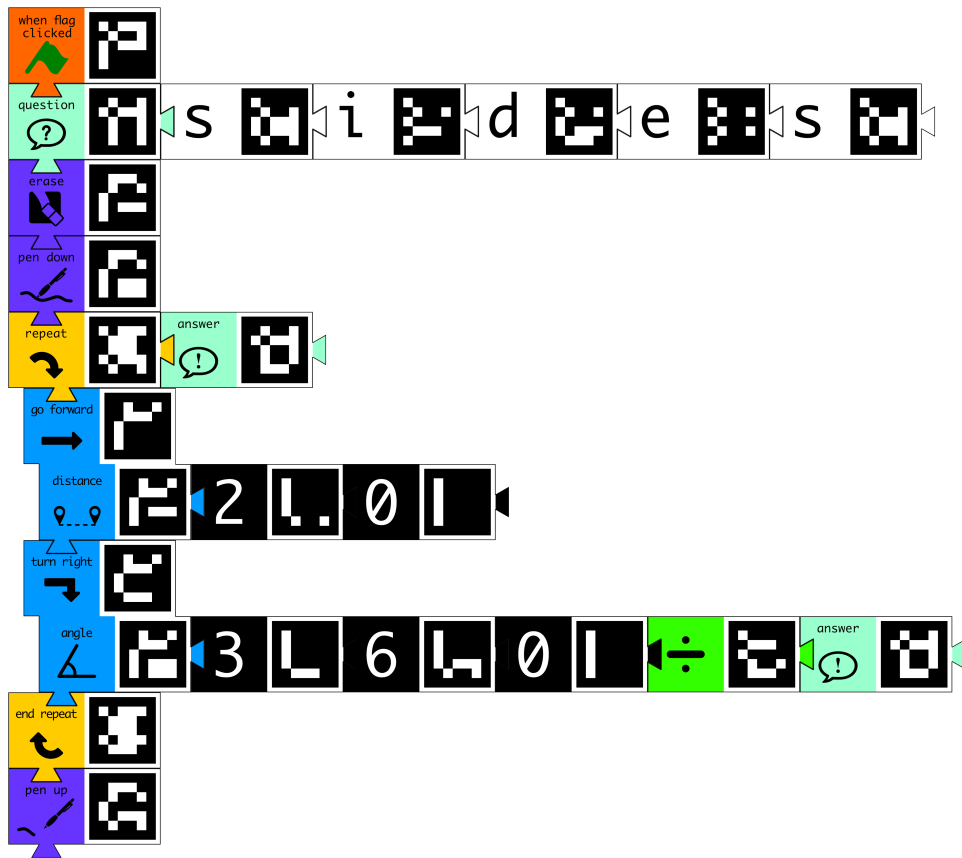


Figure 4.8: The regular polygon generation program in Tactode for Scratch.

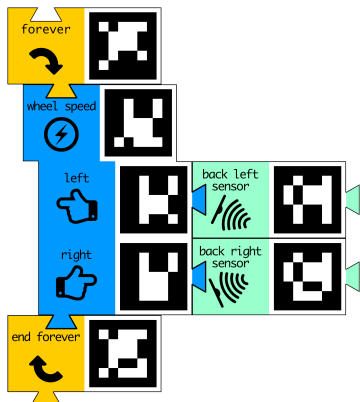


Figure 4.9: An obstacle reaction program in Tactode: run away from objects in the back.

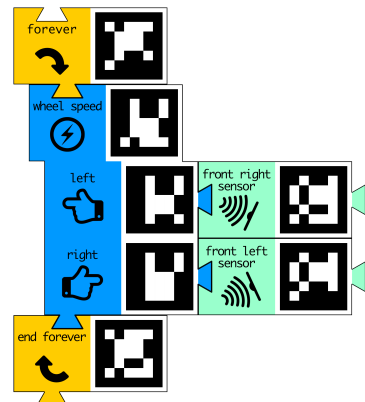


Figure 4.10: An obstacle reaction program in Tactode: follow objects in front.

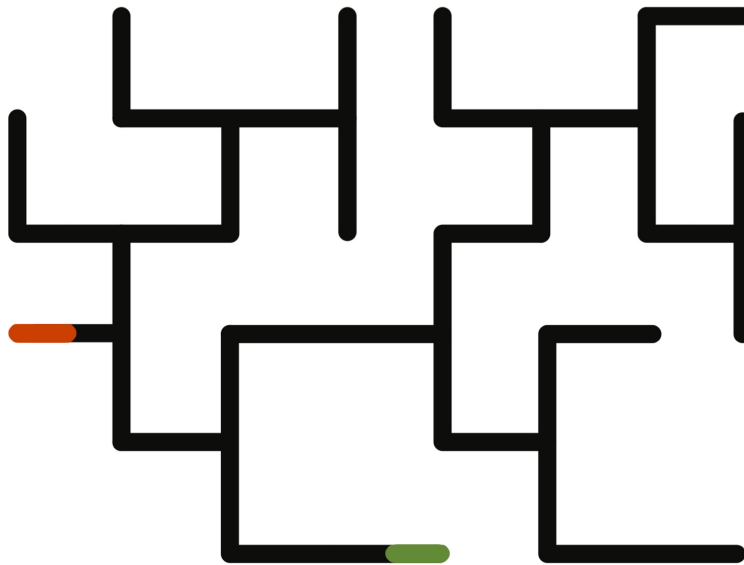


Figure 4.11: Example of a finite simple maze with the entry in red and the exit in green.

4. Otherwise, if the intersection has a line forward, continue straight;
5. Otherwise, if the intersection has a line on the right, pick the right direction;
6. Otherwise one has reached a dead end and should turn back;
7. Repeat the steps from 2 to 6 until the exit of the maze is reached.

In this case, we are identifying the exit by the color of the line (green). The `Tactode` program resulting from following this strategy is shown in Figure 4.12.

If we use the program in Figure 4.12 to solve the maze in Figure 4.11, the robot will test every possible path until it finds a solution. Had we chosen right as the preferred direction, the robot would turn right in the intersection immediately after the start point, turn left in the corner after that (because it cannot turn right or go forward), turn right in the following intersection, turn left at the next corner and reach the exit. This is the most efficient path from the red entry to the green exit for this particular maze. However, there is no point in dwelling between choosing left or right, unless one knows the maze. This is part of what the children are supposed to learn with this challenge, the program works, but it does not necessarily lead to the best path.

#### 4.4.4 Prime Number Generator

Because `Python` is a very different target from the others, it required a specific challenge. We opted for a prime number generator, because of the mathematical connection.

It is a well known number theory fact that any non-prime number  $n$  must have a prime factor that is at most  $\sqrt{n}$ . Indeed, suppose that  $n$  is a non-prime number and  $a$  is its smallest prime factor, with  $n = ab$ . Since  $n$  is not prime, we have  $b \neq 1$ . Also, given that  $a$  is the smallest prime factor,

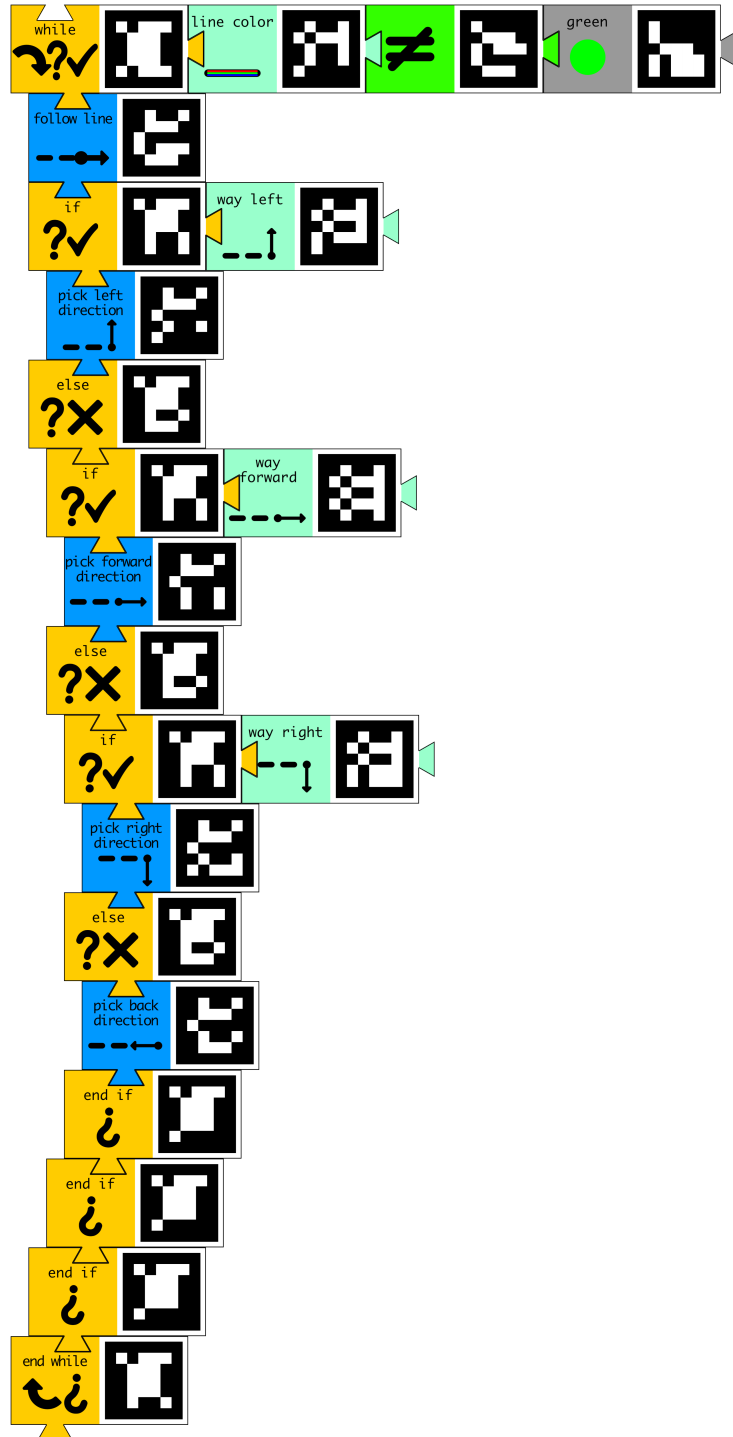


Figure 4.12: The finite simple maze solution program in Tactode.

we have  $b \geq a$ , because either  $b$  is prime or it has a prime factor that is greater or equal to  $a$ . If  $a > \sqrt{n}$ , then  $b > \sqrt{n}$ , therefore  $n = ab > \sqrt{n}\sqrt{n} = n$ , which is impossible.

Given that the smallest prime factor of a non-prime number is at most its square root, it would be more efficient to use the square root of the number to determine whether it is prime or not. We could have easily implemented this block in Tactode and will do so in the future. But there are other alternatives and when learning prime number identification, children usually start with simpler versions, such as the one shown in Figure 4.13.

It starts by printing the prime number 2. Two variables are created,  $p$  for the candidates to prime numbers, and  $d$  for the potential divisors of the prime candidates. Note that the variable blocks were tagged on the left with their names,  $p$  for variable 300 and  $d$  for variable 301. We set  $p = 3$ . While  $p$  is smaller than a previously defined maximum value (in this case 100), we set  $d = 3$  and then test if  $p$  is divisible by  $d$ , incrementing  $d$  2 by 2. The idea is to test only odd primes (2 is the only even prime) and two test only odd divisors, because odd numbers have no even divisors. When the inner loop finishes, if the  $d = p$ , that means that all possible divisors were tested and none worked, hence  $p$  is prime and should be printed to the screen. Before closing the outer loop we increment  $p$  by 2.

The prime number generator (Figure 4.13) is the most complicated challenge on our list, just like Python is the most advanced target. It is aimed at older children. But younger children can do easier examples in Python such as printing odd numbers.

Note that although this challenge was designed with Python in mind, it runs in every platform, so the children can make the Scratch cat or any of the robots say a sequence of prime numbers. They just need to add the flag block on top of the program for the platforms that implement it.

## 4.5 Evaluation

We had a few important objectives when we set out to design an educational programming language. We wanted it to be tangible, block based, with both visual and textual elements, low priced, multi-platform, easy to begin using, so that even pre-schoolers could start learning and capable of growing with the children as they advance in their programming knowledge.

With the exception of the last two main goals, it is quite simple to observe that we have accomplished what we set out to do. Tactode fills all the physical requirements it was supposed to have.

Using different materials and manufacturing processes might make it easier to obtain large quantities of Tactode pieces. But we are highly convinced that it is impossible to obtain cheaper pieces and maintain their usability and durability. Thus, if manual cutting is possible, we believe this is the best overall possibility. Note that other materials considered, including the polymer clay, are all either more expensive or require more expensive manufacturing processes.

As for the use goals, their complete evaluation requires experiments, which we did perform and present in Chapter 6. But even without looking at the results of those experiments, simply by observing the Tactode blocks and the challenges in the previous section, we can see that

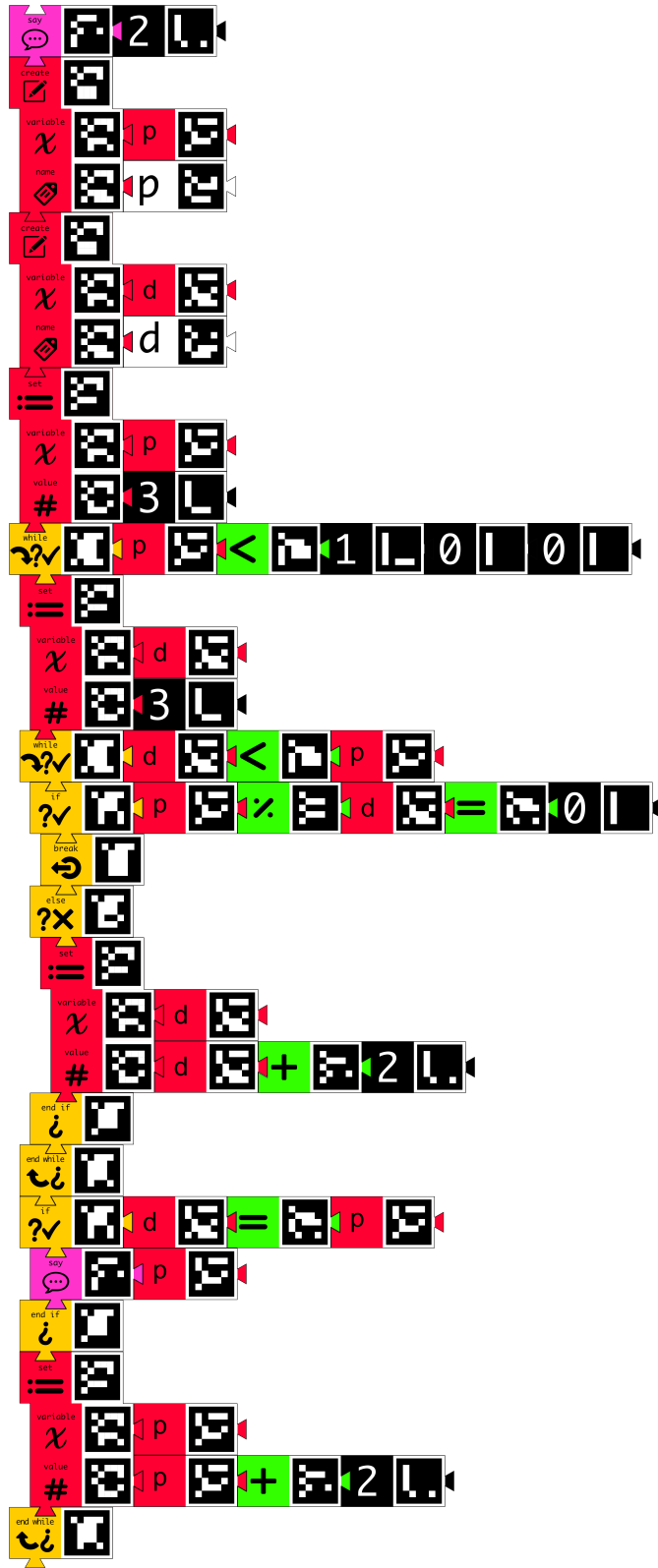


Figure 4.13: The prime number generation program in Tactode.

there are simple examples that could be used for the first steps into programming, as well as more complex programs, such as the prime number generator, that should be attempted by those with some previous experience. Note that there are even simpler examples, without any control flow blocks, like giving the robot a sequence of movement instructions.

In any case, we also wanted to implement different programming paradigms, to have more events, more different types of blocks and to add subroutines. This future work will add to the objective of having a language capable of accompanying children through a greater level in their programming education. Thus, we believe that unlike the other goals, which were fully achieved, this last one can be further improved on.

### 4.6 Conclusion

To create a programming language one must first define its syntax and semantics and then build either an interpreter or a compiler for it. In this section we described our tangible programming language `Tactode` by presenting the first of these two steps. The compiler we built is presented in the following chapter.

One of the basic steps of defining the syntax is to list the available keywords or commands. In `Tactode` these are the blocks, so there are no spelling mistakes. But one also must describe how the blocks can be fitted together to create valid programs. In `Tactode` this is done by sequencing the command blocks from top to bottom, using their top slots and bottom tabs, and by attaching sequences of left to right arguments to these command blocks, in such a way that the appropriate type of argument is supplied to each command that requires one or more arguments. As we will see, syntax errors are reported by the `Tactode` application when the program is imported, which is different from other block based languages where syntax errors cannot be made, due to the distinct shapes of the blocks.

As for the semantics, we described the meaning of each block and the meaning of any program is derived from the meaning of its blocks. Like in any other programming language, semantic errors, that is, the types of errors where the program works but does not do what it was supposed to do, must be detected by the user when the program is executed.



## Chapter 5

# Tactode Application

This chapter exposes the development of our application to transpile `Tactode` into the languages of the destination platforms. We begin with a description of our requirements for this application. After this we present the development choices we made and the general architecture of the application, paying special attention to the transpiler and how it was built. Then we show the final result and explore its capabilities, giving examples of the most common usages. In the subsequent section, we give evidence of the tests we performed on our application to ensure its compliance with the requirements. We also present an evaluation of the application, exposing its features and limitations.

### 5.1 Introduction

Designing a tangible programming language for educational purposes was only the first objective of this project. The second objective was to create an application that allowed that tangible language to be executed in a variety of target platforms. In order to keep the cost low, our application should run in as many platforms as possible, from smartphones to computers. Also, the code is to be captured by photograph, translated into the destination target and exported.

In order to obtain this translation application, we had to design a compiler for `Tactode`. Typical compilers take a high level (closer to regular human languages) code and translate it into low level (closer to machine language) code. In our case, we needed to translate `Tactode` into multiple high level target languages. This kind of compiler is known as a source-to-source compiler or a transpiler. Thus, in total we had to create six transpilers, one for each target platform. As we will see, these transpilers are not completely different from one another, as the first part of the compiler, which is called parser and transforms the source code into an intermediary language, is independent of the destination language.

Because we wanted our application to run in as many different platforms as possible, including very different operating systems, we had two main options: code our application for each operating system separately, or use a framework capable of generating applications for multiple systems from a single source code. Due to time constraints, the first option was not feasible at all, so we decided to use the `Ionic` framework [78]. This is a web based multi-platform development framework, that can create `iOS`, `Android`, `macOS` and `Windows` applications, which are precisely the operating systems where the `Tactode` application runs. It is also possible to use `Ionic` together with `Electron` to generate `Linux` applications in addition to `macOS` and `Windows`, which we will try in the future. Finally, our `Tactode Ionic` application also runs in the browser.

## 5.2 Requirements

The requirements for the `Tactode` application are a direct result of the goals we have for this project. In any case, we shall list them as thoroughly as possible, as they have guided its development.

### 5.2.1 Functional Requirements

The first and main functional requirement for the `Tactode` application is to acquire and translate programs. Each new program is to be named and captured by photograph. The application translates the program into the predefined target platform and presents it to the user. If the program contains errors, they should be listed. Otherwise, the user should be given the possibility to export the generated code file, so that it can be imported in the destination platform. This is summed in the Use Case bellow, **UC 1**

**UC 1.** *The user has composed a `Tactode` program that they wish to run, so they will name it, take a picture of it and wait for the program to be translated and either a list of errors or an export option is presented, with the intent of sharing the program with the target platform.*

The process described in the previous paragraph represents the most important feature of our application, however a few more things are still necessary. First, there must be some way to define the target platform at any given moment. Also, in order to facilitate the process of acquiring the images of the code, it should be possible to choose between taking a new photograph or using the image file of a previously taken one. Finally, although of less importance, since the application should aim at text independence, it should be possible to define the language of the interface. These requirements are collected in **UC 2**.

**UC 2.** *The user wants to configure the application, so they select the platform, the image source and the desired language, with the objective of having the desired target, the adequate source and the language they understand best.*

The third and last functional requirement for the `Tactode` application is that it should have a database of the previously captured programs that contain no errors, allowing them to be deleted and exported at will. This requirement is represented by the **UC 3**.

**UC 3.** *The user wants to see previously imported Tactode programs, so they examine a list of previous programs, with the intent of re-sharing or deleting one or more of these.*

With this we conclude the functional requirements for our application. The diagram in Figure 5.1 illustrates these functional requirements that we have described.

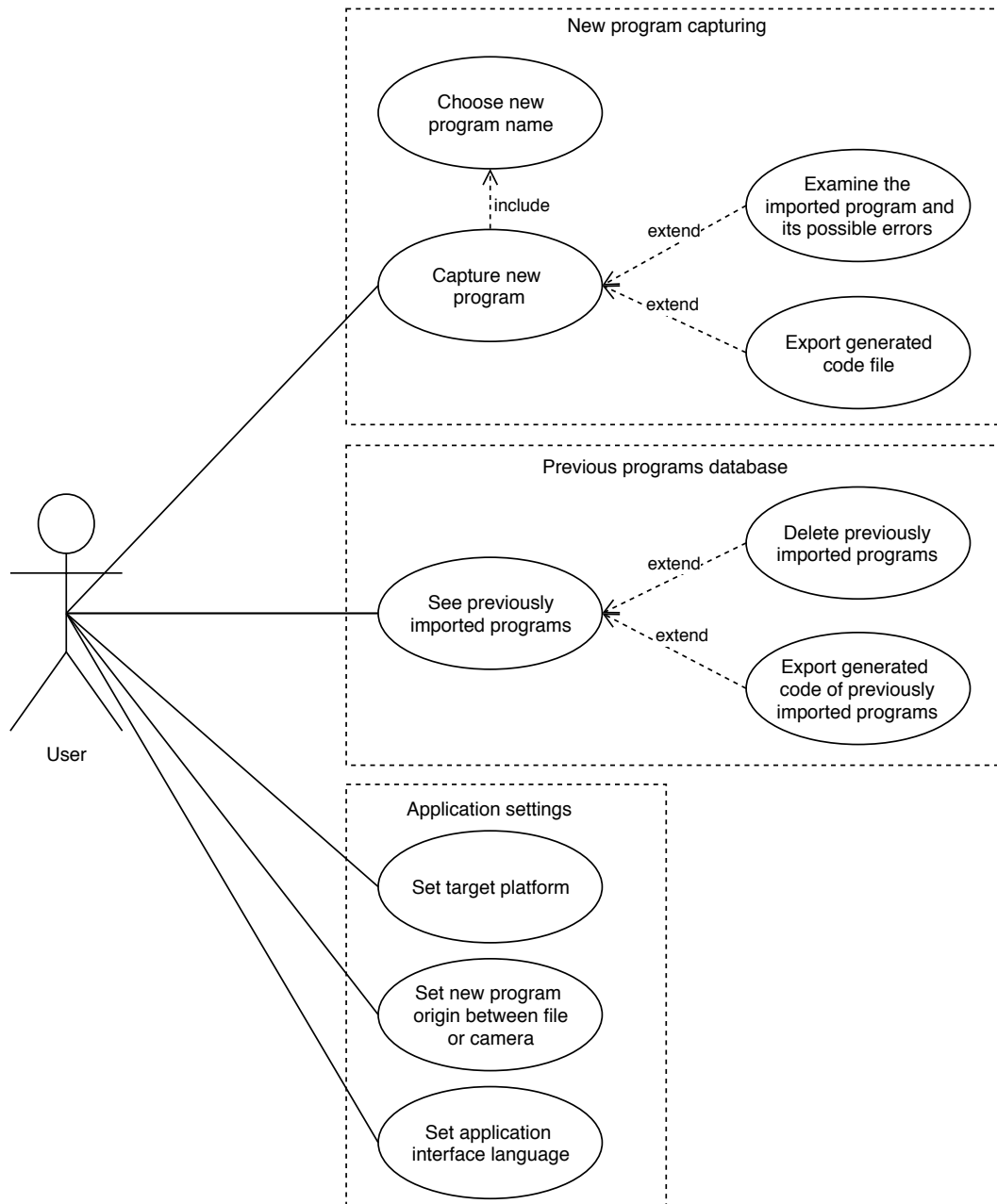


Figure 5.1: The use cases diagram for the Tactode application.

### 5.2.2 Non-Functional Requirements

As for the the non-functional requirements, given that our application is meant to be used independently by children, from ages as young as four or five years old, its usability requirements are

very important. It must be intuitive, as language independent as possible, efficient (finish its tasks quickly) and effective (accomplish what it is intended of it).

Performance is also a concern, as we have mentioned when requiring efficiency and effectiveness, because we wish the user experience to be as free from frustration as possible. However, given that there are image processing tasks involved and further tasks from the compiler, it is expected that at least a few seconds of wait will be necessary when capturing large programs.

Reliability is important, particularly given the age of the users and that any failure will add to their confusion, as they will not be certain that the mistake is not theirs but from an error in what is expected as normal behavior from the application. Much of the same can be said for availability, but this should not be a problem, as the application should be available whenever it is not already processing a new program.

Security requirements are not a concern, since the application does not deal with any sensitive data, does not publish anything to the internet, only requests access to the camera and to image files, but does not store any images that do not contain correct `Tactode` programs. Finally, at least in terms of non-functional requirements, scalability should not be a real issue, except perhaps as the number of programs previously captured and stored increases, which we should pay attention to.

### 5.3 Architecture

The main architecture pattern of the `Tactode` application is Model-View-Controller (MVC) [79]. This is a natural consequence of using `Ionic`, since that is their default pattern. `Ionic` applications are web based, having pages through which the user navigates. The view part is in the `HTML` and `CSS` files of each page. We have isolated the model in a set of `TypeScript` providers, whose simplified class diagram we present bellow. As for the controller, it is composed of the `TypeScript` files of each page, since that is where each interaction with the view is processed. There was a clear effort in the separation of these components, particularly the controller from the model, guaranteeing that only the procedures dedicated to process user interaction and correct display of the information are placed in the `TypeScript` files of each page.

As we will see below, our application is composed of three pages, one for each set of use cases: import new programs, consult previously imported programs and control the settings (target platform, image capture method and language). The model required for the first of these pages is the most complex and, for that reason, we expose the details of its implementation in the remaining of this section.

In order to guide the implementation of the acquisition of new programs, we defined the activity diagram depicted in Figure 5.2. This describes in a good detail what happens after the user clicks the import program button. As another use case specifies, there are two possibilities for importing new programs, take a new photograph or use the file of a previously taken one.

Independently of the source of the image, the next step is to detect the `Aruco` tags present as well as their positions. After this it becomes necessary to straighten the positions of the markers,

## Tactode Application

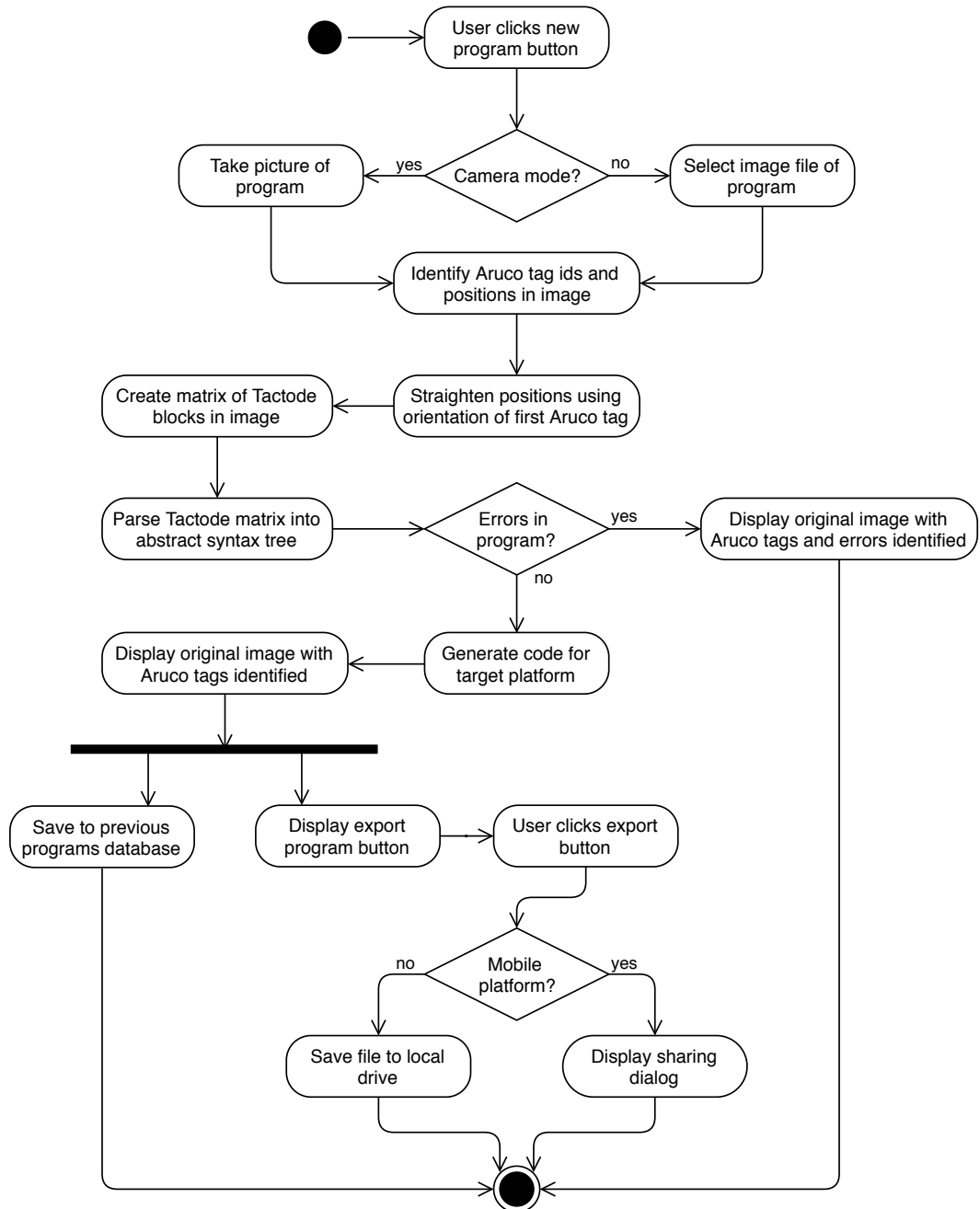


Figure 5.2: The activity diagram for importing a new program in the Tactode application.

in order to correct possible rotations when taking the photograph and make sure that the program is oriented from top to bottom and left to right.

From the set of correctly placed markers matrix is formed, so that each row of markers is processed in order. The parser will use this matrix to build an Abstract Syntax Tree (AST) [80] of Tactode blocks, which is our intermediary language. At this point, either there are syntax errors that the parser detected, and they will be reported to the user finishing the process, or there are no errors and the code generator for the previously specified target can convert the AST into a file that the target platform understands and is capable of executing.

Once the code generator finishes, the original image with the Aruco tags highlighted is shown to the user. The application also saves this new program to the database and displays an export button. If the user clicks this button, the file is either saved directly to the local drive, if the platform is the browser or a computer, or the typical sharing dialog of the smartphone is displayed, so that the user can choose where to save or send the generated coded file.

The class diagram in Figure 5.3 is also concerned with the classes involved in capturing new programs. The Program class is at the center of the whole structure. It contains the Marker matrix, once they are identified and sorted by the ImageProcessor, the AST of Block objects, once the Parser is done creating it, a list of CompilerError objects (empty when there are no errors) and once the CodeGenerator finishes, it will also contain the text of all the target code.

The Block class and its children represent each Tactode block, with the ID clearly determining the correct subclass. Indeed, this is what the BlockFactory class does, it creates the correct subclass of Block according to the supplied ID, which is just the number of the Aruco tag in that block.

This factory pattern is repeated for the CodeGenerator and the Sharer, which are decided based on the target platform, since the code that is generated is different and also the way to save it. The Parser does not need this, since it is the same independently of the target, it only uses that information to make sure that all blocks used are defined in the target platform.

In sum, the Program class is used to go through the steps identified in the activity diagram, using the ImageProcessor first, then the Parser, if there are no errors the CodeGenerator and finally the Sharer. The remaining classes are the composing elements of the Program: Marker, Block and CompilerError.

Note that the ImageProcessor class is our response to Research Question 7, since its objective is to transform an image with a Tactode program into a matrix where each block is identified, together with their relative positions. Research Question 8 is answered by the Parser and CodeGenerator classes, together they form the Tactode transpiler, which we present in detail in the next section. Finally, the Sharer class answers Research Question 9, since it is responsible for sharing the Tactode programs, either directly with the targets that support it (such as Cozmo), or by exporting the code files that can then be imported and executed.



## 5.4 Transpiler

At the core of the `Tactode` application is its transpiler, or transpilers, since we have one for each target platform. A transpiler is a type of compiler that translates programs written in one programming language into another programming language, which is why it is also called a source-to-source compiler. The difference from the more typical compilers is that the output is in a language of a similar level to the input, instead of being in a lower level.

The robotic platforms we selected all come with their own educational block based languages in which the robot can be programmed. That is why we have chosen to transpile `Tactode` into these languages. This way, children can use `Tactode` first and latter the language of the robot or even use them together depending on what is more appropriate for their objective.

Every compiler has two main components: a parser, that takes the source code and builds an AST, checking for syntax errors as it processes the input; and a code generator, which, as the name implies, uses the AST to generate the output code in the destination language.

### 5.4.1 Parser

As we saw in the previous section, the `Tactode` transpilers all share the same parser. This parser does take the destination platform as a parameter, but only to report errors in which an unknown block for that platform was used, such as using a proximity sensor in `Scratch`. In all other respects, the parser behaves the same way, independently of the target. This is the advantage of building an AST, one is simultaneously checking that the source is syntactically correct and generating an intermediate language that will then facilitate code generation, independently of the type of code that will have to be generated.

Before the `Tactode Parser` is called, the application uses the `ImageProcessor` to generate a matrix of `Marker` objects. Each row of this matrix contains a row of `Tactode` pieces identified by their `Aruco` tags. This row is always a command block followed by its arguments, unless the program has syntax errors. The `Parser` will go through this `Marker` matrix in order from top to bottom and left to right, creating the appropriate `Block` object for each marker and then parsing it, generating the AST as it goes.

Our AST is implemented in a very simple way, that can be seen by taking a closer look at the class `Block`, where we see that each object of this class has an array of other `Block` objects, known as `children`, and also a `parent` object of the class `Block`. These are precisely what their names say, instead of having a tree structure, we have each object know who is its parent and who are its children.

The `Parser` makes use of a design pattern known as visitor. In this pattern, the parser visits each `Block` object to parse it. Instead of implementing the parsing in the `Block` class, we do it in the `Parser` class. The `Block` class merely implements an `acceptParser` method that simply calls the appropriate method in the class `Parser`.

Depending on each subclass of `Block`, or in other words, on the `Tactode` block type, the parser will check for errors:



## Tactode Application

- The block is placed correctly, command blocks in vertical sequence, argument blocks to the right of the respective command blocks;
- The block is placed in the correct order, `else` after `if`, `end` after `begin`, when `flag` clicked at the top;
- An argument is present if and only if it is necessary for that block;
- The eventual argument has the correct type;
- Numerical and boolean expressions are valid;
- Each variable is created only once and with a distinct name from previously created variables.

Independently of finding errors, the parser will continue to move through the `Marker` matrix, so that it can report as many errors as possible or, in case there are no errors, finish generating the AST so that the `CodeGenerator` can do its job.

The intermediary language generated by the `Tactode Parser` is not very different from the original tangible language. Indeed, every original piece has a corresponding `Block`. But there are a few differences that facilitate the code generation, which is the function of the AST. There are some extra elements in the AST that do not have a corresponding piece in `Tactode`. These `Block` objects are:

- `BodyBlock` - child of `RepeatBlock`, `ForeverBlock`, `IfBlock`, `ElseBlock` or `WhileBlock`; its children are the commands to be executed inside that control flow, that is, between the `begin` and the corresponding `end Block`.
- `ConditionBlock` - child of `RepeatBlock`, `IfBlock` or `WhileBlock`; contains the condition to be verified by these control flow blocks;
- `RootBlock` - special `Block` with no parent that serves as root of the AST, its children are the command blocks that are not inside of any control flow block, that is, the ones that have no indentation in the tangible language.

Another difference is in the mathematical expressions. These are parsed using the well known shunting-yard algorithm originally created by Dijkstra [81], which, in our case, takes the infix mathematical expressions in `Tactode` and places them in the AST in prefix notation. For example,  $1 + 2 = 3$  becomes the tree represented in Figure 5.4.

The shunting-yard algorithm is also where the parser verifies the validity of the mathematical expressions, including whether open and close parenthesis match.

The final difference of the intermediary language is in its numbers and strings. `Tactode` has a distinct piece for each digit and each letter and uses those pieces to form numbers and strings, respectively. The AST has a single `NumberBlock` object for each sequence of numbers in the tangible language, independently of the number of digits, where the sequence of concatenated

## Tactode Application

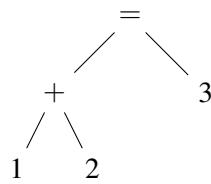


Figure 5.4: The AST for the boolean expression  $1 + 2 = 3$ .

digits is stored. The same thing for strings, instead of storing individual letters, the AST has a single `LetterBlock` object where the string of concatenated letters is stored.

### 5.4.2 Code Generators

Once the `Parser` finishes creating the `Block` AST, if there are no syntax errors, it is time for the `CodeGenerator` to start working. Similarly to the `Parser`, the `CodeGenerator` uses the visitor design pattern. Each `Block` implements an `acceptCodeGenerator` method that does two things: calls the appropriate code generation method in the `CodeGenerator` class for that `Block` object; and, for each `Block` in its children, calls its `acceptCodeGenerator` method.

Thus, in order to process the entire AST with the `CodeGenerator`, one needs only to call the `acceptCodeGenerator` method of the `AST` `RootBlock`.

The `PythonCodeGenerator` is very different from the others. It generates a plain text file with the appropriate extension, but among other things must reconvert the prefix notation of the numeric expressions into infix.

The `RoboboCodeGenerator` creates a `JSON` file, with each `Block` being represented by an attribute value pair. It is the simplest of all the code generators, as the language used is very short and to the point. The only difficulty is that it also includes a few images and sounds that must be created and together with the `JSON` file compressed to make the final file for `Robobo`.

The generators of the other targets are all subclasses of `BlocklyTypeCodeGenerator`. `Blockly` files are stored in the `XML` format and that is what the `OzobotCodeGenerator` uses. All these subclasses have in common the fact that they require a distinct alphanumeric tag to be generated to identify each `Block`.

Once `Ozobot` is taken care off, the remaining `CodeGenerator` subclasses are all subclasses of the `ScratchTypeCodeGenerator`. Note that `Robobo` also uses `Scratch`, but it is based in the old version of `Scratch`, while the others are all based in `Scratch 3.0`. Since this version of `Scratch` was influenced by `Blockly`, it also uses the unique alphanumeric tag to identify each `Block`, but it keeps the `JSON` file format, instead of using `XML`. Also, like `Robobo`, `Scratch` requires a couple of images and sounds to be compressed together with the `JSON` code file to form the final file to be exported.

The `CozmoCodeGenerator` and `SpheroCodeGenerator` are identical to each other and to the `ScratchCodeGenerator`, except that no compression or extra files are necessary, one simply saves the `JSON` code file with a specific extension for each platform.

## 5.5 Application

The `Tactode` application is composed by three pages, one for each group of use cases. The pages are organized by tabs shown at the bottom of the screen with the names `Tactode`, `Programs` and `Settings`.

The home page is represented in Figures 5.5 and 5.6. The first of these images shows the screen before the new program button (with the camera icon) is clicked. The second shows an already imported and processed program, ready to be exported by clicking the sharing button. If the user does not fill the `Program Name` box, then the default name `Tactode` will be given to the program. Otherwise, if the name chosen is too big or uses special characters, they are removed, so that a short and simple name remains.

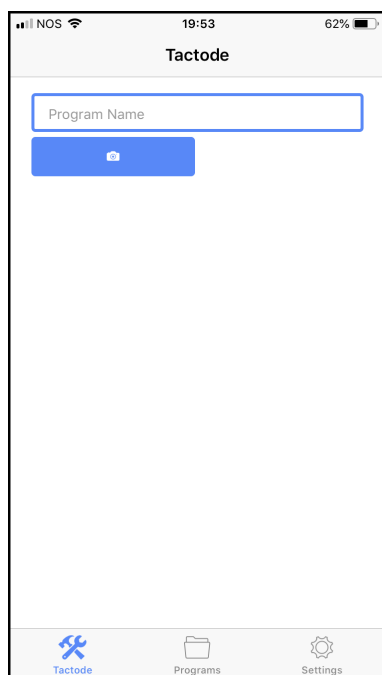


Figure 5.5: The home page of the `Tactode` application, ready to import a new program.

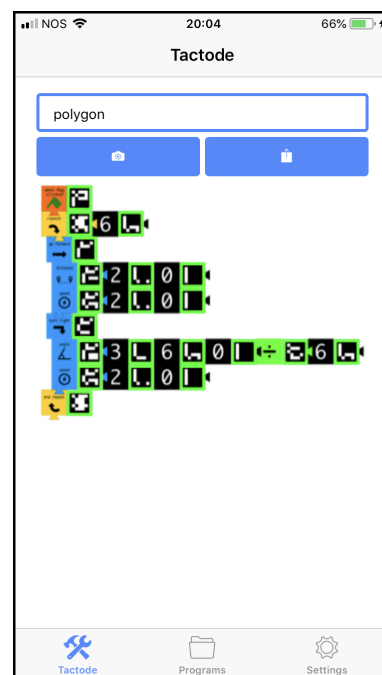


Figure 5.6: The home page of the `Tactode` application, after importing a program.

The `Programs` page, depicted in Figure 5.7, has an image of each previously imported program that had no syntax errors, together with the name of the program and the target platform on the top, and share and delete buttons on the bottom, so that the user can export again a previous program or remove it from the list, respectively.

Finally, the `Settings` page, shown in Figure 5.8, consists of three drop down choice buttons, the first for the target `Platform` (`Ozobot`, `Cozmo`, `Sphero`, `Robobo`, `Scratch` or `Python`), the second for the `Image Source` (`Camera` or `File`) and the last one for the `Language` (`English` or `Portuguese`, at the moment).

## Tactode Application

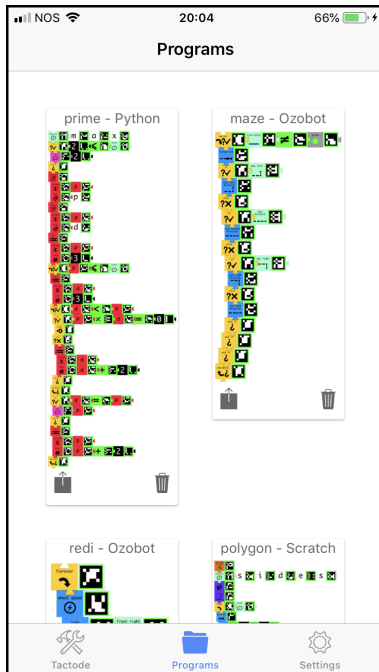


Figure 5.7: The previous programs page of the Tactode application.

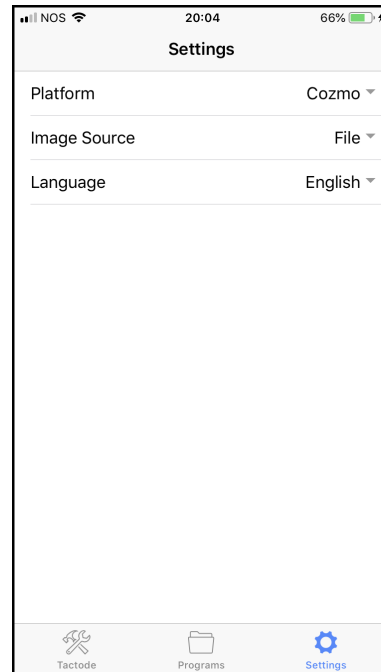


Figure 5.8: The settings page of the Tactode application.

## 5.6 Testing

We designed a series of tests for our application, to make sure that it is working as intended. These tests can be grouped into two categories: platform tests and visibility tests. The platform tests are meant to test if the translation into a given target platform is correct. The visibility tests are meant to check the limits of our code capturing process, from a given photograph.

### 5.6.1 Platform Tests

In total there are nine platform tests: one for each target; an extra one for `Ozobot`, because it has two many different pieces to fit in a single program; a test just for variables; and test for errors. We do not display all of these tests here, because they are very large and do not properly fit in A4 size paper. But we describe them and our intentions with each test.

The idea behind each target test (or set of tests, in case of `Ozobot`) is very simple, it contains all the pieces available in that platform, arranged in a way that is syntactically correct, although it does not need to make any sense semantically. Whenever changes are made to the code generator of a given target platform, the test for that platform can be run to determine if it is still working correctly. This can be done in two ways: importing the generated code file into the target platform or comparing it with a previously stored file with the same program exported from the target platform. If changes are made to the parser, then all target tests should be run.

## Tactode Application

The variables test was designed, because even though there are variables in the other platform programs, we wanted a single test for using multiple variables. The process of testing is the same as the others: import to the destination platform or compare with previously generated files.

Finally, the errors test is designed to fail, that is, it is the only file that is syntactically incorrect and the objective is to see if the `Tactode` parser correctly identifies the multiple errors. This test is shown in Figure 5.9 with the errors already identified by the parser, according to the following list:

1. End blocks cannot come before the corresponding begin blocks, missing Repeat.
2. This block cannot be placed here.
3. Else blocks cannot come before If blocks.
4. This block is not defined for the chosen platform.
5. This expression is not valid.
6. This expression is not valid.
7. Expected no block in this position, but found some block.
8. End blocks cannot come before the corresponding begin blocks, missing Repeat.
9. This block cannot be placed here.
10. Flag block cannot be used in the middle of the program, only at the top.
11. This block cannot be placed here.
12. This expression is not valid.
13. This block requires an argument, but none was found.
14. This right parenthesis block has no corresponding left parenthesis block.
15. This left parenthesis block has no corresponding right parenthesis block.
16. This expression is not valid.
17. A variable with this name has already been created.
18. This variable has already been created.
19. This block needs a closing block (End...), but none was found, missing End Repeat.
20. This block needs a closing block (End...), but none was found, missing End If.
21. This block needs a closing block (End...), but none was found, missing End Repeat.
22. This block needs a closing block (End...), but none was found, missing End While.

# Tactode Application

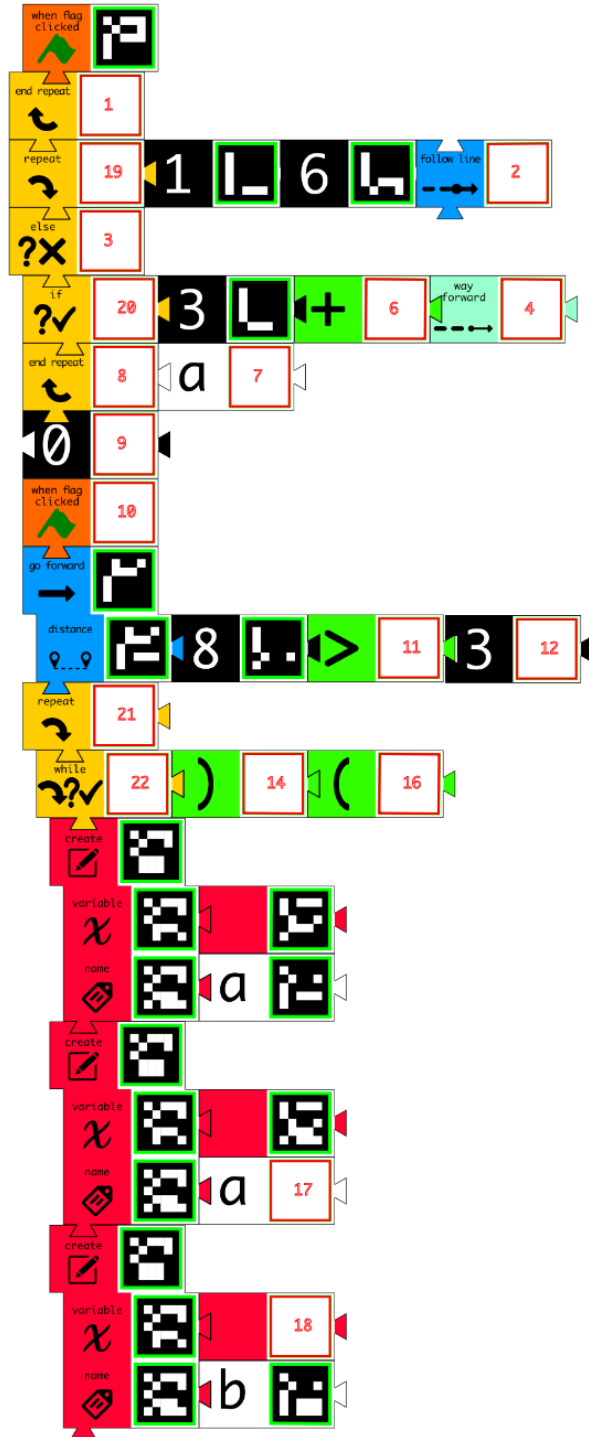


Figure 5.9: The Tactode error test after being parsed

If multiple errors are attached to the same piece, only the number of last one is shown in the image.

These platform tests validate the implementation of UC 1. The other use cases, UC 2 and UC 3, are manually validated using the settings page and the home page, respectively.

### 5.6.2 Visibility Tests

The visibility tests are a set of photographs of two programs, the Python prime number and the Scratch regular polygon generators. Not all of these tests pass in the current version of Tactode, but more of them pass now than with the initial version of image processor. Straight pictures taken from the top are the easiest to process.

Simple rotations vary according to the size of the program, especially the number of argument blocks in a single row, because the processor may place some of them in the wrong row.

Perspective pictures depend on where they are taken from and on the size of the program. Bottom perspectives are more likely to be well processed, followed by right side perspectives. This is again due to long rows of arguments being incorrectly placed in the matrix of markers.

Shadows are usually resolved but reflections that affect the Aruco tags are never fixed.

The images of all visibility tests, as well as the results (yes when they are correctly processed and no otherwise), can be seen in Figures 5.10–5.21.



Figure 5.10: Prime generator, top view, yes.

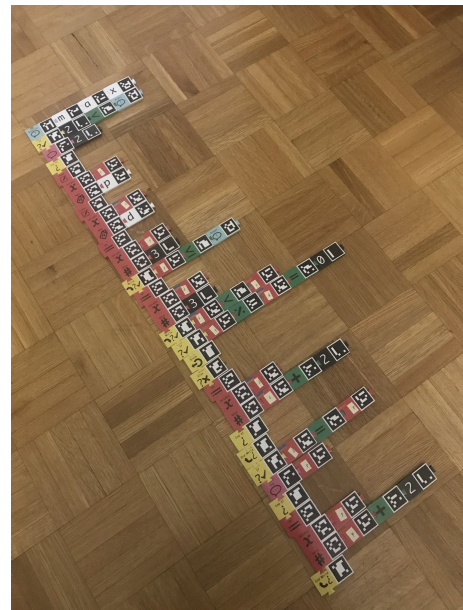


Figure 5.11: Prime generator, rotation, no.

# Tactode Application



Figure 5.12: Prime generator, 180° rotation, **yes**.



Figure 5.13: Prime generator, bottom perspective, **no**.



Figure 5.14: Prime generator, shadow, **yes**.



Figure 5.15: Prime generator, reflection, **no**.



## Tactode Application

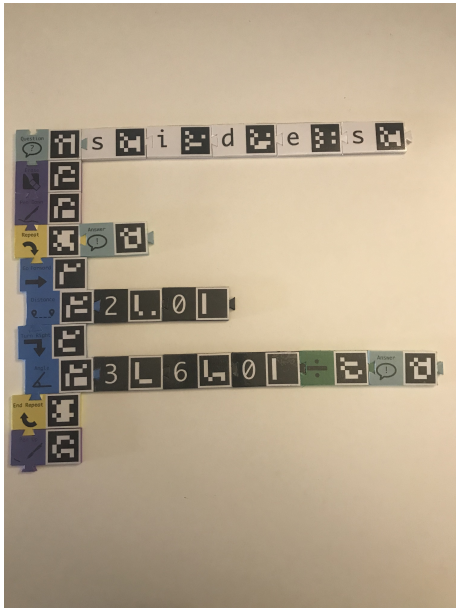


Figure 5.16: Regular polygon generator, top view, **yes**.

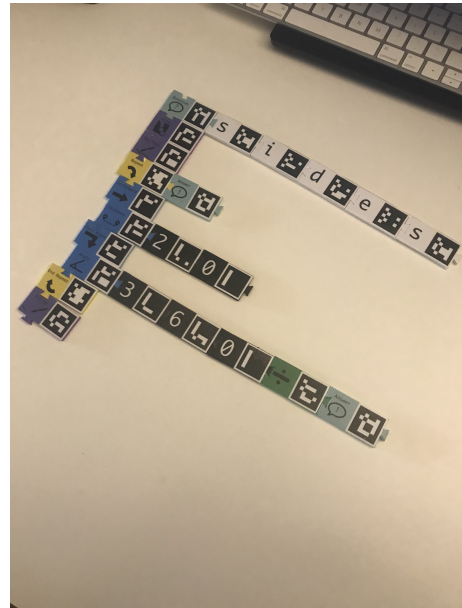


Figure 5.17: Regular polygon generator, rotation, **yes**.

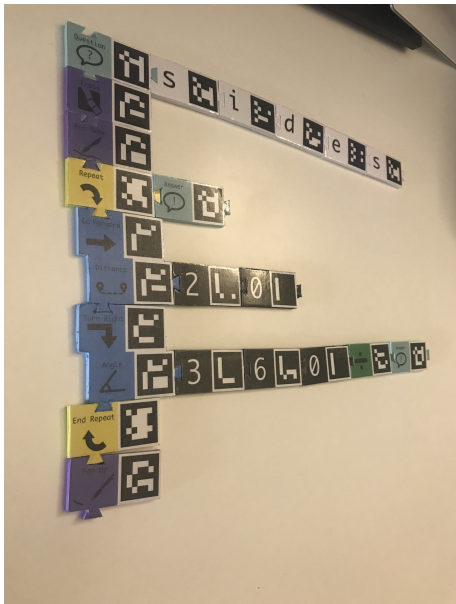


Figure 5.18: Regular polygon generator, left perspective, **no**.

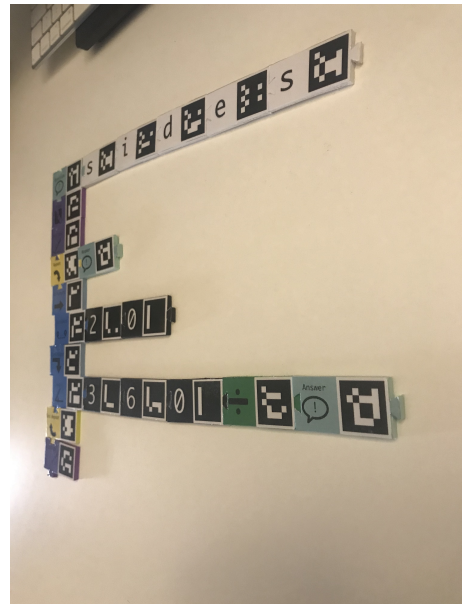


Figure 5.19: Regular polygon generator, right perspective, **yes**.

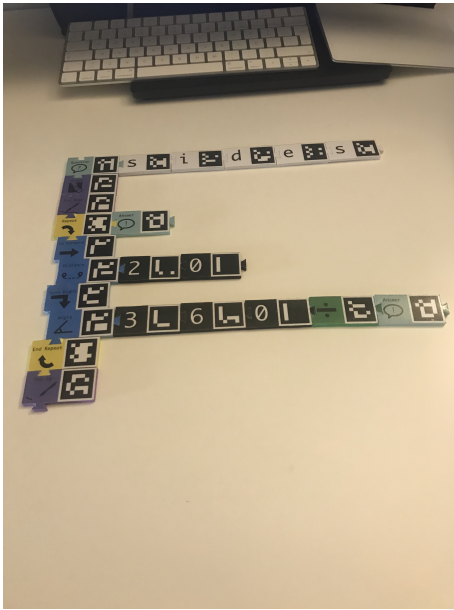


Figure 5.20: Regular polygon generator, bottom perspective, [yes](#).



Figure 5.21: Regular polygon generator, shadow [yes](#).

## 5.7 Evaluation

The proper evaluation of the `Tactode` application is shown in Chapter 6, where we present a few experiments with children and young adolescents and see how easily they use the application in a variety of platforms (smartphone, tablet and computer) and operating systems (Android, iOS, macOS and Windows).

However, we can already conclude some things from observing the current product and comparing it with the requirements we established for its development. We can also make use of the visibility tests to evaluate our application.

All functional requirements were fulfilled, with every use case implemented. However some things could be improved, such as allowing the user to zoom in on the image of the imported program, allowing previous programs to be selected and zoomed in.

As for the non-functional requirements, we made an effort to use mostly icons, but there is some text dependency, especially in the Settings page, which is probably better used by an adult or by older children. However, this is not a page to be used frequently, but only once in a while, so we deem its text dependency acceptable.

The application, particularly its home page, is intuitive with buttons only appearing when they can be used, which guides the user interaction. It is usually reliable, identifies all tags in different lighting conditions and, at least for smaller programs, even if the picture is not straight or taken from above. The image processing, parsing and code generation is also quite fast, taking only a couple of seconds for the largest programs we tested. There is also feedback so the user knows the program is being imported and they should wait. The previous programs page can also take a little while to load when there are many programs stored, this should probably be improved in future

developments.

### 5.8 Conclusion

The `Tactode` application is an essential companion for the `Tactode` tangible language. It is what makes our tangible blocks into a true programming language by allowing programs to be compiled into target languages and executed in their respective platforms.

We validated six target platforms, including `Python`, which is very different from the others. This is way above our initial idea which was to allow new platforms to be easily added, but not to do it from the start. It also goes to show just how simple it is to add a new platform, since `Sphero` was added later and its implementation was extremely fast. Indeed, it was only necessary to generate the appropriate code for each block that is available. Even `Python` was added later than the first four with relative ease, which is surprising considering its target code is very different from that of the other platforms.

Although there is room for improvement, the `Tactode` application fulfills all the requirements that were made of it. Plus, its architecture supports easy addition of new platforms and new blocks to each platform, which greatly facilitates the growth of the `Tactode` language.

## Tactode Application

## Chapter 6

# User Testing

This chapter is concerned with testing our solution with its intended users. We start with a general description of our objectives with these tests. Then we describe the experiments we performed and what they were meant to evaluate. Finally, we present and discuss the results of our experiments.

### 6.1 Introduction

Although we can evaluate some aspects of the `Tactode` language and application simply by analyzing to which degree they fulfill the initially established requirements, nothing can replace user tests in a practical project such as this one. Only by taking the product to its intended users can one see if it performs the way it is supposed to. This is especially true when the target demographic is very different from the development one, which is precisely the case of `Tactode`.

We believe testing with children is so important that we began those tests as soon as we had a minimally viable product, using small focus groups to gauge their reactions and how well they performed a couple of programming tasks. The idea was that these preliminary results could guide the continuing development of `Tactode` as well as help us to properly design the subsequent experiments.

We would have liked to perform more extensive tests, so that the volume of data would be statistically significant. This was not possible, mainly due to time constraints, because for each group of children we need a person observing them, answering some of their questions and taking notes. This means we can either have several groups, if we have enough volunteers, or we can only test one group at a time.

We did manage to test `Tactode` with a few pairs of children. These experiments were enlightening, and we are mostly pleased with their results. We also have ideas of which future developments are more important, based on the more recent of these tests.

## 6.2 Objectives

With these experiments we wanted to evaluate how well children could achieve simple programming objectives with `Tactode` and use the application to translate the code and execute it in the desired target platform. We also wanted to see how well the application performed in the different platforms (smartphone, tablet and computer) and operating systems (`Android`, `iOS`, `Windows` and `macOS`).

The experiments should determine if the children understand what each `Tactode` block does; if they are capable of creating ever more complex programs without guidance, aside from an initial explanation and example; if they identify and correct syntactic errors; if they can observe the execution and determine whether the target is behaving as intended; and if they can fix the eventual semantic errors.

Regarding the application, we wanted to check if the children were capable of using it independently once it was setup and demonstrated by their monitor. We also wanted to determine, for each target platform, how well the students managed to use the file that the `Tactode` application exported and then execute it.

In general, we wanted to see if the children enjoyed themselves while they were performing the activities, if they demonstrated interest and if both children in the group collaborated to reach the objectives. We also wanted to make a list of future improvements for both the language and the application based on the opinions of the children and on our observation of their performance.

Initially we were hoping to have a comparison between `Tactode` and some of the block based languages with graphical interfaces, namely the ones that our target platforms use. However, after the first attempts we realized this was interfering with our main goal, which was to assert that the `Tactode` language and its application are fit for the purposes they were created. We decided to focus our efforts, particularly considering that our time with children was always limited. However, there are plans to have a fully functional graphical interface for `Tactode`, in addition to the current tangible one. As such, we might again attempt a comparison, but between the two versions of the same language, which we believe would be more sensible.

## 6.3 Methodology

During our development of `Tactode` we have performed three separate tests. They were made at different stages of development and with distinct methodologies, so we shall describe them individually.

### 6.3.1 First Experiment

Although our target demographic is children or young adolescents, our very first tests were actually performed with a group of eight adolescents between the ages of fifteen and seventeen. The main reason for this is that the opportunity presented itself when the students were participating in a Junior University program on robotics, organized by Universidade do Porto. So, we decided we

## User Testing

could see how the students would interact with *Tactode* and that would at least help to guide future experiments.

We had little time to prepare and the *Tactode* pieces we now have were yet to be manufactured, so we decided to use simple rectangles of thick paper instead of puzzle like pieces, because it was easier to cut them. We did however add visual clues to the pieces, so the students would know where they were supposed to adjoin them. These pieces can be seen in Figure 6.1, they have \* signs for the vertical connections and ~ signs for the horizontal ones.

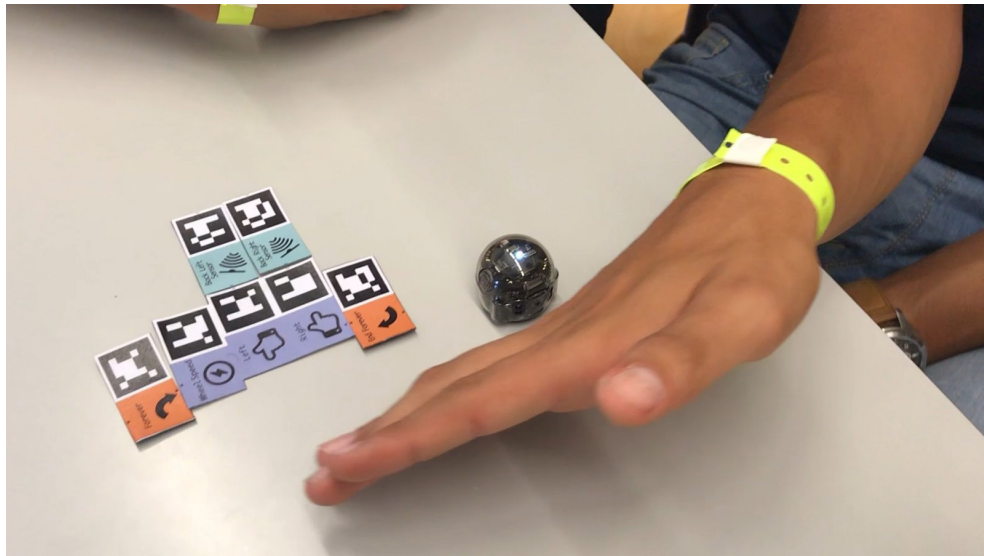


Figure 6.1: Teenagers using old *Tactode* pieces to program Ozobot.

The students of this experiment used the windows version of the *Tactode* application, taking pictures with a webcam attached to the computer. In terms of target, all students used an Ozobot robot, that was shared between the groups.

Before using *Tactode*, the students had been programming robots to do a variety of tasks. Amongst these were the obstacle reaction challenges, in which the robot had to:

1. Run away from obstacles behind it;
2. Follow obstacles in its front;
3. Run away from obstacles in its front;
4. Turn away from obstacles in its front;
5. Move forward until an obstacle is detected in its front and then stop;
6. Stay in the middle when detecting obstacles both in the front and behind.

We decided to ask the students to use *Tactode* to program Ozobot with the obstacle reaction challenges. Two pairs started with *Tactode*, while the other two used *Ozoblockly*, the programming language that comes with Ozobot. After completing a few tasks the groups switched

languages. Figure 6.1 shows a group of students using *Tactode* to program *Ozobot* to run away from obstacles on the back. Figure 6.2 shows the same challenge being programmed directly in *Ozoblockly*.



Figure 6.2: Teenagers using *Ozoblockly* to program *Ozobot*.

While the students were programming the robot, we observed them and took some notes that we present in the next section. We also asked them to fill the questionnaire in Section B.1.

### 6.3.2 Second Experiment

Our second experiment was meant as a proof of concept as well as a focus group to guide our final developments. It consisted of two separate challenges which are presented in [82] and [83], together with their results.

A total of twenty two children aged between ten and twelve years old participated in this test. Their teacher was given two tutorials, one for each challenge, which described the *Tactode* language and application, the challenge and the target platform.

The first challenge was to draw a regular polygon in *Scratch*, as shown in Figure 4.8. Fourteen of the twenty-two students participated in this activity. The second challenge was to solve a simple maze using *Ozobot*, as depicted in Figure 4.12. All twenty-two students participated. As with the first experiment, some groups began with *Tactode*, whilst others began with *Scratch* or *Ozoblockly*, depending on the challenge, then they switched.

The students were organized into groups of two or three elements and the objectives were explained to them. Then the teacher guided them through using the *Tactode* pieces to program *Scratch* and *Ozobot* for their respective challenges. Figure 6.3 shows a group of students that were using *Tactode* to program the *Scratch* cat to draw a regular polygon. The students in Figure 6.4 were tasked with helping *Ozobot* to solve a simple maze using the *Tactode* pieces.



## User Testing

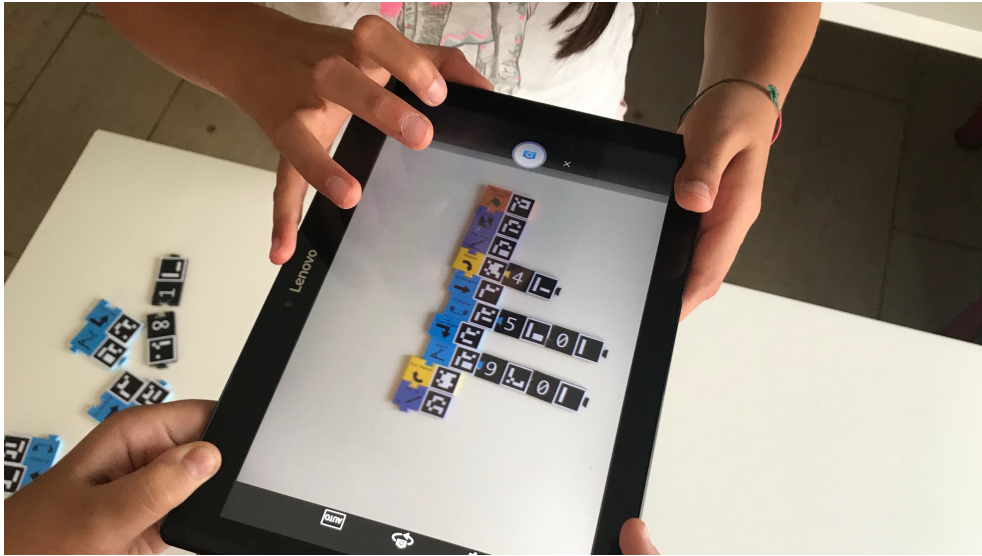


Figure 6.3: Children using the Tactode application to take a picture of their program to draw a regular polygon in Scratch.

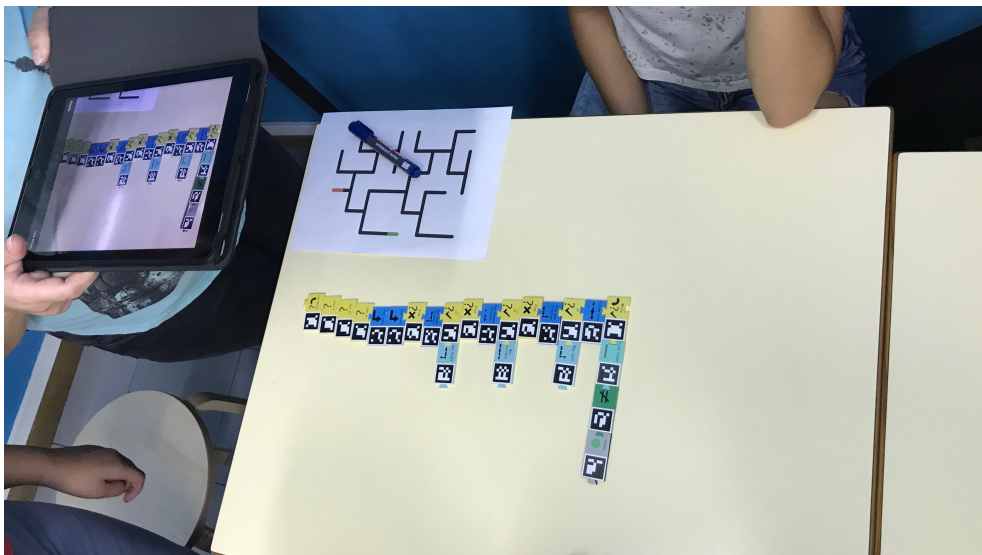


Figure 6.4: Children using the Tactode application to take a picture of their program to solve a simple maze with Ozobot.

As in the first experiment, we observed the activities and made notes. We also asked the students to answer the same questionnaire (Section B.1).

Because we felt the teacher had interfered with our intentions, especially making it impossible to evaluate how far the students would get with only an initial explanation and a simple example, we decided to have a more controlled environment for the third experiment.

### 6.3.3 Third Experiment

The third and final experiment was meant to test a nearly final version of `Tactode`. This time, each group of students was monitored by a volunteer, that was supposed to mainly observe and was only allowed to answer a few specific types of questions once the experiment began.

In this experiment, ten children with ages between eight and twelve participated. The children were organized into five pairs, and each pair was given a set of `Tactode` pieces, a robot and a device with the application installed, distributed according to Table 6.1.

Table 6.1: Material used in the third experiment.

Group	Robot	Device
1	Ozobot	Apple tablet
2	Ozobot	Lenovo tablet
3	Cozmo	Apple smartphone
4	Sphero	Apple computer
5	Sphero	Apple computer

All groups were given the same challenge, which was to have their robot move in such a way as to go through the sides of a regular polygon, as shown in Figure 4.7. Before the actual experiment began, there was a group conversation about regular polygons, which included their defining characteristics and the fact that the amplitude of an external angle of a regular polygon with  $n$  sides is  $360^\circ/n$ . This last part was deduced in such a way that it became clear that when a robot turned, to go from one side to the next, it had to turn  $360^\circ/n$ . We decided to provide this explanation initially because it is a mathematical result and it was not our objective to determine if students were capable of reaching it on their own.

Once the students were divided into pairs with their respective materials, the monitor of each group guided them through their first programming experience with `Tactode`, in which they were meant to make the robot go forward a certain distance and then turn. Then the students were shown how to use the application to take a picture of their program and export it. They were also shown how their particular robot could import and execute the resulting code file.

After the initial explanation and guided activity, the students were asked to program their robots to draw regular polygons with an increasing number of sides and using increasingly difficult programming constructs. They were meant to generate the triangle using only a sequence of movements. For the square they had to use a `repeat` block, otherwise the number of `forward` and `turn right` blocks would not be sufficient. The pentagon or hexagon should be the same as the square, but with different angles. For the heptagon, given that 360 is not divisible by 7

## User Testing

they should use the `divide` block instead of computing the angle themselves. At this point they should realize they only need to change two blocks (the argument of the `repeat` block and the number after the `divide` block) to obtain polygons with a different number of sides.

An extra challenge proposed to the students was to make the robot draw a regular polygon with a high number of small sized sides. When the robot is capable of doing this with a minimal precision, it looks very close to a circle. The idea was to teach children about approximating circles using regular polygons. The monitors were allowed to guide the children in this task and it was meant as a teaching moment more than a test.

The monitors were only allowed to answer questions about the meaning of specific `Tactode` pieces, the mathematic behind regular polygons and doubts about using the application or importing the code into the robot, but in the last case the monitors had to report the event in their observations.

Figures 6.5, 6.6 and 6.7 show a pair of children programming in `Tactode`, then loading their program onto `Ozobot` and finally placing the robot in the corner of a square to check if their program is correct.



Figure 6.5: Children programming in `Tactode` to make `Ozobot` draw a regular polygon.

In this third experiment the students used only `Tactode` and we did not ask them to make comparisons with the target languages. In spite of that, the children in this third experiment answered a longer questionnaire than those of the first two experiments. It can be seen in Section B.2.

Also, for each group, the volunteer monitor filled in an observation chart, where they had to answer the questions in Section B.3.

This last experiment was better structured, allowed us to have more control and yielded more results, not in number of children involved, which was smaller than in the second experiment, but in the number of things we observed and in the number of questions the children answered. Unfortunately, for this kind of control we need volunteers or enough time to test only a couple of groups simultaneously.

## User Testing



Figure 6.6: Children loading their program onto Ozobot.

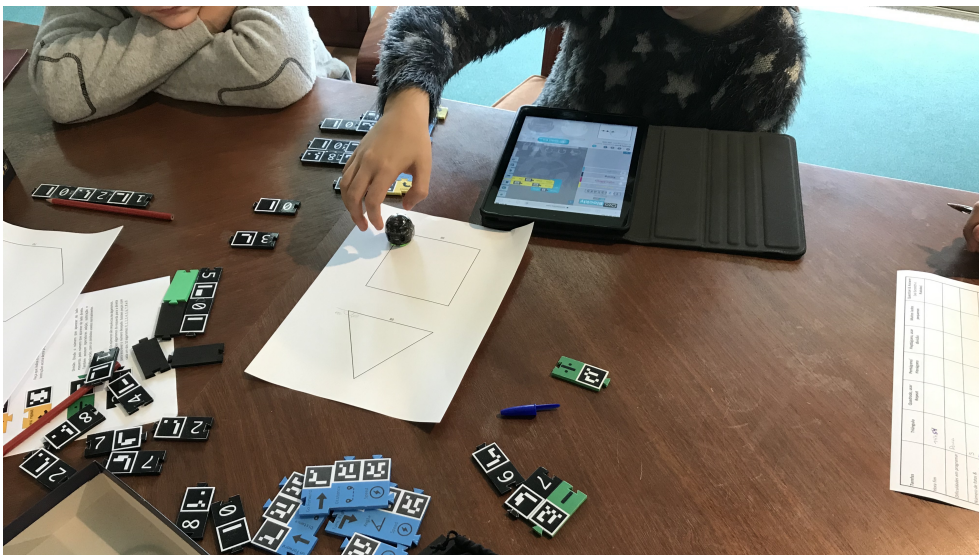


Figure 6.7: Children placing Ozobot in the corner of a square to execute their program.

This should be the model for future experiments, especially considering it is more in tune with our established goals for the tests. Indeed, we provided only an initial explanation and example and then observed how far the children went in a sequence of increasingly difficult programming challenges. We allowed them to make mistakes and then attempt to identify and fix those mistakes. The application was also used independently by the children after the first example and they were encouraged to import the code to the target platform and execute it.

## 6.4 Results

For all three experiments, we have two types of results: the answers of the students to the questionnaire at the end of the activities and the observations of those accompanying the tests. Whenever we use the 1 to 5 scale, 1 means ‘not at all’ and 5 means ‘completely’.

### 6.4.1 First Experiment

The students of this experiment were older and had already spent a few days programming robots, so we did not expect them to have many difficulties. However, both the language and the application were in its early stages, so we were anticipating some issues.

The groups that started with `Tactode` quickly started using the `proximity sensor` blocks as arguments of the `wheel speed` block, as shown in Figures 4.9 and 4.10. However, those using `Ozoblockly` used a more complex structure with `if` blocks. Their idea was to make the robot move forward (in the first challenge), if the sensors had a positive value. It works, but is not as simple and it does not make the robot move faster if the obstacle is nearer.

At first, given that the `Tactode` pieces were rectangles and did not have tabs and slots, the students did not fit them properly. The visual clues did not help, until they were specifically told that they had to align the visual clues on two consecutive pieces. Another difficulty with the `Tactode` solution was that it sometimes took several pictures for all the pieces to be identified. Also, it took some steps to get the code into the robot, while the students using `Ozoblockly` could do that faster.

Other than the difference of programming constructs used, the groups completed their tasks independently of which language they were using.

The answers of the 8 students to the questionnaire are shown in Tables 6.2 and 6.3.

Table 6.2: Answers of the students to the classification questions, in the first questionnaire.

Question	1	2	3	4	5
Did you understand the objectives of the activity?	-	-	-	2	6
Did you find the <code>Tactode</code> language easy to use?	-	-	3	3	2
Did you find the <code>Tactode</code> application easy to use?	-	-	2	1	5

Table 6.3: Answers of the students to questions comparing `Tactode` and `Ozoblockly`, in the first experiment.

Question	Tactode	Ozoblockly
Which language did you consider easier?	4	4
Which language did you prefer?	3	5

## 6.4.2 Second Experiment

Both the polygon challenge and the maze challenge of the second experiment lasted about two hours each. This includes material distribution, introductory explanation, activities and filling the questionnaires.

All groups completed the required tasks for the maze challenge. In the regular polygon challenge, the last step, which is using the `question` and `answer` blocks to let the user decide the number of sides, was not attempted by any group. The teacher deemed it too difficult and decided to skip it.

There was assistance from the teacher, when the students called, so we are not clear on how far they would get or how much longer it would take them without said assistance. In any case, we noticed the students completed at least part of the challenges on their own. Also, in the maze challenge, one of the groups immediately understood that the turning left whenever possible strategy, that the teacher advised them to use, was not the most efficient for the particular maze that was being used. This shows a deeper understanding of the problem.

This experiment made use of the final `Tactode` pieces, so all students immediately understood how the pieces were supposed to fit together. However, in one of the rooms where the experiments were conducted there were many issues with light reflections which forced the teacher to help the children to take pictures.

We observed that the groups using `Tactode` were more focussed, while the ones using `Scratch` or `Ozoblockly` (depending on the challenge), tried a wider variety of blocks and spent more time programming other things that were not asked.

The children that performed both challenges were clearly more enthusiastic about using robots than about programming without them.

The answers of the fourteen children that participated in the regular polygon challenge are in Tables 6.4 and 6.5. The answers of the twenty children that participated in the maze challenge are in Tables 6.6 and 6.7.

Table 6.4: Answers of the students to the classification questions, in the regular polygon challenge of the second experiment.

Question	1	2	3	4	5
Did you understand the objectives of the activity?	-	-	-	4	10
Did you find the <code>Tactode</code> language easy to use?	-	-	3	6	5
Did you find the <code>Tactode</code> application easy to use?	-	-	-	10	4

## User Testing

Table 6.5: Answers of the students to questions comparing `Tactode` and `Scratch`, in the regular polygon challenge of the second experiment.

Question	Tactode	Scratch
Which language did you consider easier?	5	9
Which language did you prefer?	5	9

Table 6.6: Answers of the students to the classification questions, in the maze challenge of the second experiment.

Question	1	2	3	4	5
Did you understand the objectives of the activity?	-	-	-	8	14
Did you find the <code>Tactode</code> language easy to use?	-	-	2	11	9
Did you find the <code>Tactode</code> application easy to use?	-	-	-	18	4

Table 6.7: Answers of the students to questions comparing `Tactode` and `Ozoblockly`, in the maze challenge of the second experiment.

Question	Tactode	Ozoblockly
Which language did you consider easier?	9	13
Which language did you prefer?	9	13

### 6.4.3 Third Experiment

In this experiment, each group had a monitor permanently observing them, so we have a much more detailed results. Table 6.8 gives a summary of the children involved, including their age, school year, programming experience, robot experience, percentage of participation in the activities, as well as motivation and enjoyment of programming with `Tactode`.

Table 6.8: Children participating in the third experiment.

Group	Child	Age	School Y.	Program. Exp.	Robot Exp.	Part. <sup>1</sup>	Mot. <sup>2</sup>	Enj. <sup>3</sup>
1	1	10	5	Scratch	No	60	5	5
1	2	12	7	JavaScript	No	40	4	5
2	1	10	5	Lego	Lego	50	1	1
2	2	9	4	Lego	Lego	50	1	1
3	1	8	4	Sharkcoders <sup>4</sup>	No	45	5	5
3	2	9	4	Yes	Yes	55	5	5
4	1	10	5	No	Sphero BB8	50	4	3
4	2	10	5	Yes	Yes	50	5	3
5	1	10	5	No	No	66	5	5
5	2	9	4	No	No	34	3	4

<sup>1</sup>Participation (%)

<sup>2</sup>Was the child motivated? (1 to 5)

<sup>3</sup>Did the child enjoy programming with `Tactode`? (1 to 5)

<sup>4</sup>Portuguese network of programming and robotics schools for children and adolescents.

## User Testing

The completion time of each task was measured by the monitor and the results are displayed in Figure 6.8. As shown, group 2 gave up and did not complete all the assignments. Group 3 tried the circle task, but their robot (Cozmo) was not executing it for some reason, which the monitor could not understand.

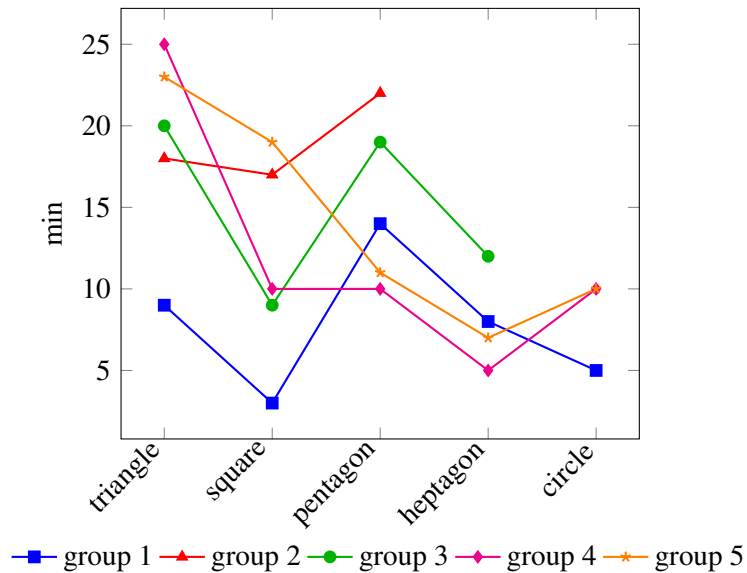


Figure 6.8: Completion times (in minutes) of each task by group.

The number of pictures taken by each group in each task is shown in Figure 6.9. Recall that group 1 was using an Apple tablet, group 2 was using a Lenovo tablet, group 3 was using an Apple smartphone and groups 4 and 5 were using Apple computers. The pictures for groups 4 and 5 were taken using smartphones, whose brand and model we did not register.

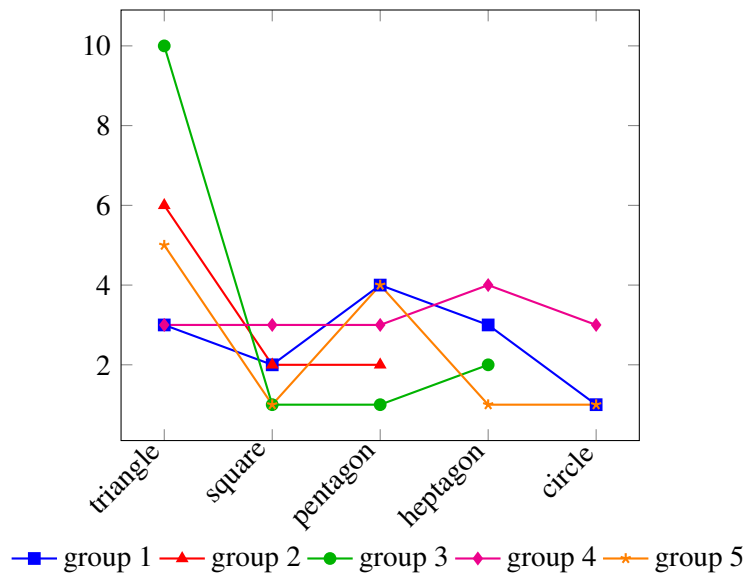


Figure 6.9: Number of photos taken for each task by group.



## User Testing

Group 1 was having problems with the first Apple smartphone used, but then changed to another Apple smartphone and all went well. After the experiment we ran some tests and identified some issues with the `Aruco` library that was being used that caused the problems with the first smartphone. We have since switched to the `OpenCV` implementation of `Aruco` and the problem seems to be fixed.

Group 2 had difficulties with the Lenovo tablet, whose pictures were very unfocused. They ended up sharing the Apple tablet of group 3. Group 5 was initially having difficulties with light reflections and then changed to another location in the room to avoid that.

According to the monitors, with the exception of group 2, the children revealed little difficulties when programming. They made some mistakes, but managed to fix them. Group 1 used the wrong length for the side of the triangle and used a  $90^\circ$  angle in the pentagon, but caught both these errors on execution and fixed them.

The children in group 2 revealed no interest in the activities. At first child 2 was playing while child 1 was programming alone. Child 1 revealed no difficulties and managed to accomplish the tasks in a timely fashion, but still had no motivation and appeared to be very bored. Eventually child 1 gave up on continuing and child 2 took over, they never cooperated. Child 2 appeared to have many difficulties and to get easily distracted. The monitor made note that child 2 completed the pentagon with some help and gave up after that.

Group 3 used simultaneously the `turn right` and `turn left` blocks in the triangle. When they started using the `repeat` block one of the children was having trouble understanding what it does, while the other picked it up immediately. In the heptagon the children had difficulties realizing they could use  $360 \div 7$  as the argument of the angle amplitude of the `turn right` command.

Group 4 only revealed difficulties in the first task with understanding the meaning of each block. They did not set the velocity in the triangle task, but they noticed this error on compilation (the `Tactode` application reports it) and they fixed it.

Group 5 had some difficulties initially. The monitor reported that they used too many pieces in the `repeat` block, but noticed this mistake and fixed it while programming.

The answers of the children to the questionnaire at the end of the third experiment are shown in Table 6.9, with the robot related questions detailed in Table 6.10 according to the robot each child was using.

## 6.5 Discussion

When examining the results of the first experiment, we need to take into account the age of the participants. In any case, some things become clear. The puzzle like tabs and slots are fundamental in making it clear how the pieces are supposed to fit together. Indeed, these were the only students who had any trouble with that and they were the only ones using rectangular pieces.

Another thing that we observed in that experiment is that the students using `Tactode` reached the objectives with simpler programs than those using `Ozoblockly`. We are convinced this is

## User Testing

Table 6.9: Answers of the students to the classification questions, in the third experiment.

Question	1	2	3	4	5
Did you enjoy this activity?	-	-	-	-	10
Did you understand the objectives of the activity?	-	1	-	2	7
Did you understand how the robot works?	-	-	-	5	5
Did you understand what the <code>Tactode</code> pieces you used do?	-	-	-	1	9
Did you find programming the robot with <code>Tactode</code> easy?	-	-	1	3	6
Did you find the <code>Tactode</code> application easy to use?	-	-	2	-	8
Did you find the robot easy to use?	-	-	1	-	9
Did you find getting the code to the robot easy?	-	1	1	5	3

Table 6.10: Answers of the students to the robot related questions, separated by the robot used, in the third experiment.

Question	Cozmo					Ozobot					Sphero				
	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Understand how robot works?	-	-	-	2	-	-	-	-	1	3	-	-	-	2	2
Find robot easy to use?	-	-	1	-	1	-	-	-	-	4	-	-	-	-	4
Find getting code to robot easy?	-	-	1	1	-	-	-	-	3	1	-	1	-	1	2

due to the fact that we gave them only the `Tactode` pieces needed for the job, forcing them to come up with a solution using only those pieces. The students using `Ozoblockly` had all blocks available simultaneously. Plus, when one chooses a sensor block, `Ozoblockly` automatically brings forward an if construct. They probably decided to do that to facilitate things, but sensor blocks can be used in other ways, so this does not always help and, in our opinion, should not be done.

In the questionnaires, all the students claimed to have understood the objectives well enough, most of them completely. Three of the eight students found the `Tactode` language neither easy nor hard to use, while the others found it easy enough or totally easy. As for the application, two students were in the middle of the scale, but most found it completely easy. In terms of comparing the two languages, they were tied in difficulty and `Ozoblockly` had a very slight advantage in preference.

We believe some of the difficulties expressed by the students regarding the language are related to the rectangular shape of the pieces. This is supported by the questionnaire data in the second and third experiments.

The results of the second experiment were somewhat tainted by the over involvement of the teacher. But still, the students did complete at least part of the tasks on their own without interference.

The children showed a clear preference for `Ozoblockly` and `Scratch` deeming them both easier and more enjoyable than `Tactode`. This is natural, given the propensity of children towards handheld devices such as tablets and smartphones. Plus, there were more blocks available in those languages and they could do more different things. Which also explains why those students were

## User Testing

more distracted.

Systematic questioning of all students revealed that the subjects reported to have understood what they were supposed to do. The large majority found the `Tactode` language easy to use, with three in fourteen and two in twenty two opting for neither easy nor hard. The application had more consensus, with every student finding it easy. However, the number of students declaring it completely easy was small. This makes sense, considering that at this point of its development the application was working but with difficulties in obtaining good pictures of the programs, sometimes due to reflections, other due to lack of focus.

The third experiment had fewer participants but more extensive findings. We can see from the report on the children that they managed to collaborate on almost all groups. The exception is group 2, where the 50-50 participation means that one child did the first two challenges alone and then the other child did the third challenge. But this was the group that showed no interest, so their inability to collaborate is likely not related to `Tactode`.

Regarding motivation and enjoyment of programming with `Tactode`, we can see that it was mostly good or excellent, with only a few exceptions.

The completion times vary in a somewhat strange pattern. We believe this happened because instead of choosing between executing the pentagon or the hexagon, some groups chose to do both and took longer because of that. Indeed, from the square to the pentagon, only the number of sides and the angle changes, no new programming constructs are introduced, so it does not make sense to take a lot longer. In any case, the heptagon was completed in twelve minutes by group 3 and less than ten by all other groups (except group 2 which gave up before that task). This shows that by the end of the progressively difficult tasks, the children had a good grasp of what needed to be done.

The number of pictures taken was not as low as hoped. Having to take many pictures is frustrating and is not the mark of a robust solution. At the end of this third experiment we were quite convinced of the need to improve the image processing of our application. We did that by switching to the `OpenCV` implementation of `Aruco` and improving the pre-processing, namely to correct orientation. We are convinced the current version of the application would yield a significantly lower number of photographs when tested, as our visibility tests support. Note that the pictures that the children took of their programs were much better than most pictures in our visibility tests.

In answer to the questionnaires, all children reported to have enjoyed the activity very much (even the ones from group 2). One child declared not to have understood the objectives, while all the others said they understood, most of them completely. How the robot works, whether it is easy to use and how easy it is to get the code into the robot received less consensus, but still the results were mostly positive. More importantly, all children understood what each `Tactode` piece they used is supposed to do, plus only one did not find it easy to program with `Tactode`. The application had better results than in the previous experiment, with only two children choosing the middle of the scale and all others opting for the top.

## User Testing

Even though all children were encouraged to take the questionnaires seriously and answer them truthfully and thoughtfully, we face their answers with some suspicion. Sometimes children think they should be nice or they answer fast without thinking very well. Indeed, it makes no sense that children who were completely uninterested and gave up before the end of the activity to report they completely enjoyed themselves. This is why the observations made by the monitors in the third experiment are so important. In any case, we think the results were mostly positive, with some suggestions for improvement (image processing and code transference to the target) becoming clear.

We wanted to determine if children were capable of using `Tactode` for programming tasks and the answer is clearly yes. The children understand what the blocks do, create programs on their own after the initial example, and identify and fix box syntactic and semantic errors.

We also wanted to check how well the children used our application. They are capable of using it on their own, but not without some frustration. We hope to have fixed this issue in the current version though. Importing the resulting code to the target platform varies a lot depending on the platform and is not within our reach to fix, because it depends on the implementation of each target. In any case, we can implement a local simulator in the application, which would make quick checks much faster.

The children and even adolescents seemed to enjoy themselves while learning with `Tactode`. Collaboration was very good, which is hard to achieve when programming in group.

Finally, these experiments tested all supported operating systems and types of devices. We also tested a variety of targets, with only `Python` and `Robobo` not used by the students. Nevertheless, these targets were tested in our platform tests.

## 6.6 Conclusion

We began testing `Tactode` as soon as we had a viable product. The earlier experiments may not have been as well prepared, but they helped to guide our development and they shaped the final experiment, where we managed to test all our objectives and obtain extensive results, despite the lack of subjects.

Overall, we validated the initially proposed goals and proved that the `Tactode` programming system works and can be used successfully by children to learn to program step by step. Note that getting children to enjoy a difficult and abstract discipline, such as programming, is not an easy task, and with the help of robots, carefully designed challenges, handheld devices and a tangible interface that seems to be quite possible.

# Chapter 7

## Conclusion

In this chapter we present a recap of the work realized throughout this dissertation. We give a report on the degree of satisfaction of our initial objectives, presenting the main contributions of our work. We also present ideas for future work in continuation of our developments.

### 7.1 Overview

We began this document with an exploration of the current trends in STEM education, educational programming languages and educational robots (see Chapter 2). Our aim was to clarify the importance of teaching children to program early on and determine the state of the art in the tools that exist for that purpose.

Chapter 3 was devoted to stating our problem. We presented what we consider to be the ideal in an educational programming language. Then we exposed the limitations of the existing languages and the consequences of those limitations. Finally, we presented our proposal, defining our goals for the `Tactode` tangible programming language and the research questions we wished to answer in this dissertation.

In Chapter 4 we presented the `Tactode` programming language. We started with the choices we made regarding block shape, materials, manufacturing process, programming paradigms, piece recognition and target platforms. Then we presented each block as well as the syntactic rules for programming with `Tactode`. We also gave examples in the form of challenges that showcase the programming constructs currently supported. Finally, we closed the chapter with an evaluation of our language, by measuring the level of achievement of our initial goals for it.

Chapter 5 exposed the `Tactode` application, which is used to capture, compile and export programs to the target platforms. We started with an elicitation of the requirements for this application. Then we presented the architecture, using activity and class diagrams, as well as explanations of the more relevant features. In the following section we gave a detailed description of the transpiler, since it is at the heart of the application and it is what makes the set of blocks from

Chapter 4 into a real programming language. We then showed the resulting application with each of its screens, giving an explanation of the features and how they should be operated. There was also a set of semiautomatic tests aimed at proofing our application after each development, both in terms of compilation of each block for each platform and in terms of visibility, that is the quality of the image processing.

In Chapter 6 we described the experiments we performed with the intended users of our programming system. We began by stating our objectives with these experiments. Then we expose the methodology of our three experiments, followed by the results of each, both in observations and in questionnaires filled by the children. We closed the chapter with a discussion of our results and their relation to our initial goals.

## 7.2 Contributions

Our main contribution with this work is an extremely inexpensive (in terms of materials) multi-target tangible programming language, together with the multi-platform application that allows it to be compiled.

Each `Tactode` piece costs as little as 0,02 €, thanks to a manual manufacturing process and the use of inexpensive materials such as paper and EVA. The pieces are also durable and fit well together, making it possible for children to easily move their programs.

A total of six target platforms were validated: `Ozobot`, `Cozmo`, `Robobo`, `Sphero`, `Scratch` and `Python`. The robotic platforms are more engaging for children while the others add a wide variety of possibilities to `Tactode`. Because we chose to implement a greater number of platforms, they have some missing features, but as our challenges show, it is already possible to do a variety of interesting programs in each.

We developed an application that works in `Android`, `iOS`, `macOS` and `Windows`, as well as any web browser. It can capture large `Tactode` programs in conditions of uneven light, as long as there are no reflections. It is also tolerant to rotations and some inclinations. It requires no network access, since the image is captured, processed and compiled locally. The whole process takes as little as two seconds for our biggest challenge, even in mobile platforms. The resulting code can be share directly with the platforms that allow it, like `Cozmo`, sent to some network location or saved locally for the target platform to access later.

The application saves previous programs, that the user can re-share or delete. It also allows configuration of the destination platform, the image source (camera or previously obtained file) and interface language (english or portuguese). Although the currently supported languages are only two, addition of new languages is extremely straightforward, as well as facilitated by the reduced amount of text.

Both the language and application were tested with the intended users and the results revealed that children are able to program a sequence of increasingly difficult challenges in a timely fashion, without any help after the initial explanation and example. The children were also able to

independently import and execute the programs in the target platforms. The details and results of some of these experiments were published in [82] and [83].

### 7.3 Future Work

Although `Tactode` is already a quite capable tangible programming system, further development can help to realize its full potential, both in the language and the application. Also, further user tests can help to achieve statistical significance, clarify our target demographic, design interesting challenges and point other development needs that we have yet to identify.

#### 7.3.1 Language

Future work in the language is essential to truly expand its capabilities, so that it allows further learning and supports more involved projects. Also, it would be interesting to provide all features of the destination targets.

The most important feature to add is subroutines, both for procedures and for functions (including return values). These will allow two different programming paradigms to be taught with `Tactode`: procedural and functional. They will also contribute immensely to the ability to create highly complex projects using `Tactode`.

Second in the list of future priorities is the addition of more events. Currently only the program start event is implemented, but with the exception of `Ozobot` and `Python`, our target platforms support a multitude of events, which can be used to teach event-driven programming.

Adding more sensor blocks for the robotic platforms, as well as more sounds and visual effects is important, so that `Tactode` can better teach programming of robots. These features should also make for more interesting and varied programs.

Finally, we should add more variable types, such as strings and booleans, instead of just having integers. This would be useful to teach children about the different types and how it is important to check that each variable has the correct type.

#### 7.3.2 Manufacturing

Alternative manufacturing methods should be explored for the `Tactode` blocks. The current method is very inexpensive, with each piece costing only 0,02 €. But it is slow and error prone, due to the pieces being manually cut. The resulting product is resilient, but the paper on top might start to show signs of wear.

Laser cutting and engraving of 4 mm EVA foam seems like a good solution, as long as the price can be kept low. This would allow quick production of large quantities and the result should be extremely resilient. Due to the malleability of EVA, the pieces fit together easily and can be moved without breaking apart. However, when it comes to engraving, we need to guarantee a minimum contrast between engraved and non engraved parts, given that the `Aruco` fiducial markers need to be reliably recognized.

## Conclusion

Another option is to use the same process that is used in commercially sold puzzles, in which the pieces are made of cardboard. However, we feel these pieces would be less resilient than those made of EVA. Note that typical puzzles are not manufactured with the intention of being assembled and disassembled a large number of times.

### 7.3.3 Application

In the future, the `Tactode` application should be enhanced into a true IDE, allowing the captured programs to be imported into a graphical edition interface and executed in a local robot simulator.

The graphical interface would make small changes to the program easier, but also serve as an independent programming environment. The transition between tangible and graphical would then become seamless.

If a text based interface is also added, such as translating `Tactode` programs to python and showing the resulting code, that could be executed in a local console, the full transition from tangible to work appropriate languages would be accomplished.

Also interesting would be to add more platforms, such as `Dash`, `Thymio`, `micro:bit` and `Minecraft Python`. The last two of these are significantly different from the current platforms, so they would add the most to the variety that `Tactode` supports.

Finally, we can add `Linux` support, using the `Electron` framework together with `Ionic`.

Another possible addition to the application would be to improve the image processing technique, so that fiducial markers would no longer be necessary in the `Tactode` blocks, whose contents would be only the ones that are useful for the programmer.

### 7.3.4 User Testing

We should repeat the third experiment with as many children as possible, in search of statistical significance, but also to test the latest version of the application, in which image capturing and processing should be both faster and more reliable. It would be interesting to test `Tactode` with younger children as well, in order to gain a clearer picture of our target demographic and to test the lower limits of our entry level.

In order to study the benefits of tangible interfaces over graphical ones, we should use the graphical version of `Tactode` as soon as it is finished. Also, it is important to make detailed notes of the performance of children with both interfaces, since it is quite possible their preferences are not aligned with better performance.

Another possible avenue for future testing is to have schools using `Tactode` independently for a period of time and obtaining feedback from both teachers and students.



# References

- [1] Noonan, Ryan. Office of the Chief Economist, Economics and Statistics Administration, U.S. Department of Commerce. Stem jobs: 2017 update. ESA Issue Brief 02-17, available at <https://www.commerce.gov/sites/default/files/migrated/reports/stem-jobs-2017-update.pdf>, March 2017. Accessed 2019-02-24. Cited on pages 1 and 5.
- [2] U.S. Department of Education. Science, technology, engineering and math: Education for global leadership. Available at <https://www.ed.gov/stem>, March 2015. Accessed 2019-02-24. Cited on page 1.
- [3] M.E. Karim, S. Lemaignan, and F. Mondada. A review: Can robots reshape k-12 stem education? volume 2016-March, 2016. Cited on pages 2 and 6.
- [4] J. Chetty. *Combatting the War Against Machines: An Innovative Hands-on Approach to Coding*, pages 59–83. Springer International Publishing, Cham, 2017. Cited on pages 2 and 6.
- [5] R. Avanzato. Integrating robot design competitions into the curriculum and k-12 outreach activities. *Communications in Computer and Information Science*, 44 CCIS:271–278, 2009. Cited on pages 2 and 6.
- [6] G. Tewolde and J. Kwon. Robots and smartphones for attracting students to engineering education. 2014. Cited on pages 2 and 6.
- [7] F.B.V. Benitti and N. Spolaôr. *How Have Robots Supported STEM Teaching?*, pages 103–129. Springer International Publishing, Cham, 2017. Cited on pages 2 and 6.
- [8] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67, November 2009. Cited on pages 2, 6, 7, and 29.
- [9] Equipa de Recursos e Tecnologias Educativas - Ministério da Educação. Programação e Robótica no Ensino Básico. Available at <http://erte.dge.mec.pt/programacao-e-robotica-no-ensino-basico-0>. Accessed 2019-02-24. Cited on page 2.
- [10] H. Uzunboylu, E. Kınık, and S. Kanbul. An analysis of countries which have integrated coding into their curricula and the content analysis of academic studies on coding training in turkey. *TEM Journal*, 6(4):783–791, 2017. Cited on pages 2 and 6.
- [11] L. Major, T. Kyriacou, and O. P. Brereton. Systematic literature review: teaching novices programming using robots. *IET Software*, 6(6):502–513, Dec 2012. Cited on page 2.

## REFERENCES

- [12] N.Ç. Özüörçün and H. Bicen. Does the inclusion of robots affect engineering students' achievement in computer programming courses? *Eurasia Journal of Mathematics, Science and Technology Education*, 13(8):4779–4787, 2017. Cited on page 2.
- [13] Y.C. Huei. Benefits and introduction to python programming for freshmen students using inexpensive robots. pages 12–17, 2015. Cited on page 2.
- [14] American Academy of Pediatrics. Media and young minds. *Pediatrics*, 138(5), 2016. Cited on page 3.
- [15] M. S. Horn and R. J. K. Jacob. Designing tangible programming languages for classroom use. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, TEI '07, pages 159–162, New York, NY, USA, 2007. ACM. Cited on pages 3 and 17.
- [16] M. S. Horn, E. T. Solovey, R. J. Crouser, and R. 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, pages 975–984, New York, NY, USA, 2009. ACM. Cited on pages 3 and 29.
- [17] M. S. Horn, R. J. Crouser, and M. U. Bers. Tangible interaction and learning: the case for a hybrid approach. *Personal and Ubiquitous Computing*, 16(4):379–389, Apr 2012. Cited on page 3.
- [18] T. Sapounidis and S. N. Demetriadis. Exploring children preferences regarding tangible and graphical tools for introductory programming: Evaluating the proteas kit. In *2012 IEEE 12th International Conference on Advanced Learning Technologies*, pages 316–320, July 2012. Cited on pages 3 and 29.
- [19] T. Sapounidis, S. Demetriadis, and I. Stamelos. Evaluating children performance with graphical and tangible robot programming tools. *Personal and Ubiquitous Computing*, 19(1):225–237, January 2015. Cited on pages 3, 14, and 29.
- [20] Institute of Medicine. *From Neurons to Neighborhoods: The Science of Early Childhood Development*. The National Academies Press, Washington, DC, 2000. Cited on page 5.
- [21] T. S. Dee and H. H. Sievertsen. The gift of time? school starting age and mental health. Working Paper 21610, National Bureau of Economic Research, October 2015. Cited on page 6.
- [22] S. Durand. Finding the 'perfect' school starting age. Available at <https://www.geteduca.com/blog/perfect-school-starting-age/>, May 2017. Accessed 2019-02-24. Cited on page 6.
- [23] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA, 1980. Cited on pages 6, 12, and 29.
- [24] P. Guo. Python is now the most popular introductory teaching language at top u.s. universities. Available at <https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext>, July 2014. Accessed 2019-02-24. Cited on page 7.
- [25] Lifelong Kindergarten Group at the MIT Media Lab. Scratch. Available at <https://scratch.mit.edu>, 2005. Accessed 2019-02-24. Cited on page 7.

## REFERENCES

- [26] Wikipedia. Scratch (programming language). Available at [https://en.wikipedia.org/wiki/Scratch\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Scratch_(programming_language)), December 2017. Accessed 2019-02-24. Cited on page 7.
- [27] J. Mönig. Snap! Available at <http://snap.berkeley.edu/about.html>. Accessed 2019-02-24. Cited on page 8.
- [28] J. Chung. Stencyl. Available at <http://stencyl.com>. Accessed 2019-02-24. Cited on page 8.
- [29] N. Fraser. Ten things we’ve learned from blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 49–50, October 2015. Cited on page 8.
- [30] E. Pasternak, R. Fenichel, and A. N. Marshall. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B B)*, pages 21–24, October 2017. Cited on page 8.
- [31] Google for Education. Blockly. Available at <https://developers.google.com/blockly/>. Accessed 2019-02-24. Cited on page 8.
- [32] Microsoft. Makecode. Available at <https://makecode.com>. Accessed 2019-02-24. Cited on page 9.
- [33] R. Pausch, T. Burnette, A. C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. Alice: Rapid prototyping for virtual reality. *IEEE Computer Graphics and Applications*, 15(3):8–11, May 1995. Cited on page 10.
- [34] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: A 3-d tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5):107–116, April 2000. Cited on page 10.
- [35] Carnegie Mellon University. Alice. Available at <https://www.alice.org>. Accessed 2019-02-24. Cited on page 10.
- [36] Alan Kay et. al. Squeakland. Available at <http://www.squeakland.org>. Accessed 2019-02-24. Cited on page 10.
- [37] Wikipedia. Visual programming language. Available at [https://en.wikipedia.org/wiki/Visual\\_programming\\_language](https://en.wikipedia.org/wiki/Visual_programming_language), January 2018. Accessed 2019-02-24. Cited on page 11.
- [38] Lifelong Kindergarten Group at the MIT Media Lab. Scratchjr. Available at <https://www.scratchjr.org>. Accessed 2019-02-24. Cited on page 11.
- [39] Microsoft Research. Kodu. Available at <https://www.kodugamelab.com>, 2009. Accessed 2019-02-24. Cited on page 12.
- [40] Lego. Mindstorms: Learn to program. Available at <https://www.lego.com/en-us/mindstorms/learn-to-program>, 2013. Accessed 2019-02-24. Cited on page 12.
- [41] Yaroslavski, D. Lightbot. Available at <http://lightbot.com>, 2017. Accessed 2019-02-24. Cited on page 12.

## REFERENCES

- [42] Fisher Price. Think & learn code-a-pillar application. Available at [https://www.fisher-price.com/en\\_US/brands/think-and-learn/learning-apps/index.html](https://www.fisher-price.com/en_US/brands/think-and-learn/learning-apps/index.html). Accessed 2019-02-24. Cited on page 13.
- [43] Hopster. Coding safari. Available at <https://www.hopster.tv/coding-safari/>. Accessed 2019-02-24. Cited on page 13.
- [44] SpriteBox LLC. Spritebox. Available at <http://spritebox.com/hour.html>. Accessed 2019-02-24. Cited on page 13.
- [45] codeSpark. codespark academy: Kids coding. Available at <https://codespark.com>. Accessed 2019-02-24. Cited on page 13.
- [46] H. Suzuki and H Kato. Algoblock: a tangible programming language, a tool for collaborative learning. In *Proceedings of the 4th European logo conference*, pages 297–393, 1993. Cited on page 15.
- [47] P. Wyeth and H. Purchase. Designing technology for children: moving from the computer into the physical world with electronic blocks. *Information Technology in Childhood Education Annual*, 2002(1):219–244, 2002. Cited on page 16.
- [48] Modular Robotics. Cubelets. Available at <https://www.modrobotics.com/cubelets/>, 2012. Accessed 2019-02-24. Cited on page 16.
- [49] Nikolaus Correll, Chris Wailes, and Scott Slaby. A one-hour curriculum to engage middle school students in robotics and computer science using cubelets. In M. Ani Hsieh and Gregory Chirikjian, editors, *Distributed Autonomous Robotic Systems*, pages 165–176, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. Cited on page 16.
- [50] B. Wohl, B. Porter, and S. Clinch. Teaching computer science to 5-7 year-olds: An initial study with scratch, cubelets and unplugged computing. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, WiPSCE '15, pages 55–60, New York, NY, USA, 2015. ACM. Cited on page 16.
- [51] Fisher Price. Think & learn code-a-pillar. Available at <https://fisher-price.mattel.com/shop/en-us/fp/think-learn/think-learn-code-a-pillar-dkt39>. Accessed 2019-02-24. Cited on page 16.
- [52] KUBO Robotics. KUBO. Available at <https://kubo-robot.com>, 2017. Accessed 2019-02-24. Cited on pages 17 and 22.
- [53] D. Wang, C. Zhang, and H. Wang. T-maze: A tangible programming tool for children. In *Proceedings of the 10th International Conference on Interaction Design and Children*, IDC '11, pages 127–135, New York, NY, USA, 2011. ACM. Cited on page 18.
- [54] D. Wang, T. Wang, and Z. Liu. A tangible programming tool for children to cultivate computational thinking. *The Scientific World Journal*, 2014, 2014. Cited on page 18.
- [55] Osmo. Osmo coding family. Available at <https://www.playosmo.com/en/coding-family/>. Accessed 2019-02-24. Cited on page 19.
- [56] S. Goyal, R. S. Vijay, C. Monga, and P. Kalita. Code bits: An inexpensive tangible computational thinking toolkit for k-12 curriculum. In *Proceedings of the TEI '16: Tenth International Conference on Tangible, Embedded, and Embodied Interaction*, TEI '16, pages 441–447, New York, NY, USA, 2016. ACM. Cited on page 20.

## REFERENCES

- [57] Lego. Mindstorms. Available at <https://www.lego.com/en-us/mindstorms>, 2013. Accessed 2019-02-24. Cited on page 20.
- [58] Anki. Cozmo. Available at <https://www.anki.com/en-us/cozmo>, 2013. Accessed 2019-02-24. Cited on page 20.
- [59] A. Sullivan, M. U. Bers, and C. Mihm. Imagining, playing, & coding with kibo: Using kibo robotics to foster computational thinking in young children. *Proceedings of the International Conference on Computational Thinking Education*, 2017. Cited on page 23.
- [60] KinderLab Robotics. KIBO. Available at <http://kinderlabrobotics.com/kibo/>, 2018. Accessed 2019-02-24. Cited on page 23.
- [61] Makeblock. mBot. Available at <http://store.makeblock.com/product/mbot-robot-kit>, 2013. Accessed 2019-02-24. Cited on page 23.
- [62] Lego. Boost. Available at <https://www.lego.com/en-us/boost>, 2017. Accessed 2019-02-24. Cited on page 23.
- [63] F. Riedo, M. Chevalier, S. Magnenat, and F. Mondada. Thymio ii, a robot that grows wiser with children. In *2013 IEEE Workshop on Advanced Robotics and its Social Impacts*, pages 187–193, November 2013. Cited on page 24.
- [64] F. Mondada, M. Bonani, F. Riedo, M. Briod, L. Pereyre, P. Retornaz, and S. Magnenat. Bringing robotics to formal education: The thymio open-source hardware robot. *IEEE Robotics Automation Magazine*, 24(1):77–85, March 2017. Cited on page 24.
- [65] Thymio. Thymio. Available at <https://www.thymio.org/home-en:home>. Accessed 2019-02-24. Cited on page 24.
- [66] Wonder Workshop. Dash. Available at <https://www.makewonder.com/dash>, 2017. Accessed 2019-02-24. Cited on page 24.
- [67] Ozobot & Evollve. Ozobot. Available at <https://ozobot.com>, 2013. Accessed 2019-02-24. Cited on page 24.
- [68] Sphero. Sphero. Available at <https://www.sphero.com/sphero>, 2017. Accessed 2019-02-24. Cited on page 25.
- [69] F. Bellas, M. Naya, G. Varela, L. Llamas, A. Prieto, J. C. Becerra, M. Bautista, A. Faiña, and R. Duro. The Robobo project: Bringing educational robotics closer to real-world applications. In W. Lopuschitz, M. Merdan, G. Koppensteiner, R. Balogh, and D. Obdržálek, editors, *Robotics in Education*, pages 226–237, Cham, 2018. Springer International Publishing. Cited on page 25.
- [70] Manufactura de Ingenios Tecnológicos. Robobo. Available at <http://www.theroboboproject.com>, 2017. Accessed 2019-02-24. Cited on page 25.
- [71] Python Software Foundation. Python Documentation - Functional Programming HOWTO. Available at <https://docs.python.org/3.7/howto/functional.html?highlight=paradigm>, 2019. Accessed 2019-02-24. Cited on page 38.
- [72] I. Čukić. *Functional Programming in C++*. Manning Publications, 2018. Cited on page 38.

## REFERENCES

- [73] M. Fiala. Artag, a fiducial marker system using digital techniques. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 590–596 vol. 2, June 2005. Cited on page 39.
- [74] E. Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*, pages 3400–3407, May 2011. Cited on page 39.
- [75] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco Madrid-Cuevas, and Rafael Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition*, 51, 10 2015. Cited on page 39.
- [76] Francisco Romero Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded up detection of squared fiducial markers. *Image and Vision Computing*, 76, 06 2018. Cited on page 39.
- [77] OpenCV Team. OpenCV. Available at <https://opencv.org>, 2018. Accessed 2019-02-24. Cited on page 39.
- [78] Ionic. Ionic Framework. Available at <https://ionicframework.com>, 2019. Accessed 2019-02-24. Cited on page 54.
- [79] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. Cited on page 56.
- [80] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. Cited on page 58.
- [81] Edsger Dijkstra. Algol 60 translation: An algol 60 translator for the x1 and making a translator for algol 60. January 1961. Cited on page 61.
- [82] A. Cardoso, A. Sousa, and H. Ferreira. Programming for young children using tangible tiles and camera-enabled handheld devices. In *ICERI2018 Proceedings*, 11th annual International Conference of Education, Research and Innovation, pages 6389–6394. IATED, 12-14 November, 2018 2018. Cited on pages 76 and 91.
- [83] A. Cardoso, A. Sousa, and H. Ferreira. Easy robotics with camera devices and tangible tiles. In *ICERI2018 Proceedings*, 11th annual International Conference of Education, Research and Innovation, pages 6400–6406. IATED, 12-14 November, 2018 2018. Cited on pages 76 and 91.




# Appendix A

## Tactode Pieces

This appendix presents all the currently available Tactode pieces.


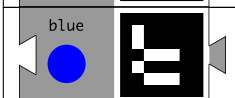
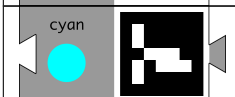
### A.1 Numbers

Table A.1: Tactode number pieces


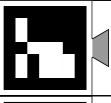

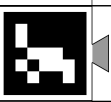




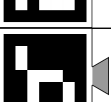
Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	0	number 0	yes	yes	yes	yes	yes	yes
	1	number 1	yes	yes	yes	yes	yes	yes
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	9	number 9	yes	yes	yes	yes	yes	yes

### A.2 Colors

Table A.2: Tactode color pieces






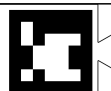





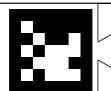
Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	50	color black	yes	yes	yes	no	yes	no
	51	color blue	yes	yes	yes	yes	yes	no
	52	color cyan	yes	yes	yes	no	yes	no

## Tactode Pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
green  	53	color green	yes	yes	yes	yes	yes	no
magenta  	54	color magenta	yes	yes	yes	no	yes	no
orange  	55	color orange	yes	yes	yes	no	yes	no
red  	56	color red	yes	yes	yes	yes	yes	no
white  	57	color white	yes	yes	yes	no	yes	no
yellow  	58	color yellow	yes	yes	yes	no	yes	no

## A.3 Letters

Table A.3: Tactode letter pieces

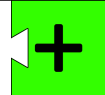
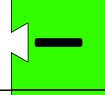

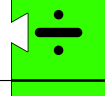
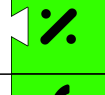
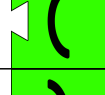
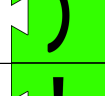
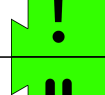
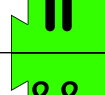
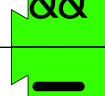



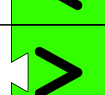
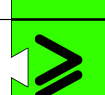
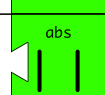

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
 	100	letter A	yes	yes	yes	yes	yes	yes
 	101	letter B	yes	yes	yes	yes	yes	yes
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
 	125	letter Z	yes	yes	yes	yes	yes	yes
 	132	letter a	yes	yes	yes	yes	yes	yes
 	132	letter b	yes	yes	yes	yes	yes	yes
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
 	157	letter z	yes	yes	yes	yes	yes	yes

## A.4 Operators



## Tactode Pieces

Table A.4: Tactode operator pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	200	addition operator	yes	yes	yes	yes	yes	yes
	201	subtraction operator	yes	yes	yes	yes	yes	yes
	202	multiplication op.	yes	yes	yes	yes	yes	yes
	203	division operator	yes	yes	yes	yes	yes	yes
	204	remainder operator	yes	yes	yes	yes	yes	yes
	205	left parenthesis	yes	yes	yes	yes	yes	yes
	206	right parenthesis	yes	yes	yes	yes	yes	yes
	207	negation operator	yes	yes	no	yes	yes	yes
	208	disjunction operator	yes	yes	yes	yes	yes	yes
	209	conjunction operator	yes	yes	yes	yes	yes	yes
	210	equality operator	yes	yes	yes	yes	yes	yes
	211	inequality operator	yes	no	yes	no	no	yes
	212	less than operator	yes	yes	yes	yes	yes	yes
	213	less than or eq. op.	yes	no	yes	no	no	yes
	214	greater than operator	yes	yes	yes	yes	yes	yes
	215	greater than or eq. op.	yes	no	yes	no	no	yes
	218	absolute value op.	yes	yes	yes	yes	yes	yes

## Tactode Pieces

### A.5 Variables

Table A.5: Tactode variable pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	300	variable 300	yes	yes	yes	yes	yes	yes
	301	variable 301	yes	yes	yes	yes	yes	yes
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	394	variable 394	yes	yes	yes	yes	yes	yes
	399	create variable						
	398	a variable, 300-394	yes	yes	yes	yes	yes	yes
	397	variable name, string						
	396	set variable value						
	398	a variable, 300-394	yes	yes	yes	yes	yes	yes
	395	value, numerical exp.						

### A.6 Events





Table A.6: Tactode event pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	400	when flag clicked event	no	yes	yes	yes	yes	no

### A.7 Control




## Tactode Pieces

Table A.7: Tactode control pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	502	repeat a number of times	yes	yes	yes	yes	yes	yes
	503	end of repeat	yes	yes	yes	yes	yes	yes
	504	repeat forever	yes	yes	yes	yes	yes	yes
	505	end of forever	yes	yes	yes	yes	yes	yes
	506	execute if cond. true	yes	yes	yes	yes	yes	yes
	507	execute if cond. false	yes	yes	yes	yes	yes	yes
	508	end of if	yes	yes	yes	yes	yes	yes
	509	repeat while cond. true	yes	yes	yes	yes	yes	yes
	510	end of while	yes	yes	yes	yes	yes	yes
	511	break out of loop	yes	no	no	no	no	yes

## A.8 Sensors

Table A.8: Tactode sensor pieces

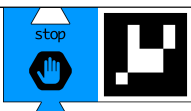
Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	600	front left prox. sensor	yes	no	no	yes	no	no
	601	front right prox. sensor	yes	no	no	yes	no	no
	602	back left prox. sensor	yes	no	no	yes	no	no

## Tactode Pieces


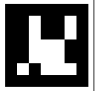





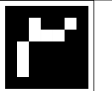





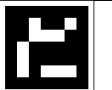





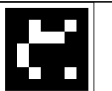





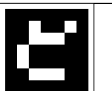




Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	603	back right prox. sensor	yes	no	no	yes	no	no
	604	front far left prox.	no	no	no	yes	no	no
	605	front far right prox.	no	no	no	yes	no	no
	606	front center prox.	no	no	no	yes	no	no
	607	back center prox.	no	no	no	yes	no	no
	612	is there line on left	yes	no	no	no	no	no
	613	is there line on right	yes	no	no	no	no	no
	614	is there line forward	yes	no	no	no	no	no
	615	is this the line end	yes	no	no	no	no	no
	616	get color of line	yes	no	no	no	no	no
	618	ask a question, string	no	no	no	yes	yes	yes
	619	user answer to quest.	no	no	no	yes	yes	yes

## A.9 Movement





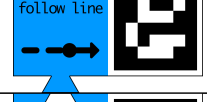

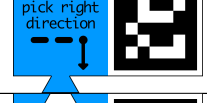
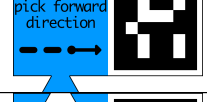
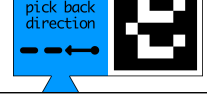
Table A.9: Tactode movement pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	700	stop wheels	yes	yes	yes	yes	no	no

## Tactode Pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
 	701	set wheel speed						
 	702	left wheel, num. exp.	yes	yes	yes	yes	no	no
 	703	right wheel, num. exp.						
 	704	move forward						
 	708	distance, num. exp.	yes	yes	yes	yes	no	no
 	710	speed, num. exp.						
 	705	move backward						
 	708	distance, num. exp.	yes	yes	yes	yes	no	no
 	710	speed, num. exp.						
 	706	turn left						
 	709	angle, num. exp.	yes	yes	yes	yes	no	no
 	710	speed, num. exp.						
 	707	turn right						
 	709	angle, num. exp.	yes	yes	yes	yes	no	no
 	710	speed, num. exp.						



## Tactode Pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
 	704	move forward						
 	705	move backward						
 	706	turn left						
 	707	turn right						
	711	follow line	yes	no	no	no	no	no
	712	pick left direction	yes	no	no	no	no	no
	713	pick right direction	yes	no	no	no	no	no
	714	pick forward direction	yes	no	no	no	no	no
	715	pick back direction	yes	no	no	no	no	no

## A.10 Sound



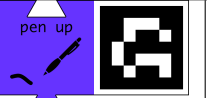
## Tactode Pieces

Table A.10: Tactode sound pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	800	say a phrase, string	yes	yes	yes	yes	yes	yes
	801	think a phrase, string	no	no	no	no	yes	no

## A.11 Visual

Table A.11: Tactode visual pieces

Piece	Tag	Function	Ozobot	Cozmo	Sphero	Robobo	Scratch	Python
	900	erase screen content	no	no	no	no	yes	no
	901	put pen down to write	no	no	no	no	yes	no
	902	lift pen to stop writing	no	no	no	no	yes	no

## Tactode Pieces



## Appendix B

# Questionnaires for the experiments

This appendix contains the questionnaires answered by the students and monitors in the experiments described in Chapter 6.

### B.1 First and Second Experiment

The students in the first experiment answered the following questionnaire:

1. Did you understand the objectives of the activity?  
not at all ①            ②            ③            ④            ⑤ completely
  
2. Did you find the `Tactode` language easy to use?  
not at all ①            ②            ③            ④            ⑤ completely
  
3. Did you find the `Tactode` application easy to use?  
not at all ①            ②            ③            ④            ⑤ completely
  
4. Which language did you consider easier?  
 `Tactode`  
 `Ozoblockly`
  
5. Which language did you prefer to use?  
 `Tactode`  
 `Ozoblockly`

## B.2 Student Questionnaire for the Third Experiment

The students in the third experiment answered the following questionnaire:

1. Which robot did you use?

- Cozmo
- Ozobot
- Sphero

2. Did you enjoy this activity?

not at all ①      ②      ③      ④      ⑤ completely

3. Did you understand the objectives of the activity?

not at all ①      ②      ③      ④      ⑤ completely

4. Did you understand how the robot works?

not at all ①      ②      ③      ④      ⑤ completely

5. Did you understand what each `Tactode` piece you used does?

not at all ①      ②      ③      ④      ⑤ completely

6. Did you think that programming the robot with the `Tactode` pieces was easy?

not at all ①      ②      ③      ④      ⑤ completely

7. Did you find the application easy to use?

not at all ①      ②      ③      ④      ⑤ completely

8. Did you find the robot easy to use?

not at all ①      ②      ③      ④      ⑤ completely

9. Did you find transferring the code from the `Tactode` application to the robot easy?

not at all ①      ②      ③      ④      ⑤ completely

## B.3 Monitor Questionnaire for the Third Experiment

The monitors in the third experiment answered the following questionnaire:

1. Material:

(a) Which robot did the group use?

## Questionnaires for the experiments

- Cozmo
- Ozobot
- Sphero

(b) Which device did the group use to run the `Tactode` application?

- Lenovo tablet
- Apple tablet
- Apple smartphone
- Apple computer

2. For each child:

- (a) Name: \_\_\_\_\_
- (b) Age: \_\_\_\_\_
- (c) School year: \_\_\_\_\_
- (d) Previous programming experience: \_\_\_\_\_
- (e) Previous experience with robots: \_\_\_\_\_
- (f) Percentage of participation: \_\_\_\_\_
- (g) Motivation  
not at all ①      ②      ③      ④      ⑤ completely
- (h) Enjoyment  
not at all ①      ②      ③      ④      ⑤ completely

3. For each task between triangle, square (using `repeat` block), pentagon and hexagon (using `divide` block):

- (a) Completion times: \_\_\_\_\_
- (b) Programming difficulties: \_\_\_\_\_
- (c) Difficulties in taking pictures/number of pictures taken: \_\_\_\_\_
- (d) Difficulties in using the application: \_\_\_\_\_
- (e) Difficulties getting the code into the robot: \_\_\_\_\_
- (f) Number of attempts: \_\_\_\_\_
- (g) Programming errors:
  - i. Number and type of errors: \_\_\_\_\_
  - ii. Level of debugging
    - there are no errors
    - children do not notice that there are errors

## Questionnaires for the experiments

- children notice the errors
  - children are able to determine error location
  - children know which errors they have committed
  - children fix the errors
- iii. Debugging moments
- when programming
  - when compiling
  - when executing