Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Elettronica, Informazione e Bioingegneria

Design and implementation of Robo^3: an applied game for teaching introductory programming

Advisor:
Ing. Daniele Loiacono

Author:
Filippo Agalbato
850481

Academic Year 2016–2017

Abstract

This work discusses the design and implementation of Robo^3, a web game to teach programming skills, to be used as a supplement during introductory programming courses. The game helps students to visualize and understand the effects of the code they write; at the same time it allows the instructor to easily author levels on the topic of their choice and to gather data on players' performance, which can be visualized in aggregate form by a companion dashboard environment.

Sommario

Questa tesi discute la progettazione e l'implementazione di Robo^3, un gioco web volto a insegnare a programmare, il cui scopo è di essere usato come strumento di supporto nei corsi di introduzione alla programmazione. Il gioco permette agli studenti di vedere e capire gli effetti dell'esecuzione del codice da loro scritto; allo stesso tempo, permette a chi tiene il corso di creare facilmente nuovi livelli sugli argomenti che ritiene utili e raccoglie dati sull'andamento dei giocatori, che possono poi essere esaminati in forma aggregata tramite un ambiente *dashboard* specializzato creato appositamente.

Gli ultimi anni hanno visto una crescita nell'importanza data alle competenze informatiche e di programmazione, che sempre più vengono usate da molti nella vita di tutti i giorni. In aggiunta a ciò si è assistito a un incremento nell'uso e nell'efficacia di metodi di insegnamento alternativi, come l'uso di giochi applicati all'ambito in questione, in funzione di supporto all'insegnamento tradizionale. Alla luce di questi aspetti è stato deciso di progettare e sviluppare Robo^3. Dopo uno studio attento e accurato della letteratura scientifica correlata e degli esempi commerciali di giochi sulla programmazione, si è stabilito che gli aspetti importanti di tale gioco devono essere rivolti sia ai giocatori, che possano così vedere con i propri occhi gli effetti del codice che scrivono, sia a coloro che tengono i corsi in questione, che possano facilmente creare livelli personalizzati sugli argomenti che ritengono adatti e possano esaminare i dati che il gioco raccoglie, in modo da poter monitorare l'andamento dei propri studenti sotto molti aspetti diversi.

Questa tesi discute dettagliatamente tutte le scelte di design che portano al raggiungimento di tali obiettivi progettuali, oltre a fornire una panoramica sul gioco in sé e sull'ambiente *dashboard* associato.

Acknowledgements

My greatest thanks go to professor Loiacono, for his calm and kindness, for his helpfulness when I was in need, for proposing me a thesis subject which I especially liked and for making working with him on this thesis enjoyable and a great experience under every possible point of view. None of these I will ever forget.

I would also like to thank all the friends made along the way, for they are the real treasure, my family, for always feeding me so that I did not die, my legs, for always supporting me, and my arms, for always being at my side.

Contents

List of Figures

Chapter 1

Introduction

1.1 General scope

In recent years there has been a rise in the awareness given to the importance of learning and developing skills related to programming and, more generally, to analytical and computational thinking. In a world where technology is pervasive and computers are used in many tasks during both life and work, the benefits related to knowing how to think, design and write a piece of code grow year by year. There have been awareness campaigns and efforts to bring the topic of computer science and programming to school curricula, such as the week-long 2013 *Hour of Code* initiative, promoted by the *Code.org* foundation in the United States.

The use of serious or applied games for educational and training purpose is also becoming pervasive and it has often shown to be successful (Rajaravivarma, 2005), (Drake and Sung, 2011), (Lee, 2013). While not able to fully substitute traditional teaching strategies, this approach has proven valuable as a supplementary learning method, allowing students to see and apply what they learn in the classroom under a different light, from a different perspective, or just plainly in a more relaxed and fun setting, all of which contributes to the alternative learning experience. Coupling the two approaches can thus provide many benefits.

1.2 Purpose of this work

The aim of this thesis is to discuss the design and the development of Robo^3, a web game to supplement the teaching of introductory programming courses, designed after a careful study of similar examples from scientific literature and related games from the commercial world. Key points of Robo^3 include:

- allowing students to visualize the consequences of the code they write in the game world, providing a concrete representation for abstract concepts, strengthening understanding and retention;
- the possibility for the instructor to quickly and easily author new levels, so that these may always be strictly related to the topic they wish to focus on;
- the gathering of data related to how players fare while playing, such as which levels they were able to complete and metrics related to their solutions, coupled with an online dashboard to aggregate and show these data through appropriate graphical representations, providing the instructor with up-to-date feedback.

Chapter 2

State of the art

This chapter provides an overview of games from the scientific literature that have been developed with the express purpose of teaching programming skills and computational thinking abilities. Then, the most important commercial games available today that cover the same topic are presented. Finally, some remarks and a brief analysis of the games presented is provided as a guideline for the design choices in the reminder of the thesis.

2.1 Why to use games?

Using games as a reinforcing and exercising tool for teaching programming skills and analytical thinking can be very beneficial for students. In fact, games allow to better visualize concepts that might otherwise be perceived as too abstract. They also help to familiarize with knowledge and methods that would maybe be tedious to study in the standard fashion, offering a cycle of challenges and rewards that propels the learning experience forwards with less perceived effort and fatigue.

These benefits are extensively discussed in the literature: Lee (2013), Lee, Ko and Kwan (2013) and Lee and Ko (2011) discuss how using game approaches, like presenting programming tools as human-like characters, may improve a student's retention, interest and speed; Theodoropoulos, Antoniou and Lepouras (2016) present the results of a study on 70 Greek high-schoolers that correlates the different cognitive styles of the students with how successfully they learn from serious games for teaching programming.

Rajaravivarma (2005) studied how notorious word and number games (such as Jeopardy or Hangman) can be of value for teaching skills related to computer science and computational thinking, and similarly Drake and Sung (2011) discuss the use of several board games to introduce and demonstrate concepts related to programming such as data structures.

While not properly a game, and thus less related to this analysis, *Scratch*[1] exemplifies very well how using games for teaching can be beneficial. Developed by the MIT Media Lab, it is a visual programming language associated with a browser-accessed environment that allows to create interactive stories, animations and small games. Such a game-like approach is much more captivating than the traditional way of studying for an introduction to programming topics.

Another approach that successfully employs game-like methods is the one carried on by the *CS Unplugged* project[2], a collection of free learning activities that teach computer science topics through games and puzzles that use cards, string, crayons and physical activity, aimed at younger students. It was explicitly designed to allow to learn computational abilities without any prior knowledge in programming.

2.2 Programming games in the literature

By examining the games proposed in the scientific literature it is possible to distinguish two general trends. The first is to design games whose aim is to help and aid programming skills by focusing on the act of writing code, on the syntax of the programming languages, on the written language itself, allowing students to familiarize themselves with the actions they will carry out while programming in the real world and maybe to feel less intimidated by the act. The second trend is to design games that take a more abstract point of view on the problem, often disregarding the act of writing code and focusing on the skills related to problem-solving and analytical thinking. Both trends are analyzed and discussed in the following subsections.

2.2.1 Games for writing code

Many of the games created and analyzed in the literature focus on the syntax of programming languages and the act of writing code. A certain number of authors have produced games that are focused on teaching one single programming language. Mitamura, Suzuki and Oohori (2012) present a showcase of four games for learning programming concepts: the first three are very narrowly focused on learning the syntax of Java, while the fourth has a much broader scope and consists in moving a character in several top-down levels by entering a small program detailing its movements. Eagle and Barnes

[1]`https://scratch.mit.edu`

[2]`csunplugged.org`

(2009) present the game *Wu's Castle*, made with *RPG Maker*, where players visually interact with loops and similar structures and may change them interactively, following a C-like syntax. Anderson and McLoughlin (2007) present a 3D game-environment where a sheep is controlled by writing code in C-Sheep (a subset of ANSI C) or via graphical programming. Paliokas, Arapidis and Mpimpitsos (2011) present a game employing the *LOGO* language, where one controls a 3D spaceship by entering commands and seeing the resulting behavior. Jordine, Liang and Ihler (2014) present a mobile game for learning Java. Sierra et al. (2016) present a mobile RPG-like game where one controls a character that explores a world through programming in Java.

As it can be seen, there are many examples of proposals of serious games for teaching specific programming languages. Other authors advocate for more universal approaches. Thus Serrano-Laguna et al. (2015) state that, even if using games to teach programming has been successfully used in many cases in the literature, all games so far proposed are narrow in their scope. Since, however, it is stated that writing code is an essential part in learning how to program, they advocate for the construction of a general game engine able to handle any possible language and different kinds of games with minimal configuration, where the games consist of typing small programs to solve each level. They provide one example of a game implemented with such engine and two case studies to test its effectiveness.

While certainly effective, this *syntactic* approach has been deemed too focused on the act of writing code and not enough on the broader and more abstract analytical thinking abilities required for programming.

Related to this specific point is the study by MacLaurin (2011) on *Kodu*, a 3D game where players control robots by writing rules for triggered actions. Stolee and Fristoe (2011) have analyzed 346 user-made *Kodu* programs to examine which programming concepts can be learned. While the rules that one writes in *Kodu* are still textual in nature, its approach is a step towards teaching more abstract concepts without leaving them implicit in the act of writing code.

2.2.2 Games for teaching concepts

There are other proposals of games less focused on teaching syntax and how to write code and more in teaching programming concepts. Masso and Grace (2011) present *Shapemaker*, an augmented reality card game (played on a *table-based tactile interface*) where the core concepts of programming are learned while explicitly avoiding the added complexity of having to learn burdensome syntax and typing code: while players are still lead to familiarize with syntax, this is achieved without the effort that using a real language would entail. Schmitz et al. (2011) discuss a browser game resting atop a back-end made by several online services, made to train its player in *IT skills*. Sajana, Bijlani and Jayakrishnan (2015) present a game made of several minigames, each dedicated to a different aspect of programming (data structures, control flow and so on). Horn et al. (2016) analyze the use of *GrACE*, a game for middle school children tailored at teaching not *the craft of programming* but *abstraction and computational thinking* through the solution of minimum spanning tree procedural problems. Kazimoglu et al. (2012) present the the game *Program your robot*, specifically developed to teach computational thinking.

2.2.3 Surveys in the literature

There have been some surveys in the literature on the topic of games for teaching concepts related to programming and computer science. Gibson and Bell (2013) conducted a survey on games to teach computer science topics in general, concluding that the coverage of topics is very patchy and incomplete. More to the point, Vahldick, Mendes and Marcelino (2014) produced a study presenting a classification and examination of the skills required by a programmer that can be trained by appropriate games and a survey of those games that exist in the literature. As conclusions, it is noted that those games do not adapt themselves to different skill levels as the student learns and grows, do not allow more than one language and provide no feedback during the gameplay experience, only after it has ended.

2.3 Commercial programming games

While the genre is not vast or popular by any means, there are several commercial programming games. Their aim is not to teach, but to challenge the player with programming puzzles: as a result, while they can be valuable in teaching, what they offer is a series of challenges, often very difficult, that usually require good programming skills to be solved in the first place instead of teaching them.

Lightbot[3] (Danny Yaroslavski, 2008) is a game where players control a robot that can move around a chessboard and activate specific tiles on the ground, lighting them; the player can control the robot by writing a program with symbolic instructions (see Figure 2.1). Programs are limited in size and consist of one main and two secondary parameterless functions, allowing for

[3]`http://lightbot.com`

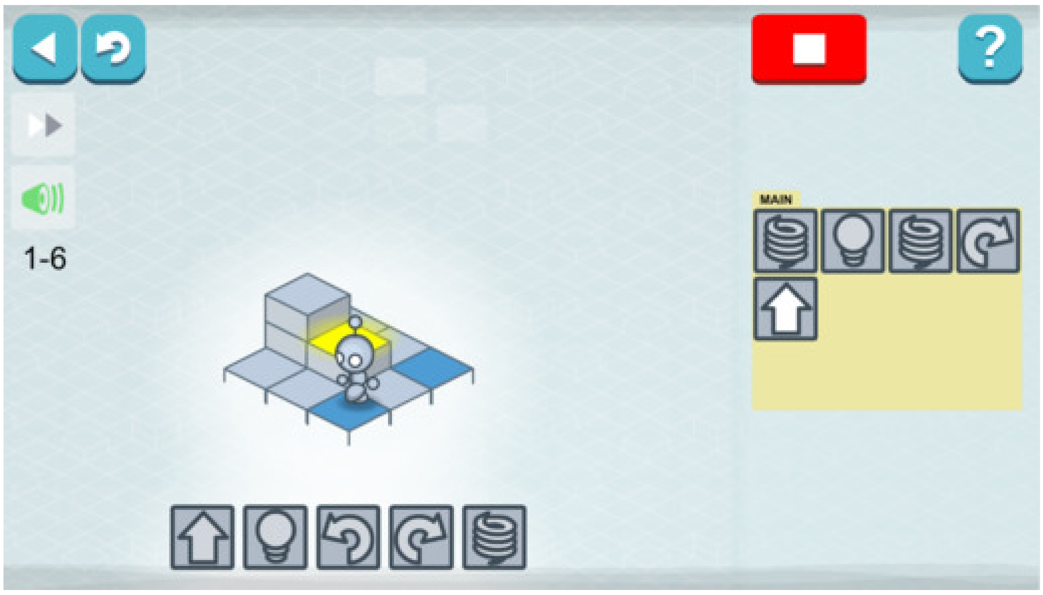


Figure 2.1: *Lightbot*

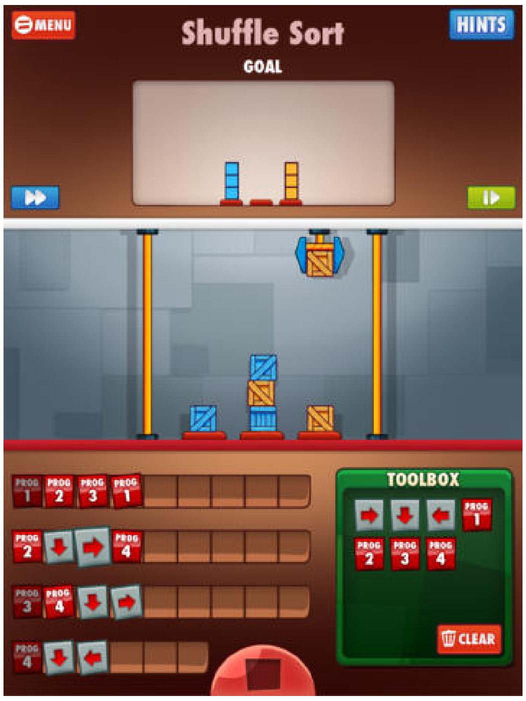
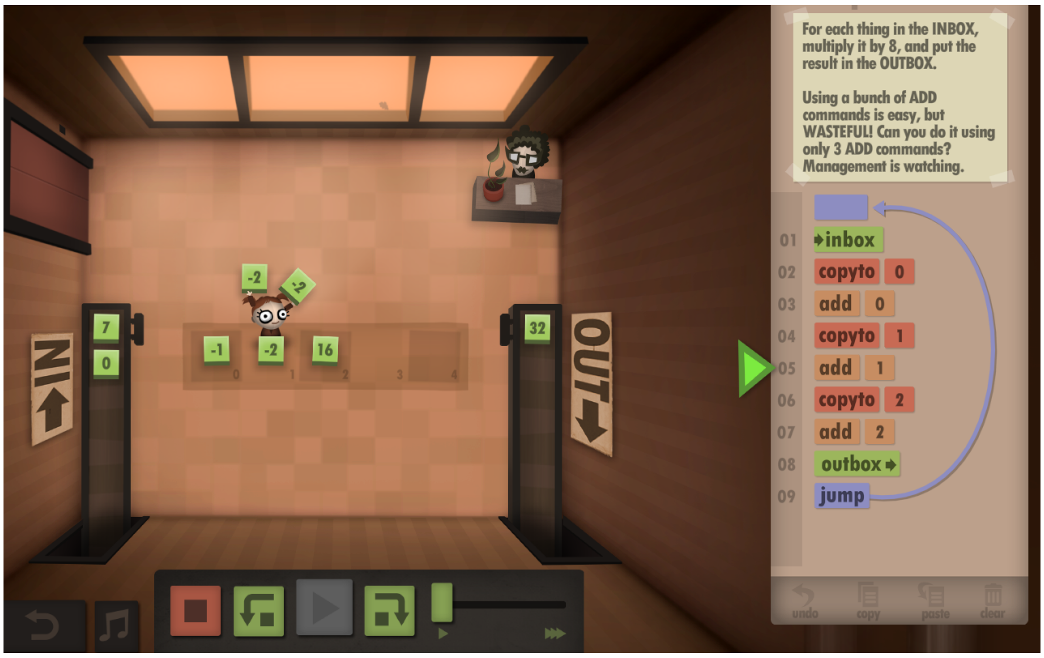


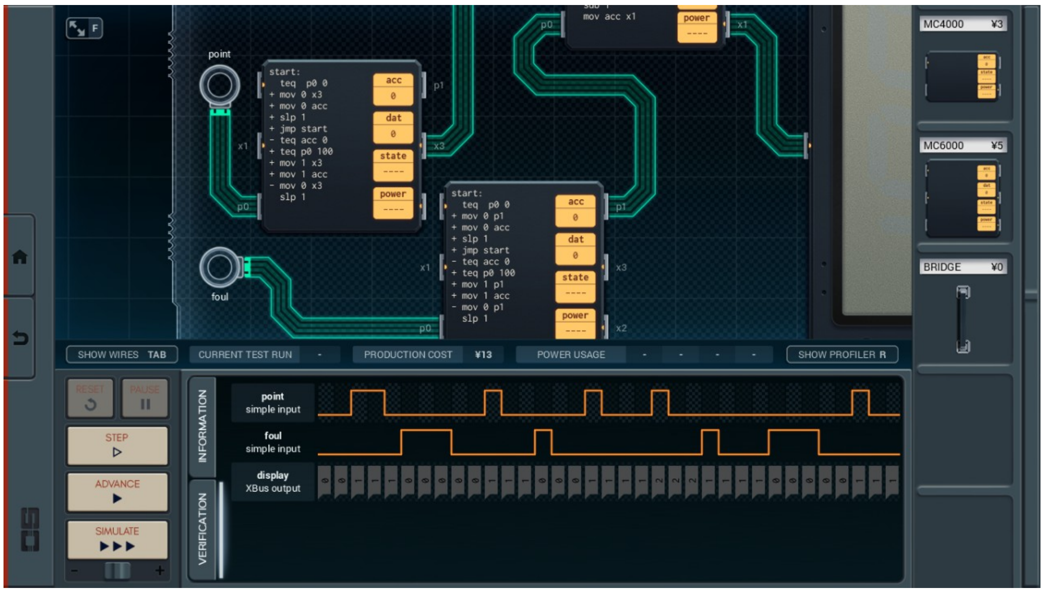
Figure 2.2: *Cargo-Bot*


For each thing in the INBOX, multiply it by 8, and put the result in the OUTBOX.

Using a bunch of ADD commands is easy, but WASTEFUL! Can you do it using only 3 ADD commands? Management is watching.

```
01 inbox
02 copyto 0
03 add 0
04 copyto 1
05 add 1
06 copyto 2
07 add 2
08 outbox
09 jump
```

Figure 2.3: *Human Resource Machine*


```
start:
  teq  p0 0
+ mov 0 x3
+ mov 0 acc
+ slp 1
+ jmp start
- teq acc 0
+ teq p0 100
+ mov 1 x3
+ mov 1 acc
- mov 0 x3
  slp 1
```

```
start:
  teq  p0 0
+ mov 0 p1
+ mov 0 acc
+ slp 1
+ jmp start
- teq acc 0
+ teq p0 100
+ mov 1 p1
+ mov 1 acc
- mov 0 p1
  slp 1
```

```
mov acc x1
```

Figure 2.4: *Shenzhen I/O*

recursive calls, which are actually the only way to obtain loop-like control flows; the game also includes a `break` instruction in some of the different versions released. Each instruction can receive a colored conditional flag, meaning that it is executed only if the robot is standing on a tile of the same color of the conditional flag. There is no form of input/output and no state, except for the position of the robot on the chessboard and the lighting of tiles. The programming model is thus very abstract and very much akin to functional programming. *Lightbot* was analyzed by Gouws, Bradshaw and Wentworth (2013), who developed a framework for evaluating the effectiveness of computational thinking material and used this game as a case study, concluding it performs very well according to their metrics. It was also re-implemented for the *Hour of code* initiative promoted by *Code.org*[4]. This new implementation maintains most of the core mechanics but overhauls both the user interface and the level design, becoming more of a teaching game and less of a simple puzzle game.

Cargo-Bot[5] (Two Lives Left, 2012) is a mobile (iOS) game where the player controls a crane to shuffle colored boxes around, by writing a program with symbolic instructions (see Figure 2.2). The programs are very limited in size, but the game offers up to four parameterless functions, allowing for recursion much in the same way *Lightbot* does. Colored conditions may be applied on single symbols to activate them only when the crane holds a box of the same color or holds nothing (marked symbols get otherwise skipped). The crane has very limited movement, and program state is represented by its position and by the disposition of boxes on different platforms. It can be argued that boxes act as variables of sorts, but only their position is mutable, not their color. *Cargo-Bot* was deemed useful for teaching recursion by Tessler, Beth and Lin (2013), who discuss the effort made to re-implement it for browsers (with permission from the developers). This study also briefly analyzes *Lightbot* as another similar game.

Human Resources Machine[6] (Tomorrow Corporation, 2015) is a multi-platform game where one instructs an office employee on how to move letters and numbers from an input conveyor belt to an output conveyor belt (see Figure 2.3) by writing a program in an assembly-like language where each instruction is a symbol-like block (no typing is needed: whole instructions are selected from a menu). Being based on simple assembly, it lacks any form of function call, but it offers low-level conditional and unconditional jump instructions. Input is read and removed from an automatically-refilled register,

[4]`code.org`
[5]`https://twolivesleft.com/cargobot`
[6]`https://tomorrowcorporation.com/humanresourcemachine`

and output is deposited in a similarly self-emptying register. There are also several work registers available, with operations to store and retrieve, and even a dedicated pair of instructions to treat the work registers as an indexed array. Values may be both integers and characters. Allowed operations are addition and subtraction.

Spacechem[7], *Infinifactory*[8], *TIS-100*[9] and *Shenzhen I/O*[10] are all made by the same producer, *Zachtronics*, and thus present strong similarities. All are released for computers. The strongest similarity among these games is that they all rate the player's solutions based on three metrics: time (simulation cycles elapsed), program size (usually symbol count) and execution memory footprint (varies with the game); then compare it against the solutions of other players, so that one can instantly receive, after a level ends, a feedback on how well their solution was compared to other players. Other similarities are smaller and less interesting and relate to game design, such as with the execution modules of *TIS-100* and *Shenzhen I/O*.

Spacechem (2011) challenges the player with designing the sequence of operations to manipulate atoms and molecules in a science-fiction-like industrial setting. Overall, it is more related to industrial automation than to pure programming. Abstracting from the industrial presentation, code is represented by symbols on a 2D surface, connected by a colored line, executed when an actuator that travels the screen along the the line encounters them. Lines must loop by design to create repetitive execution patterns. Conditions are obtained through symbols that may change the direction of movement for the actuator. Input structures must be fetched from an input area, transformed to output structures and transferred to an output area to win. State is represented by the position of atoms on the screen. Operations on data is through symbolic manipulation instructions such as binding or separating atoms in molecules.

Infinifactory (2015) maintains the industrial automation setting, this time in 3D. The player must construct an industrial pipeline that assembles simple blocks in complex structures by combining very simple block-like machines. Similarly to *Spacechem*, code is represented by the disposition of the building blocks that make up the factory, such as conveyor belts, pistons and welders. Input structures must be fetched from an input area, transformed to output structures and transferred to an output area to win. State is represented by the position of the blocks in the factory. Operations on data is through symbolic manipulation machines, such as welders to join two blocks, cutters

[7]`http://www.zachtronics.com/spacechem`
[8]`http://www.zachtronics.com/infinifactory`
[9]`http://www.zachtronics.com/tis-100`
[10]`http://www.zachtronics.com/shenzhen-io`

to separate them or other that alter their shape.

TIS-100 (2015) requires the player to repair an old and corrupted command-line computer system by solving assembly programming challenges. Code is written by typing in a written assembly-like language. Program size is limited by the dimensions of small input areas, in the form of programmable modules, but there are up to twelve such modules, each connected to its neighbors. Each can thus represent a different function, lacking the language any form of function call. The connections among modules are through registers and the execution is parallel by design, since each module executes asynchronously and blocks only when waiting in read mode on a register or once its execution terminates. Conditional and unconditional jumps allow low-level loops and conditional constructs. Several input registers automatically read from several input streams and bring in integer values. In order to complete a level, specific values must be written into one or more output registers in a specific sequence. As mentioned, program size for each module is limited to a handful of instructions, and moreover each module offers only one work register, one backup register and (up to) four input/output registers. Allowed operations are addition and subtraction.

Shenzhen I/O (2016) maintains some core concepts from *TIS-100*, being based on designing integrated circuits by assembling electronic components and small programmable CPUs to create complex behaviors (see Figure 2.4). The main differences with *TIS-100* are that the typed assembly language that the CPU components accept is richer, there are many non-programmable modules doing specific things and the connections among each piece of the puzzle must be decided by the player. Program size is limited, but the circuit allows as many CPUs and components as there is space to fit. The result is usually a complex entity made of several parallel interconnected programmable modules, each connected by the player through registers and buses to some of its neighbors. Conditional and unconditional jumps allow low-level loops constructs and there are dedicated "test" instructions and a block label syntax for implementing conditional constructs. There are several input registers to automatically read from several input streams and bring in integer values. In order to complete a level, specific values must be written into one or more output registers in a specific sequence. There are different registers according to module type, but usually one work register and some input/output ones. Operations are addition, subtraction, multiplication and other basic mathematical functions.

The programming styles present in these commercial games are either very abstract or very low-level: there is no trace of the middle ground of a modern high-level syntax. Moreover, as mentioned, they are meant as challenges rather than teaching tools: the difficulty often spikes unpredictably and the last levels are exceedingly complex, especially for someone who should learn how to program. For example, both *TIS-100* and *Human Resource Machine* have the last level that requires writing an assembly program to sort several zero-terminated sequences of integers.

2.4 Conclusions

This chapter has analyzed and discussed several proposals in the scientific literature, corroborated by some examples of commercial products, for the problem of using games to teach programming skills and analytical abilities. A comparison of the different approaches highlights some possible disadvantages for games focused on the act of writing code, which might be confusing and tricky for learners and might even carry the risk of missing the broader point, which is learning how to solve a problem in one's mind before setting out to write its implementation. Of the commercial games analyzed, *Lightbot* and *Cargo-bot* follow more closely these precepts. As discussed, however, the programming paradigm that even the commercial game follow is probably too abstract or too low level or lacking in certain programming constructs.

Chapter 3

Conceptual design

This chapter reports the design choices that have been made in the creation of ROBO^3 and contains a detailed description of all its elements, as well as a discussion of these design choices.

3.1 Overall design

3.1.1 High level design concepts

Of the two approaches present in the literature and in the commercial world, as discussed in chapter 2, the one that focuses less on the act of writing code and more on the high-level structure of the program has been chosen. While they are not in contrast, the chosen one offers probably less disadvantages for the learning student, since it abstracts away from the little details of how to write code and allows the player to focus on the problem itself, training the broader and more abstract capabilities of how to *think* and organize programs.

3.1.2 Game description

The game presents several levels, each being focused on a particular concept or aspect of programming. In each level, the player has to control a robot on a tiled surface and program it to move some colored cubes around. The program is written by putting specific symbols inside regions of the screen that represent the program itself. Each symbol is an instruction with a well-defined meaning. Once the player starts the simulation by clicking on the *Run* button, the instructions are read one at a time and used by the game to control the robot accordingly. The cubes in the level are organized in stacks, each stack standing in a certain tile. The robot can pick up the topmost cube

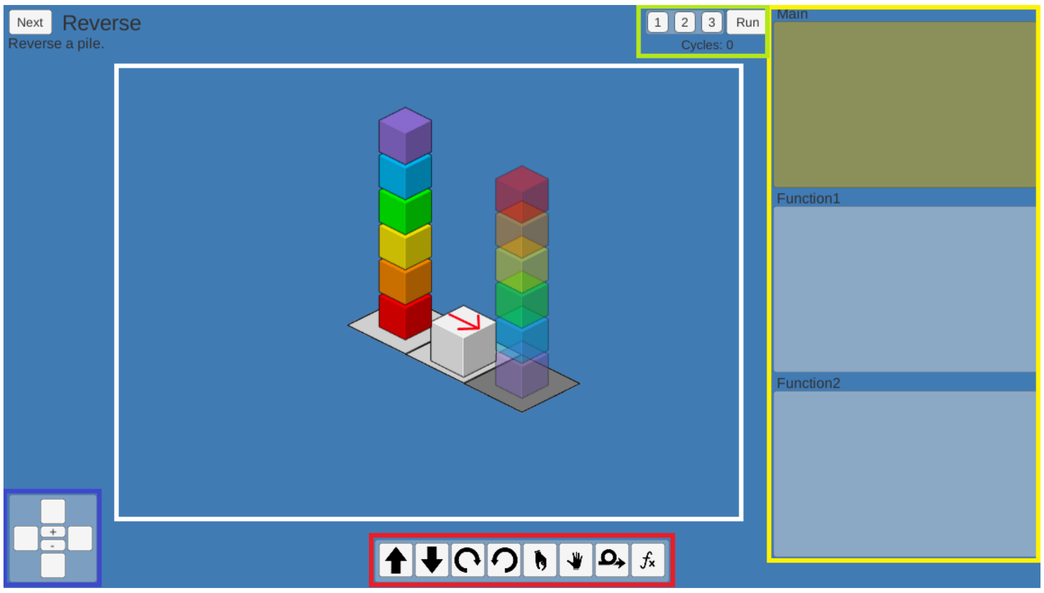


Figure 3.1: An overview of the main game screen: in white the main viewport, in yellow the function panels, in red the instruction toolbox, in green the simulation controls, in blue the camera controls

and drop it in any other tile, possibly on top of the existing stack for that tile. The cubes must be carried to certain marked tiles in certain patterns to successfully complete the level. The player must thus write the program that correctly builds the required output sequences of cubes starting from the input sequences. An overview of the game is depicted in Figure 3.1.

3.1.3 Inspiration sources

The game design draws its heaviest inspiration mainly from *Lightbot* and *Cargo-Bot*. As in *Lightbot*, the player writes a program through symbolic instructions, which control and move a robot on a tiled surface, and as in *Cargo-Bot*, the player writes a program to shuffle pre-existing stacks of colored cubes to obtain the required output sequences. One of the key design differences, though, is the inclusion of a specialized loop symbol in the instruction set, whereas both the aforementioned games only allowed loop-like structures to arise through recursion.

3.2 Game elements

3.2.1 Overview

A view of a typical game screen is shown in Figure 3.1. This is the main screen for the game and it is structured so that there is no need for the player to look at other screens during gameplay. It is divided in different parts:

- main viewport: it shows an isometric view of the game world, visualizing the whole level. The robot stands on top of a tiled floor (which may change shape depending on the level). Stacks of cubes also stand on top of the same tiled floor. The robot may move around on these tiles, pick the cubes up and drop them;

- function panels: the place where the player must insert the instruction symbols to compose its program. It offers one main function and several secondary functions;

- instructions toolbox: allows to drag the instruction symbols to the function panels;

- simulation controls: allow to start and stop the simulation and to select its speed;

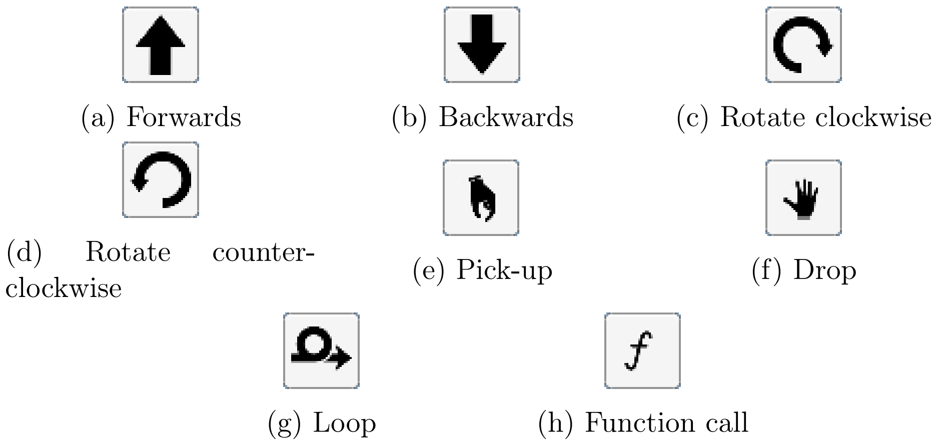
- camera controls: allow to move the viewport around the level.

3.2.2 Levels

The robot moves on a tiled floor. Each step in a certain direction moves the robot of exactly one tile. Thus, the robot is always centered on exactly one tile. Similarly, tiles can contain cubes.

Some tiles start with cubes already on them, but any tile may accept dropped cubes. So-called *validation tiles*, which are visually different from standard tiles, represent the areas where the robot must carry cubes in a certain pattern in order to complete the level. This pattern is shown by having the tile contain the ghost outline of the requested cube stack.

Levels data is separate from the rest of the game and may be freely edited by the designer without need of touching the program, allowing for easy authoring of new levels according to need.



(a) Forwards (b) Backwards (c) Rotate clockwise (d) Rotate counter-clockwise (e) Pick-up (f) Drop (g) Loop (h) Function call

Figure 3.2: Instructions and their symbols

3.2.3 Cubes

Cubes are the elements that must be moved around by the robot in order to complete a level; they can be picked up and dropped by the robot and they are organized in stacks: when picking a cube, the topmost one is picked, and when dropping a cube, it is deposited on top of the stack. There is exactly one stack of cubes per tile. Any tile can thus accept dropped cubes, and any cube can be picked up and moved elsewhere. Cubes are present in the game world and cannot be created by the player. They can however be destroyed: if the robot picks up a new cube while already holding another one, the held one is destroyed and the new one replaces it.

Cubes are colored, in six different tints: red, orange, yellow, green, blue, purple. Cubes of the same color are functionally identical for the game: if an output sequence requests a red cube, for example, any red cube will satisfy that request.

3.2.4 Instructions

There are eight different instructions with well-defined meanings, represented by symbols that get selected from a certain panel and brought over to the code panels to compose the program that the robot will follow. Symbols can be dragged and dropped and freely reordered. They are listed in Figure 3.2. Their meaning is as follows:

- *forwards*: moves the robot in the direction it is facing by exactly one tile;

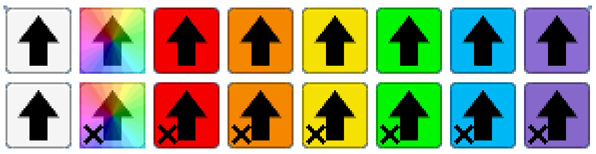


Figure 3.3: The different conditions applied to the *forwards* instruction

- *backwards*: moves the robot in the direction opposite to the one it is facing by exactly one tile;
- *rotate clockwise*: rotates the robot of 90° clockwise, making it face another direction;
- *rotate counter-clockwise*: rotates the robot of 90° counter-clockwise, making it face another direction;
- *pick-up*: picks up a cube from the top of the stack of the current tile, if there are any, and keeps it with its hands. This cube is now being carried by the robot and will follow it when it moves around. If there is already a cube being carried when picking up a new cube, the old one is destroyed;
- *drop*: drops the carried cube (if present) on top of the cube stack for the current tile;
- *loop*: jumps back in execution to a previous instruction. What instruction to jump to can be selected by the player. An arrow visually binds the loop symbol with the destination symbol. The jump is unconditional and the program never terminates if no condition is associated to this symbol (see subsection 3.2.5). It is akin to a do/while loop;
- *function call*: calls one of the other functions. Function call executes exactly as expected: calls can be recursive and once a call terminates execution continues from the next symbol.

There is one code panel dedicated to a *main* function. This is the starting point for the execution. There are also some other secondary function panels, each associated with a specific function.

3.2.5 Conditions

Conditions are applied to instruction symbols rather than being a symbol themselves. They are represented by the instruction symbol, normally black on white background, changing up to both colors. A conditioned symbol gets executed only if the condition evaluates to true, otherwise it gets skipped. Conditions perform a check on the color of the cube that is being currently held by the robot (or, in the case of the *pick-up* and *drop* instructions, on the color of the cube that is on top of the stack of the current tile). There is a condition color for each cube color (red, orange, yellow, green, blue, purple), which means that the instruction gets executed only if a cube of that specific color is present, and there is the rainbow color, which means that the instruction gets executed if any cube is present. There is also the possibility of negating a condition: negated solid colors evaluate to true if there is not a cube of that color and the negated rainbow condition evaluates to true if there are no cubes at all. See Figure 3.3 for an overview of the different conditions and their negated versions.

Conditions are essential when using the loop instruction, because the simple loop symbol represents an unconditional jump, leading to non-terminating programs.

3.2.6 Simulation

Once the player has written the program that solves the level, it must run. By clicking on the appropriate button, the player can start the simulation. While running, each instruction is sequentially read, interpreted, and its effects are passed to the robot. After a brief delay, which allows the player to see the results of each instruction, the next one is read and the cycle repeats. The currently executing instruction is highlighted in the code panels, to allow an even easier visualization of what is happening. Different selectable simulation speeds allow the player to look closely at the effects of their program on the world or to quickly skip over parts they already know.

At the end of the simulation, the validation step takes place, checking the cubes on the validation tiles against the expected sequence. Programs that do not terminate never enter the validation steps and are thus invalid. The designer can moreover choose to limit the maximum number of computation cycles allowed in a solution: programs that exceed this number are forcefully terminated and considered invalid.

3.2.7 Test cases

In order to force the player to write the code that solves the general problem instead of the specific case of a level (thus discouraging reliance on the specific disposition and order of cubes), each level supports an unlimited number of test cases. Each test case is made by a sequence of input cube stacks and a sequence of expected output cube stacks. Floor shape cannot change from one test case to the other, nor the position of the validation tiles.

What the player sees when loading a level is the first test case. Once the simulation runs, if the first test case is successful, the next one is fetched, if present, and the simulation restarts from the beginning with the new configuration of inputs and outputs, but maintaining the same program. This process continues until all test cases have been successfully exhausted or a single test case returns with a wrong result: in such a case, that single test case is shown again to the player, to highlight the problem, and the simulation terminates unsuccessfully.

By aptly configuring different test cases, the designer can obtain a level that forces the player to think more broadly in order to develop a general solution.

3.3 Discussion

The stated aim of the game is to help the teaching of programming skills and analytical thinking abilities by being paired as a reinforcement and exercising tool along with a more traditional, introductory course to computer science and programming. The game should thus cover similar topics, presenting a programming environment where the same concepts can be found. This has lead to the choice of an instruction set containing the high-level concepts of loops and function calls, as opposed to an assembly-like language that would only have presented conditional and unconditional jumps.

Following one of the two approaches discussed in chapter 2, the game avoids focusing on writing code by hand or on the syntax of existing languages, but rather provides a programming toolbox made of functional blocks, so as to abstract away from the subtleties of the act of writing code, allowing the full attention of the player to focus on the program they are creating. This is achieved by having the code being made by a composition of well-defined instruction symbols.

The game environment allows the player to see the results of the code they created. This is achieved by having the robot visibly execute all of the instructions, meaningfully modifying the state of the world (i.e. the cubes and their disposition).

Finally, the game leads to the development of critical and analytical thinking capabilities that allow to think of a solution in general terms. This is achieved by requiring several variations of the same level to be solved, through different test cases.

Chapter 4

Level design

The design of levels in ROBO^3 is inspired by the typical exercises that are presented in introductory programming courses. Such exercises can be classified according to the following thematic groupings:

- simple deterministic sequences;
- conditions;
- loops;
- array and data structure manipulation;
- functions;
- recursion;
- files and streams.

4.1 Design implications

The programming environment offered by ROBO^3 is not the same as that of a typical programming language such as C. Some preliminary considerations are due when approaching the topic of level design.

4.1.1 Colors and stacks

Cubes are colored and colors lack both an order relation and arithmetic operators, excluding most of the standard exercises about numbers, but they do allow equality checks: as a result they are very similar to characters, so exercises about string manipulation are useful as a design source.

There are moreover strong similarities between stacks of cubes in and input/output streams, such as the fact that input streams only allow non-random access to the next element in queue, much like input stacks do, or the fact that cubes can be taken and manipulated but not created, only destroyed, so exercises on data streams are another good design source.

4.1.2 Complexity

Levels of ROBO^3 that specify a fixed size for the input stacks (particular cases) are easier to solve than the ones that do not (general cases), since these latter necessarily require the use of iteration constructs and conditional checks, while for the former a player can just *unroll* the loop and repeat the necessary instructions the correct number of times. If this is not the intended behaviour, the designer should aptly configure the test cases of a level to avoid fixed-sized input stacks. On the other hand, levels that explicitly allow a non-looping solution due to this may be beneficial to teach introductory topics.

4.1.3 Ease of authoring

Particular care was taken to separate the level specification from the rest of the game. Levels are defined in textual data files that the game then reads and interprets, allowing an easy authoring of new levels without need of changing the game code.

4.2 Sample levels

This section details several typical exercises inspired by the categories defined previously, lists their possible implementations in the C language and presents sample levels of ROBO^3 that cover the same topics.

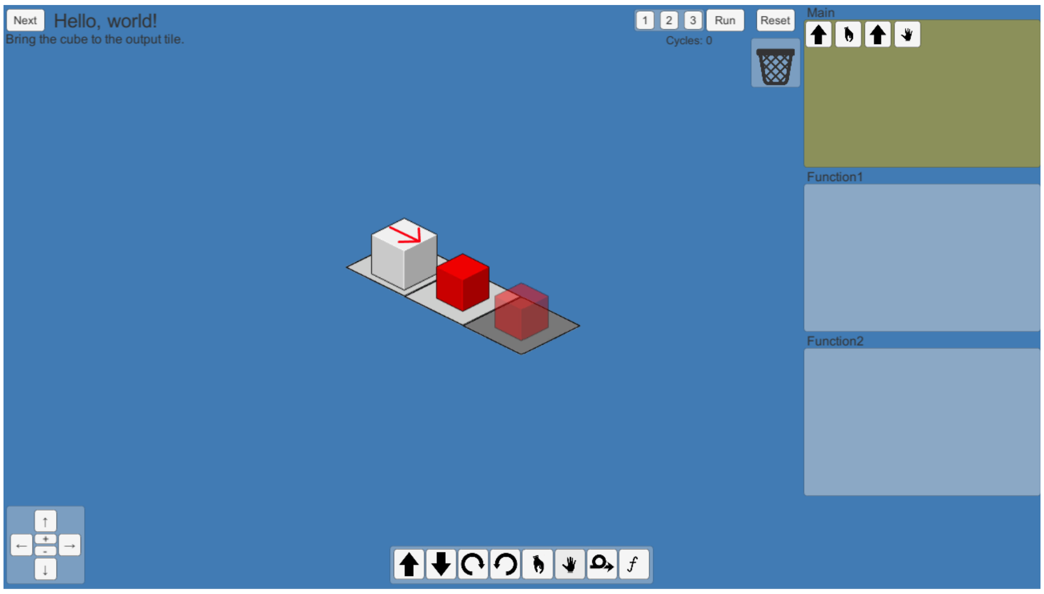
4.2.1 Hello world

Simple deterministic sequences are defined as very simple programs that execute linearly, without any form of control flow constructs. The simple *Hello world* program, that prints the phrase “Hello, world!” on the screen, is one such example. What this program does is writing a constant string to the output stream. Another similar example would be a *greeter* program, where the program asks for the user’s name and then prints a tailored greeting: in this case, the program acquires some data from the input stream and

```
void main()
{
    int num;

    scanf("%d", &num);
    printf("%d", num);
}
```

Figure 4.1: Code specification for simple deterministic sequences



Bring the cube to the output tile.

Figure 4.2: Sample level and solution for simple deterministic sequences

```
\begin{lstlisting}
void main()
{
    FILE *s1, *s2;
    int num;

    /*Acquisition of the streams...*/

    scanf("%d", &num);
    if(num >= 0)
    {
        fprintf(s1, "%d", num);
    }
    else
    {
        fprintf(s2, "%d", num);
    }
}
```

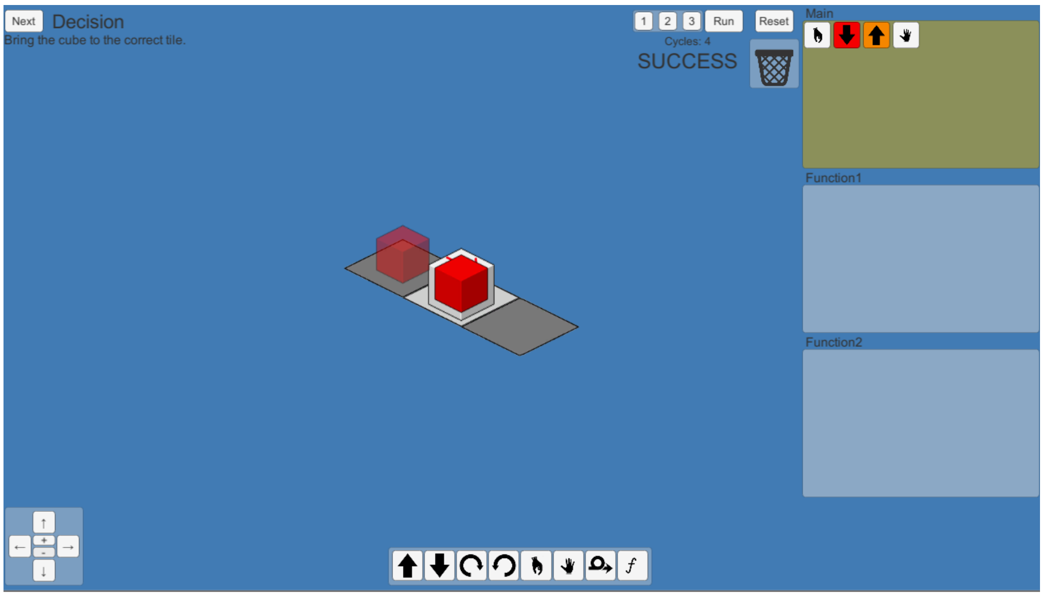
Figure 4.3: Code specification for for conditions

then writes it to the output stream. The barebone specification that implements this pattern acquires one integer from *standard in* and prints that same number to *standard out*. Such a program is, of course, of limited use, but it is a first step towards familiarizing with programming. It employs the simplest possible form of input/output, executes sequentially and terminate. See Figure 4.1.

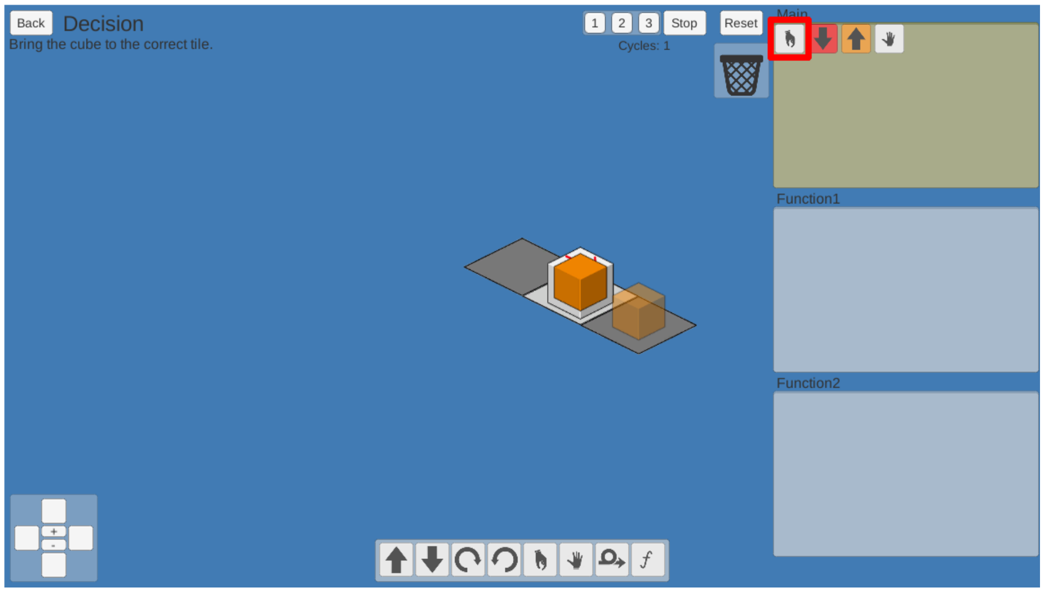
Keeping in mind the similarities between streams and cube stacks, we can easily design the simplest ROBO^3 level that requires to move one cube from the input tile to the output tile. See Figure 4.2. Much as is it helps a novice familiarize with the language and with new concepts, such a program is also helpful in a game environment for the same reasons: it allows the player to learn how to *use* the game.

4.2.2 Property comparison

Typical introductory exercises on conditional constructs ask the user to check if a particular property of some input data is true or false: a number is positive, a character is alphabetic, a year is a leap year and so on. As mentioned, cube colors lack a total order relation and therefore typical comparison exer-



(a) First test case



(b) Second test case

Figure 4.4: Sample level and solution for conditions

```
void main()
{
    int num;

    do
    {
        scanf("%d", &num);
        printf("%d", num);
    }
    while(num != 0);
}
```

Figure 4.5: Code specification for loops (1)

cises, such as finding the maximum of two or three numbers, are not possible. Maintaining the parallel between stacks and streams, checking a property in Robo^3 would lead to carrying the cube to a different validation tile, meaning writing the data to a different output stream. See Figure 4.3.

An implementation in the game requires the use of several test cases. Each test case has exactly one input cube, for example either red or non-red. There are two distinct validation tiles: the first requests the red cube if the input cube is red and the second requests the non-red cube if the input cube is non-red; they do not otherwise request anything. Thanks to the different test cases, the player is forced to write a program that transports the cube in the first validation tile if it is red and in the second validation tile if it is not. See Figure 4.4.

4.2.3 Input stream copy and reversal

Let us consider a program that processes a sequence of integers coming from an input stream. The length of the sequence is unknown, but it is, for example, zero-terminated. This is a textbook case of an exercise requiring a loop to acquire the data. The simplest operation that can be done on the input data is replicating it on another output stream, maintaining order. See Figure 4.5.

A more complex program would ask to reverse the sequence before writing it out. This requires memorizing each value in an appropriate data structure and then reading them in reverse. For simplicity's sake, we can assume that there is an upper bound on the length of the sequences. See Figure 4.6.

Due to the nature of stacks, the level of Robo^3 that implements the first

```c
void main()
{
    int num, i, j;
    int buf[16];

    i = 0;
    do
    {
        scanf("%d", &num);
        buf[i] = num;
        i++;
    }
    while(num != 0);

    j = i - 1;
    while(j >= 0)
    {
        printf("%d", buf[j]);
        j--;
    }
}
```

Figure 4.6: Code specification for loops (2)

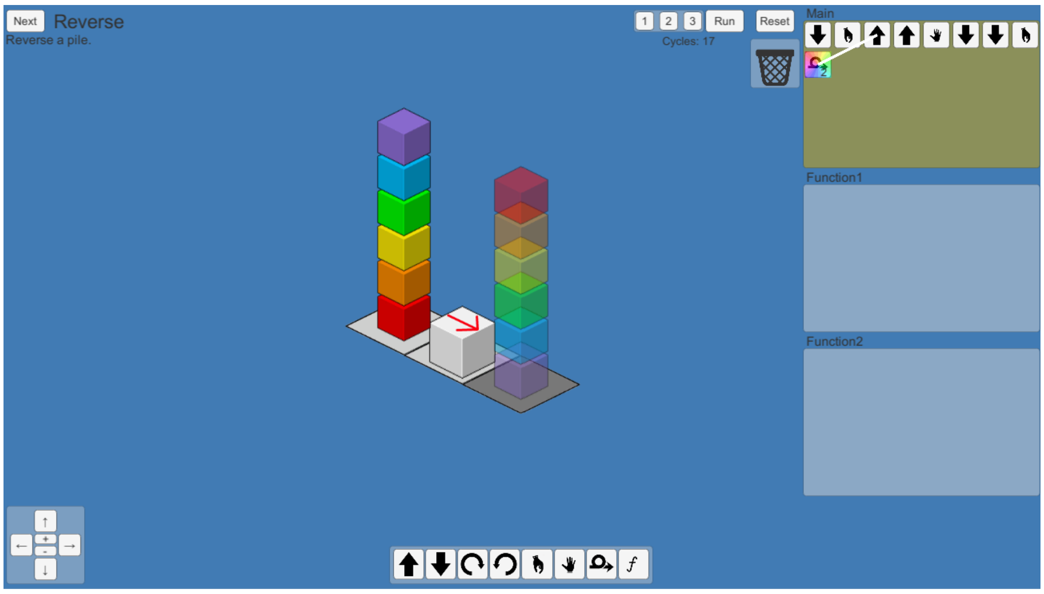


Figure 4.7: Sample level and solution for loops (1)

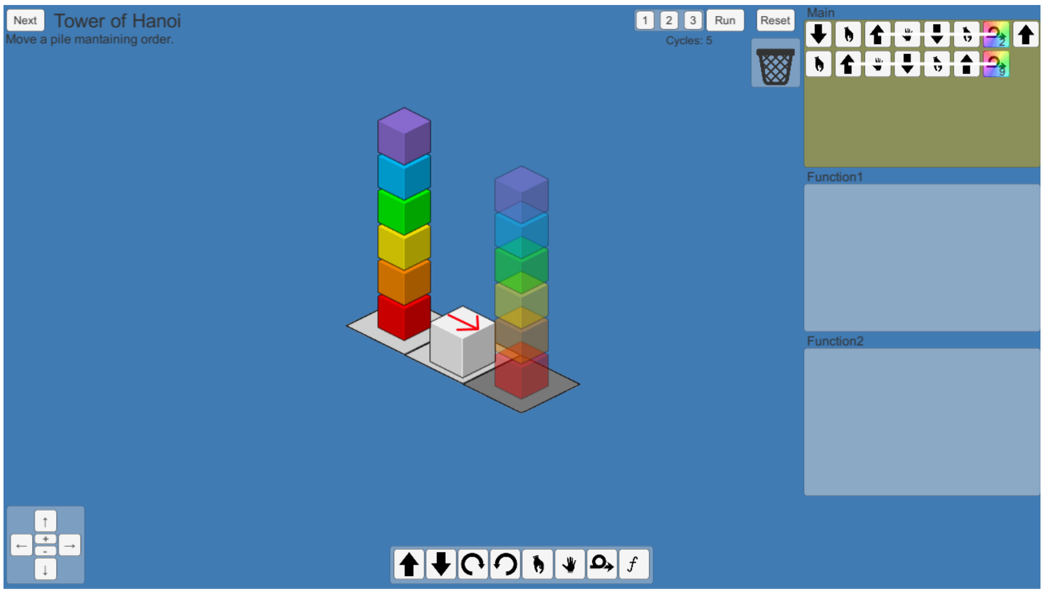


Figure 4.8: Sample level and solution for loops (2)

pattern is the *reversal* of an input stack; similarly, the level that implements the second pattern is the movement of an input stack, maintaining order. This behaviour makes sense when one considers that what would be the first element for an input stream rests on top of an input stack of cubes, while what would be the first element for an output stream stays on the bottom of an output stack.

In the first case, the difficulty is all in understanding how to structure the loop so that it breaks when there are no more cubes. This is achieved by having the condition on the loop check if there is any cube still in the hands of the robot. Since it is a do/while loop, this requires a few adjustments to the structure that one might intuitively draw having a while loop in mind. See Figure 4.7.

In contrast with the C specification, the second case is still very easy and only requires to apply the solution for the first case twice. Conceptually, of course, the player still has to use an intermediate data structure as temporary storage. See Figure 4.8.

4.2.4 Functions

Functions in Robo^3 are parameterless, so they are mostly useful as a code reuse tool. They do have, however, an interesting interaction with conditions: since conditions are applied symbol-by-symbol, wrapping some instructions

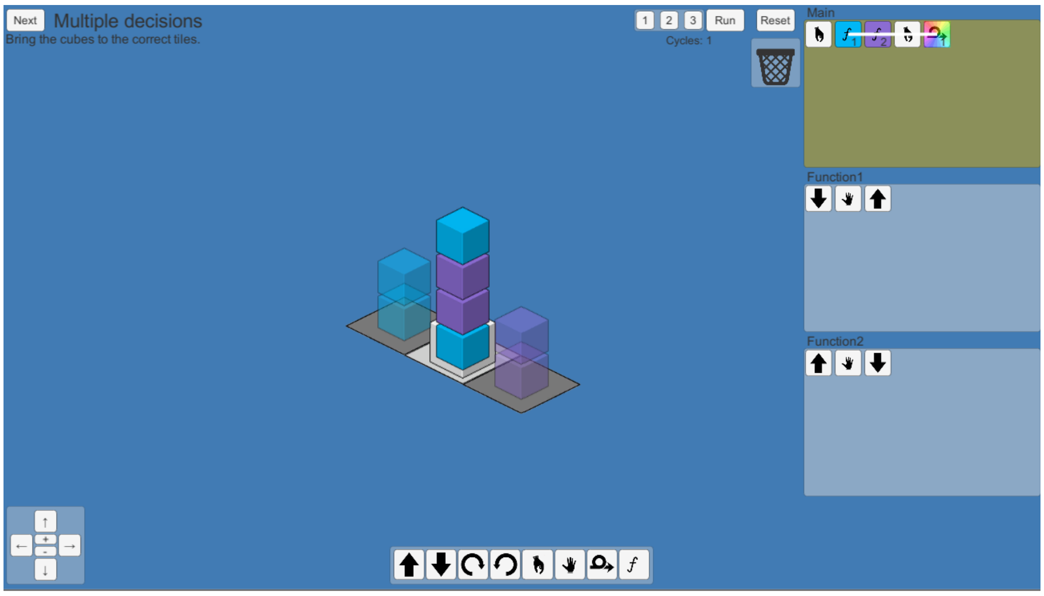


Figure 4.9: Sample level and solution for functions

in a function and applying a condition to that function is similar to having a condition on a whole block of code. This allows more complex behaviours, such as, for example, a version of the level described in subsection 4.2.2 with an unbounded number of input cubes: in fact, it is possible to use two functions, each bringing the held cube to a different place and then making the robot come back to the starting position, which could not otherwise be done due to how conditions work. See Figure 4.9.

4.2.5 List fill

The lack of numbers in ROBO^3 prevents from having indexed arrays in the game. Stacks are the native data structure. It is possible, however, to mimick other kinds of structures, such as lists.

Let us consider a program that appends an element to the end of a linked list. Lists are not accessed by index but by traversal, so to reach their tail one must scan the whole list. See Figure 4.10.

The corresponding level works thanks to the property of conditional checks applied to the *pick-up* instruction, which checks the topmost cube of the tile stack instead of the held cube. This can be used to find the nearest free tile. See Figure 4.11. Managing this representation of a list in the game is easier than in C, since ROBO^3 allows to abstract away from complex details like pointers and memory allocation. Going back to the starting

```
void append(node_t *list, int num)
{
    node_t *node = malloc(sizeof(node_t));
    node -> next = NULL;
    node -> value = num;

    while(list -> next != NULL)
    {
        list = list -> next;
    }

    list -> next = node;
}
```

Figure 4.10: Code specification for data structures

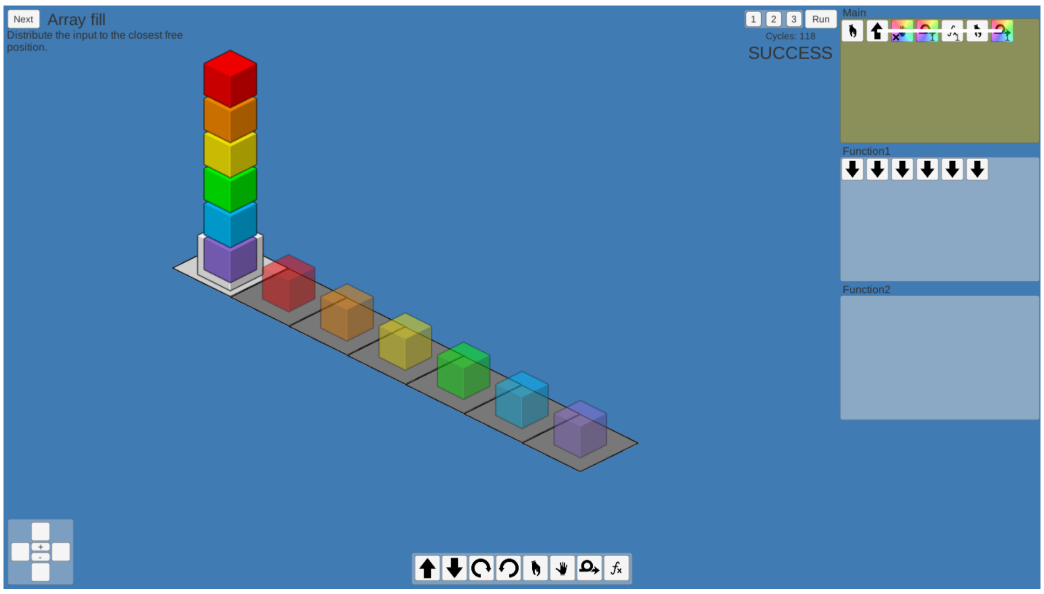


Figure 4.11: Sample level and solution for data structures

```
int COUNT = 5;

void main()
{
   act();
}

void act()
{
   COUNT--;
   printf("First part of the call.\n");
   if(COUNT > 0)
   {
      act();
   }
   printf("Second part of the call.\n");
}
```

Figure 4.12: Code specification for recursion

tile to pick another cube and repeat the procedure is quite inelegant in this solution. A cleaner approach is to use recursive function calls, as discussed in subsection 4.2.6.

4.2.6 Stack-based counter and recursive stream copy

Since Robo^3 functions are parameterless, recursion loses some of its power. The game does however fully support recursive calls of functions and keeps track of the whole call stack. Since stack-based *Push Down Automata* are notoriously able to count, it is easy to see how one could use this property to execute certain actions an undefined number of times and then, once a breaking condition occurs, to execute other actions *the same number of times.*

There is no meaningful related programming exercise in the C language, since for the simpler cases there is no real reason to use the call stack to keep track of a number when one can just increase a counter; real-life applications of the benefits of this technique are rather complex. The code in Figure 4.12 is however a proof of concept for this behaviour.

Lacking other options, using the call stack to count becomes much more useful in Robo^3. A level that easily uses this property is one where there are at least two possible input tiles (in different test cases) arranged in a line, so

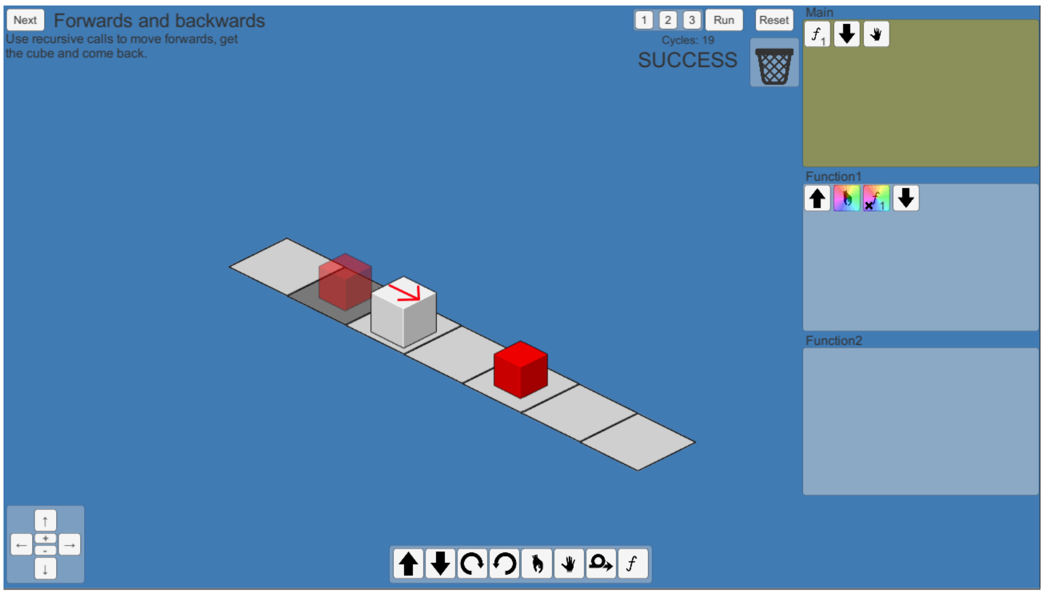


Figure 4.13: Sample level and solution for recursion (1)

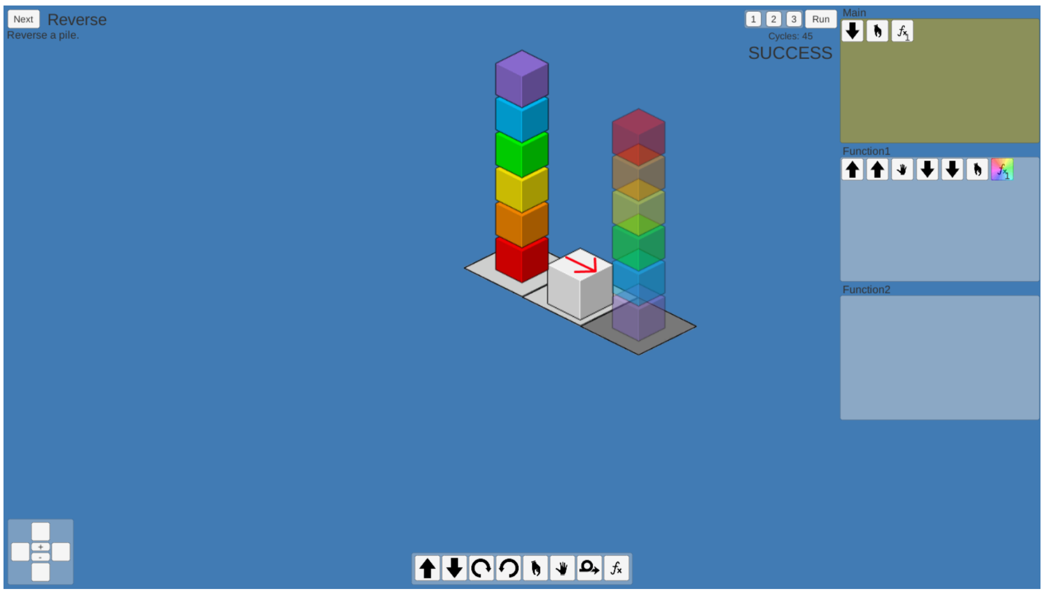


Figure 4.14: Sample level and solution for recursion (2)

that the robot does not know *a priori* how far to go. Using recursion, it is possible to write a program that allows the robot to move forwards until a cube is found, pick it up and then move backwards of the same number of steps. See Figure 4.13.

The special case of tail recursion notoriously boils down to looping. The same stream copy or stack reversal level presented in subsection 4.2.3 can also be quite elegantly solved with tail recursion, as depicted in Figure 4.14.

4.3 Conclusions

The design choice of a simplified programming model has both advantages and disadvantages. While it is true that the language is easier to understand and use as a result, it is also true that the lack of meaningful exercises soon leads to a brisk increase in level complexity due to the lack of certain elements such as numbers and indexed arrays, which would allow a wider level design space. In fact, the last levels presented in this chapter employed the properties of advanced data structures such as lists and stacks.

Chapter 5

Dashboard

In order to allow the instructor to keep track of the progress of the students playing Robo^3, a data gathering system has been integrated into the game. Robo^3 sends messages to a remote server on certain specific game events; these messages are then retrieved and displayed in aggregated form by a companion dashboard environment. The whole system allows for a deeper understanding of users' progress and opens the door to more complex analyses over time.

5.1 Data model

Each time a user performs an action that is deemed relevant, the game composes and sends a message to a remote server. These messages will then be downloaded and processed by the dashboard in order to be graphically shown.

5.1.1 Game events

There are four events that generate messages:

- entering a level (*ENTER*);
- exiting a level (*EXIT*);
- failing a simulation in a level (*FAILURE*);
- successfully completing a level (*SUCCESS*).

The game sends exactly one message every time one of these events takes place. *ENTER* and *EXIT* messages allow to keep track of user movement and time spent inside levels, while *FAILURE* and *SUCCESS* messages allow to keep track of user behaviour and successes.

5.1.2 Game messages

A message is structured as follows, containing several different fields:

- message type: identifies the event that spawned this message (*ENTER*, *EXIT*, *FAILURE*, *SUCCESS*);
- user identifier: a unique identifier associated to each user, likely the same identifier used by the institution holding the course Robo^3 is used in;
- timestamp: records the moment the message was created and sent;
- level name: the level related to the fired event (the one just started for *ENTER*, the one just left for *EXIT* and where failure and success took place for *FAILURE* and *SUCCESS*);
- level session unique identifier: newly and uniquely associated by the game to each level every time the user enters inside it; allows easier matching of *ENTER* and *EXIT* messages to compute time spent inside a level. All messages generated during the same level session have the same value for this field.

Messages generated by a *SUCCESS* event contain some additional fields:

- solution: a complete listing of all instructions composing the successful solution to the level, including all data related to conditions, loops and function calls;
- cycles: the average over all test cases of how many simulation cycles (program steps) were necessary to complete the level with the user program;
- instructions: how many instructions compose the solution. While redundant (one could obtain this value from the solution field), it allows to slightly optimize the processing of data done by the dashboard;
- functions: how many functions compose the solution. While redundant (one could obtain this value from the solution field), it allows to slightly optimize the processing of data done by the dashboard.

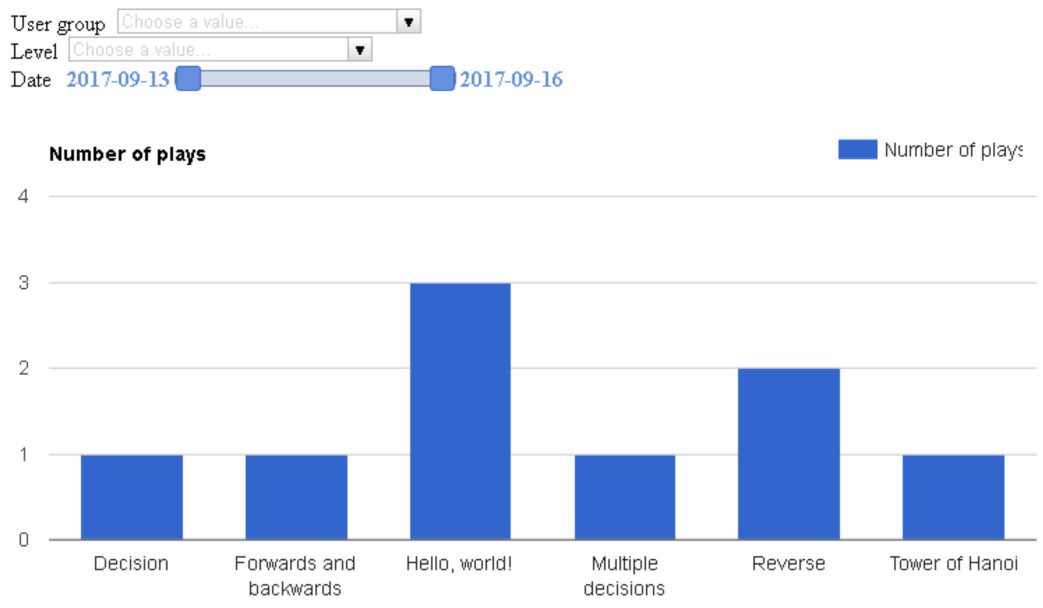


Figure 5.1: Sample chart for the number of plays

5.2 Dashboard design

The goal of the companion dashboard environment is to offer a flexible data visualization tool, so as to allow analyses on game use and player performance, progress and perceived difficulty. The dashboard is a web application that connects to the remote storage server, fetches all stored messages, processes them and builds a variety of charts on several different topics. All charts are interactive and allow the data they show to be filtered according to different metrics, like time period, user or level. The dashboard is organized in several different parts, each dedicated to certain charts. They are detailed in the following subsections.

5.2.1 Time spent playing

Three charts detail how much time users spend on levels. The first shows the total time spent inside a level, the second shows the average user time inside the level and the third shows the total number of plays that a level had. They consist of column charts, with a column on the x-axis for each level and the y-axis dedicated to the aforementioned values. They can be filtered so as to show data only from certain time ranges, from certain users or from certain levels. See Figure 5.1 and Figure 5.2.

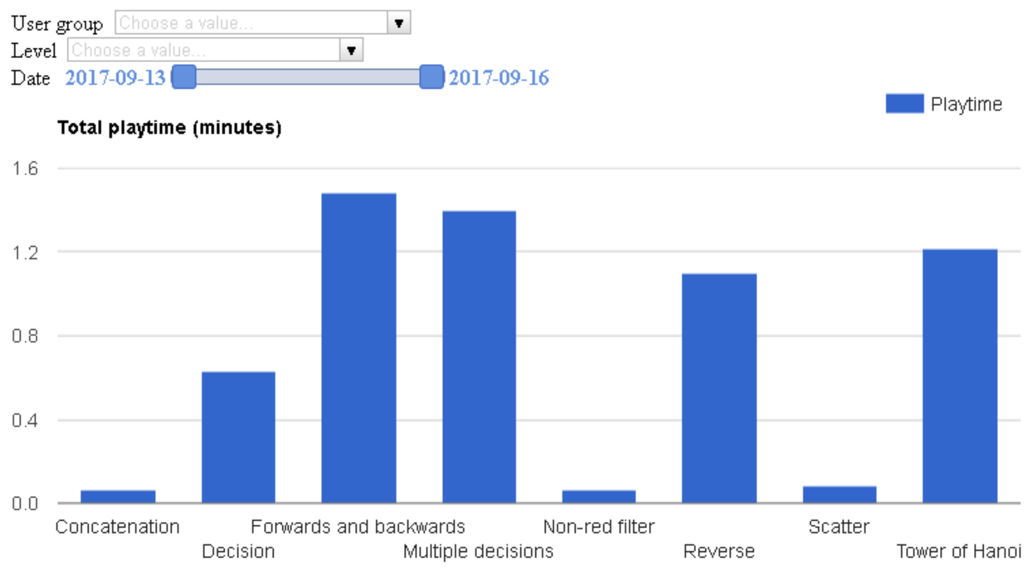


Figure 5.2: Sample chart for the total playtime

5.2.2 Successes

Two charts detail how easy a level is for players. The first shows the percentage of successes over all attempts at solving the level, the second shows how many players successfully completed that level at least once. They consist of column charts, with a column on the x-axis for each level and the y-axis dedicated to the aforementioned values. They can be filtered so as to show data only from certain time ranges, from certain users or from certain levels. See Figure 5.3.

5.2.3 Solutions content

Three charts detail summarize the content of user solutions. The first shows how many instructions, and of which kind, were used for the level, the second shows how many function calls were (directly) recursive versus how many were not and the third shows how many instructions were conditioned versus how many were not. They consist of stacked column charts, with a column on the x-axis for each level and the y-axis dedicated to the aforementioned values. They can be filtered so as to show data only from certain time ranges, from certain users or from certain levels. See Figure 5.4.

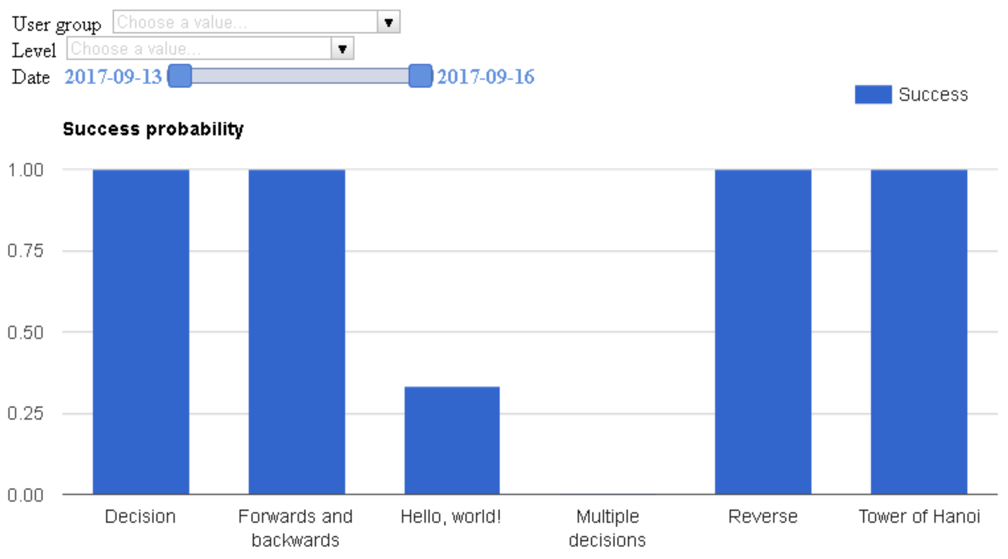


Figure 5.3: Sample chart for the success probability

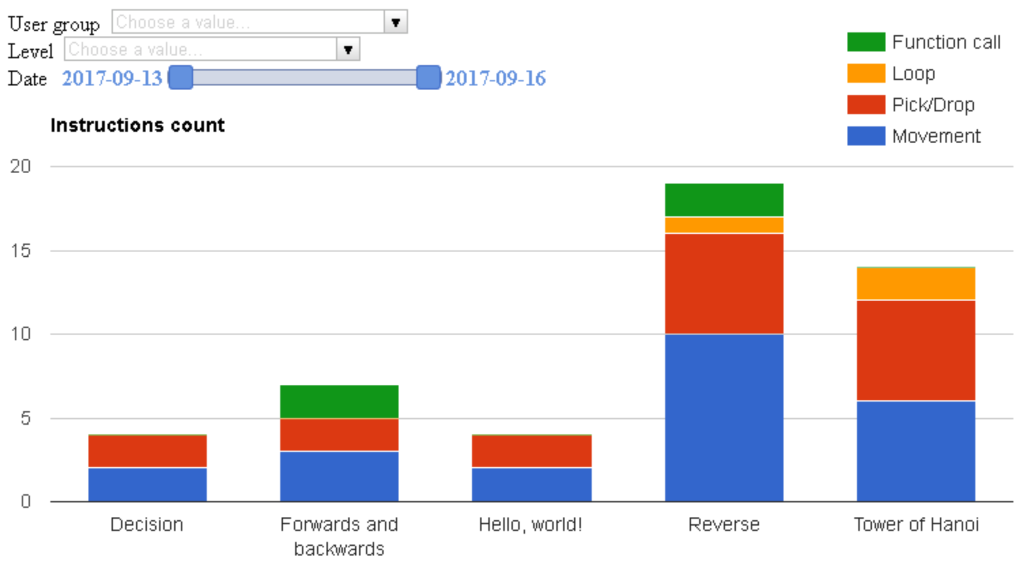


Figure 5.4: Sample chart for the instructions count

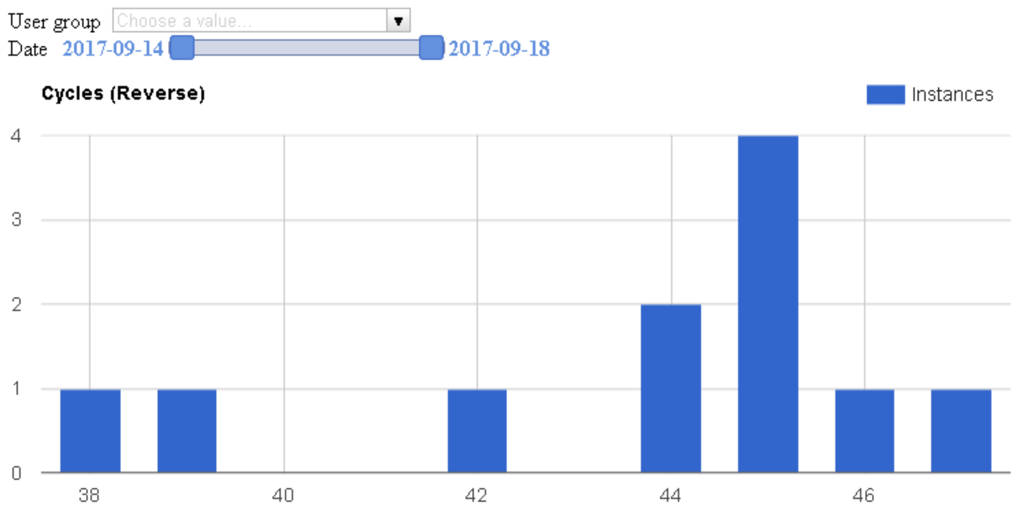


Figure 5.5: Sample chart for cycles performance

5.2.4 Solutions performance

Three charts for each level detail the performance of user solutions using different metrics. The first shows the cycles elapsed, the second shows the instructions count and the third shows the functions count. They consist of column charts, with a column on the x-axis for each unique value in solutions and the y-axis dedicated to how many solutions achieved that particular value. They can be filtered so as to show data only from certain time ranges or from certain users. See Figure 5.5

5.2.5 User overview

This part of the dashboard repeats all charts previously described, but filters them so as to only show data pertaining to a specific user.

Chapter 6

Conclusions

This thesis discussed the design and implementation of ROBO^3, a web game to help the teaching of programming skills and analytical thinking abilities. Its purpose is being paired as a reinforcement and exercising tool along with a traditional introductory programming course; its focus is both on helping students learn by visualizing the effects of the code they write and on helping instructors to easily design appropriate levels and gather useful data on students' behaviour.

An analysis of the scientific literature and of the commercial games available allowed for a comparison of different approaches, highlighting advantages and disadvantages of each, leading to the choice of a higher-level approach for ROBO^3, where the player does not write the code directly with a real-life syntax but uses atomic symbols to compose a program. An effort was made to provide an instruction set that, at the same time, is easy to use and provides the structures of modern high-level languages, like function calls and loops. Support for different test cases in each level also allows the instructor to force the players to think of their solutions in general terms, training critical thinking abilities.

The design choice of not including the handling of numbers in the game has allowed to simplify the instruction set, making it easier to learn and use. It has also, however, reduced the design space for levels, preventing many of the standard exercises found in typical introductory courses to be implemented as levels. This has resulted in a rather steep increase in difficulty once leaving the more trivial exercises behind, due to the lack of indexed data structures such as arrays.

The game is naturally paired with a dashboard environment that allows the instructor to easily visualize all data gathered by the game on the performance of players. This allows the instructor to analyze the results, perform meaningful comparisons and make informed choices when guiding the learning process.

6.1 Future development

Robo^3 has been designed to constantly gather data on the performance of its players, but time constraints have prevented from actually employing it on the field and carrying out a campaign of analysis to test its actual impact on the learning experience. This aspect should certainly be investigated, especially since its purpose is being used in the real world by instructors for introductory programming courses.

In order to further expand the ease of use of the whole system, in the case of formal adoption from one such course, the game might be integrated with the institution's own authentication system, allowing to keep better track of the performance of students that use it. In a similar fashion, the dashboard environment might be expanded as needed to support the visualization of new forms of aggregate data, empowering the analysis tools offered to the instructor.

Further developments on the game itself may include the expansion of the instruction set and the inclusion of numbers in lieu of or in addition to colors, allowing for a more diverse set of exercises, to better train other aspects taught in the courses or to smooth out the difficulty curve of the levels. Such an extension should of needs however to be carefully designed and validated through user studies, as discussed.

Bibliography

[1] Eike Falk Anderson and Leigh McLoughlin. 'Critters in the Classroom: A 3D Computer-game-like Tool for Teaching Programming to Computer Animation Students'. In: *ACM SIGGRAPH 2007 Educators Program*. SIGGRAPH '07. San Diego, California: ACM, 2007. ISBN: 978-1-4503-1830-3. DOI: 10.1145/1282040.1282048. URL: http://doi.acm.org/10.1145/1282040.1282048.

[2] Peter Drake and Kelvin Sung. 'Teaching Introductory Programming with Popular Board Games'. In: *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*. SIGCSE '11. Dallas, TX, USA: ACM, 2011, pp. 619–624. ISBN: 978-1-4503-0500-6. DOI: 10.1145/1953163.1953338. URL: http://doi.acm.org/10.1145/1953163.1953338.

[3] Michael Eagle and Tiffany Barnes. 'Experimental Evaluation of an Educational Game for Improved Learning in Introductory Computing'. In: *SIGCSE Bull.* 41.1 (Mar. 2009), pp. 321–325. ISSN: 0097-8418. DOI: 10.1145/1539024.1508980. URL: http://doi.acm.org/10.1145/1539024.1508980.

[4] Ben Gibson and Tim Bell. 'Evaluation of Games for Teaching Computer Science'. In: *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*. WiPSE '13. Aarhus, Denmark: ACM, 2013, pp. 51–60. ISBN: 978-1-4503-2455-7. DOI: 10.1145/2532748.2532751. URL: http://doi.acm.org/10.1145/2532748.2532751.

[5] Lindsey Ann Gouws, Karen Bradshaw and Peter Wentworth. 'Computational Thinking in Educational Activities: An Evaluation of the Educational Game Light-bot'. In: *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '13. Canterbury, England, UK: ACM, 2013, pp. 10–15. ISBN: 978-1-4503-2078-8. DOI: 10.1145/2462476.2466518. URL: http://doi.acm.org/10.1145/2462476.2466518.

[6] Britton Horn et al. ‘Design Insights into the Creation and Evaluation of a Computer Science Educational Game’. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE ’16. Memphis, Tennessee, USA: ACM, 2016, pp. 576–581. ISBN: 978-1-4503-3685-7. DOI: `10.1145/2839509.2844656`. URL: `http://doi.acm.org/10.1145/2839509.2844656`.

[7] T. Jordine, Y. Liang and E. Ihler. ‘A mobile-device based serious gaming approach for teaching and learning Java programming’. In: *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. Oct. 2014, pp. 1–5. DOI: `10.1109/FIE.2014.7044206`.

[8] Cagin Kazimoglu et al. ‘A Serious Game for Developing Computational Thinking and Learning Introductory Computer Programming’. In: *Procedia - Social and Behavioral Sciences* 47 (2012). Cyprus International Conference on Educational Research (CY-ICER-2012)North Cyprus, US08-10 February, 2012, pp. 1991–1999. ISSN: 1877-0428. DOI: `http://dx.doi.org/10.1016/j.sbspro.2012.06.938`. URL: `http://www.sciencedirect.com/science/article/pii/S1877042812026742`.

[9] Michael J. Lee. ‘How Can a Social Debugging Game Effectively Teach Computer Programming Concepts?’ In: *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. ICER ’13. San Diego, San California, USA: ACM, 2013, pp. 181–182. ISBN: 978-1-4503-2243-0. DOI: `10.1145/2493394.2493424`. URL: `http://doi.acm.org/10.1145/2493394.2493424`.

[10] Michael J. Lee and Andrew J. Ko. ‘Personifying Programming Tool Feedback Improves Novice Programmers’ Learning’. In: *Proceedings of the Seventh International Workshop on Computing Education Research*. ICER ’11. Providence, Rhode Island, USA: ACM, 2011, pp. 109–116. ISBN: 978-1-4503-0829-8. DOI: `10.1145/2016911.2016934`. URL: `http://doi.acm.org/10.1145/2016911.2016934`.

[11] Michael J. Lee, Andrew J. Ko and Irwin Kwan. ‘In-game Assessments Increase Novice Programmers’ Engagement and Level Completion Speed’. In: *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. ICER ’13. San Diego, San California, USA: ACM, 2013, pp. 153–160. ISBN: 978-1-4503-2243-0. DOI: `10.1145/2493394.2493410`. URL: `http://doi.acm.org/10.1145/2493394.2493410`.

[12] Matthew B. MacLaurin. ‘The Design of Kodu: A Tiny Visual Programming Language for Children on the Xbox 360’. In: *SIGPLAN Not.* 46.1

(Jan. 2011), pp. 241–246. ISSN: 0362-1340. DOI: `10.1145/1925844.1926413`. URL: `http://doi.acm.org/10.1145/1925844.1926413`.

[13] N. Masso and L. Grace. 'Shapemaker: A game-based introduction to programming'. In: *2011 16th International Conference on Computer Games (CGAMES)*. July 2011, pp. 168–171. DOI: `10.1109/CGAMES.2011.6000334`.

[14] T. Mitamura, Y. Suzuki and T. Oohori. 'Serious games for learning programming languages'. In: *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Oct. 2012, pp. 1812–1817. DOI: `10.1109/ICSMC.2012.6378001`.

[15] I. Paliokas, C. Arapidis and M. Mpimpitsos. 'PlayLOGO 3D: A 3D Interactive Video Game for Early Programming Education: Let LOGO Be a Game'. In: *2011 Third International Conference on Games and Virtual Worlds for Serious Applications*. May 2011, pp. 24–31. DOI: `10.1109/VS-GAMES.2011.10`.

[16] Rathika Rajaravivarma. 'A Games-based Approach for Teaching the Introductory Programming Course'. In: *SIGCSE Bull.* 37.4 (Dec. 2005), pp. 98–102. ISSN: 0097-8418. DOI: `10.1145/1113847.1113886`. URL: `http://doi.acm.org/10.1145/1113847.1113886`.

[17] A. Sajana, K. Bijlani and R. Jayakrishnan. 'An interactive serious game via visualization of real life scenarios to learn programming concepts'. In: *2015 6th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. July 2015, pp. 1–8. DOI: `10.1109/ICCCNT.2015.7395173`.

[18] Birgit Schmitz et al. 'Transferring an Outcome-Oriented Learning Architecture to an IT Learning Game'. In: *Towards Ubiquitous Learning: 6th European Conference of Technology Enhanced Learning, EC-TEL 2011, Palermo, Italy, September 20-23, 2011. Proceedings*. Ed. by Carlos Delgado Kloos et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 483–488. ISBN: 978-3-642-23985-4. DOI: `10.1007/978-3-642-23985-4_43`. URL: `https://doi.org/10.1007/978-3-642-23985-4_43`.

[19] A. Serrano-Laguna et al. 'Building a Scalable Game Engine to Teach Computer Science Languages'. In: *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje* 10.4 (Nov. 2015), pp. 253–261. ISSN: 1932-8540. DOI: `10.1109/RITA.2015.2486386`.

[20] A. J. Sierra et al. 'Educational resource based on games for the reinforcement of engineering learning programming in mobile devices'. In: *2016 Technologies Applied to Electronics Teaching (TAEE)*. June 2016, pp. 1–6. DOI: `10.1109/TAEE.2016.7528384`.

[21] Kathryn T. Stolee and Teale Fristoe. 'Expressing Computer Science Concepts Through Kodu Game Lab'. In: *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*. SIGCSE '11. Dallas, TX, USA: ACM, 2011, pp. 99–104. ISBN: 978-1-4503-0500-6. DOI: `10.1145/1953163.1953197`. URL: `http://doi.acm.org/10.1145/1953163.1953197`.

[22] Joe Tessler, Bradley Beth and Calvin Lin. 'Using Cargo-bot to Provide Contextualized Learning of Recursion'. In: *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. ICER '13. San Diego, San California, USA: ACM, 2013, pp. 161–168. ISBN: 978-1-4503-2243-0. DOI: `10.1145/2493394.2493411`. URL: `http://doi.acm.org/10.1145/2493394.2493411`.

[23] Anastasios Theodoropoulos, Angeliki Antoniou and George Lepouras. 'How Do Different Cognitive Styles Affect Learning Programming? Insights from a Game-Based Approach in Greek Schools'. In: *Trans. Comput. Educ.* 17.1 (Sept. 2016), 3:1–3:25. ISSN: 1946-6226. DOI: `10.1145/2940330`. URL: `http://doi.acm.org/10.1145/2940330`.

[24] A. Vahldick, A. J. Mendes and M. J. Marcelino. 'A review of games designed to improve introductory computer programming competencies'. In: *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. Oct. 2014, pp. 1–7. DOI: `10.1109/FIE.2014.7044114`.