HaskellQuest: a game for teaching functional programming in Haskell

Karolina Drobnik

Master of Science

School of Informatics
University of Edinburgh

2018

Abstract

Learning through games gathers growing enthusiasm among researchers and lecturers. Programming, as a complex skill to master, requires a substantial amount of practice and reading, which discourages many inspiring computer scientists. To address this issue, researchers proposed the introduction of educational games that is believed to make the learning process more attractive and approachable. Still, such an approach has been rarely seen in functional programming education, which lacks entertaining forms of teaching.

In this dissertation we propose a game-based approach for teaching Haskell, a purely functional programming language. The main goal of this work is to increase the learners' motivation and enjoyment by presenting programming ideas through play. During the playing session, players program objects in the game world through a block-based programming language based on Haskell. For the purposes of this work, we developed a 3D game prototype called *HaskellQuest*, that aspires to teach basic concepts of the functional paradigm. This application is aimed at novice programmers who wish to develop their programming skills in an engaging and entertaining way. Our results show that the game positively influences learning the basics of Haskell concepts, engages users in playing challenges that require block-based programming, and motivates them to learn more advanced concepts. However, our solution is far from perfect and only implements a concept of game-based learning for programming purposes. We believe that this work will serve as a reference for future work and game-based projects for teaching functional programming.

Acknowledgements

I would like to express my deep gratitude to Professor Sannella for his valuable suggestions during each stage of the project and patient guidance throughout this work. His generously invested time in this dissertation is very much appreciated.

Appreciation is due to Mr. Lujan for giving us a free access to to the beta release of *ScalaQuest*. His game greatly inspired our work and showed that teaching functional programming can be entertaining and engaging.

I wish to thank Mr. Kozłowski for sparking the idea of programmable game objects, his advice on an effective usage of Unity engine and general game development tips. I am deeply grateful for his support.

Finally, I want to extend my thanks to Janek, my fiancé and a great Haskell enthusiast, and my brother, Łukasz, for their ongoing encouragement and support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Karolina Drobnik*)

Table of Contents

Chapter 1

Introduction

Programming is a creative process touching on many fields of technology and allowing to solve problems of every scale. Because of that power, myriad enthusiasts are trying their utmost to acquire that skill. However, the majority of them face obstacles, even on basic concepts, that seem to be impossible to overcome. Such an experience discourages newcomers and left them feeling overwhelmed because this activity demands understanding abstract concepts, formulating solutions using language constructs and a basic understanding of a computer architecture. Moreover, bigger problems await when learning the functional programming paradigm, which relies heavily on abstract mathematical concepts.

This project aims to facilitate the process of learning functional programming concepts by introducing a game-based learning environment. For the purposes of the project, we developed a prototype of an educational game. *HaskellQuest* is a 3D game that represents functional programming concepts as real world objects that can be programmed using the Haskell language. The previous work suggests that teaching programming concepts through a game-based approach positively influences learners' motivation and engagement[41, 43, 23] and enhances their persistence[59]. Furthermore, some studies suggest that incorporating games in the teaching process results in increased learning outcomes[23, 12, 60].

In this work, we formulate a hypothesis that a game-based approach, coupled with programming in a visual environment, is likely to increase students' interest in the language, improve their engagement and understanding of the basics of functional programming.

The results of our evaluation show that the game encourages players to learn more about Haskell and makes the process of learning the basics of functional programming

more enjoyable. Moreover, the analysis of learning outcomes suggests that playing sessions positively influenced the learning process but, as the evaluation was performed on a small sample, we cannot argue that this change was statically significant. Further confirmation of these findings should be addressed in future work, together with technical improvements to our solution.

This dissertation first briefly presents a motivation for providing a game-based framework for teaching Haskell and the results of our approach. Subsequently, the theoretical background of the learning process is discussed, followed by a review of related work and problems in current learning approaches. After presenting the context of our work, two following chapters describe the design process and its outputs, later used in the game implementation. We provide there a high-level view of the project architecture, completed with class diagrams and description of data flow during execution. Later on, the testing phase is evaluated, together with the project performance. Finally, we discuss the possible improvements to our project and point directions for future work.

Chapter 2

Background

2.1 Learning process

The complexity of the learning process have been observed from many perspectives. For the purposes of designing learning environment, Ertmer et al. distinguish the main three: behaviourism, cognitivism and constructivism. Each of these theories is "a source of verified instructional strategies, tactics, and techniques"[20]. Accordingly, a choice of a theoretic foundation depends on the learning context, such as a target teaching group or a complexity of a topic. Building a solution on top of well established approaches, as Ertmer et al. argue, allows to make predictions about the success of used teaching techniques[20].

Behaviourism perceives an act of learning as an observable change in performance of a learner caused by a stimulus, such as a question or, mentioned by Ertmer et al., a use of cue cards[20]. By a repetitive practice of actions and responses from the environment, the learner acquires a specific habit of reacting to the stimuli. This approach places a major emphasis on actions and behaviour but have no consideration for "the structure of a student's knowledge"[20].

The shift in focus towards cognitive processes can be seen in cognitivism, where we reason about the process of knowledge acquisition and its mental representation. This model is able to handle the theory behind the development of complex cognitive processes such as thinking and problem solving[20]. Behaviourism concentrates on the observable performance of the student, whereas cognitivism emphasises processing and storing the information. A learning session is based on a continuous practice with corrective feedback, that aims to support building patterns of knowledge. Unfortunately, the robustness of this theory is overshadowed by its major drawback. This

perspective assumes that knowledge acquired by a student is the same as it was presented, and the mental structures are assumed to not depend on the previous experiences.

Constructivism emerged as a third view to address cognitivism limitations. In this approach, it is assumed that learning is based on prior knowledge and experiences, actively used throughout the learning session[20]. Here, a student infers knowledge based on his or her interpretation of the feedback from the environment to his or her actions. Constructivism assumes that instructional materials should be tailored to prior learners' experiences to provide the most effective knowledge acquisition. Another aspect that distinguishes this approach from cognitivism is the role of instruction, which is used to provide a source for information interpretation, and does not just to make learners memorise facts.

The last approach, together with cognitivism, was identified as the most popular learning theory foundation for game-based learning environments[34]. It is argued that constructivism, thanks to its emphasis on active interaction, closely relates to the activity of playing a game[45]. In addition to this foundation, many researchers[27, 40, 22, 34] use the concept of "flow" to describe an effective learning process.

Flow is defined as a particular type of experience that consists of a deep focus on the activity, ignorance of external stimuli and feeling pure enjoyment of being in this state[17]. This phenomenon was first described by Csikszentmihalyi and can be seen in many cases – doing sports through socialising to listening to music. This concept can be also applied to the effective learning activity, as argued by Csikszentmihalyi: "if the decision is to [follow a given course] because of an inner feeling of rightness, the learning will be relatively effortless and enjoyable."[17]. Moreover, this concept was used to describe the enjoyment felt during playing games, as they make "the activity as distinct as possible from the so-called "reality" of everyday existence"[17]. In order to induce flow, Csikszentmihalyi argues, three conditions need to be met: (1) a clear goal of the task is provided, together with reports on progress; (2) there is immediate and understandable feedback; (3) an appropriate balance of challenges and skills of the person is preserved[17]. Maintaining a flow state requires a specific adaptation of task difficulty to learners' skill improvement over time, which is also true for educational environments. This idea is illustrated by a *flow channel* diagram that can be seen in Fig. 2.1. A_1 represents a beginner that faces an adequately hard task, in contrast to someone in state A_3, who experiences distress when encountering the challenge. With continued practice, both of them can enter the flow channel, with A_1 facing boredom

at some point.

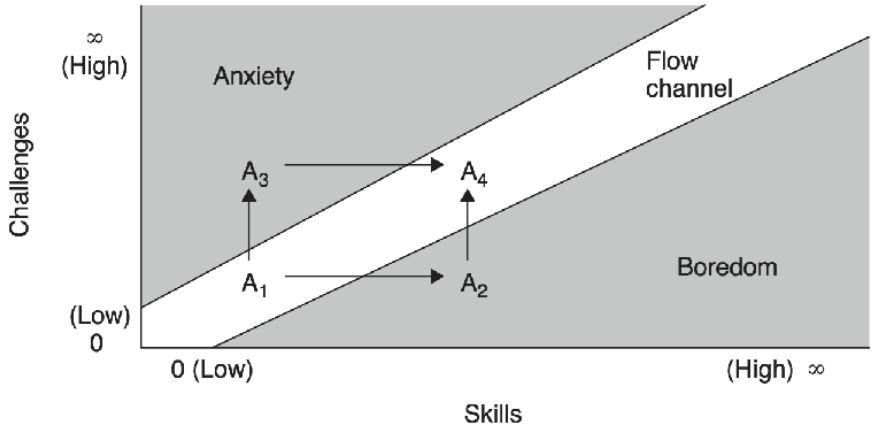

Figure 2.1: Diagram of flow states by Csikszentmihalyi[17].

In this theory, when the skill level is low, flow can appear only when objectives are relatively easy to meet. Otherwise, this process provokes too much anxiety and may discourage from the learning activity. Later on, as the skills increase, too little challenge may cause boredom, likely to decrease motivation to perform a task. In our work we aim to sustain the state of flow for as long as possible by providing puzzles with a gradual increase of difficulty.

The concepts of clear goal and feedback are also a part of a learning activity. At the confluence of gaming and learning activities, Garris et al. put forward a theory of the game cycle model[23]. As it was identified in our project proposal, this model "captures iterative process"[19] of learning that happens during the playing session and can be accommodated for teaching complex ideas. The diagram of this process is presented in Fig. 2.2.

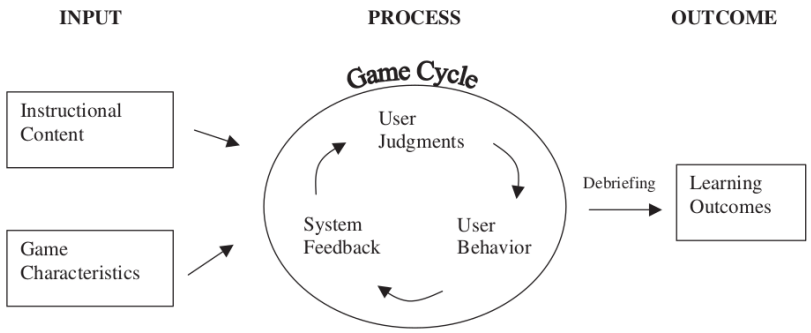

Figure 2.2: Game cycle model diagram[23] by Garris et al.

The game cycle begins with *inputs*, such as instructions and game characteristics,

that significantly influence *user judgement*. The formulated perspective has a further impact on the *user behaviour*, as he or she tries to effectively solve encountered problem. Accordingly, the environment provides feedback on the outcome observed by a user and, depending on the final result, learns if that was a correct approach or if it should be changed.

2.2 Games in teaching programming

A video game is an interactive medium that has a specified system of rules within which a player tries to achieve a specified goal[46]. A user interacts with that system using input devices such as a keyboard or game controllers. Based on his or her actions, feedback is presented in a graphical, often appealing, way. Because this interaction is engaging, and generally enjoyable, games captured the interest of researchers. They proposed their usage for non-entertainment purposes such as training. The term *serious games* describes a group of games that have a different aim, for example practising a particular skill or acquiring knowledge. Over the past twenty years, there have been numerous projects to tackle the problem of teaching programming basics this way. Researches and game developers alike have been working on attractive and effective solutions, a small selection of which is reviewed below.

One of the attempts at lowering entry barriers for learning programming is Scratch[35]. It is a visual programming language presented in an entertaining way aimed at older children that wish to learn how to program. Students can create interactive applications or animations by placing colourful blocks together, which are thought to make the activity more “fun”. As one might notice, this solution implements a purely constructive approach and gives space for experiments. Scratch gathered interest all around the world[29] and is successfully used as a teaching aid[52]. Despite its popularity, this project has a great limitation. A student learns concepts by trial and error without any guidance, which is thought to develop bad programming habits and inability to grasp more complex ideas[33, 37]. We assume that this problem can be addressed with proper guidance from an experienced instructor or by providing informative examples.

The idea of block-based programming language was also developed in Reduct[5], a puzzle game, where a user can execute JavaScript code. It presents programming concepts as concrete pictures. For instance, a function concept is represented by a hole and pipe and a collection of items looks like a bag of the same shapes. As the game progresses, these representations “fade” into actual syntax of the language as

a player builds more intuition about programming. A study involving young adults showed that the great majority of participants performed excellently on questions involving JavaScript recognition and recall. Unfortunately, their knowledge transfer was not satisfactory, as this game places no emphasis on language syntax which was confused even when tested in very similar contexts. Still, the concrete representation of programming concepts as metaphors were still effective for language recognition. Because of these results, we decided to take similar approach and represent data structures as concrete objects, described in section 3.5.2.

Another interesting project in introducing programming with a game-based approach is Giguette's work[24]. He designed a series of small games, each of them representing a concept of data structures or algorithm design. Students first play these games before the teaching session, learn about the concept and then they are asked to implement a simplified version of the played game. He proposed this approach as he identified that students of introductory courses had been struggling with following written instructions and solving algorithmic problems[24]. It is said that providing such an environment for experiments consisting of concrete examples will result in better understanding of the subject and improved students' perception of programming. Unfortunately, his work only describes the implementation of his ideas with no experimental data to support his claims.

The enthusiasm for game-based learning in computer science can be also seen beyond academia. We can categorise these efforts into two groups: usage of commercial entertainment games modified for teaching purposes, and custom educational games designed for developing programming skills.

A solution that falls into the first category is Minecraft[38], a 3D sandbox game that allows to create own world objects from resources in the game and explore the virtual world. Thanks to its overwhelming popularity[28], it was thought that it could be used for serious purposes such as programming. There were several approaches developed for creating a coding environment in this game. MakeCode[53] is an code editor that allows to program modifications in the game using a block-based language. Its visual appearance is similar to that in Scratch, with added components for representing in-game objects. For now, there are no identified studies verifying the effectiveness of this approach but we assume that this approach might work similarly well as Scratch. Moreover, there were also attempts at using actual programming languages in Minecraft. The "Coding with Minecraft"[50] book uses a game modification called ComputerCraft[44] that adds a possibility to write programs inside the game with the

Lua scripting language. Also, the author of the modification claims that using this tool is an “excellent way to learn a real world skill in a fun, familiar environment”[44]. The book itself serves as a guide to ComputerCraft and shows how to program an in-game robot to automate common game tasks such as collecting materials or crafting items[50].

A commercial product that was developed with educational purposes in mind is Human Resource Machine[15]. It is a visual puzzle game about office workers that need to organise items on two conveyor belts, which are the metaphors for input and output. The player needs to create a list of commands to steer this office avatar to organise data according to instructions provided at the beginning of the puzzle. This game shows concepts of programming such as values, addition, arrays, conditions and iteration. The game gathered very positive reviews from the gaming community and was praised for accurate representations of programming concepts.

Lightbot[31] is another game that gained popularity in teaching programming basics. This is a simple video game where a player commands a robot to travel through levels to light up particular squares and make his way out of the maze. The commands issued by a user are stored in a sequence of symbols representing walking, jumping and turning on the light. After learning the basic instructions, a user learns to organise them into functions or compose them using loops. It’s important to emphasise that this solution does not use any kind of programming language and relies only on symbols, allowing even pre-school children to learn basics of computational thinking. Lightbot is one of the few games which was examined for effectiveness and players enjoyment. As we have identified in the project proposal, this game was reviewed by Gouws et al.[25] and they claim that it is a “useful game for practising computational thinking”[25].

The ongoing popularity of instructional methods using games makes us ask if there are convincing proofs of their effectiveness. When reviewing all games in the context of adolescents, Granic et al. claimed that “ findings suggest that video games provide youth with immersive and compelling social, cognitive, and emotional experiences”[26], thus the games alone can positively influence players. Unfortunately, studies for serious games are slightly ambiguous in their assessment of these aspects. It is partially caused by the great diversity of metrics being measured, usage of different methodologies and theoretical models, as it was pointed out by Boyle et al.[9]. They claim that the quality of evidence of game-based learning effectiveness is inadequate and still recommend development of a system for “experimental work”[9] to be able to find attributes

contributing to increased engagement and learning outcomes. Fortunately, the first of them has clear evidence that it is an essential part of an effective learning experience, mentioned by Hamari et al.[27], also confirmed by Boye et al. and Iten et al.[9, 32]. Finally, it was concluded that the enjoyment of the playing session is said to have little or no impact on learning outcomes but influences motivation and engagement[32]. With regards to learning outcomes, a review provided by Backlund and Hendrix claims that serious games "are not always superior to other types of learning material"[6] but still are an effective aid in learning. On the other hand, O'Neil argues that educational games cannot be directly evaluated with regards to "educational benefits"[40], as they heavily depend on the quality of the "instructional design" embedded in the game.

2.3 Teaching functional programming

Functional programming is still being taught through standard approaches or during practice-based workshops. However, we can identify several solutions that took a different direction. One of them is "The Little Schemer" book that "present[s] abstract concepts in a humorous and easy-to-grasp fashion."[21] using Scheme, a Lisp dialect. It presents each subject by a series of question-answer pairs built around basic arithmetic, gradually progressing into recursion and other advanced concepts. Each concept is discussed several times and used as a building block for harder examples, which strongly reinforces recently gained knowledge.

Another book that tries to present functional programming in an approachable way is "The Land of Lisp"[7], which presents Common Lisp concepts with concrete examples and humorous pictures that allow to make connections between concepts. Throughout this book, the reader learns how programs are build and is encouraged to program his or her own versions of the games presented. Additionally, the author introduces comics in between chapters to show a story line in the book to make it more entertaining.

As was mentioned in the previous paragraphs, Haskell relays on the standard approaches, such as introductory courses at universities that assume no previous exposure to programming. These courses are based on textbooks such as "Haskell: The Craft of Functional Programming"[58] and "Programming in Haskell"[30] and require reading in order to make students understand discussed concepts that are later used in project-based assignments[14]. We identified in our previous work that "independent learners ... rely on practical courses available online[3] or massive online open

courses[48].”[19], which are thought to be a great introduction to functional programming. Nevertheless, this approach lacks extended practice under supervision and may contribute to problems with understanding more advanced concepts.

Still, even with these efforts, functional programming has a reputation of an academic language that is *too abstract* to be grasped by a normal person. It was thought that introduction of practical and interesting methods can change this outlook, thus there were a few game-based solutions proposed. One of them is ScalaQuest[36], “a game that provides a series of short Scala-related puzzles that can be solved by explicitly writing one-line programs in Scala”[19]. We suggest that this method adequately addresses the need of novice functional programmers but the beta version of the game suffered from many imperfections. In our project proposal we pointed out the problem of no instructional content, which means that the game is aimed at people with basic knowledge of Scala[19]. Additionally, it was found that the feedback in the game is quite limited and does not depend on the context of the answer[19].

Chapter 3

Design

3.1 Process and methods

In the design phase, we had to make an outline of planned game concepts, such as the story, game world and its content. The latter is, as defined by Ernest Adams, "an artificial universe, an imaginary place in which the events of the game occur"[2], thus it is essential to plan an immersive environment that supports effective learning. In order to design a game in an organised way, we combined design guidelines from the "Level Up!" book[46] with suggestions on planning an effective game puzzle[11], and our own ideas on carrying out the planning phase. The design process is as following:

1. **Establish general story** – describe a context in which the playing session takes place. Firstly, we define a **character**, which will encounter many events that thwart obtaining a **goal**. These events will be the source for problems to solve, presented by puzzles in our game.

2. **Design the game world** – design possible locations and prepare a list of objects that can appear in the game and can be used for building our puzzles.

3. **Describe programming concepts** – create a list of functional paradigm concepts we intend to teach. Their description includes ideas for representing these in the game environment.

4. **Design a gameplay** – a gameplay is defined as a set of actions available to the player to be performed in the game environment. We create a list of what our user can use to interact with to achieve his or her goals. These activities are later referred as **game mechanics**.

5. **Design puzzles** – plan puzzles following the procedure described in section 3.2.

6. **World map** – add details to locations established in 2 and draw a map and storyboards, for later reference.

3.2 Puzzle design

A puzzle is a problem that a player needs to identify and later discover the elements used in the process of solving. This approach favours active exploration and requires increased focus to overcome challenges. We believe that by introducing a moderated constructionism, together with building a player's intuition using the game environment and visual simulation of programming concepts, we can stimulate the learning process in an attractive way.

The process of designing an effective and interesting puzzle isn't clearly established in the industry and the known approaches[46, 47, 18] slightly differ from one another. We decided to follow guidelines provided by Mark Brown[11], which are the output of a thorough analysis of multiple puzzle games and interviews with their creators. After establishing the real-life representations of programming concepts and rules of the game, we can focus on incorporating these elements inside the game cycle model[23] using following the process:

1. **Set the goal of a challenge**

2. **Provide a clear representation** – it is a part of the game cycle *input* component[23].

3. **Show a logical contradiction** – the heart of every puzzle, a situation where some elements are in a state that they are not supposed to be in. For example, our player makes an assumption that every Terminal is an interactive object. Once he or she sees a non-working one, he or she is looking how to get back to the "natural" game state and make it working.

4. **Provide tools to solve a puzzle** – our in-game character uses **Terminal** objects to manipulate objects in the puzzle. To solve the puzzle, a user needs to understand code describing objects that can be manipulated through this mechanism.

5. **Place puzzle in the context** – challenges faced during the gameplay are tied together and support in achieving the main goal of the game. It partially influences *User Behaviour* and maintains immersion.

After establishing all of the above, we can connect the pieces together into one challenge that follows the Input-Process-Outcome Game Model[23]. For the objects in the puzzle, we provide names that are the same as definitions on the Terminal screen. We argue that *Game Characteristics* are described by perceived attributes of interactive objects. For instance, when a player sees a bookcase standing in the way, he or she expects that it can be moved in some direction to make it possible to go around it.

Figure 3.1: In-game instructions

Once the player accesses the Terminal, the first iteration of a game cycle begins. Code presented on the screen is designed to be filled in with multiple-choice answers. Some of them are either syntactically incorrect or perform an action that does not contribute to solving the puzzle. Each listing is intended to be *Instructional Content*, so it is written as close to natural language as possible. Based on the *Judgements*, the user fills in code blanks and, once he or she has finished, presses the "Run" button that evaluates answers and performs actions in the scene. Actions triggered afterwards contribute to *System Feeback* and a player sees the consequences in the game world and compares them to his or her intended goal. The final feedback is gathered from game state observation and enables the user to check the answers and understand his or her mistake, if any. By this approach, he or she tries to look at the problem from different angles, greatly contributing to an active learning experience.

Once we finished gathering ideas for puzzles, we described each of them considering the following aspects:

1. Brief description
2. Game mechanics
3. Location
4. Used world objects
5. Programming concepts

6. Code and available answers

3.3 Puzzle case study – Moving platform

1. **Brief description** – the puzzle is activated once a player restores energy by solving the previous two puzzles. Player goes upstairs and sees an incomplete wooden bridge with a movable platform. The main goal here is to make the platform move from one side to the other. The platform can move only by 6 meters, thus the final solution requires a player to apply *movePlatform* function with the correct direction three times, together with the *stopFor2Seconds* function.

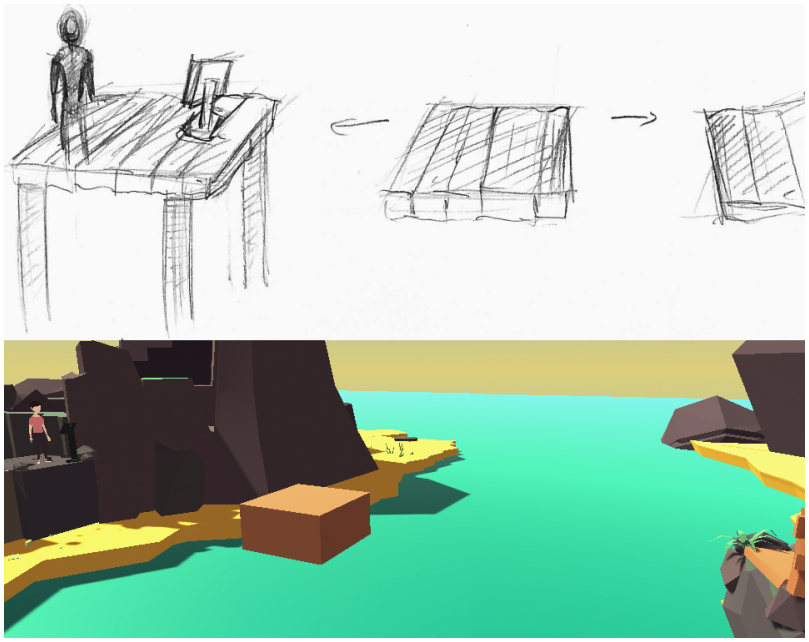

Figure 3.2: Moving platform concept and the in-game look

2. **Game mechanics** – interaction with Terminal, moving platform
3. **Location** – Beach
4. **Used world objects** – programmable platform
5. **Programming concepts** – conditionals, numeric expressions, function composition, algebraic datatypes.
6. **Code and available answers** – full code and answer options can be seen in Appendix C.

3.4 General story and the world

The story in this game is intentionally simple to avoid distraction from the learning process. We are playing with a castaway that arrives at an island that looks usual at first glance. The first thing she sees is a lighthouse, that might have some connection with the outside world, so the protagonist wants to get there and call for help. This is more complicated than expected – the path is often impassable and there are non-working machines that need power to be restored. These challenges can be overcome by programming on small computers standing nearby.

Figure 3.3: First game frame and the interaction with the first Terminal

The game world consist of four main locations. First is a beach, and consists of a pair of small islands connected by a bridge, which is the first puzzle to be solved. By solving this puzzle, a player learns how to interact with the world and how objects in the scene change their behaviour based on the Terminal input. The code can be run multiple times, so the player can experiment with his or her answers as many times as necessary. Other puzzles that can be encountered in this location are the main entrance that needs a password to open, and a power plant that consists of two sub-challenges.

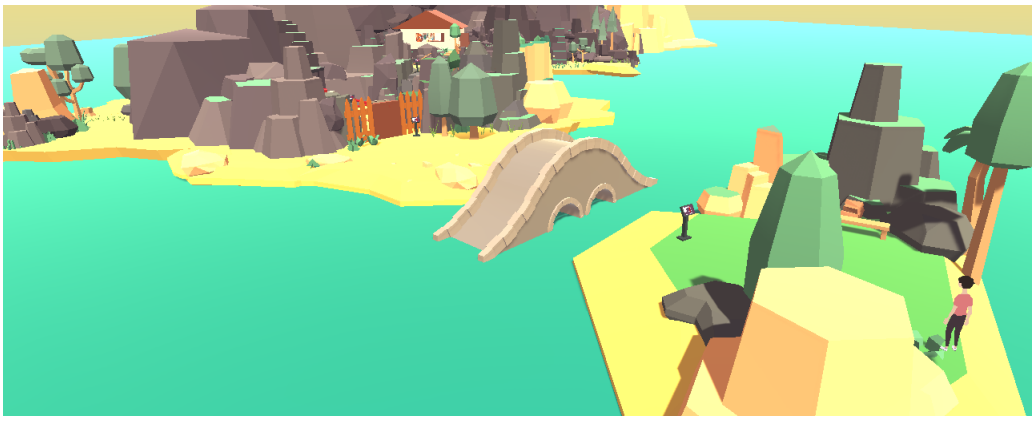

Figure 3.4: Beach location

The second location is a reading room – an interior that has four challenges and each of them contributes to opening of a secret passage to the next location. Here, user explores a small area to find game objects linked to solving sub-challenges. For example, it includes taking an appropriate book from the shelf to find information on the mechanism that needs to be fixed.

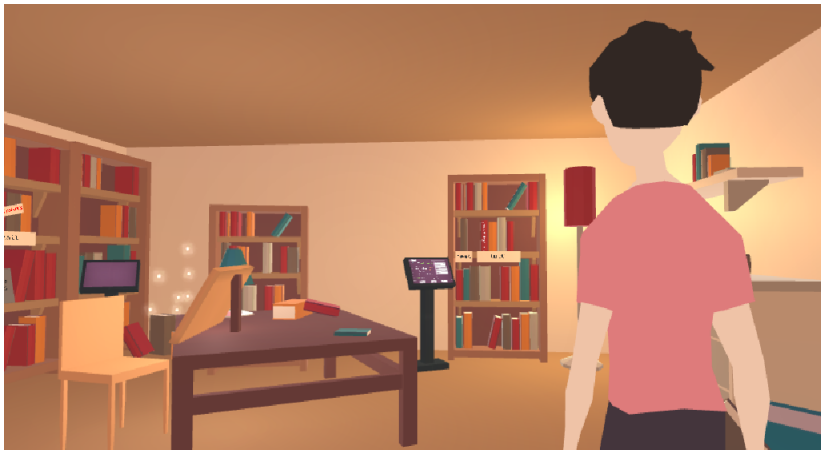

Figure 3.5: The Reading Room

The third one is a dungeon and underground laboratory. The player needs to solve more puzzles to find a way out. Puzzles in this level are about list comprehensions and basic aspects of recursive functions.

Figure 3.6: Dungeons and lab location

The final location is Lambda Tower, which includes highlands above the laboratory and a lighthouse. It only has two challenges to solve, including the final one.

Figure 3.7: The Lighthouse – Lambda Tower

The top-down look of the designed map, and its in-game representation, can be seen in Fig. 3.8.

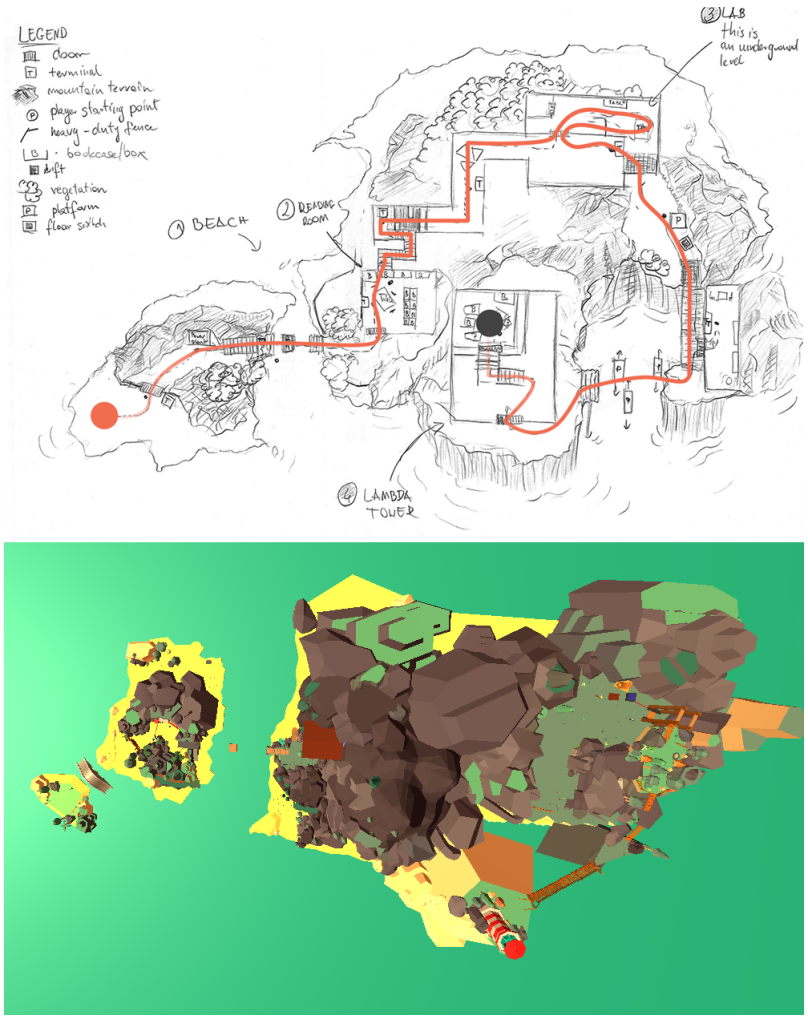

Figure 3.8: Top – early-design world map, bottom – top-down view on the game world.

In our design, we decided to place a strong emphasis on the constructionist approach that is constrained by the domain of the puzzles. First of all, we paid a great attention to the representation of programming concepts in the context of world objects and their attributes. We believe that connecting the real-world examples to abstract programming ideas will ease the learning process. Secondly, the feedback on the correctness of provided solutions is provided by visual effects and changes in the game state, aided with information pop-ups on the Terminal. Finally, we made exploration of the world an important part of the solving process. HaskellQuest focuses on a problem-solving approach that can be seen in commercial puzzle games such as the Witness[8], Portal 2[16] or Obduction[61].

As we decided to represent programming concepts using real-world metaphors, implemented puzzles are resemble imperative programs, as they change a state in the world instead of returning new states of values. These concepts are too abstract to be easily represented using real-world phenomena, so in our work we just present concepts of lists, data types and functions by their usage in this context, not exactly as they would be used in the functional programming paradigm. Originally, we wanted to represent changes in objects not by changing their state but by returning new objects with new characteristics produced by applying functions. Unfortunately, the time available for the implementation of puzzles was limited and we decided to focus on presenting more ideas that can be built together instead of relying only on a few aspects being repeated throughout the playing session.

3.5 Game interaction

3.5.1 Terminal

The Terminal is a core mechanism of our game to interact with the world and provides a way to learn Haskell. It is a represented as a standalone terminal game object that is placed in the front of each puzzle. A user can interact with it by aiming at it with a cursor and pressing the 'E' key, zooming into a code screen. This script is incomplete and the player needs to fill it in with provided answers and run code to change the game state. If there are any errors identified, the Terminal shows a message that resembles the compilation error from the Haskell compiler.

We decided to embed the task of programming as an in-game activity, as we believe it is likely to increase immersion[10] and enrich the gameplay. After establishing

Figure 3.9: Terminal concept art and the in-game view

the first design decision, we had to consider how a player will input the answers on the Terminal. The first option was a text-based solution. A user needs to write short snippets of code which are evaluated by an interpreter. The second one was block-based programming where code is represented visually with blanks to be filled-in with ready-to-use code blocks. We decided to implement the latter as it was indicated that such visualisation eases the learning process and makes it more enjoyable[60]. Weintrop and Wilensky claim that "students in the blocks condition showed greater learning gains"[60] compared to their colleagues working in a text-only environment. Moreover, O'Neil et al. in his work argued that "effective problem solving in games can place a large cognitive load on working memory."[40] and supported this claim with Cognitive Load Theory, proposed in Sweller's paper[51]. This is a theory which claims that a short-term memory has a limited capacity, implying that the amount of information that can be processed at one time is restrained by the cognitive potential of a person. Projecting that to our case, we aim to limit the number of presented topics and activities during our teaching process to avoid overwhelming a newcomer with new information. However, this is not a flawless approach. We decided to lower the cognitive load, thus relaxing the requirement of memorising language syntax and moved the emphasis of learning and recalling the material to recognising familiar concepts and reoccurring patterns. It is likely that our participants would have difficulties in writing error-free code without any guidance but would still being able to read Haskell listings. Still, we decided to teach people abstract ideas, thus we think that issues of syntax ambiguity should be omitted for some time.

The design decision of using blocks of answers allowed to implement the interpreter only as the part of the game, which substantially simplified the implementation of the puzzle creation and processing framework, explained in chapter 4. Furthermore, we gained full control over the possible outcomes of the challenges, so we were able

to design and provide tailored feedback to the user's actions.

3.5.2 Interactive world elements

Puzzles in the game are presented using a set of objects, whose attributes can be changed by scripts running on the Terminal. A certain group of them directly represents concepts existing in programming, such as a bookshelf of book stacks can represent a list that stores values of the same type, namely books. These elements can be values on their own or serve as building blocks in other entities such as algebraic data types. Additionally, we have in-game objects that are only used to implement concepts such as function composition, which can be implemented through pipes being connected by a connector that changes some characteristics of them. Additionally, we have other sequences that represent lists, such as fuse boxes or sequence of platforms.

Elements of the world are as following:

1. **Books** – a unit item that can be found in a list. Books are used to teach the idea of accessing the elements of lists using head/tail functions. Also **Fuse** values in fuses box are used in this way.

2. **Bookshelves/Stacks** – a container that stores a list and usually is defined as a custom data type. In order to get access to the list, a user needs to apply a special function.

3. **Circuits** – a representation of a function composition. From the high-level perspective, functions act as logic gates and return some results on conditions.

4. **Platforms** – in puzzles which have multiple platforms, they are a part of a bigger construction, for example a step in stairs.

5. **Bookcases/boxes** – objects that can be moved in four directions – forward, backward, left, right – and travel a particular distance, defined in the puzzle..

6. **Moveable platforms** – flat floating objects that can be used by a player to travel horizontally.

7. **Lift** – a mechanism built on top of a **moveable platform**, which allows a player to travel vertically up to a defined height.

8. **Ground switches** – buttons that are placed on the ground and can be pushed only with programmed objects.

9. **Programmable doors** – doors that open on a correct solution, such as fixing a power mechanism or providing a good password.

3.5.3 Puzzles

The game revolves around solving problems on the way to the final goal of the game. Player can observe his or her progression only by the number of finished puzzles and how far they are from the final destination. We do not provide points or any other system for awarding a user as it is believed that this may interfere with the learning experience[4]. Puzzles are heavily embedded in the game world and the player can progress from one to another by solving all previous challenges. Additionally, puzzles were designed with an idea of gradual progression, where concepts presented in one puzzle are used in following puzzles to reinforce recall of the material.

The matrix of concepts covered by each puzzle in the game is presented in Fig. 3.10. Its appearance is based on the matrix of concept progression in Arawjo et al.[5].

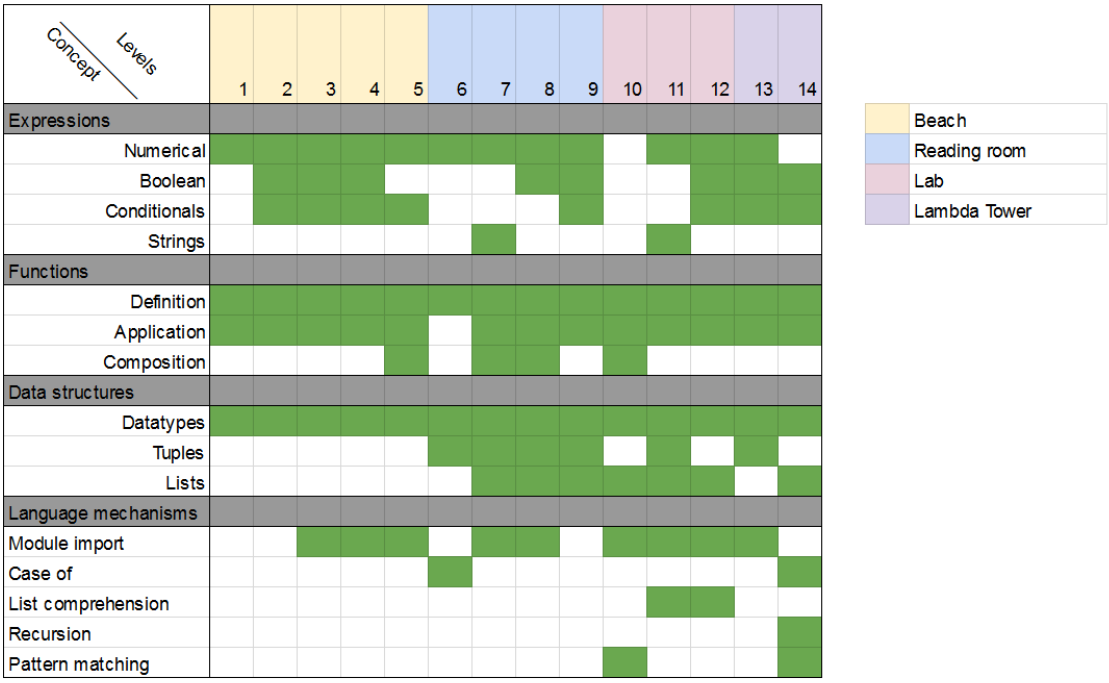

Figure 3.10: Puzzles concept coverage. Levels correspond to game challenges.

Chapter 4

Implementation

4.1 Methods

Implementation of the game was carried out using the Unity game engine[57], a special software environment designed for developing games. This solution provides full support for the rendering process, physics simulation, collision detection, and more, and its usage significantly reduces time spent on implementing a game. Furthermore, the Unity engine features a platform of ready-to-use objects and scripts developed by a community called the Unity Asset Store[56]. We decided to use several packages from there to enhance graphics and gameplay, as we believe it contributes to general game enjoyment and immersion. The list of used assets can be found in Appendix A.

4.2 High-level architecture

The architecture of our project can be divided into two parts – definitions of game-specific actions and interactions, and a framework for puzzle development. The first part consist of components handling user input, player's actions and camera behaviours, defined using Unity-supported scripts. Behaviours, as they are called, are C# scripts that are loaded by the game engine and attached to objects in the game scene. They support creating actions that can be triggered on the object's state change, for example entering a collision with other object. Definitions of the player's movement and camera behaviour are modified scripts from the Standard Assets package[55], which significantly reduced time spent on general-purpose code.

In our work, we focused on developing an abstract way of representing a puzzle, both in terms of game objects and scripts describing actions triggered by code on a

Terminal. Puzzles consist of interactive objects, stored in an object with **PuzzleController** behaviour, and a Terminal, that accepts and validates data provided by the user. Apart from storing interactive objects, a PuzzleController provides the configuration of a Terminal display, a **PuzzleConfiguration** and the description of the puzzle itself, **PuzzleScript**. The first script is used in the initialisation phase to temporarily store answers that can be placed in code blanks. A PuzzleScript, on the other hand, is responsible for building code, running the provided script and checking the final outcome. The "Building" option refers to checking if the code inserted in the blanks is valid, thus if they contain no syntax errors and are filled-in, whereas "running" means iterating over a script defined in a configuration file. This class maps entities of PuzzleConfiguration to methods to which they refer, and later maps methods to objects on which they are invoked. Relationships between Puzzle components can be seen in Fig. 4.1.

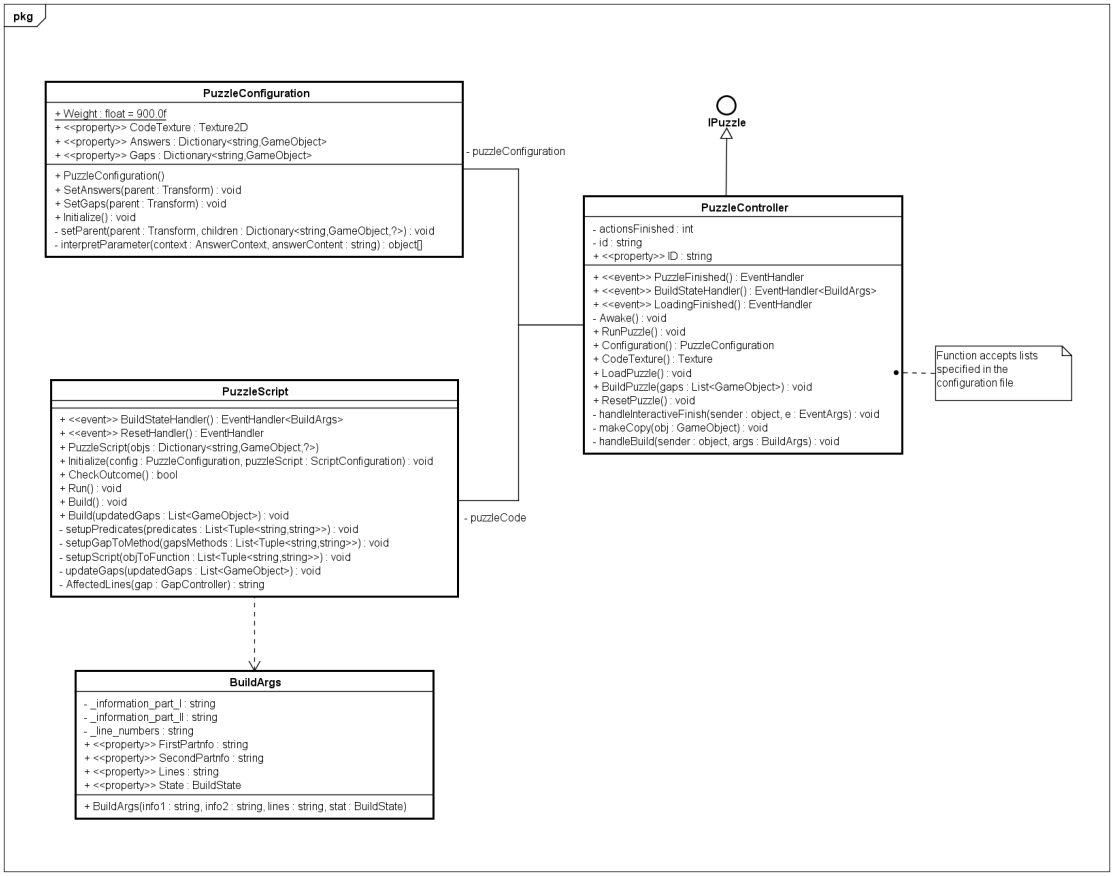

Figure 4.1: Puzzle-specific scripts

In order to make an object programmable, it needs to have a script implementing the **IScriptable** interface. This allows to apply functions as specified in PuzzleCon-

troller, revert state to that before running the code on Terminal, signal the end of action and check if the object has reached the accepted state. Specific controllers implementing that interface can be reused in different puzzles and game objects, given that they provided the required components. A sample controller is shown in Fig. 4.2.

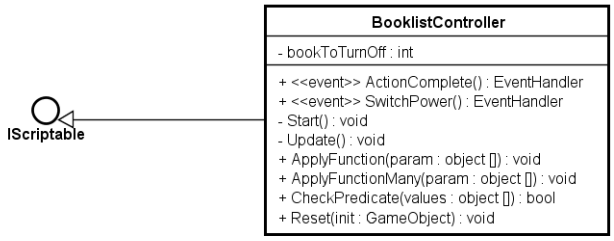

Figure 4.2: An example of a component implementing *IScriptable* interface

All puzzles and terminals are observed by a **GameState** script, which is attached, together with **LoadManager**, to an empty game object. This pair of scripts is designed to enhance communication between connected puzzles and handle the initialisation process. The first one also listens for user interaction with Terminals and adjusts the camera in TerminalMode and is just a helper class for all puzzles. The second one is responsible for loading the puzzle configuration, specified in an external file.

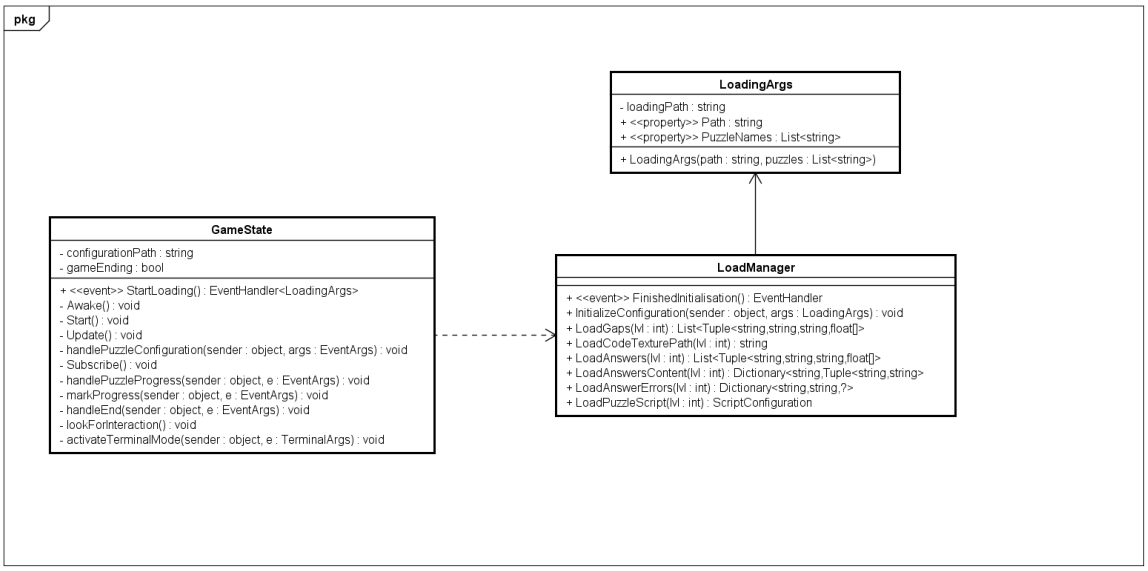

Figure 4.3: GameState and LoadManager scripts

4.3 Puzzle creation framework

To provide a fast and scalable solution for creating puzzles, we developed a special system to support object creation from a script instead of using the Unity editor. The configuration for every puzzle is stored in a table, named after a game object with a PuzzleController script attached. Inside that table, there are following arrays:

1. **gapList** – a list of code blanks, where each of them is specified by its name, its coordinates on the Terminal screen and the expected context of the answer. This implementation assumed no rigid connections between answer options and gaps, which caused some ambiguity described more in the following chapter.

2. **answerList** – a list of answers, defined by its name, coordinates, their syntax correctness and displayed text.

3. **answerContent** – a list that maps answers defined in an earlier list to their contexts and actual content. The content interpretation depends on the contexts of gaps and answers. For example, for a puzzle involving movement of an object, the answer and matching blank will be of a MovementDirection type.

4. **answerErrors** – in the case when some of the answers are marked as syntactically incorrect, a pop-up message will appear on the Terminal. This list defines the contents of that pop-up.

5. **gapToMethod** – a list that maps the contents of a code blank to an IScriptable method that will take it as an argument.

6. **paramsToPredicate** – as some puzzles are considered as solved only under specific conditions, this list provides a list of accepted values for interactive objects. These may include the distance travelled by a platform or a calculated value as a part of a puzzle.

7. **code** – a list that defines a sequence of executed functions together with the name of the object on which it will be executed.

8. **codeTexturePath** – a relative path to a texture of puzzle code showed on the Terminal, as the code displayed on the Terminal is a PNG file loaded from *Resources* file.

The contents of these arrays closely resemble the components in PuzzleScript. Despite the fact that this approach allows to freely define puzzle and Terminal contents, it has a major drawback. If a Unity user needs to use files from a particular directory, it needs to be copied together with a game binary file in order to work. This means that every person holding a copy of a game is able to freely modify configuration file, which may lead to uncontrolled changes in the game or to an application crash. For future work, we propose usage of binary format to eliminate the possibility of direct changes on the client side.

The decision to use images of the code instead of the Unity-supported Text feature was made due to technical issues of text fonts. Later on in the game development, that issue was fixed. Unfortunately, implementation work was too advanced to introduce such change, thus we continued development with the initial approach.

The second script used for puzzle initialisation, the PuzzleConfiguration class, is responsible for handling creation of answer and blank game objects. It uses two specialised classes – **AnswerFactory** and **GapFactory** – to create instances of answers and code blanks and attach scripts defining their behaviour – **AnswerController** and **GapController** respectively. As both classes share many similarities, it was decided to abstract out the creation of the initial game object to the **GeneralFactory** class. Its work is defined by **ObjectConfiguration** which specifies general properties such as background colour, size and position of a component. The output of a general factory is further modified by concrete factories of answers and code blanks. The connections between PuzzleConfiguration and factories are presented in Fig. 4.4.

4.4 Terminal mechanism

The Terminal provides a way for users to interact with the game world. The mechanism is build up from three parts, where each of them communicates with another using event handlers. These are provided by the native C# event system[13]. Scripts used for defining the Terminal mechanism are as follows:

1. **TerminalState** – a logic part of the Terminal that links puzzle and Terminal display together and passes information from one script to another. It is subscribed for events in **TerminalInteraction** and when one is invoked, it triggers **PuzzleController** to run its own script.

2. **TerminalInteraction** – a helper function that stores functions triggered by *OnClick*

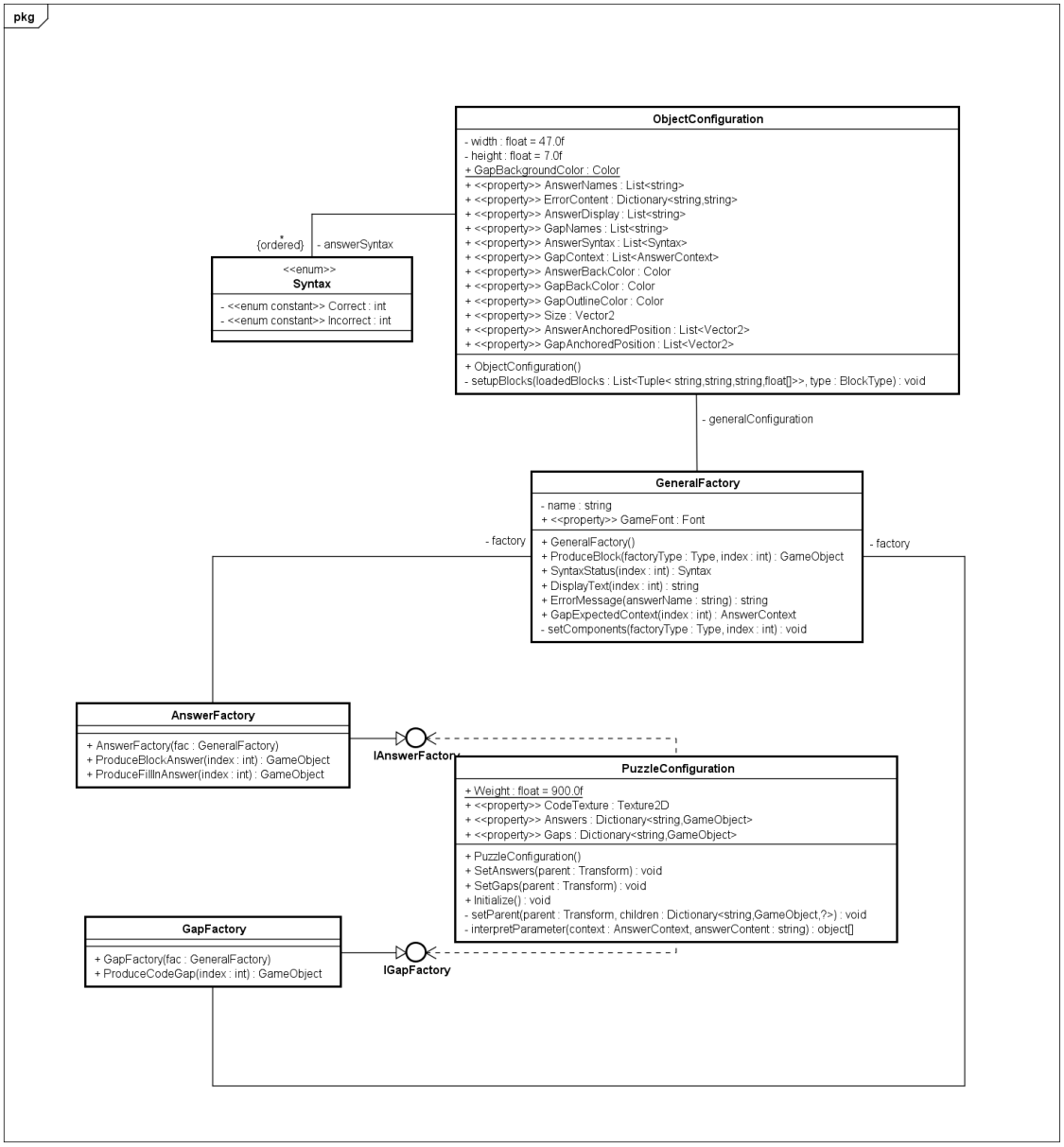

Figure 4.4: Terminal elements factories

events of on-screen buttons.

3. **TerminalUIState** – a script responsible for visual changes on the Terminal. It stores references to blanks and answer objects and runs the function for changing components position or the drag-and-drop feature. The mappings between these objects are set in a template object, defined using the Unity editor.

In order to interact with the Terminal, a user needs to point in the direction of an instance he or she wishes to use. If a player presses a specific key when the cursor is pointing to a Terminal, **GameState** will set a variable **TerminalMode** to *On*. This

value triggers a camera view change and stops a player object. The dependencies of the described components are shown in Fig. 4.5.

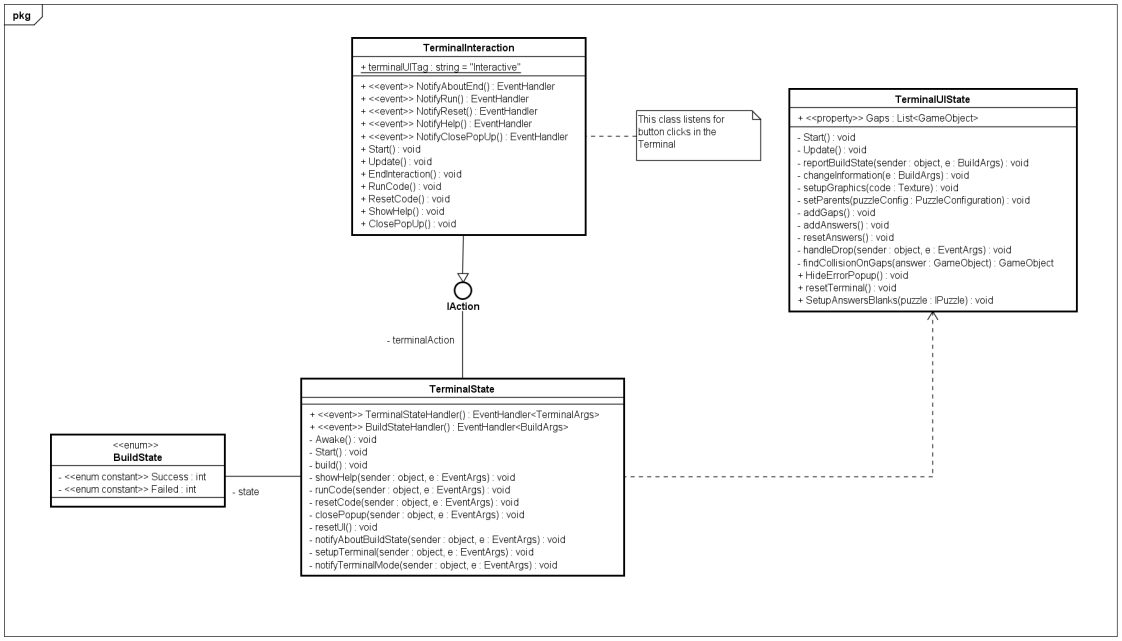

Figure 4.5: Terminal components.

A player can interact with a terminal using a mouse to drag answers into blanks and scroll the Terminal content. He or she can check their answers by pressing the “Run” button, triggering a sequence of actions for code checking and running. After all of these steps, the control over puzzle state comes back to PuzzleController, which informs Terminal about the final outcome.

Although the significant part of that mechanism is working correctly, we have identified an issue with accepting some answers that, from Haskell language perspective, are not correct. This is connected to a problem with describing answers and blanks with an enum, that serves both for parsing and code correctness checking. In future work we would like to separate this into two independent entities, a generalised one for parsing, and a specialised type for error checking.

4.5 Game initialisation

Initialisation of our game is based around two mechanisms – initialisation functions provided by the Unity engine and the event system constructed around puzzle and terminal creation. There are two functions that can be used for configuring game objects:

1. **Awake** – a first function called right after loading the scene and objects in it.

2. **Start** – a function called before the rendering of the first frame, after applying all game engine initialisation scripts.

As functions are executed independently on each object in the scene, we needed to introduce our own system for synchronisation between loading components, **GameState** script and puzzles. We decided to use again the native C# event system and define event handlers to signal request configuration loading, *StartLoading*, and finishing loading from the file, *FinishedInitialisation*. Event handlers for loading purposes are defined in GameState's the *Awake* function. PuzzleController, on the other hand, uses the *Awake* function to find children objects that have an IScriptable script attached and subscribes to their *ActionFinished* event handlers. Additionally, it instantiates an event for building information passed from its script.

The great majority of scripts use *Start* for initialisation, including interactive puzzle components, GameState and Terminal scripts. Before rendering the first frame, GameState subscribes to UI and Camera events and locates puzzles with their Terminal objects to initialise them. At that point, GameState invokes an event to begin configuration loading and waits for its output.

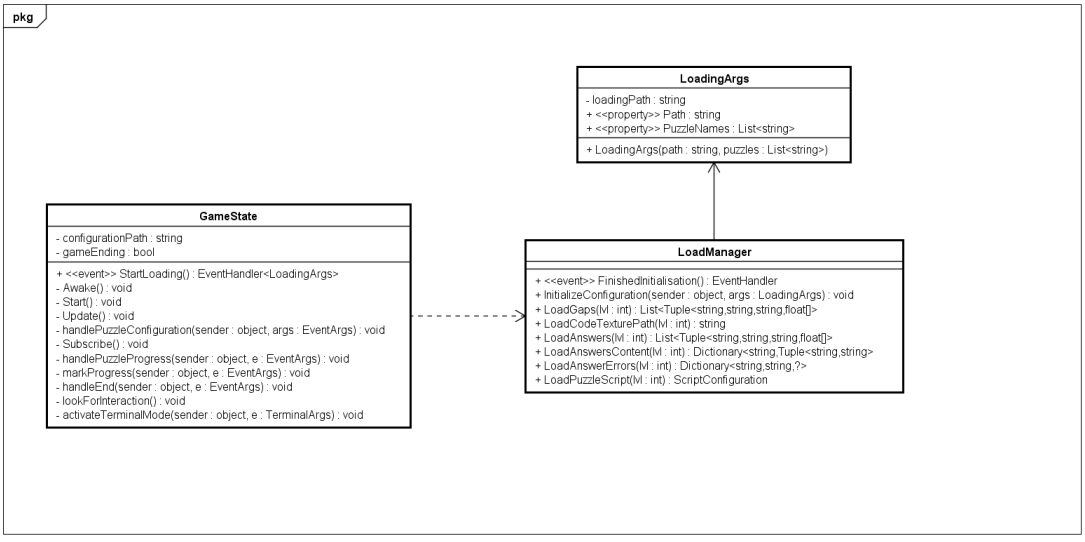

Figure 4.6: GameState and LoadManager scripts

A file used for puzzle configuration uses TOML file format[42], that is designed to be easy to write and understand and has established libraries in many programming languages including C#. We chose the Nett[1] parsing library and wrote our

own class library to handle parsing the format of the puzzle file. That class library, **TOMLLoader**, is a singleton class that provides static methods for loading particular arrays from the file. Thanks to usage of the **ScriptComponent** enum value passed to *GetScriptComponent* method, we can load different arrays that have the same structure with a consolidated interface. A similar approach was used for loading the UI elements of a Terminal, using **BlockType** enum. The dependencies in TOMLLoader library can be seen in Fig. 4.7.

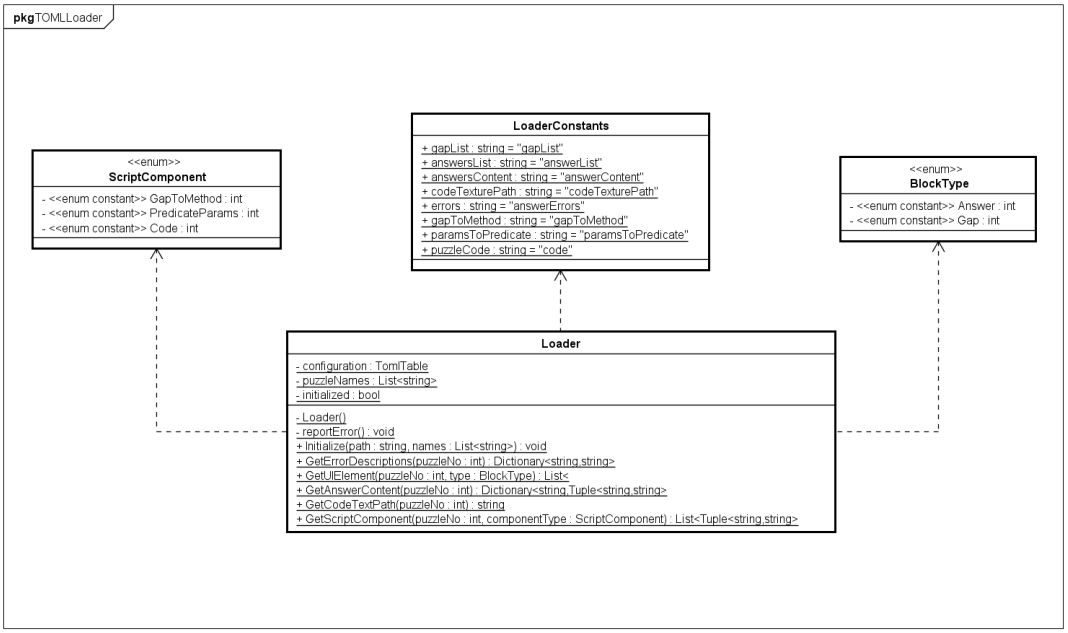

Figure 4.7: TOMLLoader class

4.6 Game loop

The runtime behaviour of our application is defined by a constant iteration of three major actions – reading the user's input, updating the state of the game logic and rendering the scene popularly known as a *game loop*[39]. The sequence of actions happening in our game is defined by the implementation of Unity engine as a sequence of updates followed by scene rendering phase[54]. Every object defined in the game can be updated using three kinds of updates:

1. **FixedUpdate** – this function updates the game logic multiple times per frame. In our implementation, it is used for physics calculations and updates of the camera position.

2. **Update** – the most widely used form of a state update, done once per frame.

3. **LateUpdate** – additional update function that gets executed after *Update* finishes.

As the order in which objects in the scene execute each update function is unpredictable, so the following description of our game loop assumes no connection between objects. Update begins with a call of *FixedUpdate*, which is defined for a script of a player object, accordingly known as **ThirdPersonUserControl** and **ThirdPersonCharacter**. The first one handles user input and translates to ground movement, the second one updates the physics component according to the change of the first one. Other scripts that get updated during this phase are moving **IScriptable** controllers and camera scripts responsible for following the player and zooming in on **TerminalMode** turned on by the user-Terminal interaction.

The second phase of game state update is a single update function that gets executed for **GameState** and **TerminalInteraction**, which work together on detecting a request for interaction with a Terminal. Inside this Update part, a special type of functions get executed – coroutines. They address a major drawback of the Unity engine design where every function executed during runtime has to finish its work during one iteration of an update. Coroutines are able to pause and resume execution between **Update** calls, so they can be used for optimising the number of function calls during runtime or provide a way of smoothing graphics effects. Here, we used this feature to implement a timed checker of player position and fade in/out GUI effects. After finishing these processes, Unity internally updates the animation, tries to execute **LateUpdate** if defined, and moves on to scene rendering.

Chapter 5

Evaluation

5.1 Methods

The "HaskellQuest" game was evaluated with respect to user experience and learning outcomes. The first part focuses on feelings of a player during gameplay and how effectively we maintain a players' "flow" state and foster their engagement. In the second part we test the programming knowledge of users before and after the playing session and to see if there was some improvement afterwards. The evaluation in based on qualitative data gathered from three on-line surveys: programming knowledge pre- and post-test and game evaluation. The knowledge tests consist of eight to nine multiple choice questions testing understanding of programming concepts using short Haskell code snippets. The evaluation of the playing experience contains eight questions which are statements with which a user can agree or disagree using five point rating system. Used statements describe different aspects of "flow" experience, which are based on the EGameFlow[22] scale, and motivation for learning more about functional programming.

Unfortunately, as this is a self-report questionnaire, we might have problems with data validity. For example, participants might have problems with recalling how they felt during the gameplay if there was a long pause between playing session and filling in the survey. Moreover, the declared confidence in programming skills might be different than that in reality, as our results showed a discrepancy between what a user feels about his or her skills and their post-test performance.

In our evaluation, we focused on gathering quantitative data to ease the process of comparing knowledge on functional programming before and after the experiment. Furthermore, a limited scale for assessing the game allowed us to reason about general

satisfaction from playing the game and see the trends in motivation towards learning about the functional paradigm. However, limiting the number of possible answers may not give us in-depth information about the perception of the game. To address this issue, we added an open, voluntary question to leave comments about the game. We consider this to be successful as the overwhelming majority also answered that question. The full questionnaires, together with their answers, can be seen in Appendix B. A substantial part of the numeric data presented in the following section is based on reports generated on SurveyMonkey[49] platform.

Initially, evaluation of the "HaskellQuest" game was planned to be done on much larger group of incoming Informatics undergraduate students. This big group, which would consist of about fifty participants, would allow us to verify if a change in knowledge after playing our game was big enough to claim that the playing session helped in learning basic functional programming concepts. Unfortunately, the planned arrangements for communicating with incoming students via the ITO fell through shortly before the evaluation was due to start and we had to perform evaluation on a smaller group of volunteers instead.

The testers in our evaluation were anonymous, identified only by nicknames to enable their pre-test/post-test/evaluation questionnaires to be matched. The process for gathering data was approved by the Informatics ethics process and, since the personal data collected was anonymous, was GDPR-compliant.

5.2 Learning outcomes

The evaluation of learning outcomes was carried out on a group of 8 participants with previous programming experience, where only one of them had used Haskell before. This was a mixed group of three incoming undergraduate Informatics students together with five postgraduate volunteers who had a background in Computer Science. They are the only students who solved both pre-test and post-test questionnaires, out of a group of ten students who evaluated the playing experience. Participants performed very well in both quizzes, which was probably caused by the form of our questionnaires. They were aimed at people with no prior programming experience and tested recognition of Haskell structure and reasoning about what might be described with that language. Moreover, students rarely took the "I don't know" option and tried to understand the code snippets in questions. The mean score in the first test was 91%, with the lowest score of 80% and the highest of 100%. The results for this test can be seen

below in table 5.1.

Mean	Median	Standard deviation
91%	90%	8%

Table 5.1: General pre-test scores

The majority of questions were answered correctly, with a few exceptions. Surprisingly, the hardest question was that on conditional expressions – only 50% of participants answered it correctly. The question asked about the outcome of a correctly formatted expression evaluating to true, and 25% of students choose the answer "There will be an error as baz doesn't know foo definition". This shows that half of the participants weren't aware of the declarative aspect of the Haskell language and might assume its behaviour by analogy with C/C++, where each function must be at least declared before it can be used. Two other questions that users had problems with were about list comprehension and operations on lists. There was only one incorrect response to each of them, where the one for lists comprehension indicates misunderstanding of the concept, while the second seems to be an oversight while reading the snippet. Specific information on troublesome questions can be found in table 5.2.

Question	Percent correct	Average score	Standard deviation
5	50%	0.5/1.0	0.53
8	88%	0.9/1.0	0.35
9	75%	0.8/1.0	0.46

Table 5.2: Results for the most difficult pre-test questions

Participants were asked to complete a second tests after the playing session which was constructed in the same way as the first one was. The content of questions was more advanced than that in the pre-test and referred to concepts presented in the game challenges. Despite this rise in difficulty, the users generally improved their general score to 94% mean, with the lowest score of 75% and the highest of 100%.

Mean	Median	Standard deviation
94%	96%	9%

Table 5.3: General post-test scores

Performance on this test was almost the same as in the pre-test – only three questions were harder for participants. The first one, which was considered the easiest from the question set, caused the biggest trouble for half the of students. Two of them weren't aware that a type error will be caught at compile time rather than during runtime, where the latter is typical for dynamic typed languages. Moreover, other two chose the answer which suggested that Haskell would do dynamic type conversion and run the code despite the error. These answers indicate that maybe the idea of static typing was not introduced clearly in our game and this needs to be addressed in the next version of the game. The second question that confused one person showed a definition of an algebraic type, which could be seen in the previous test. The participant was not aware of naming conventions for programming concepts but still we consider mistaking data definition for function definitions to be a severe mistake which should be addressed by instructional content. The last troublesome question examined knowledge on accessing contents of data-type instances. One participant thought that function accessing a type field will return a new value of the storage type, instead of the value inside of it. The comparison between final scores for pre- and post-tests can be seen in Fig. 5.1

Question	Percent correct	Average score	Standard deviation
2	50%	0.5/1.0	0.53
4	88%	0.9/1.0	0.35
6	88%	0.9/1.0	0.35

Table 5.4: Results for the most difficult post-test questions

As a part of our evaluation, we compared results of pre- and post-test to verify if our solution contributed to per-user performance improvement. Overall, with the exception of one participant, the entire group improved their knowledge after the playing session. As the initial scores were very high, the mean relative increase was not as big as expected and is equal to 3% with standard deviation at 9%. The change was relatively small but approximately uniform among students, which we consider to indicate a success in our approach.

Mean	Median	Standard deviation
3%	1%	9%

Table 5.5: Per-user relative performance change.

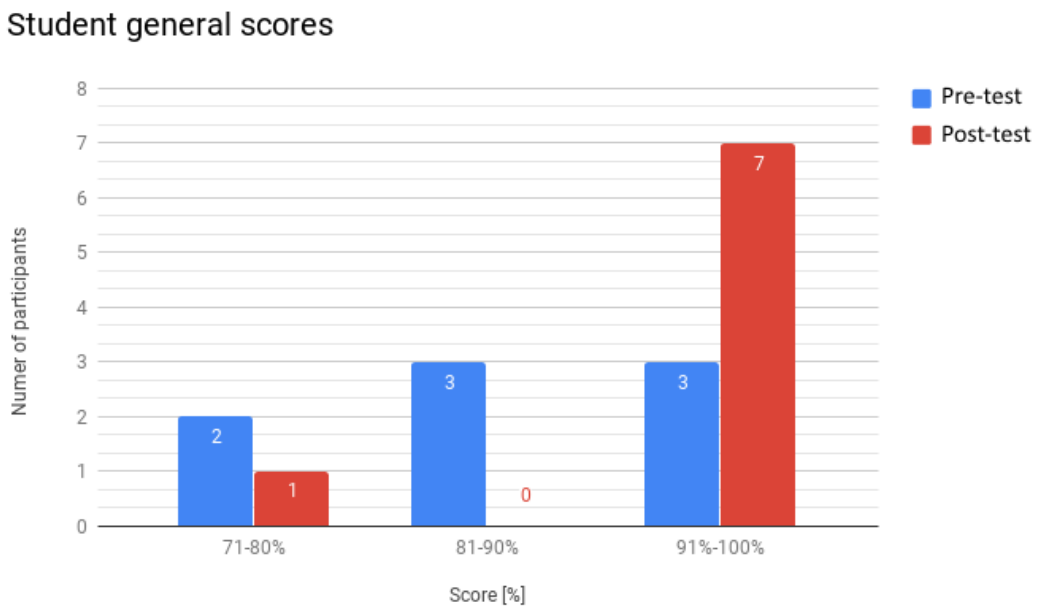

Figure 5.1: Pre- vs. post-test results.

A closer inspection of results of the student who performed worse on post-test showed an interesting phenomenon. In the first test, the participant struggled with questions on lists and list comprehension and had no problems with answering questions on algebraic datatypes. After the playing session, he or she excelled at list-related questions but made mistakes on these connected to datatype definitions and Haskell type system. These findings further suggest that "HaskellQuest" did not enforce understanding of static typing and only indirectly explained datatype definitions. Results of particular participants can be seen in Fig. 5.2.

Despite the fact that there is a notable change in test results, we cannot claim that these findings are statistically significant. The sample size was way too small to apply any statistic methods to confirm that playing "HaskellQuest" had a positive influence on the learning process. Moreover, we did not have a control sample to further assure that our results are reliable. Additionally, all participants had some experience with imperative programming which makes it hard to say how true programming novices would perform at these tests. In future work, we plan to test our solution on much bigger groups that are also diverse in prior programming knowledge to see the effectiveness of our approach on programmers of all experiences.

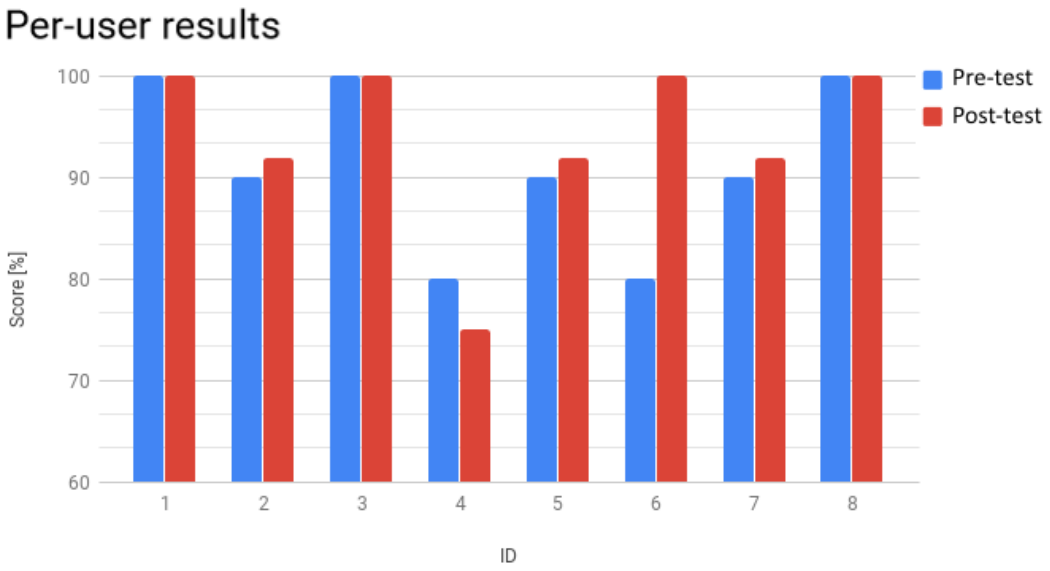

Figure 5.2: Pre- vs. post-test results for particular participants.

5.3 Game evaluation

As a part of evaluating our solution, we measured satisfaction among our participants from playing the game and attempted to ascertain if they were likely to enter the "flow" state. The user group consisted of ten people, where eight of them were tested for learning outcomes. Three out of ten completed whole game, with four others who stopped playing at the second-to-last puzzle. It is hard to speculate on the cause of such behaviour but it is possible that they got bored at some point or maybe they played the game only for evaluation purposes and felt that it had no more to offer to them. In future studies, we should also consider collecting data if participants do not finish the game. Furthermore, three users stopped playing in the first or second location, suggesting problems with understanding puzzles or technical problems that influenced their game perception.

As we explained in chapter 2.1, the feeling of "flow" consists of deep concentration on the task, seeing clear goals in the activity and working towards them and general enjoyment of performing these actions. The overwhelming majority of players were concentrated on the game for most of the time, with 60% of them seeing the goals of the game. Three participants could not say if they agree or disagree with this statement and only one of them had problems with clearly seeing the goals. Almost 60% enjoyed playing the game without feeling anxious or bored about encountered

challenges, which was further supported by positive comments on the game, such as “I really enjoyed playing it” or “It was fun, even though I sometimes wondered if I solved a puzzle”. The last comment refers to one of the identified flaws of our solution, which is the limited facilities for feedback.

The problem with feedback concerned the lack of information on how the actions of a player contributed to the change in the game world or if the answer was correct. It was mentioned in three of eight received comments that on an erroneous answer, the game would not provide hints or clear feedback on what went wrong. Unfortunately, because of time constraints, this feature was not fully implemented and it had a major influence on players’ experience. This issue is critical and should be addressed as a priority in future work.

The last aspect measured in this evaluation was motivation and perceived usefulness of functional programming. According to our findings, only one person was not interested in learning more about Haskell in the future, with others being strongly motivated to do so. We consider this to be a major success, as one of the users declared “ I want to learn Haskell, it brought my attention to this language”. Moreover, over 80% of users felt that they learned some basics of the Haskell language, with the same number of responses agreeing on the fact that knowing these basics helped in solving in-game puzzle.

Overall, players were generally happy about this learning environment, despite technical problems that were not addressed during development. Almost all comments mention problems with navigating through the game world, which is connected to faulty collision detection between the player and game objects. The standard movement controller attached to the player had special requirements that sometimes could not be met in our game. For future work, we would like to write our own implementation of that script, to provide a more pleasant playing experience.

5.4 Project performance

As a part of the evaluation process, we decided to do a self-report on the project performance. First we will discuss the project log which was maintained throughout the development process and compare it to the planned schedule for version 2.0 of the project. Accordingly, we will review if the project met the criteria for a successful project, formulated in our project proposal[19].

The first phase of our project focused on design of the puzzles, game world and

the Terminal mechanism. We decided to plan all puzzles to cover all topics planned for version 1.0 and make introduction of new concepts possible. It was not specified in the initial project plan but we wrote a design document describing the challenges, what mechanisms will they use and how they contribute to the story. This approach saved us much time during the implementation phase.

Subsequently, we moved on to programming basic game elements such as player movement, camera and game interaction. Here, we had to spend more time on configuring the camera than expected, which caused a delay in implementation by three days. Still, as we did more work in this stage on a design part, there was no serious delay in our development process.

The design decision to implement block-based programming system indirectly eliminated the biggest risk of our project, which was access to an in-game interpreter described in detail in our previous work[19]. The interpreter implemented for our purposes is embedded in-game only and is connected to the puzzle creation system, which was not included in our initial plans. The amount of time spent on implementation of that system was about eight days, comparable to what was planned for the initial system. The introduction of that automated system significantly reduced time spent on implementing puzzles and made the introduction of modifications flexible and simple, limiting it to editing the configuration file.

Overall, the project went smoothly over the time with minor interruptions for script corrections. Unfortunately, we underestimated the time needed for implementing a feedback system and building the game scene within the Unity editor, which resulted in problems identified by participants in previous sections. It seems that during our work we should have focused more on providing verbose instructions and understandable feedback on player's actions. Still, we were able to provide a working and complete solution within a very demanding time frame.

We can assess the performance with respect to our completion criteria, which were defined as follows[19]:

1. **Understanding of basic programming concepts** – a test conducted after a playing session suggests an improvement in understanding functional programming concepts. Unfortunately, the participants did not feel sure about their skills and perceived them to be low.

2. **Game enjoyment** – we measured game enjoyment by asking questions on participants' concentration, engagement and feeling about enjoyment levels. De-

spite technical difficulties occasionally appearing in the game, the majority of players enjoyed the game and express interest in learning more about Haskell. We therefore consider this criterion to be fulfilled.

3. **Time to complete tasks** – we did not implement a data collection system in the game as it was decided that quantitative data from surveys would be sufficient.

4. **Version 2.0 goals** – we covered all functional paradigm concepts except for lazy evaluation because time needed for puzzle creation framework implementation had been underestimated. Moreover, the feedback system was not fully implemented as we had to focus more on implementing puzzles. As the lack of a more helpful feedback system was problematic for participants, we consider that we partially succeeded in that aspect.

Chapter 6

Conclusion

This project was undertaken to develop a game-based learning environment for teaching functional programming and evaluate its influence on learning outcomes and students' engagement. The experiments with participants show that this approach sparks an interest in learning more about Haskell and positively influences the learning process. Students generally enjoyed the learning sessions and were focused on solving the puzzles. However, the implementation had certain limitations that significantly affected this activity. We identified that the most influential factors were: over-simplified feedback system, unbalanced difficulty between puzzles, and game-specific flaws such as imperfect collision detection.

The analysis of knowledge tests before and after the playing session revealed that users have improved their scores after the playing session. This might suggest that "HaskellQuest" contributes to understanding of basic concepts in functional programming and reinforces recognition of Haskell syntax. Nevertheless, we cannot say how big the influence of our solution was because of the limited evaluation of learning outcomes. The sample size was too small to apply statistic methods in order to infer if this positive change was statistically significant thus a deeper analysis is necessary.

The main topics that should be addressed in future work include refinement of the feedback system, fixing bugs and difficulties identified during experiments with users, and lowering the difficulty of implemented puzzles. Work on the feedback system would include introduction of a clearer indication of failure, introduction of a system of hints on the Terminal screen and a clearer presentation of goals in the game. Ideally, this process would involve regular testing with potential players to receive feedback on our ideas and find potential problems during runtime. We would adapt our approach based on their suggestions expressed after the playing session. Additionally, we argue

that providing additional teaching material, not only this embedded in the game world, would also positively contribute to the learning process. We identified that some concepts of functional programming are too complex to be explained using metaphors and require some commentary or hints to help the user achieve an accurate understanding.

Despite the project's limitations, we believe that this study sheds more light on game-based learning for teaching functional programming. This non-standard form of learning seems to be an attractive introduction to Haskell, whose academic reputation discourages even seasoned imperative programmers, not to mention novices. It is suggested that this way of presenting the concepts of the functional paradigm, coupled with possibility of applying new knowledge in practice, is worth exploring further. Games for educational purposes can strongly support standard ways of teaching and invoke motivation for learning concepts in-depth, what is an acknowledged problem with teaching programming.

Appendix A

List of used game assets

1. *Standard Assets*, author: Unity Technologies. https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351; Accessed 15 Aug 2018
2. *Liam — Stylized character*, author: Akishaqs. https://assetstore.unity.com/packages/3d/characters/humanoids/liam-stylized-character\-100007; Accessed 15 Aug 2018
3. *FREE Skybox – Cubemap Extended*, author: Cristian Pop. https://assetstore.unity.com/packages/vfx/shaders/free-skybox\-cubemap-extended-107400; Accessed 15 Aug 2018
4. *3D Scifi Kit Starter Kit*, author: Creepy Cat. https://assetstore.unity.com/packages/3d/environments/3d-scifi-kit-starter-kit-92152; Accessed 15 Aug 2018
5. *Old Telephone*, author: Rokay3D. https://assetstore.unity.com/packages/3d/old-telephone-62434; Accessed 15 Aug 2018
6. *Pipes Kit*, author: mojo-structure. https://assetstore.unity.com/packages/3d/props/industrial/pipes-kit-64170; Accessed 15 Aug 2018
7. *Simplistic Low Poly Nature*, author: Acorn Bringer. https://assetstore.unity.com/packages/3d/environments/simplistic-low-poly-nature-93894; Accessed 15 Aug 2018
8. *Low Poly Survival Game Props*, author: Timothy Porter. https://assetstore.unity.com/packages/3d/props/low-poly-survival-game-props-107984; Accessed 15 Aug 2018
9. *Low Poly Game Kit*, author: Matthias Pieroth. https://assetstore.unity.com/packages/templates/packs/low-poly-game-kit-110455; Accessed 15 Aug 2018
10. *Simple Home Stuff*, author: Mohammed Nafiz. https://assetstore.unity.com/packages/3d/simple-home-stuff-69129; Accessed 15 Aug 2018
11. *Simple Home Stuff*, author: Mohammed Nafiz. https://assetstore.unity.com/packages/3d/simple-home-stuff-69129; Accessed 15 Aug 2018
12. *Low Poly: Free Pack*, author: Axey-Works. https://assetstore.unity.com/packages/3d/environments/low-poly-free-pack-58821; Accessed 15 Aug 2018
13. *Sci-Fi Styled Modular Pack*, author: Denis Valcu. https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-styled-modular-pack-82913; Accessed 15 Aug 2018

14. *Nature Pack (Extended)*, author: Kenney. https://assetstore.unity.com/packages/3d/environments/landscapes/nature-pack-extended-66146; Accessed 15 Aug 2018
15. *Blueprint texture*, author: Pixelmator Ninja. https://gumroad.com/l/reqp; Accessed 15 Aug 2018
16. *3D low poly Lighthouse model*, author: GizArt. https://www.turbosquid.com/3d-models/3d-lighthouse-corona-model-1153070; Accessed 15 Aug 2018
17. *Low Poly Gear 3D*, author: Nix Interactive. https://www.turbosquid.com/FullPreview/Index.cfm/ID/1262631; Accessed 15 Aug 2018

Appendix B

Questionnaires

B.1 Pre-test

1. **Provide your nickname**

   _____

2. **Select all programming languages you have used so far.**

   - None, I haven't tried programming before
   - Java
   - C/C++
   - C#
   - Visual Basic
   - Haskell
   - Other (please specify)

3. **What is the outcome of running this code snippet?**

```
bar = 2.0
foo = 8.0 + bar
```

   - foo will be equal to 8.0
   - there will be an error as we add bar to a number
   - foo will add bar to 8.0, so it be equal to 10.0
   - I have no idea

4. **What is the outcome of running this code snippet?**

```
bar = 2.0
foo = if bar > 2.0
  then "That's a good bar"
  else "That's a problem"
```

   - foo will be equal to "That's a problem"
   - foo will be equal to "That's a good bar'
   - there will be an error as there are no parentheses around "bar >2.0"
   - I have no idea

5. **What is the outcome of running this code snippet?**

```
baz = foo 2

bar = 2
foo baz = if bar == 2
      then 1
      else 0
```

   - baz will be equal to 0
   - baz will be equal to 1
   - there will be an error as baz doesn't know foo definition
   - I have no idea

6. **What is defined in the code snippet?**

```
data SteeringWheel = UsualWheel
     | PremiumWheel

data Vehicle = Vehicle{
   numberOfWheels :: Int
 , engineWorking :: Bool
 , steeringWheel :: SteeringWheel}
```

   - SteeringWheel that can be a UsualWheel but it does not compile because of |

- SteeringWheel that can be either UsualWheel or PremiumWheel
- Vehicle that stores a number of wheels, information if engine is working and an object of a SteeringWheel type
- Vehicle that has an engine, numberOfWheels defined as a floating-point number and an object of a SteeringWheel type
- I have no idea

7. **What is the outcome of running this code snippet?**

```
bestList = [1, 2, 3, 4, 5]
bestElement = head bestList
```

- it defines a list of numbers called bestList and a list of top three elements of it in bestElement
- it defines a list of numbers called bestList and a first element of it in bestElement
- I have no idea

8. **What is stored in bestElement?**

```
bestList = [1, 2, 3, 4, 5]
bestElement = head
  (tail bestList)
```

- [2]
- 1
- 2
- I have no idea

9. **What is stored in evenBetterList?**

```
increment x = x + 1
bestList = [1, 2, 3, 4, 5]

evenBetterList = [increment y
    | y <- bestList]
```

- [1, 2, 3, 4, 5]
- it won't compile as we can't add 1 to the list
- [2, 3, 4, 5, 6]
- I have no idea

10. **What this function does?**

```
fact 0 = 1
fact n = n * fact (n-1)
```

- As it takes a parameter, checks on condition if it's zero and returns 1
- If a passed number is 0, it matches the first function and returns 1
- If the parameter does not match the first pattern, it multiplies it by passed number and output of a function of a smaller number
- This function keeps on executing forever and never stops
- It keeps calling the function until parameter is equal to 0
- I have no idea

B.2 Game Evaluation

Questions 4 - 9 have the same answers as in Question 3.

1. **Provide your nickname**

   _____

2. **What was the last puzzle you have solved in the game?**

   _____

3. **I was concentrated on the game for most of the time.**

- Strongly disagree
- Disagree
- Neither agree nor disagree
- Agree
- Strongly agree

4. **I was concentrated on the game for most of the time.**
5. **Goals of the game were clear to me.**
6. **I received feedback on success or failure of my actions immediately.**
7. **I generally enjoyed playing the game without feeling bored or anxious.**
8. **I feel I learned some basics of functional programming.**
9. **I think the things I learned about Haskell helped me in solving puzzles.**
10. **I would like to learn more about that language in the future.**
11. **Feel free to comment on your playing experience below.**

B.3 Post-test

1. **Provide your nickname**

---

2. **What is the outcome of running this code snippet?**

```
foo x y = x + y
bar = foo 10 "testTheType"
```

- bar will be a string value "10testTheType"
- An error will be detected after running the code
- There will be an error before running the code
- I have no idea

3. **What is the value of bar?**

```
check x y = x - y > 10

foo z w = if check z w
          then z + 1
          else z + w + 1

bar = foo 12 1
```

- 14
- 13
- 12
- I have no idea

4. **What does this code snippet define?**

```
data Bookcase = Bookcase{
  position :: (Float, Float)
  , height :: Float
  , width :: Float
  , numberOfShelves :: Int}
```

- It defines a datatype that stores position, described with two numbers, height, width and the number of shelves
- It defines a function that takes position, described with two numbers, height, width and the number of shelves
- It defined a datatype that stores position, height and width, all described with floating-point numbers
- I have no idea

5. **What is livingRoomTable?**

```
data Top = WoodenTop | MetalTop

data Table = Table {
   top :: Top
 , numberOfLegs :: Int}

woodenTable n = Table{
    top = WoodenTop,
     numberOfLegs = n}

livingRoomTable = woodenTable 3
```

- A definition of a Table value that has n legs
- A definition of a Table value that has wooden top and three legs
- A function that returns WoodenTop in a ‘top‘ value
- I have no idea

6. **What is the value of slabs?**

```
data Slab = Slab (Float, Float)
data Pavement = Pavement {
   isThereAcrub :: Bool
   , pavingSlabs :: [Slab]
}

example = Pavement {
  isThereAcrub = True
, pavingSlabs = [
     Slab (2.0, 2.0),
          Slab (1.0, 1.0)]}

slabs = pavingSlabs example
```

- A value of type Pavement
- A list of two values of a type Slab
- An empty list
- I have no idea

7. **What is the value of slabs?**

```
data Slab = Slab (Float, Float)

data Pavement = Pavement {
  pavingSlabs :: [Slab] }

example = Pavement {
  pavingSlabs = [Slab (2.0, 2.0),
  Slab (1.0, 1.0)]}

slabs = head
   (tail (pavingSlabs example))
```

- Slab (2.0, 2.0)
- Slab (1.0, 1.0)
- An empty list
- I have no idea

8. **What is the value of slabs?**

Note: Pavement and Slab are defined same as in (7) from now on.

```
example = Pavement {
  pavingSlabs = [Slab (2.0, 2.0),
    Slab (1.0, 1.0)]}

slabs = [makeBiggerSlab x
   | x <- pavingSlabs example]

makeBiggerSlab (Slab (x, y)) =
  Slab (w, z)
       where (w, z) =
          (x + 1, y + 1)
```

- [Slab (2.0,2.0), Slab (1.0,1.0)]
- Undefined, because the code will not compile
- [Slab (3.0,3.0), Slab (2.0,2.0)]
- I have no idea

9. **How slabs function works? What is its value?**

```
example = Pavement { pavingSlabs =
  [Slab (1.0, 1.0),
  Slab (1.0, 6.0)]}

slabs [] = []
slabs list = differentSize
  (head list) : slabs (tail list)

differentSize (Slab (x, y)) =
  Slab (w, z)
       where w = x + 2.5
             z = y + 1.0
```

- It stops executing on an empty list

- When it gets an empty list, it keeps on executing forever
- It gets a list, applies differentSize on its first element and is run on the rest of the list
- differentSize function is applied on the first element and added to the rest of the old list
- In the end, slabs is a list: [Slab (3.5,2.0), Slab (3.5,7.0)]
- In the end, slabs is a list: [Slab (3.5,2.0), Slab (1.0,6.0)]
- I have no idea
- In the end, slabs is a value Slab (1.0,1.0)
- I have no idea

10. **What is the outcome of running this code snippet?**

```
example = Pavement { pavingSlabs = [Slab (1.0, 1.0),
                    Slab (1.0, 6.0), Slab (2.0, 3.0)]}
hole = (1.0, 1.0)

slabs [] = []
slabs (x:xs) = if checkIfItFits x hole
               then x : (slabs xs)
               else slabs xs

checkIfItFits (Slab (x, y)) (width, height) =
                x <= width && y <= height
```

- If a slab fits, slabs saves an element and goes on with the rest of the list
- If a slab fits, slabs returns the first element of the list that fits the hole and stops
- slabs takes the first element of the list, checks if it fits the hole to cover
- slabs takes a list and matches against a pattern: first element - other elements
- In the end, slabs is a list [Slab (1.0,1.0)]

Appendix C

Puzzle case study code

In the following listing "?????" stands for code blanks to be filled in.

```
import PlatformUtil (stopFor2Seconds)

data Platform = Platform{
  horPosition :: Float }

data Direction = Backward | Forward

currentPlatform = Platform{
  horPosition = 8.0 }

totalDistance = 16.0
moveDistance  = 8.0

travel platform = movePlatform
   (movePlatform stoppedPlt Forward)
      ?????
       where stoppedPlt = stopFor2Seconds ?????

movePlatform plat dir = Platform{
  horPosition = ?????
           }
calcPosition plat direction =
   if direction == Forward
   then (horPosition plat) + moveDistance
   else(horPosition plat) - moveDistance
```

Available answers:

- Backward
- Forward
- Stop
- (movePlatform platform Backward)
- (movePlatform Platform Backward)
- (movePlatform platform Forward)
- calcPosition cPosition dir
- calcPosition plat
- calcPosition plat dir

Bibliography

[1] Nett – .net library for toml. https://github.com/paiden/Nett. Accessed 15 Aug 2018.

[2] Ernest Adams. Game worlds. In *Fundamentals of Game Design*, chapter 4. New Riders, 2009.

[3] Chris Allen. Functional education. http://bitemyapp.com/posts/2014-12-31-functional-education.html. Accessed 15 Aug 2018.

[4] Erik Andersen, Yun-En Liu, Richard Snider, Roy Szeto, Seth Cooper, and Zoran Popović. On the harmfulness of secondary game objectives. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, FDG '11, pages 30–37, New York, NY, USA, 2011. ACM.

[5] Ian Arawjo, Cheng-Yao Wang, Andrew C. Myers, Erik Andersen, and François Guimbretière. Teaching programming with gamified semantics. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 4911–4923. ACM, 2017.

[6] P. Backlund and M. Hendrix. Educational games - are they worth the effort? a literature survey of the effectiveness of serious games. In *2013 5th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pages 1–8, Sept 2013.

[7] Conrad Barski. No Starch Press, 2010.

[8] Jonathan Blow. The witness. https://store.steampowered.com/app/210970/The_Witness/, 2016. Accessed 15 Aug 2018.

[9] Elizabeth A. Boyle, Thomas Hainey, Thomas M. Connolly, Grant Gray, Jeffrey Earp, Michela Ott, Theodore Lim, Manuel Ninaus, Claudia Ribeiro, and João Pereira. An update to the systematic literature review of empirical evidence of the impacts and outcomes of computer games and serious games. *Computers & Education*, 94:178 – 192, 2016.

[10] Emily Brown and Paul Cairns. A grounded investigation of game immersion. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '04, pages 1297–1300, New York, NY, USA, 2004. ACM.

[11] Mark Brown. *What Makes a Good Puzzle? Game Maker's Toolkit*. https://www.youtube.com/watch?v=zsjC6fa_YBg&t, 2018. Accessed 15 Aug 2018.

[12] Douglas B. Clark, Emily E. Tanner-Smith, and Stephen S. Killingsworth. Digital games, design, and learning: A systematic review and meta-analysis. *Review of Educational Research*, 86(1):79–122, 2016.

[13] Microsoft Community. Handling and raising events. https://docs.microsoft.com/en-us/dotnet/standard/events/. Accessed 15 Aug 2018.

[14] Haskell comunnity. Haskell in education. https://wiki.haskell.org/Haskell_in_education. Accessed 15 Aug 2018.

[15] Tomorrow Corporation. Human resource machine game. http://tomorrowcorporation.com/humanresourcemachine. Accessed 15 Aug 2018.

[16] Valve Corporation. Portal 2. http://www.thinkwithportals.com/, 2011. Accessed 15 Aug 2018.

[17] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, March 1991.

[18] Marsh Davies. A good puzzle game is hard to build. https://www.rockpapershotgun.com/2015/01/22/how-to-make-a-puzzle-game/. Accessed 15 Aug 2018.

[19] Karolina Drobnik. *HaskellQuest: a game for teaching functional programming in Haskell – Informatics Project Proposal*. page 4, 2018.

[20] Peggy A. Ertmer and Timothy J. Newby. Behaviorism, cognitivism, constructivism: Comparing critical features from an instructional design perspective. *Performance Improvement Quarterly*, 26(2).

[21] D. P. Friedman and Mattias Felleisen. *The Little Schemer*. MIT Press, Cambridge, MA, USA, 1996.

[22] Fong-Ling Fu, Rong-Chang Su, and Sheng-Chin Yu. Egameflow: A scale to measure learners' enjoyment of e-learning games. *Computers & Education*, 52(1):101 – 112, 2009.

[23] Rosemary Garris, Robert Ahlers, and James E. Driskell. Games, motivation, and learning: A research and practice model. *Simulation & Gaming*, 33(4):441–467, 2002.

[24] Ray Giguette. Pre-games: Games designed to introduce cs1 and cs2 programming assignments. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 288–292. ACM, 2003.

[25] 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, pages 10–15. ACM, 2013.

[26] Isabela Granic, Adam Lobel, and Rutger C. M. E. Engels. The benefits of playing video games. *American Psychologist*, 69(1):66–78, 2014.

[27] Juho Hamari, David J. Shernoff, Elizabeth Rowe, Brianno Coller, Jodi Asbell-Clarke, and Teon Edwards. Challenging games help students learn: An empirical study on engagement, flow and immersion in game-based learning. *Computers in Human Behavior*, 54:170 – 179, 2016.

[28] Samuel Hort. Minecraft had 74 million active players in december, a new record for the game. https://www.pcgamer.com/minecraft-had-74-million-active-players-in-december-a-new-record-for-the-game/, 2018. Accessed 15 Aug 2018.

[29] Samuel Hort. Scratchers worldwide diagram. https://scratch.mit.edu/statistics/, 2018. Accessed 15 Aug 2018.

[30] Graham Hutton. *Programming in Haskell*. Cambridge University Press, Cambridge, 2007.

[31] LightBot Inc. Lightbot. http://lightbot.com/index.html. Accessed 15 Aug 2018.

[32] Nina Iten and Dominik Petko. Learning with serious games: Is fun playing the game a predictor of learning success? *British Journal of Educational Technology*, 47(1):151–163.

[33] Filiz Kalelioglu and Yasemin Gülbahar. The effects of teaching programming via scratch on problem solving skills: A discussion from learners' perspective. *Informatics in Education*, 13:33–50, 2014.

[34] Ming-Chaun Li and Chin-Chung Tsai. Game-based learning in science education: A review of relevant research. *Journal of Science Education and Technology*, 22(6):877–898, 2013.

[35] MIT Media Lab Lifelong Kindergarten Group. Scratch, a visual programming language. https://scratch.mit.edu/about/. Accessed 15 Aug 2018.

[36] Alejandro Lujan. Scalaquest - the game to learn scala. https://www.kickstarter.com/projects/andanthor/scalaquest-a-game-to-learn-scala/description. Accessed 15 Aug 2018.

[37] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 168–172. ACM, 2011.

[38] Mojang. Minecraft. https://minecraft.net/en-us/, 2011. Accessed 15 Aug 2018.

[39] Bob Nystrom. Game loop – game programming patterns. http://gameprogrammingpatterns.com/game-loop.html#the-pattern. Accessed 15 Aug 2018.

[40] Harold F. O'Neil, Richard Wainessa, and Eva L. Bakerb. Classification of learning outcomes : evidence from the computer games literature. 2005.

[41] Maja Pivec. Editorial: Play and learn: potentials of game-based learning. *British Journal of Educational Technology*, 38(3):387–393, 2007.

[42] Tom Preston-Werner. Toml – tom's obvious, minimal language. https://github.com/toml-lang/toml. Accessed 15 Aug 2018.

[43] Rathika Rajaravivarma. A games-based approach for teaching the introductory programming course. *SIGCSE Bull.*, 37(4):98–102, 2005.

[44] Daniel Ratcliffe. Computercraft modification. http://www.computercraft.info/download/, 2012. Accessed 15 Aug 2018.

[45] Carlton Reeve. Constructivism and games. http://playwithlearning.com/2012/01/20/constructivism-and-games/, 2012. Accessed 15 Aug 2018.

[46] Scott Rogers. *Level Up! The Guide to Great Video Game Design*. Wiley Publishing, second edition, 2014.

[47] J. Schell. *The Art of Game Design: A Book of Lenses*. CRC Press, second edition, 2014.

[48] Jeremy Singer and Blair Archibald. Functional baby talk: Analysis of code fragments from novice haskell programmers. *6th International Workshop on Trends in Functional Programming in Education*, 2017.

[49] SurveyMonkey. Surveymonkey platform. https://www.surveymonkey.com. Accessed 15 Aug 2018.

[50] Al Sweigart. No Starch Press, 2018.

[51] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285, 1988.

[52] José-Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. Visual programming languages integrated across the curriculum in elementary school: A two year case study using "scratch" in five schools. *Computers & Education*, 97:129–141, 2016.

[53] Microsoft MakeCode Team. Makecode for minecraft. https://minecraft.makecode.com/. Accessed 15 Aug 2018.

[54] Unity Technologies. Execution order of event functions. https://docs.unity3d.com/Manual/ExecutionOrder.html. Accessed 15 Aug 2018.

[55] Unity Technologies. Standard assets. https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351. Accessed 15 Aug 2018.

[56] Unity Technologies. Unity asset store. https://assetstore.unity.com/. Accessed 15 Aug 2018.

[57] Unity Technologies. Unity game engine. https://unity3d.com/unity. Accessed 15 Aug 2018.

[58] Simon Thompson. *Haskell : the craft of functional programming*. Addison Wesley, Harlow, third edition, 2011.

[59] Matthew Ventura, Valerie Shute, and Weinan Zhao. The relationship between video game use and a performance-based measure of persistence. *Comput. Educ.*, 60(1):52–58, 2013.

[60] David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Trans. Comput. Educ.*, 18(1):3:1–3:25, October 2017.

[61] Cyan Worlds. Obduction. https://store.steampowered.com/app/306760/Obduction/, 2016. Accessed 15 Aug 2018.