HaskellQuest: a game for teaching functional programming in Haskell

Neel Amonkar

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

The primary goal of the HaskellQuest project was to design and implement an educational game that would introduce programmers to Haskell and assess their skills along the way. Functional programming languages like Haskell are notoriously difficult for programmers to get accustomed to, as they're fundamentally different from the commonplace imperative paradigm - placing the learning experience within the framework of a game may help bridge the gap for newcomers and provide entertainment to ease frustrations.

The current version of HaskellQuest sequentially covers the concepts of types, expressions, function definitions, lists and list comprehensions, and recursion, along with a few extras (tuples and pattern-matching). The story follows a superhero, Lambda-Man, as he learns Haskell from his mentor and deals with villains and threats to Haskell City; each in-game battle is themed around specific Haskell concepts, and the player is tasked with writing Haskell functions to foil enemy attacks and escape unharmed. The core approach of the game is to provide an entertaining story, varied settings and fun graphics to hook the player into learning Haskell - thus, special emphasis was placed on the game's presentation.

The project has succeeded in implementing the core mechanics of the Haskell challenges, battle dialogue and themed enemy attacks, though some ideas in the initial concept had to be cut due to time constraints. A user test was carried out at the end of the development, which yielded positive results; the feedback validates the core approach of the game, while also providing suggestions for improvements.

The current version of HaskellQuest is available for Windows, MacOS and Linux. The game was designed to easily accommodate expansion and further development - the final section of this paper proposes multiple ideas for future work that have the potential to greatly enhance the game's educational and entertainment value.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: 297129
Date when approval was obtained: 2024-02-18

The participants' information sheet and a consent form are included in the appendix.

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.

(*Neel Amonkar*)

Acknowledgements

I would like to thank my project supervisor, Don Sannella; his guidance and advice were invaluable in the development of this project.

I would also like to thank my parents, whose help and support in my dissertation (and in general) cannot be overstated, as well as my friends, who gave me encouragement and helpful feedback during development.

Table of Contents

Chapter 1

Introduction

HaskellQuest is an educational game aimed at programmers wishing to get to grips with the functional programming language Haskell; it takes players through Haskell's features and tests their understanding through themed battle segments. While some design decisions were made with University of Edinburgh students in mind, since they'd be the demographic playtesting the game, it is not **specific** to them; any newcomers to Haskell can play through and enjoy the game.

1.1 Project goals and approach

Computing students can find it difficult to get to grips with Haskell and other functional programming (FP) languages, as it requires a radically-different thought process from the imperative languages that they're used to. This project's fundamental goal is to explore how teaching functional programming through the framework of a game might help bridge the gap for newcomers, i.e. people who have some prior familiarity with programming, but none whatsoever with FP.

One of the appeals of video games as a medium is allowing the player to interact with a story or world. The core approach of this project is based around that appeal - to provide an engaging story and presentation to hook the player into wanting to progress through the game, where learning and writing Haskell is their main 'power'. A personal motivation for this decision was my pre-existing experience in graphic design and digital artwork - as this would be my first time being the solo developer for a full game, I wanted to lean on those strengths.

HaskellQuest covers the following topics, as specified in the project description:

- Types, including simple algebraic data types
- Expressions
- Function definitions and pattern-matching
- Tuples, lists and list comprehensions
- Recursion

1.2 Dissertation structure

- **Background**: Chapter 2 explores the core concepts of FP, Haskell and serious games, as well as previous work on how games can be used to teach programming.
- **Initial decisions**: Chapter 3 talks about the foundational decisions made at the start of development, like the game engine, core mechanic and storyline.
- **Battle design**: Chapter 4 describes the design decisions behind each section of the game, as well as some scrapped concepts.
- **Implementation**: Chapter 5 goes into detail about how game elements were implemented and why, as well as some drawbacks of the approaches taken.
- **Evaluation**: Chapter 6 discusses the method of recruiting playtesters and evaluating the game, as well as the feedback received.
- **Conclusion**: Finally, Chapter 7 summarises the achievements of this project and discusses some potential avenues for future work.

Chapter 2

Background

2.1 Functional programming and Haskell

2.1.1 Introduction to functional programming

Functional programming is a programming paradigm that is wholly separate from the imperative paradigm that most programmers would be familiar with. Where the latter is based on writing a sequence of statements that update a program state, FP is based around applying and composing functions - defined as expressions mapping values to other values, similar to the mathematical definition. Most commonly-known FP languages are in fact **pure** functional programming languages; functions are reliant on their arguments only, without considering any global or local state.

2.1.2 Haskell

Haskell is one of the most popular functional programming languages; on a 2023 list of the most commonly-used languages on GitHub, ranked by number of pull requests, it placed 26th [39]. Its genesis was at a meeting at the conference on Functional Programming Languages and Computer Architecture (FPCA) in 1987, where it was agreed that "more widespread use of this class of functional languages was being hampered by the lack of a common language". [26] The FP language most used previously was Miranda [55], but it was proprietary, restricting further development. Haskell was designed and developed by committee, and is open-source. Given this context and its aforementioned popularity, it would make the most sense to use Haskell as the FP language for teaching purposes.

2.1.3 Why teach FP?

Functional programming languages are already included in many CS university programs as part of an introductory course, like at the University of Oxford [21], the University of Nottingham [12], and of course here at Edinburgh [29]. According to Joosten et al., who began the introduction of an FP course to the CS program at the University of Twente in 1987, FP languages allowed students to "denote appropriate

abstractions" and write "clear and concise programs... that express the essence of the algorithm, and nothing more" [35]. They deemed the former aspect to be particularly relevant, as the university department assessed first-year courses by their ability to separate better students from poor performers - the FP course was able to do this based on the students' abstraction ability, where before programmers lacking that skill "would sometimes pass only because they can make programs work".

Additionally, in a 2001 letter to the Budget Council at the University of Texas, Austin, Edsger W. Dijkstra endorsed the use of Haskell in their introductory programming course over Java [15]. He noted that first-year students would probably already be familiar with imperative programming languages, so the novelty of the FP paradigm would immediately alert them to the existence of alternatives. Furthermore, in his words, FP "elegantly admit(s) solutions that are very hard (or impossible) to formulate with" imperative languages, and "functional programs are more readily appreciated as mathematical objects than imperative ones", providing a more natural segue into rigorous reasoning about programs.

2.2 Serious games

2.2.1 Introducing the term

The HaskellQuest project, being a game designed to teach FP, falls under the umbrella of 'serious games'. The term's current widespread meaning is attributed to Clark C Abt in his book 'Serious Games' [2]: games having "an explicit and carefully thought-out educational purpose" and "not intended to be played primarily for amusement", though of course he stressed that this "does not mean that serious games are not, or should not be, entertaining."

While this definition of serious games is quite broad, any modern discussion of the topic is specifically about video games - the medium is increasingly in the mainstream and the industry itself is valued at hundreds of billions of dollars [30]. The use of video games for this purpose has increased since the 2000s, and now includes fields such as healthcare (both for training and treatment purposes), military training and recruitment, and education in classrooms. In particular, a literature review of works in the last category, covering use in elementary school up to higher education and based on a broad range of topics, concluded that "(even) if they are not always superior to other types of learning material, the evidence that serious games can be effective learning materials in their own right is quite strong." [5]

2.2.2 Why video games are effective learning tools

In 'Learning and Studying: A research perspective' [23], Professor James Hartley outlines a list of principles of learning emphasised in behavioural psychology, which include:

- **"Activity is important"**: According to the book, "learning is better when the learner is active rather than passive", later noting that one reason for this is

because it allows the student to pace their own learning. This straightforwardly correlates to the interactive medium of games, as progression is driven by the player's own actions and decisions.

- **"Repetition, generalisation, discrimination are key notions"**: This emphasises that "frequent practice - and practice in varied contexts - is necessary for learning to take place". Most games take the player through varied environments and situations, thereby fulfilling the requirement for variety, as well as providing feedback and allowing the player to try again after an unsuccessful attempt.
- **"Reinforcement is the cardinal motivator"**: The book states that "the effects of the consequences on subsequent behaviour are important - whether they be extrinsic (reward from a teacher) or intrinsic (self-reward)". The extrinsic rewards that games offer are obvious - they can reward players for success in various ways, e.g. progress in the game's story, in-game currency, increasing the player character's capabilities, and so on.

Hartley later outlines a fundamental split in individual learning strategies - 'deep' learners, for example, would try to extract the meaning of a text, while 'surface' learners would concentrate on just memorizing the text itself. The book then goes on to list some teaching strategies to encourage deep processing, which include 'using problem-based learning' and 'setting assignments that cannot be completed by memory work alone'. Once again, games fit into this quite naturally - the basic mechanic of any game is setting problems within a set of rules for players to solve, and well-designed games generally avoid rote memorization as the solution to these problems (at least, not on the first playthrough).

2.2.3 The use of games in teaching programming

Given the recent popularity of serious games, there have been prior attempts to use them to teach programming languages and concepts. Among the most prominent is the CodinGame platform [11], which has multiple single-player competitive coding challenges based around common game genres like puzzle games, shooters, and so on. Each challenge is a full-fledged competitive mini-game, where the actions taken by objects within the game (e.g. a turret attacking oncoming enemies, as seen in Figure 2.1, or a car heading from checkpoint to checkpoint as quickly as possible) are controlled by the code that the player submits.

Other prominent examples of programming-related serious games include Code Hunt by Microsoft Research [6] and Human Resource Machine by Tomorrow Corporation [27] - both focus on writing efficient imperative algorithms, although in slightly different ways.

Miljanovic and Bradbury conducted a systematic review of existing literature about serious games in programming [42] - they found that most games surveyed were effective at teaching introductory programming concepts, but there were some deficiencies in terms of teaching further software development (which wouldn't be a concern with Haskell, since it isn't really used for that) and a lack of consensus in evaluating their effectiveness. Additionally, Zhan et al. examined the effect of gamification in program-

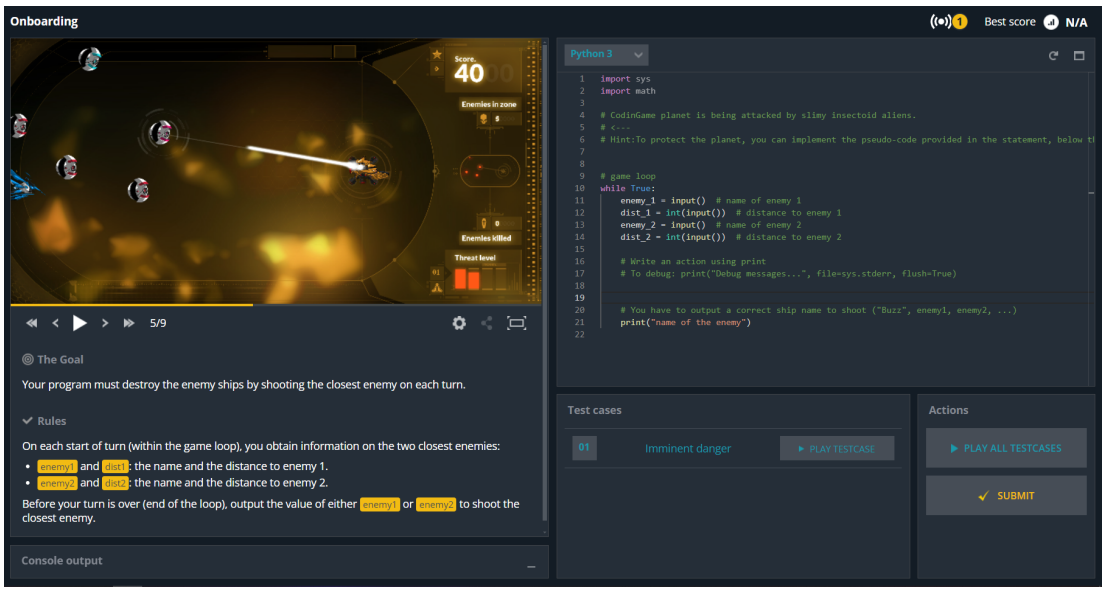


Figure 2.1: The onboarding exercise in CodinGame [11] - the player is tasked to write Python code to shoot the closest enemy on each turn, which in practice is about learning conditional statements in Python

ming education [63], and found that role-playing games were found to emphasise the students' academic achievements, while puzzle games "were most effective in terms of enhancing students' learning motivation and thinking skills".

In general, though, existing games very much focus on imperative programming, with a few exceptions.

2.3 Gaming and teaching FP

2.3.1 Learning FP through game development

There have been previous attempts to teach functional programming that have incorporated gaming elements. The Soccer-Fun project [3], implemented in 2011 at the Radboud University Nijmegen in the Netherlands, was an attempt to teach functional programming - students modelled the decision-making of a virtual football player as functions in the pure FP language Clean. The game framework itself was fully implemented in Clean and used its GUI library - students could then pit their virtual football players against one another to see who performed better.

'Haskell in Space' [38] is a similar project. Second-year computing students were tasked with implementing a version of the arcade game *Asteroids* [4] - specifically, writing Haskell functions to model the physics-based behaviour of the ship, its projectiles, and the titular asteroids. Both were found to be effective: the competitive element of Soccer-Fun served as a good motivating factor for students, while the graphical element of Haskell in Space proved highly popular (to the point where students embellished their submissions graphically even though they weren't required to).

Of course, these wouldn't come under the category of serious games - they are, first and foremost, programming assignments challenging students to **make** a game (or, in the case of Soccer-fun, a sub-section of the game). Even so, the primary appeal of both projects was that the behaviour of the game was directly influenced by the students' programs, exactly the same as with CodinGame.

In the case of these FP projects, however, the **entire game** was built from the ground up in an FP language, which is potentially limiting in terms of the scope. Games are inherently about manipulating a game state, which clashes with the fundamentally stateless nature of pure FP. The Soccer-fun and Asteroids games handled this by defining a custom datatype representing a state, and then creating functions that take a state as an input and return a state as an output. However, the complexity of this state type would increase drastically alongside the complexity of the game itself. While it is possible to create a full game using Haskell [25], the technical challenge involved is far beyond my current capabilities.

2.3.2 Serious games teaching FP

For a solution, we can examine previous HaskellQuest games - in particular, the 2018 thesis by K. Drobnik [16] and the 2023 thesis by E. Bogomil [7]. They share a common approach - they both use the Unity engine [58] for developing the game itself, meaning the game state and variables are kept track of in a standard imperative approach. The games then have specific instances of puzzles/obstacles which involve writing Haskell functions, where the interaction between said functions and the game itself is **simulated**; a Haskell compiler is used to check if the player's written function works as expected, and the game code (written in C#) alters the state based on the result. These games can be seen in action in Figures 2.2 and 2.3.

With these examples, the Haskell functions themselves are in some way related to the game state: prompting the player to write a function to move a platform so that they can progress to the other side in the 2018 game, for instance, or allowing the player to define a function to attack enemies in a fantasy role-playing-game (RPG) format in the 2023 thesis. As such, the main appeal of assignments like Soccer-fun and Asteroids and platforms like CodinGame is preserved without presenting too much of a technical hurdle.

Figure 2.2: The introductory battle in E. Bogomil's HaskellQuest [7]: the player is tasked with writing actions that can modify the player's character and enemy's stats i.e. dealing damage to the enemy, recovering player health, etc.

Figure 2.3: An early puzzle in K. Drobnik's HaskellQuest [16]: the player is given Haskell code that rotates a bridge in the environment and has 3 'blocks' that they can slot into place

Chapter 3

Initial decisions

3.1 Approach to development

The traditional method of organising game development in the industry is a linear, step-by-step affair [8]:

- Planning, where you lay out the basics of the game - the genre, the scope, the story, the engine to use
- Pre-production, where you create more detailed plans, create prototypes to test concepts, and finish conceptualising as much of the game as possible
- Production, where you actually build everything
- Testing, where you hunt down and fix any bugs and glitches that might have cropped up, ranging from the annoying to the game-breaking
- Pre-launch, where the marketing for the game begins
- Launch, the final opportunity to polish the game
- Post-launch, where the team can work on bugfixes, patches, and downloadable content (DLC)

However, given I was working as a solo developer on a full project for the first time, in a genre I wasn't familiar with (educational games), I decided not to fully use this approach. To me, it didn't make much sense to spend time writing detailed design documents when in most cases I wouldn't know if a mechanic actually made sense until I played it, or creating a detailed timeline when I didn't know how long each task would take.

The approach I ended up taking was much more similar to an iterative and incremental approach [36] - I came up with an idea, implemented it, tested it to see if it worked well, and built on the results for the next idea.

3.2 Choice of game engine

The first decision I had to make, apart from the game's mechanics, was how to actually develop it. While, again, it would be possible to write the game from scratch in another language (since, as discussed in section 2.3, the only requirement is that it can communicate with a Haskell compiler), this would once again be far beyond my current expertise. It made the most sense to use a game engine - they, by default, come with a lot of tools, libraries and support to make development easier. The choice then lay between the most commonly-used engines today:

1. Unity [58], as with the two previously-cited HaskellQuest games
2. Unreal Engine, developed by Epic Games [59]
3. GameMaker, developed by YoYo Games [22]
4. Godot Engine, a free and open-source engine gaining popularity [19]

Unreal Engine theoretically supports creating 2D games, but Epic Games has focused their efforts in developing its ability to make 3D games with photorealistic graphics, which is not at all what I had in mind for HaskellQuest. Godot Engine supports both 2D and 3D games, and the community has added support for Haskell to be used in scripting [53]. Unfortunately, I had absolutely no experience with Godot at the beginning of development, and didn't want to subject myself to the learning curve and growing pains on top of the design challenge of creating an educational game. That left the two options I had prior experience with: Unity and GameMaker.

While GameMaker is ideal for 2D games, it doesn't actually include a toolkit for UI elements - to make a button, for example, you'd need to use GameMaker's functions for a) checking if the mouse is over the object and b) checking if the mouse button has been clicked, and then manually position and align the sprite for the button.

Unity, on the other hand, has much more support for UI elements - using the button example again, positioning and aligning it is far easier. Additionally, the included Button class has a pre-built method to add a function that executes after a click, set whether the user can interact with the button or not, and so on - it basically allows developers to focus more on implementing what the button actually does in-game. The concept for the game involved players writing code, something that needs a robust UI. As such, I decided to go with Unity for this project.

3.3 Central mechanic

When coming up with the game's fundamental mechanic, I knew I wanted to have the players actually write code - the two HaskellQuest entries I'd tried out in the last semester of third year, E. Bogomil's game [7] and M. Despinoy's game [14], had worked this way. An alternative would be to offer players 'code blocks' to fill in gaps in code, like with K. Drobnik's game [16], or in general offer multiple choices for the correct answer to each challenge. The problem with this, I felt, is that it would be too easy to guess wildly or try every choice and still get the correct answer without actually

understanding what you're doing. Additionally, when playing through E. Bogomil's HaskellQuest last year, I found that it was all too easy to miss out on vital Haskell knowledge (since that game incentivises exploration of the world through collectible Haskell tips). With my design, I wanted to make sure the player would have no way of going through the game without gaining Haskell knowledge - a more constrained approach.

3.3.1 Abortive first concept

The first concept I came up with was a side-scrolling 2D platformer where the main character is armed with a projectile weapon, similar to Capcom's *Mega Man* series [40] - indeed, in a holdover from this concept, the main character in the final game looks very similar to the character X. In those games, the player attempts to clear levels in any order - defeating the boss at the end of each level rewards the player with a new weapon that fires a different type of projectile (themed around the defeated boss).

In my version, I would have gone with a more linear approach to introduce the player to Haskell concepts and challenges of increasing difficulty. To gain new weapons, rather than defeating boss characters, the player character would have instead needed to interact with a terminal at some point in the level. A Haskell challenge would then pop up, and upon completion the player would be rewarded with a new weapon themed around the concept the challenge was based on (e.g. a gun that is based around recursion). The idea here was that the player character (PC) created the weapon using Haskell, and that the character's capabilities increased alongside the Haskell skills of the player.

The problem with this concept was two-fold. The first was that I was having difficulty coming up with ideas for weapons to build a proof-of-concept around: what does a 'recursion gun' look like? How can the player use it in a 2D platforming situation? The second and more important problem was that most of the time spent playing a level would not be spent **writing Haskell** - the platforming mechanics would actively take focus away from the Haskell challenges. As such, I had to start over - thankfully, this wasn't a problem, as I hadn't begun implementing anything yet.

3.3.2 Second, final concept

The second concept I came up with was based on *Undertale* [20], a wildly popular 2D role-playing game (RPG) released in 2015. In it, the player interacts with monsters in an underground world - the battles in that game are turn-based, meaning the player performs an action and then the enemy performs an action. *Undertale* implements a 'choice' system for the player's action - the player can choose to 'ATTACK' and ultimately kill the enemy in a manner akin to other RPGs, or they can 'ACT' i.e. attempt to talk to and befriend the monster they're fighting. The latter option is heavily encouraged by the game as part of its story and morality, and introduces a puzzle aspect as players attempt to find the correct options to ensure a peaceful outcome. Meanwhile, during the enemy's turn, the player is tasked with dodging projectiles in a segment similar to the bullet hell / manic shooter genre [60], adding urgency to the battle. I

thought this would be a good fit for HaskellQuest, seeing as the puzzle and role-playing aspects had the potential to enhance the game's educational value (as discussed in section 2.2.3).

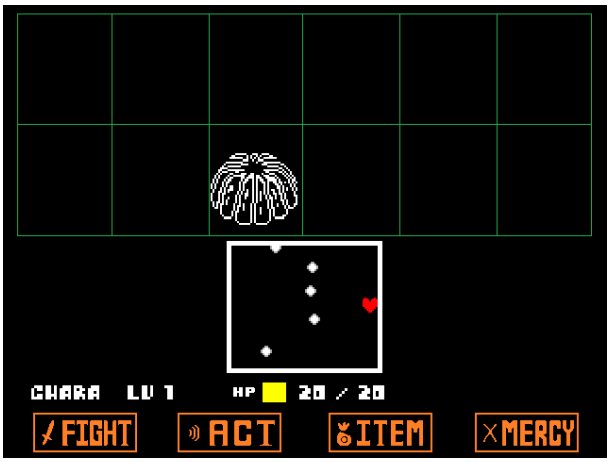


Figure 3.1: The enemy's turn in an early battle in Undertale: the player, represented by the heart, dodges projectiles to avoid damage

To adapt the mechanic, I thought: "What if, instead of 'ACT'ing to end the battle, the player had to write Haskell code?" In Undertale, successfully 'ACT'ing would advance the battle to the next phase, but you could still take damage from the enemy's turn. In some cases, however, the enemy's attack would be weakened, or become totally ineffective, and this was used as the springboard for HaskellQuest's game loop: if you successfully complete the Haskell challenge, the enemy's projectiles essentially cannot harm you, and the battle itself would proceed to the next phase. Thus, the Haskell challenges have to be patterned around directly interfering with the enemy's attack in some way, in order to maintain the idea from Section 2.2.3 and Section 2.3 that the player's written code is affecting objects in the game world. I removed the choice between 'FIGHT'ing and 'ACT'ing because it would be completely superfluous to HaskellQuest.

3.4 Story premise

The thing I wanted to accomplish with the story was essentially adding entertainment value. Its role would be to provide a sense of forward momentum and anticipation for the player - essentially, they would be incentivised to learn Haskell skills and complete challenges to see what situations they would be put in next. I also knew I wanted to make it tongue-in-cheek, and cater it to University of Edinburgh students, seeing as they'd be the primary audience playing and testing the game. As such, I thought it would be fun to base it around 'Lambda-Man', Professor Philip Wadler's Superman-esque 'alter-ego' at conferences [48], who we were introduced to as part of a gag during lectures.

I decided that the main character would be a successor to the original Lambda-Man, who acts as a mentor character in-game, serving as a metaphor for us students in first-year

learning Haskell. To make this even more absurd and thus potentially more intriguing, I decided to have the original Lambda-Man recovering from a climactic battle, and only able to communicate with the main character through a hologram (thereby necessitating a successor to begin with), like the character Zordon from *Mighty Morphin' Power Rangers: The Movie* [41]. Crucially though, to allow for people not in on the joke (and thus a broader audience), this is mostly unaddressed within the game itself - the character just so happens to look like Prof. Wadler, as opposed to Prof. Wadler himself being in the game. This character's role would be to grant the player character the Lambda Gauntlet, which allows the user to rewrite their adversaries' Haskell code - this would be the in-game justification for the mechanics.

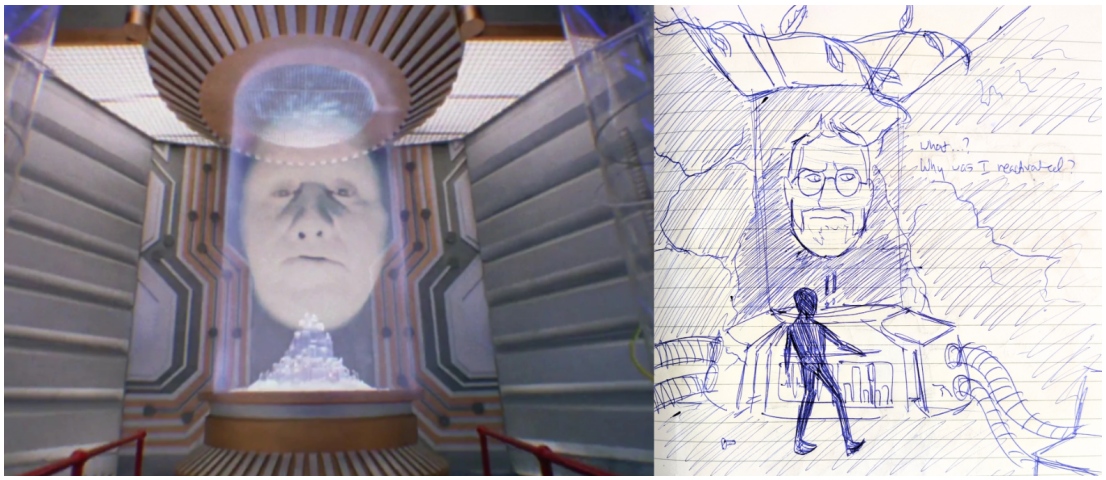


Figure 3.2: **L:** Zordon, as seen in *Mighty Morphin' Power Rangers: The Movie* [41]. **R:** The initial concept sketch for the original Lambda-Man; the visual inspiration is clear.

With this story idea in place, the structure for the rest of the game began to take shape. My original idea for the story was a grand adventure following the player's fight against some vast organisation responsible for the world being in disarray, who were also the ones that the original Lambda-Man had fought against and lost. As such, I wanted the game to have an overworld section (again, similar to Undertale), where the player could explore the environment and encounter enemies with simpler Haskell challenges, all leading up to a boss at the end of an area with a multi-phase battle - none of this made it in, for reasons that will be expanded on later.

3.5 Method of Haskell integration

After deciding what engine to use, this was the first implementation-related decision; how exactly would the game compile the player's Haskell code? The first option would be to require the user to have the Glasgow Haskell Compiler (GHC) installed, and make calls to that using C#'s Process class [47]. The main downside to this is that it adds extra steps to the installation process; even with the recently-streamlined GHC installation guide [31], players who don't want to go through the extra hassle of installation would be entirely barred from playing the game.

A solution to this would be including the compiler within the game files. This would

be less of a hassle, but it would also inflate the file size for the game itself. More importantly, it isn't a foolproof solution. When attempting to play M. Despinoy's HaskellQuest [14] last year, which used this exact method, any compilation requests in-game seemed to time out entirely, despite GHC being included with the game, as well as already being installed elsewhere on my system. With both methods, the Haskell code would be compiled on the player's system, which can introduce problems. If, for instance, the player writes erroneous code that doesn't actually terminate, neither would the process - you would need to specifically assign a timeout for that process in the game's code.

The option I decided to go with was using an external online compiler, following the lead of E. Bogomil's HaskellQuest [7]. There are lots of online compiler services that provide Haskell as an option, most of which are intended for practicing programming online [24] [45] [46]. Some of these offer APIs to their compiling services [34] [10] [44], accessed via sending the script in a POST request to a REST endpoint - the companies provide a free trial with a limited number of API requests per day, and then offer payment plans to increase that number. Of these, I decided to go with JDoodle for HaskellQuest, as it seemed the easiest to start with - it had documentation on how to use the API, as well as a Java code example to send the POST request, which was straightforward enough to convert to C#.

This method solves the problems mentioned previously: the online compiler obviously wouldn't affect the file size, and the only requirement on the player's part to get the game to work is to have an internet connection, which would almost certainly be true for the game's intended demographic. Additionally, any potential problems with non-terminating code or input sanitation can be assumed to be handled by the service - after all, they charge for access to the compiler. (Indeed, JDoodle handles the former case by timing out.) There is one shortcoming here: the game needs to be kept up-to-date to handle any changes or updates to the API. Additionally, the service costs money; the free trial only allows 200 requests per day, which is infeasible for multiple players attempting to play the game simultaneously. As such, for the period of user evaluation, the JDoodle account was upgraded to allow 20,000 calls per day for $20.

Chapter 4

Battle design

4.1 Initial proof-of-concept

After deciding on the central mechanic detailed in section 3.3.2, I wanted to flesh it out into a proof-of-concept "battle" (in reality, a slideshow in Canva [9]) in order to figure out how the game loop would actually work. I started with the 'Mad Dummy' enemy from Undertale [20] as a basis. In the original game, that character launches homing missiles that zero in on the player, who then has to position themselves such that the missiles fly back into the enemy.

To adapt this for HaskellQuest, I thought: "what if the player instead wrote a function to re-target the missiles?" My main inspiration here was the weekly FP tutorials in the University's first-year Introduction to Computation course [29]. There, we would be tasked to write simple Haskell functions that, for example, reversed a string or split a list into smaller lists - the hope was that by doing something similar here, players would gain practical experience with Haskell's syntax and understand the purely-functional nature of the language. With the tutorials, these functions would be checked using an auto-grader that compared the function's output to that of an example implementation hidden from the students - I decided to do the same thing with all the challenges.

The fundamental idea for this proof-of-concept was to define a simple algebraic data type (ADT) named 'Missile', representing the in-game object, and have the player write a function of type '[Missile] -> Target -> [Missile]' that changes the target of every single missile to whatever the function took as input. A hidden "main" IO monad would then check the output of the written function to see if it matched the expected behaviour, printing a Boolean value. If that value was False, the game would proceed to the enemy's turn; the enemy fires the missiles at the player, who is allowed to try and dodge them to avoid taking damage. If True, the missiles would instead hit the enemy and the player would take no damage, acting as the reward.

4.1.1 First idea of progression

Often, for the tutorials, we were tasked with implementing a function one way (e.g. with a list comprehension), and implementing the same function a different way (e.g.

using recursion instead). In the proof-of-concept, I went about it the same way, as the player could write the function in 3 ways:

- List comprehension, an obvious method
- Recursion (by pattern-matching the list against 'x:xs'), though in this case it yields no benefit whatsoever
- Creating a function of type 'Missile -> Target -> Missile' that only changes one Missile, then sending that and the original list as inputs to the 'map' function

I had the thought that each phase of the fight would require the player to implement the function a different way, progressing from list comprehension to recursion to 'map' use. This would allow the reuse of the script that handled the missiles (I hadn't started implementation yet, but I had some idea of how it would be done), because the enemy's actual attack would behave identically in each section. However, when it came time to implement this, I ran into two problems. The first was that it would be quite hard to distinguish whether a function was implemented using list comprehension or recursion in a way that wasn't error-prone; you could try and match a syntax format, but that wouldn't be foolproof. The second was that from the player's point of view, going through the exact same attack multiple times in a row would be **incredibly** dull.

4.1.2 Second, final idea of progression

To move away from this problematic initial idea of progression, I decided to revisit one of the ideas from Section 3.3.1: what if each level in the game was themed around a specific Haskell concept? This helped clarify the role of the weaker enemies mentioned in Section 3.3.2 - they would hopefully act as warm-ups for the player to understand the Haskell concept before testing that understanding in the area's boss, where each phase would be an increase from the last in terms of complexity.

Thus, the overall structure of the game was decided, based on the concepts that the game had to cover as part of the project description:

- The first battle would cover **types, expressions** and **function definitions** - the basic building blocks of the language.
- The second battle would cover **list comprehensions**.
- The third battle would cover **recursion**.
- If I had enough time, I planned to include a fourth battle themed around **higher-order functions** - however, this was always a remote possibility.

4.2 List comprehension battle - Col. Trigger-Finger

I decided to include the 'Mad Dummy'-inspired fight as a boss; it made sense to use it as the battle themed around list comprehension. The individual phases would then be designed around this progression:

1. Basic list comprehension

2. List comprehension with a Boolean guard
3. List comprehension with multiple generators

4.2.1 Haskell challenge design

Since the design for the first phase was largely taken from the proof of concept, I won't be elaborating on that again - the actual implementation, where most changes happened, is detailed in Section 5.1.

For the second phase, rather than using more homing missiles, I decided the enemy would fire multiple volleys of regular missiles at the player that were all lined up horizontally, such that it would be difficult for the player to dodge them all. The conceit here was that one of these missiles would be inactive, meaning even when colliding with the player it wouldn't do any damage - this would be determined by a Boolean in the Missile data type definition. The player would be given another list of Missiles, and would need to write a list comprehension that only included the inactive missile, thereby needing to apply guards in list comprehensions.

The original idea here was that it would merely **highlight** the inactive missile in an obvious way, such that the player could still be hit by all the other missiles if they didn't move fast enough. However, in terms of the story, it was already a stretch that the enemy would fire a single inactive missile in a barrage of live ones, and I thought it'd be even more of a stretch for that same enemy to have any sort of method to highlight individual missiles. Additionally, when play-testing, I found that getting hit by missiles even after beating the Haskell challenge was somewhat aggravating - as such, I changed the battle mechanic.

For the final phase, I decided to include both kinds of missiles to fulfil the requirement of having multiple lists. The enemy would fire another volley of the horizontal missiles, and then quickly fire several homing missiles. The idea was for an all-out attack that would hem the player in; the slowly-descending regular missiles would prevent the player from moving around too much, thereby making it more likely that they'd get hit by the regular missiles. With this one, the solution would be to retarget the homing missiles to home in on one of the regular missiles each, eliminating both sets of missiles.

The solution here, though, could not be implemented with the standard method of including multiple generators in a list comprehension. There, the rightmost generator is iterated through first - in an imperative language, the list comprehension would be more similar to nested loops, whereas the concept required that both lists be iterated through simultaneously. This could be done using the 'zip' function, included in the Haskell Prelude, that takes two lists and returns a list of tuples where the elements are paired - the player could then use this list in the comprehension. Haskell has also recently included a new feature as an extension known as **parallel list comprehension** [1], which effectively does the same thing as the 'zip' function, except with syntax more akin to multiple list comprehension.

4.2.2 Characterisation

I knew I wasn't going to keep the Mad Dummy character in place in the game, though, as I wanted to create something new that fit the storyline better by being a straightforward villain. The original was characterised as somewhat psychotic and trigger-happy; these aspects were maintained as the basis for the final character. Given that all of the attacks in this fight involved the enemy recklessly firing missiles, I thought it'd be fitting to have the character be a satire of macho militarism who would, quite literally, have missile-launchers built into his body.

Figure 4.1: One of the sprites for Colonel Trigger-Finger.

This was then reflected in the design of 'Colonel Trigger-Finger', seen in Figure 4.1. The sea of medals was inspired by Jason Isaacs' character in *The Death of Stalin* [28], while the torn sleeves and pant legs, exaggerated bulging muscles (even though the character is a robot), stubby proportions and cigar were meant to convey the exaggerated "manly man" nature of the character. The lack of subtlety in the name was also intentional; it was meant as a tribute to the 'pulpy' nature of classic superhero comic books.

4.3 Tutorial 'battle' - Lambda-Man Hologram

By this point, the main sections of the list comprehension battle had been completed. Looking at the time I had left to get the game ready, I realised that I had been far too ambitious in my plans; I wouldn't be able to implement the weaker enemies designed as warm-up exercises. Additionally, the current framework was more suited to **assessing** the player's Haskell skills, since it had its roots in Inf1A tutorials; I still needed some way to actually **introduce** the concepts of list comprehension and recursion to the player. My plan for the first battle was a tutorial where the original Lambda-Man instructs his successor in using Haskell, and tests the player by using mock attacks that would be easy for the player to foil, covering the foundational Haskell concepts of types, expressions and function definitions. I decided to expand this to include a tutorial for list comprehension and recursion as well.

4.3.1 Haskell challenge design

Since this would be the introduction to the game and its specific mechanic of altering attacks, I decided to go with the simplest concept possible; there is a function of type

'Player -> Player' (Player being another ADT) that reduces the player's health by some fixed amount. The player would then be tasked to alter the function so they are completely unharmed; if the player took damage before succeeding, the Lambda-Man hologram would heal them for the next phase.

I wanted the player's alterations to the damage value in the function to be reflected in-game; this would make these challenges more of a 'playground' for the player and would cement the idea that the Haskell code was actually affecting the game's world. I also intended for astute players to notice that they could alter the function to **increase** their health instead.

Since this section would also need to introduce the list comprehension and recursion mechanics prior to the bosses themed around them, I decided that the second and third phases would be based around exactly that. The second phase would introduce a more complex version of the prior function, this time with type "Player -> [Int] -> Player". This function would use a list comprehension to go through the list, excluding values less than or equal to 0, and then reduce the player's health by the sum of that list (using the 'sum' function from the Haskell Prelude). The third phase would then behave identically, except implemented using recursion and guards.

In keeping with the 'playground' nature of the first battle, I also wanted the player to be able to alter the [Int] that would be passed to the functions; I somehow missed the obvious exploit of setting the list to include values less than 0 exclusively, until after the game had already been released to playtesters and it was too late to change it.

4.3.2 Characterisation

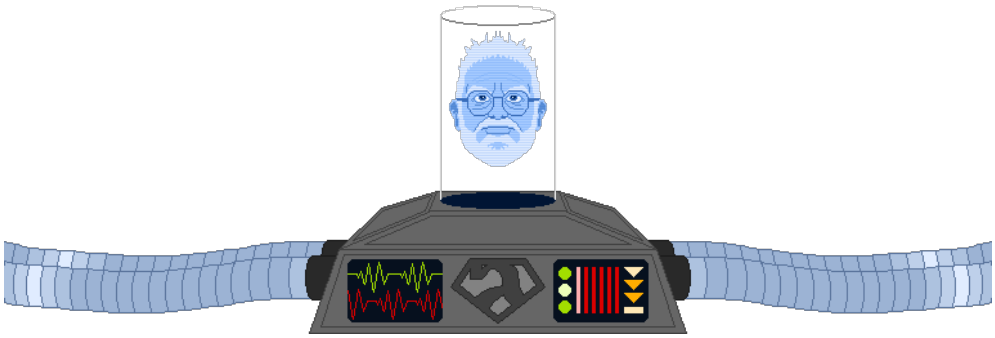


Figure 4.2: One of the sprites for the Hologram.

In keeping with the narrative role he would have to play, I wanted the Hologram to be an encouraging and supportive mentor, applauding the player for their progress in Haskell. The dialogue here would have to mostly serve as introductions to and explanations of Haskell features, some of which was taken from *Learn You a Haskell for Great Good* [37] - when introducing concepts, I made a point to have them be linked to analogues from pure math that players might already be familiar with (e.g. list comprehension to set notation). To make this less of a slog for players, I thought it'd be good to make it more of an actual **dialogue** between the two characters; Lambda-Man (the player character) would take the role of someone inexperienced with Haskell, asking questions and coming to realisations at about the same time I thought the player would.

The character design remained the same in essence as the concept sketch in 3.2 - I added the Lambda-Man crest as a nod to the initial gag, as well as a heartbeat indicator to signal that this was supposed to be healing and monitoring the original Lambda-Man's body.

4.4 Recursion battle - Dr. Fractal

4.4.1 Haskell challenge design

When designing, I reasoned that the concept of recursion lacked the intuitive progression in complexity that the other two battles had; at the time, I couldn't think of specific features to expand upon, unlike with list comprehensions. As such, the progression would be more dependent on the actual gameplay scenarios, rather than the progression being defined first and the gameplay scenarios following.

For the first phase, I was inspired by a friend's suggestion to create a scenario where the player uses recursion to split a large enemy into smaller enemies that can do less damage; it literalises the common refrain about recursion, "splitting big problems into smaller, more manageable ones". This function would, logically, have the type 'Fractal -> [Fractal]' (Fractal being yet another ADT, this time representing the enemy).

Originally, this function would divide the enemy into half, and then call the function on those halves, stopping when the enemy's Size parameter got to 1. However, in pre-release testing, it was pointed out to me that the behaviour could be replicated by a list comprehension that created 'n' copies of a Fractal with Size 1, where 'n' is the input Fractal's Size value. To fix this, I changed it so that the recursion would stop when the enemy's Size was an odd number. To deal damage, the enemy fires projectiles aimed at the player - unlike the homing missiles, they don't change direction after they're fired, but they travel faster to test the player's dexterity. These projectiles would then be greatly weakened upon successfully splitting the enemy into smaller copies.

The second (and in this case final) phase emerged entirely from the story for this battle. I'd had the idea for this enemy to be a scientist whose experiments with recursion had gone wrong, so the second phase would be based around rescuing a Person from deep within a nested Fractal structure, i.e. "Fractal (Fractal (Fractal (Fractal ...." Failing would result in another round of projectiles; succeeding would skip this entirely, meaning there'd be no attack at all.

Originally, the problem would be that the scientist had forgotten to include a base case in the function creating the nested Fractal structure; the player would have to first add that Person base case and then write the function extracting the Person from the Fractal. However, I immediately realised that if the player was in charge of writing the base case, they could then just make that the definition of the second function too; if I cheated in writing the auto-tester by using a different Fractal structure, it would potentially break the central illusion of the game. As such, only the second function survived into the final game.

The fact that this was the final battle was reflected in a design choice for the first phase;

if the player completes the Haskell challenge successfully, the resulting smaller Fractals would fire projectiles doing **less** damage, not **zero** damage. Additionally, because there were more Fractals, there'd be more projectiles to deal with; while the damage values were later tuned so that solving the Haskell challenge would still be better for the player, taking damage would still result in a score penalty per hit (more in Section 4.5).

4.4.2 Characterisation

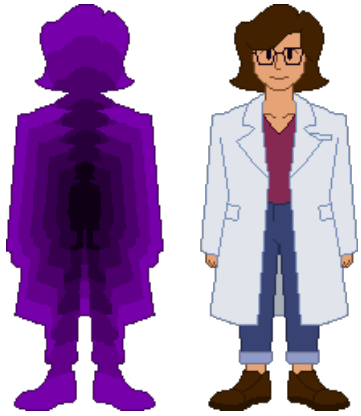


Figure 4.3: **L:** One of the sprites for Dr. Fractal. **R:** The sprite for Dr. Kowalewski.

To fulfil the story's purpose of adding variety and keeping the player guessing, I decided for this enemy to not have the character be a straightforward villain. There is a long tradition in superhero stories of the hero having to help a scientist whose experiments have gone wrong, transforming them into a threat - take Curt Connors / The Lizard from Spider-Man [61], for instance. In this case, Dr. Kowalewski's failed experiments with recursion make her into a Fractal, afflicted by having her every thought constantly echoing until she can't think clearly. This causes her to attack Lambda-Man, who is consistently endeavouring to stop her peacefully and save her - this plot element was included to add a sense of urgency and stakes to the battle. The character design, once again, was intended to reflect this storyline - while Dr. Kowalewski looks pretty much normal, the Fractal form was designed to look like an endless tunnel moving further and further away.

4.5 Scoring system

While this wasn't part of the initial concept, I added a score tally at the end of each battle to add some replay value for players. The two metrics for judgement were code attempts and damage taken. These values start at a maximum, 'perfect' value, and are decreased by a fixed penalty every single time the player a) takes damage or b) submits incorrect code. The trick here is, of course, that if the player submits working code on the first try every time to get a perfect code score, then they'll never take damage, ensuring a perfect damage score (with the exception of the first phase of the Dr. Fractal fight). The hope here was to encourage players to try the battles again and again to get a perfect score, which is indicated on the result screen - this would reinforce their

Haskell skills, as players would be more likely to retain solutions once they'd found them. Unfortunately, since this was a relatively last-minute addition, I couldn't flesh out this scoring system beyond the bare essentials.

4.6 Scrapped concepts - overworld and items

As mentioned in section 3.4, I originally wanted the game to have a larger scope, leaning heavily on its RPG inspirations by including an overworld where the player could explore the world they're trying to save, encounter smaller enemies and talk to non-player-characters (NPCs). Part of the overworld would be a series of stores, where the player could purchase items to help in battle by either restoring their health or providing hints for the current challenge.

However, by the time the list comprehension battle had been completed, I realised I didn't have enough time left to include all of this; the smaller enemies and the overworld in particular would be massive time sinks to design, implement and test. There would also be the additional concern of needing to implement a game-saving system to accommodate the longer length of the game - as such, those concepts were cut. Any necessary story details prior to each battle would then be conveyed through short interstitial cutscenes.

I still wanted to include the items, though - the plan was then to include the item shop screen in between battles, as the player would gain money at the end of each battle. The problems arose when I tried to figure out how the player's monetary rewards would scale. The current scoring system was based on dodging attacks successfully and submitting as little incorrect code as possible; if I used the same system for the monetary rewards, then the players getting the most money would be the least likely to actually use the items anyway. I couldn't come up with an alternative reward system for players, though, and the higher priority was getting the other battles ready so the game could cover all the required Haskell topics, so the items had to be cut as well.

Chapter 5

Implementation

The final version of the game loop in battles can be seen in Figure 5.1. As before, the emphasis was put on the two core aspects of 1) the Haskell puzzles and enemy attacks, and 2) the story progression through the battle to keep the players' interest. The following sections detail the implementation of the key mechanics and features

5.1 Implementing the Haskell challenges

In the final game, each script consists of two parts; the first is the code displayed to the player, which they are tasked with modifying, while the second is the code that checks if the player's solution behaves as expected. Let's use the Haskell code for the first phase in the Colonel Trigger-Finger fight as an example.

5.1.1 Code shown to player

```
1   type Target = String
2   type Angle = Float
3
4   data Missile = Missile Target Angle
5   -- represents homing missiles; the angle is the starting
6   -- angle of the missile
7
8   launch :: Target -> Int -> [Missile]
9   launch t n = [Missile t (angleGen i) | i <- [1..n]]
10  -- this launches n missiles at target t with random start angles
11
12  -- write a function to retarget all Missiles
13  -- we'll be using this to fire them back at the enemy!
14  retarget :: [Missile] -> Target -> [Missile]
15  retarget missiles t = -- INPUT HERE --
```

As I touched on before, the script defines custom data types using the 'data' keyword that map to the in-game objects; in this case, it's the Missile type. The 'type' keyword is also used to define aliases for certain pre-existing types; this is mainly done to improve readability. Details were added to help tie this code to the in-game behaviour, after

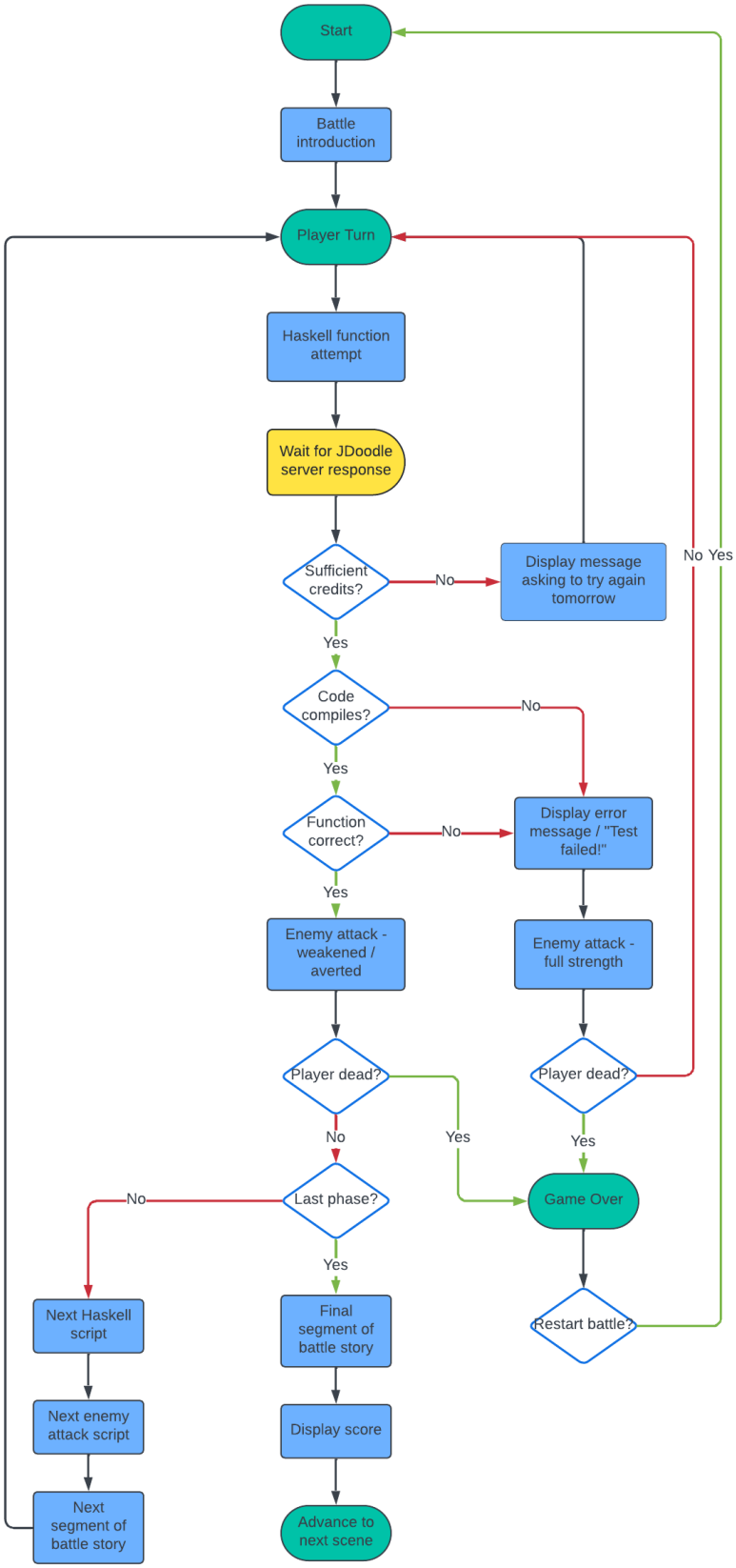


Figure 5.1: A flowchart detailing the game loop in battles.

feedback from the second marker indicated a lack of clarity in that area; in this case, it's the 'Angle' type and the 'launch' function.

However, the latter seems to violate the rules of pure FP languages; the 'angleGen' function seems to be a pure function that returns different values on each call, which is normally impossible. While Haskell does provide an interface to a pseudo-random number generator, that would be **far** outside the scope of the game, and too difficult for a new Haskell programmer to deal with - as such, the definition is hidden away from the player.

The Missile type having an 'Angle' value also helps verify that the retarget function written by the player takes a [Missile] and returns a modified version of that same [Missile], rather than ignoring the input and generating a separate list. A similar method was used for any other challenges where the idea was to modify a list of in-game objects.

The comments here are quite important; in each challenge, they help clarify what the purpose of each line is, as well as providing the player with instructions on what their function should do.

5.1.2 Auto-tester

```
16  -- TEST CODE
17  angleGen :: Int -> Float
18  angleGen i = fromIntegral i -- the player's not going to see
    this, the definition is just here to stop a compiler error
19
20  main :: IO()
21  main = do
22    let missiles = launch "player" 5
23    let newMissiles = retarget missiles "enemy"
24    let retargetCheck = (and [t == "enemy" | (Missile t _) <-
    newMissiles]) && not (null newMissiles)
25    let wrongTargets = [t | Missile t _ <- newMissiles, t /= "
    enemy"]
26    if null wrongTargets
27      then
28        print(retargetCheck)
29      else if head wrongTargets == "player"
30        then
31          print("error: Missiles still target player")
32        else
33          print("error: " ++ head wrongTargets ++ " is not a valid
     target")
```

Here, the problem of 'angleGen' from earlier is handled; it is a completely trivial function that just converts the input Int into a Float. This was done because implementing it using the aforementioned pseudo-random interface would lead to a completely-unnecessary hit to performance - the player never sees the expression at any point. Similar 'helper functions' in the other challenges are also handwaved away; in some cases, the tests themselves completely disregard the function, and the definition is only present to avoid a compiler error.

The code here is written in a do-block using an IO monad to allow for more standard imperative-style programming. The do-block allows for let-statements for multiple Booleans; usually, the tester needs to perform multiple checks on the value returned by the player's function. The 'main' monad is in place because JDoodle's compiler service is primarily built with imperative languages in mind, and therefore needs the equivalent of a main function - the IO monad also allows for the results of those checks to be printed, meaning they're in the output string returned by JDoodle.

In this specific instance, the auto-grader also prints custom error messages to provide more informative feedback if the player sets the missiles' Targets incorrectly (or more playful players who might want to mess around and set it to random string values). However, due to time constraints, I couldn't implement custom feedback for the other challenges - even in this one, while the code checks if the Angle values remain unaltered, there isn't a corresponding error message for if they **were** changed.

5.2 The in-game code editor

Figure 5.2: The in-game code editor, taken from the first phase of the Col. Trigger-Finger fight.

The code editor is an incredibly important part of the game, given that the entire central mechanic is built on writing and editing Haskell scripts. As such, the UI was designed to be as simple and unobtrusive as possible while maintaining the game's pixel-art aesthetic, in keeping with Nielsen's Usability Heuristics for interface design [43] - specifically, Principle 8, "Interfaces should not contain information that is irrelevant or rarely needed."

It was meant to resemble an actual text editor, since the intended demographic for the game was programmers with experience in imperative languages who had no experience

with Haskell - this is in accordance with Principle 2 of the Usability Heuristics, "The design should speak the users' language."

The code editor takes up half of the screen when active, and most of it is taken up by the Haskell script view - the lower dark blue area is devoted to displaying status/error messages. In addition to the obvious 'Submit' button, the player is also given a 'Reset' button in case they want to erase all their modifications.

5.2.1 Displaying Haskell code, JDoodle compilation requests

In each battle (represented as a Scene within Unity), the code editor has an associated array of text files, one for each phase. The code files themselves have the "-- TEST CODE" line immediately preceding the autotester, as seen in Section 5.1.2; this is present to allow the code string to be split into two halves, the first of which is the editable section shown to the player. Unity's InputField system [57], which was used for the editable text, allows for the addition of a vertical scrollbar - while most of the Haskell scripts are quite short, this was added anyway to allow the player's input to be as long as necessary. However, the usefulness of the scrollbar was limited by some strange behaviour in the InputField itself; when attempting to select the text so you can start typing, it jumps right back to the top anyway, so you'd have to manually scroll down using the arrow keys anyway.

Sending the code to JDoodle for compilation was also fairly straightforward; the service has REST endpoints that, respectively, return the number of credits you've used so far (though strangely not the ones **remaining**, which seems like it'd be the more useful value given that different plans have different limits), and return the results of the script compilation itself. The game code checks if the player can access the endpoints and if the account has sufficient compile credits remaining, and then sends the request; errors with being unable to access the URL or having insufficient credits are highlighted to the player using the method in section 5.2.3.

5.2.2 Comment highlighting

Modern text editors include various quality-of-life features like syntax highlighting and code indentation. The most necessary one to implement for HaskellQuest was the automatic highlighting of comments; since they were used to provide instructions and clarification in the Haskell scripts, it was essential to allow players to distinguish comments from code at a glance.

Unity's TextMeshPro UI system allows the use of HTML-esque rich text tags [49], particularly a '<color>' tag allowing you to change the colour of a substring. The code for the editor then includes a CommentHighlighting() function, executed any time the player adds a character to the code, that changes the colour of any comment text, starting from "--" up until the end of the line. The function also includes a corresponding check for the opposite scenario, where any text that doesn't have a "--" present has its <color> tags removed.

However, the use of these tags introduced some bugs. Most notably, once these invisible

characters were added to/removed from the text, the cursor in the InputField jumped forward. The InputField has a 'caretPosition' attribute controlling the position of the cursor, but forcing it to stay the same at the end of the CommentHighlighting function didn't resolve the issue for some reason. I also made the mistake of underestimating the impact this bug would have on players' experience, releasing the game to playtesters before actually fixing the issue. The fix was ultimately quite simple; the caret position was jumping on the frame **after** the alterations to the text had been made, so it simply had to be forced back on that frame instead.

5.2.3 Error display

With JDoodle's compilation errors, I made the decision to only show the first line describing the error type, e.g. "parse error (possibly incorrect indentation or mismatched brackets)". The reason is because while the JDoodle Haskell compiler does provide multi-line error descriptions highlighting where the errors are, leaving the full error risks showing the player the lines in the autotester code, as the error descriptions usually include a few lines after the erroneous one. Additionally, the error would only ever be on-screen for the duration of the enemy attack; including the full error description might be too much for the player to parse. The error display also displays custom error messages output by the autotester code, like in Section 5.1.2.

While most errors result in the enemy attacking the player, there is a special case to handle compiler timeouts - the game simply alerts the player to a timeout with yellow text (similar to how the errors in Section 5.2.1 are handled). This is because while the player causing an infinite loop (the usual reason for a timeout) is definitely a code error, there was once a point in testing the game where JDoodle sent continuous timeout responses for no discernible reason whatsoever. As such, I didn't want to have the risk of the player being penalised for something that wasn't their fault.

5.2.4 Animation code

I knew I wanted to only have the code editor on-screen when strictly necessary, as the alternative would be having the view of the enemy take up only half of the screen during dialogue segments that didn't involve writing code. At the time of writing the CodeEditor script, I didn't have much experience with Unity's Animator system; if I did, I would have known that it allows for developers to create animations for UI elements. Currently, though, all code for moving the editor on- and off-screen is done within the script frame-by-frame with the use of Unity's Coroutines [56] - this holds true for adjusting the position of the enemy view as well. Since these functions would need to be called by various other components throughout the game's execution, and since each battle scene would only ever have one instance of the editor, I decided to make the CodeEditor class a singleton.

5.3 Attack controllers

Each Haskell script needed to have an associated attack, so I created the AttackController abstract class in C# to denote the attack scripts - this way, when the CodeEditor gets the compilation result and needs to call the appropriate function to start the enemy's turn, it doesn't actually need to know the specific **type** of attack. This abstract class then defines a Trigger function that takes a Boolean representing the autotester result, which gets overridden with concrete implementations by all the actual attack scripts.

Optionally, the Trigger method is also overloaded with a version that accepts an 'additionalConditions' string, used for any extra values. In the final game, this is only used for the tutorial attacks that allow the player to change the damage value (as detailed in section 4.3.1), but the inclusion does future-proof the attack scripts a little by allowing for more complex interactions between the Haskell code and the attack behaviour.

As mentioned before, each AttackController only maps to one attack, meaning the enemy object in the attack phase has multiple associated AttackController scripts. In hindsight, I would have perhaps gone with one script per enemy, which initiates a different attack depending on the current phase of the battle. The current system leads to a lot of code duplication between different attack scripts. In the tutorial's case, there are 3 copies of the same attack script, as each actual attack there is functionally the same - it's only the Haskell script that changes.

5.4 In-battle story segments

5.4.1 Dialogue box

This was a key inclusion, both to allow character dialogue to play out and tell the story, as well as the more baseline requirement of allowing the Lambda-Man hologram to explain Haskell concepts in the tutorial battle. The DialogBox class was implemented as a singleton too, as there would also be only one instance per scene. Its main functionality is the StartDialogue method that takes a text file as a parameter, where each line of text is treated as a separate line of dialogue - the line is displayed character-by-character, though there is special handling for rich text tags.

The StartDialogue method is overloaded to allow for character names to be displayed, custom colours for those character names, and a sound effect to be played as each character in the line is drawn on-screen. The reason for the last inclusion is to allow each character to have a 'voice', in a sense, once again similar to Undertale [20]. Lambda-Man (the player character) has a fairly non-descript blip while the Hologram has the same blip pitched down an octave, Colonel Trigger-Finger's sound effect is harsh and robotic, while Dr. Fractal's sound effect has a high amount of reverb - these sound effects were created using an online tool called ChipTone [52].

However, this method of implementation has its downsides - in any given exchange between characters, they will most likely have to alternate lines of dialogue. However, each call to StartDialogue (and, thus, each associated text file) corresponds with the speech for only one character, meaning in most cases there are plenty of text files that

are only 1-2 lines long.

5.4.2 Enemy event controllers

The job of calling the StartDialogue function to have the dialogue segments play out correctly falls to these controllers, which all inherit from a common EnemyController superclass. This class defines an abstract PhaseTransition function, overridden by each enemy controller script, which takes an integer parameter representing the current phase of the fight. It is called by the CodeEditor script when the player has successfully finished one Haskell challenge, as can be seen in Figure 5.1. The class also includes a BattleEnd function, as well as some common functionality for the subclass implementations. Apart from the dialogue sequences, these controllers are also in charge of other aspects e.g. lowering the speed of Col. Trigger-Finger's animation to reflect him overexerting himself, or switching Dr. Fractal's animation at the end of the fight to represent Dr. Kowalewski being rescued.

5.5 Sprite art and character animation

From the beginning of the project, I knew I wanted to focus on the aesthetics of the game to attract players' attention and move away from Undertale's deliberately-minimalist style; I have extensive prior experience with digital art, and had experimented with retro-styled pixel art (a popular style for modern 'indie' games). Each battle would involve several elements:

- The 'main' view, a first-person view through Lambda-Man's eyes at the enemy, as seen in Figure 5.3
  - An animated large sprite of the enemy
  - The enemy's background, depicting the location of the battle
  - The code editor, depicting the Lambda Gauntlet on Lambda-Man's right hand, as seen in Figure 5.2 (obviously, this could be reused for all battles)
- The 'overhead / bird's eye' view of the battle, as seen in Figure 5.4 - this was intended to allow me to reuse overworld artwork for the battle segments, before the overworld was cut
  - A top-down, smaller sprite for the enemy
  - A top-down view of the battle environment
  - Projectiles fired by the enemy, with the exception of the missiles in the Col. Trigger-Finger fight, for which I used license-free art from OpenGameArt.org [51]
  - Four sets of sprites for Lambda-Man moving in four directions - in a holdover from the initial idea, these sprites took inspiration from the 'overworld' sprites in *Final Fantasy VI* [18]

All of these were created using a free pixel art tool, Pixelorama [32] - any animations were done frame-by-frame using the same tool, and implemented using Unity's Animator features. Originally, the cutscenes would have had images depicting the events described - due to time constraints, they ended up as simple text over licence-free pixel art sky backgrounds [13].

All the pixel art assets created for the game can be seen in Appendix B.

Figure 5.3: The 'main view' in Dr. Fractal's battle.

Figure 5.4: One of the 'overhead' attack phases in Dr. Fractal's battle.

Chapter 6

Evaluation

6.1 Method of evaluation

For playtesting the game, I sent out emails to undergraduate Informatics students in year 1, 2, 3 and 4. Playtesters would have their names included in the credits of the game as a 'thank you': the initial email only included a link to a sign-up form, so that only the people interested would be contacted with further details, and the actual evaluation forms could remain fully anonymous.

While UG1 students are theoretically closer to the 'beginner' demographic the game was designed in mind with, they would have had the most recent experience with Haskell through the University's mandatory Introduction to Computation course [29] - students in later years would have largely spent the overwhelming majority of time with imperative languages. As such, even though all UoE Informatics students have some prior experience with Haskell, the later-year students' comparative "rusty" Haskell skills would make them closer to a 'beginner' level, though again this wouldn't strictly apply to **all** later-year students.

21 students signed up as playtesters; they were provided with links to Windows, Mac and Linux versions of the game to maximise compatibility (this was made easy by Unity's native support for Mac and Linux builds), as well as links to two forms:

- The main evaluation form, allowing one response per person
- A form for reporting bugs, allowing unlimited responses per person

This scheme was taken, as with a lot of things, from E. Bogomil's HaskellQuest [7]. The playtesters are allowed to complete the game and report back in their own time, as opposed to a scheduled and observed session. With the large number of testers, scheduling might have proven difficult, but this method has the downside of needing to continually follow up with people to ensure they've submitted.

In the end, I received 17 responses for the evaluation form and 12 responses for the bug report form - the forms, along with the responses, are included in appendix sections A.1 and A.2.

6.2 Evaluation form

6.2.1 Questionnaire design

The evaluation form largely relies on subjective self-evaluation on the playtesters' parts. Objective evaluation of the testers' Haskell skills within the form would have been completely pointless, as the challenges in the game can't be cleared without understanding the Haskell concepts they're themed around. That said, the form does include a question asking testers where they stopped playing the game - this was done to measure any significant trends, as it would clearly be indicative of a problem if a large majority of the players were unable to get past a certain challenge, for instance.

The form includes fields for the testers to rate how **enjoyable** the game was and how difficult the hardest challenge they played was, as well as text fields for them to elaborate on what they liked and disliked the most about the game. Keeping the aforementioned variance in Haskell knowledge in mind, a field asking the tester to self-evaluate their Haskell experience prior to playing the game was added to the main evaluation form. Additionally, since none of the playtesters were exactly beginners, a field was added asking how they thought the game would fare for a complete newcomer.

The questionnaire usually talks about the game as a whole, allowing testers to mention specific aspects as they see fit; the sole exception is a question assessing the effectiveness of the tutorial dialogue. This is because it's a particularly load-bearing part of the experience; if it's too impenetrable for a beginner to understand, then the later sections would be too difficult to play through. Conversely, if it's unnecessarily detailed, then the player would try to skip through it, potentially missing key information, or they would get bored and quit the game entirely.

6.2.2 Overall results

The testers had a variety of answers for the question assessing their prior experience level with Haskell. On a scale of "1 - Complete novice" to "6 - I'm an experienced Haskell programmer", the most common answer was 3, selected by 7 testers (41%), followed by 2 and 4 (both selected by 4 testers, 24% each), and finally 1 and 6 (both selected by 1 tester, 6% each), leading to an average of 3.06.

Overall, the game was received positively; out of a maximum of 6, it scored a 4.71 for how enjoyable it was and a 4.29 for how effective it'd be at introducing people to Haskell. Additionally, in the text answers for what part of the game they liked the most, people overwhelmingly mentioned the presentation, story, and challenge design, validating the core approach:

> "I think the design, visuals, music and sound effects are all amazing. Clearly a lot of time has been put into these. I love how you tell a story with it too. You slowly introduce concepts and give simple explanations on how each of them work."

> "I liked the old school game vibe, the music, sfx, art, all of it just fit together well. The story was really cool too, and the challenges fit the theme really

well, so the immersion was great!"

Furthermore, in the responses for the question assessing where the testers stopped playing, the most frequent response (7 testers, 41%) was that they finished the game; operating on the prior assumption that clearing a section means the player understood the concept, this reflects well on the game's teaching potential.

6.2.3 Problems

However, the testers did have issues with the game. This is reflected in the score for the question assessing the difficulty; the average was 3.88 out of 6, skewing a little higher than I was expecting. Additionally, with the tutorial dialogue (assessed on a scale of "1 - Didn't cover enough at all" to "5 - Too excessive even for a beginner), 9 testers (53%) rated it a "3 - Just right", but that means that around half of the testers thought the dialogue was either too excessive or not helpful enough.

There were also some common trends observed in the answers for what testers would change about the game. Testers frequently requested a hint system or some kind of Haskell cheatsheet to help them when they were stuck in puzzles, as well as reminders of past errors. For instance:

> "I would maybe have some option for hints, or some reminders of syntax, because by the time I went through the dialogue where some syntax was explained, and arrived at the exercise, I did not 100% remember how [sic] guards look like."
>
> "The lack of hints when stuck, maybe with a cheatsheet of Haskell syntax."

While hints were at one point planned to be included as items (as mentioned in section 4.6), the syntax cheatsheet was **not**; given that a lot of the core design of the game was inspired by the Introduction to Computation course, though, it really should have been, seeing as that course offered a Haskell syntax cheatsheet that I personally referred to extremely often. It's also worth noting that E. Bogomil's HaskellQuest [7], which this project takes no small amount of inspiration from, implemented one as well i.e. the in-game Grimoire - there, it incentivised exploration and interaction with NPCs. Even though I wanted this game's design to be more linear, a cheatsheet should still have been included.

Additionally, testers complained about all the Haskell explanation being given right at the beginning of the game. For example:

> "I think that all of the explanation being right at the beginning would make it hard to remember by the last challenge. If I was a beginner I think that I would have struggled because the tutorial was too long ago. It might be better to have a more gradual introduction to Haskell's concepts throughout the game."

This is an issue of pacing, which could easily have been resolved by splitting up the tutorial section such that the 'battle' introducing a concept is placed right before the battle assessing said concept - arranging the game this way could have also potentially

mitigated the harm caused by the lack of hints, since the explanation for any given concept would be fresh in the player's mind as they fight the next enemy. The link between this problem and the lack of any hint system was well-explained by one tester:

> "A huge infodump at the beginning is definetely [sic] unfit for the zoomer attention span. With the combination of not being able to access the information afterwards, it will inevitably lead to skipping through the dialogue, forgetting everything you read, and then having to look stuff up when you actually need them. In essence you're perpetuating the problem that educational games are trying to solve -> remembering stuff is boring."

Some testers complained about the lack of information in error messages (an unintended consequence of the decision to only keep the first line), while others mentioned that they would have liked to go back through prior error messages to see what went wrong and how to fix it. Additionally, one tester mentioned that they occasionally missed some dialogue and thus would have liked the option to re-read prior lines; this kind of "dialogue log" has become commonplace in modern games, and could greatly help with the lack of clarity that some testers experienced.

6.3 Bug reports

The bug report form included a field for the tester to indicate what area of the game the bug was spotted in, as well as a text field for them to describe the bug in more detail and include steps to recreate it (if they knew the cause), and a field for any supporting screenshots / videos. The bug reports fell into a few broad categories detailed below.

6.3.1 Code editor bugs and shortcomings

One of the very first reports was of the cursor jumping after adding/removing comments, as described in section 5.2.2. Some were to do with issues with Unity's InputField class, like the scroll-bar issue mentioned in section 5.2.1, or not being able to position the cursor at the end of a line without using the arrow keys. One bug was that the backtick character wasn't being rendered correctly; this was fixed in the same update that fixed the comment highlighting bug, along with some typos in dialogue text.

There was also one bug reported that I'd encountered sporadically during my own testing, but which I could never recreate consistently - when pressing Backspace at the beginning of a line, sometimes the two lines would be displayed atop one another. This is only an issue with the rendering - the actual text itself is stored properly, meaning any compilation requests would work as expected - but it still negatively impacts the user experience.

In hindsight, with all the issues caused by the InputField being used as a code editor, it may have been easier and better to use a premade asset instead, since the editing functionality there was the primary focus of the developers. Trivial Interactive offers a pre-made 'In-Game Code Editor' asset for purchase on the Unity Asset Store [33] that handles highlighting for keywords and comments, and allows the user to customise the rules for recognising both. A more expensive alternative [50], by Sandro Ropelato,

includes more quality-of-life features such as undo and redo operations (which the InputField does not offer by default), which one tester had specifically requested in their response to the evaluation form.

6.3.2 Exploits and unintended behaviour

This category includes behaviour like the exploit of "setting the damage list to be all zeros" mentioned in section 4.3.1, which bypasses the intended point of the tutorial in modifying the damage functions themselves and understanding how they work through that. Additionally, one tester reported an exploit in Col. Trigger-Finger's 'homing missile' attack: if the player stands behind / on top of the Colonel in the attack phase, the missiles can never hit the player. They mentioned that the game didn't progress as though this was unexpected behaviour, but didn't mention if the missiles were hitting the Colonel or not, which is the in-story reason for the battle to progress to the next phase. This speaks to some lack of clarity in the story text, bolstered by another tester's confusion in the evaluation form at why the Colonel started exploding at the end of the battle.

In general, though, some of the homing missiles can hit the Colonel in the attack phase even when the Haskell puzzle hasn't been solved. This is because the behaviour of the homing missiles is identical to the original battle in Undertale [20]; the homing missile continually rotates towards the player's position for some duration of time before it 'locks on' to a straight path, and the original solution in Undertale was for the player to position themselves so that the missiles were pointed towards the enemy and then get out of the way once they'd locked on. The specific values set for the missiles' locking timer and rotation speed in HaskellQuest make it difficult for the player to cause **all** the missiles to collide with the enemy (with the potential exception of the scenario described in the bug report), but this behaviour nevertheless causes some dissonance between the game's intended narrative and how the attack actually plays out.

6.3.3 Platform compatibility issues

In the pre-release testing for the Mac and Linux versions, I ran into issues that were later reported by testers as bugs. The Mac version has security issues to do with the lack of entitlements [17] required for applications on the platform, which could potentially require signing up as an Apple developer. The workaround (included in a README) involves running a command in the Terminal and repeatedly attempting to open the application until a prompt allows the player to bypass the security check, which is frustrating and error-prone; one tester reported having to repeat the steps 5 times before the application opened.

The Linux version, when tested on DICE machines, opened in an incorrect aspect ratio, meaning players could see past the intended boundaries of the screen and the bottom of the screen was slightly cut off; this was reported by a tester using Debian as well. Unfortunately, I could find no reference to this issue online, and the window resolution was set to a fixed 1920x1080 for all versions of the game in Unity's build settings, so I have no idea what could possibly be causing this.

Chapter 7

Conclusion

7.1 Achievements

The main achievement of this project is the successful design and implementation of the core game mechanics e.g. player movement, compiler-game interface, dialogue boxes, and so on, as well as varied attacks, enemies and Haskell challenges to make the game feel sufficiently eclectic and enjoyable. Additionally, this implementation is scalable - the AttackController and EnemyController class structure, for example, allows for new battle mechanics to be added easily, and adding/altering a battle's aesthetics and story is as simple as swapping out the art and dialogue text.

Despite the emphasis on presentation, the game's design still ensures that the core learning objectives are met - the game's challenges were intended to interweave the introduction and assessment of all the required Haskell concepts with storytelling and creative gameplay scenarios. The results of user testing have shown that this design satisfies the requirement for effective and entertaining teaching; the testers praised the aesthetics, story and design, positively rated the game's effectiveness for newcomers (though with the caveats mentioned in section 6.2.3), and 41% enjoyed the game enough to play it all the way through.

In terms of personal development, I've learned a lot in conveying an interesting story in the framework of a game, i.e. through dialogue, game mechanics and graphical assets. In the grand scheme of things, I've also learned what to do and what **not** to do when developing a game. For instance, the original scope was, in hindsight, clearly far too ambitious for the time frame I actually had; however, I didn't have the necessary experience and knowledge to figure that out at the very start of development.

7.2 Future work

In terms of the game in its current form, some tweaks and improvements could definitely be made to greatly improve the player experience - for instance, adding the quality-of-life text editor features mentioned in sections 5.2.2 and 6.3.1, or splitting the tutorial section up, adding a Haskell cheatsheet and improving the error messages as described

in section 6.2.3. Furthermore, my supervisor suggested recursion that could have been included in the Dr. Fractal battle: recursion with multiple base cases, recursion with guards on the recursive case, and having multiple recursive calls in a single function call - the progression of the battle could have been based on these.

Throughout this report, I've mentioned concepts and sections that were originally intended to be in the game, but couldn't be because of time constraints - some of these inclusions could go a long way in alleviating the problems found during playtesting.

- The overworld, 'regular' enemies and item system, as described in section 4.6 could be added in full - this could curb any potential worries about excessive difficulty by adding more warm-up challenges and creating a better sense of progression, as well as allowing more time for those concepts to fully sink in. Additionally, allowing players to explore the world would increase their immersion in the story, and therefore their enjoyment.
- The game could be expanded to include a section themed around higher-order functions - the player could be introduced to the in-built map, fold and filter functions, as well as having to write functions that take other functions as an input. This, in itself, could greatly increase the game's educational value; higher-order functions and lambda expressions are two of the FP features that have made their way over to imperative languages like Java, Python, Kotlin, etc.

Additionally, in discussions with my supervisor, a potential alternative expansion was brought up. The game could be reworked to include a multiplayer campaign - two players could go through the game simultaneously, working on Haskell code together and dodging more complex projectile patterns from the enemy. This would require a lot more research on my part into networking in general and game netcode [62] specifically - however, the gameplay experience of having two players bouncing off ideas and working together to beat the Haskell challenges could prove to be an enjoyable learning experience.

A much simpler multiplayer aspect could be added by including a high score table for each battle, with players competing to get to the top. However, the current scoring essentially means that players get a perfect score if they replay the battle while knowing the answers - this would make it far too easy for experienced players to reach the top, so the score system would have to be reworked beforehand.

Essentially, while this initial release was largely a success proving the validity of the approach, the core concept can be taken much further to enhance the game's entertainment factor and educational effectiveness.

Bibliography

[1] *6.2.6 Parallel List Comprehensions - GHC User's Guide*. URL: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/parallel_list_comprehensions.html.

[2] Clark C Abt. *Serious Games*. New York: Viking Press, 1970.

[3] Peter Achten. "The Soccer-Fun project". In: *Journal of Functional Programming* 21.1 (2011), pp. 1–19. DOI: 10.1017/S0956796810000055.

[4] *Asteroids*. [Arcade]. Atari, 1979.

[5] Per Backlund and Maurice 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)*. 2013, pp. 1–8. DOI: 10.1109/VS-GAMES.2013.6624226.

[6] Judith Bishop et al. "Code Hunt: Experience with coding contests at scale". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 398–407.

[7] Eva Bogomil. "HaskellQuest: a game for teaching functional programming in Haskell". Bachelor's Thesis. School of Informatics, University of Edinburgh, 2023.

[8] Ross Bramble. *The Seven Stages of Game Development - GameMaker Blog*. May 10, 2023. URL: https://gamemaker.io/en/blog/stages-of-game-development.

[9] *Canva - Visual Suite For Everyone*. URL: https://www.canva.com/.

[10] *Code Execution APIs | OneCompiler*. URL: https://onecompiler.com/apis/code-execution.

[11] CoderPad. *CodinGame*. URL: https://www.codingame.com.

[12] *Computer Science BSc Hons - University of Nottingham*. URL: https://www.nottingham.ac.uk/studywithus/ugstudy/courses/UG/Computer-Science-BSc-Hons-U6UCMPSC.html.

[13] CraftPix. *Free Sky Backgrounds*. URL: https://free-game-assets.itch.io/free-sky-with-clouds-background-pixel-art-set.

[14] Maxim Despinoy. “Having fun learning - A hidden component to success”. Bachelor’s Thesis. School of Informatics, University of Edinburgh, 2023.

[15] Edsger W Dijkstra. *To the Budget Council (concerning Haskell)*. Apr. 2001. URL: https://www.cs.utexas.edu/users/EWD/OtherDocs/To%20the%20Budget%20Council%20concerning%20Haskell.pdf.

[16] Karolina Drobnik. “HaskellQuest: a game for teaching functional programming in Haskell”. Master’s Thesis. School of Informatics, University of Edinburgh, 2018.

[17] *Entitlements | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/bundleresources/entitlements.

[18] *Final Fantasy VI*. [SNES]. Square, 1984.

[19] Godot Foundation. *Godot Engine*. URL: https://godotengine.org/.

[20] Toby Fox and Temmie Chang. *Undertale*. [PC, Mac, Linux, PS4, PS Vita, Nintendo Switch, Xbox One]. 2015. URL: https://undertale.com/.

[21] *Functional Programming (2023-24) - University of Oxford*. URL: https://www.cs.ox.ac.uk/teaching/courses/2023-2024/fp/.

[22] *GameMaker*. YoYo Games. URL: https://gamemaker.io/en.

[23] James Hartley. *Learning and Studying: A Research Perspective*. Routledge, 1998.

[24] *Haskell Online IDE & Code Editor for Technical Interviews*. URL: https://coderpad.io/languages/haskell/.

[25] HaskellWiki. *Applications and libraries/Games — HaskellWiki*. [Online; accessed 14-March-2024]. 2023. URL: https://wiki.haskell.org/index.php?title=Applications_and_libraries/Games&oldid=66354.

[26] Paul Hudak et al. “Report on the programming language Haskell: a non-strict, purely functional language version 1.2”. In: *SIGPLAN Notices* 27 (Jan. 1992).

[27] *Human Resource Machine*. [Windows, Mac OS X, Linux, iOS, Android, Wii U, Nintendo Switch]. Tomorrow Corporation. URL: https://tomorrowcorporation.com/humanresourcemachine.

[28] Armando Iannucci et al. *The Death of Stalin*. 2017.

[29] *Informatics 1 - Introduction to Computation: University of Edinburgh*. URL: http://www.drps.ed.ac.uk/23-24/dpt/cxinfr08025.htm.

[30] *Insights: Global Entertainment and Media Outlook 2023-2027 | PwC*. June 21, 2023. URL: https://www.pwc.com/gx/en/industries/tmt/media/outlook/insights-and-perspectives.html.

[31] *Installation - GHCUp*. URL: https://www.haskell.org/ghcup/install/.

[32] Orama Interactive. *Pixelorama*. URL: https://orama-interactive.itch.io/pixelorama.

[33] Trivial Interactive. *InGame Code Editor | GUI Tools | Unity Asset Store*. URL: https://assetstore.unity.com/packages/tools/gui/ingame-code-editor-144254.

[34] *JDoodle - Integrate online compiler plugin and API*. URL: https://www.jdoodle.com/intergrate-online-ide-compiler-api-plugins.

[35] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. "Teaching functional programming to first-year students". In: *Journal of Functional Programming* 3.1 (1993), pp. 49–65. DOI: 10.1017/S0956796800000599.

[36] C. Larman and V.R. Basili. "Iterative and incremental developments. a brief history". In: *Computer* 36.6 (2003), pp. 47–56. DOI: 10.1109/MC.2003.1204375.

[37] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011.

[38] Christoph Lüth. "Haskell in Space: An interactive game as a functional programming exercise". In: *Journal of Functional Programming* 13.6 (2003), pp. 1077–1085. DOI: 10.1017/S0956796803004891.

[39] Madnight. *GitHut 2.0 - GitHub language stats*. URL: https://madnight.github.io/githut/#/pull_requests/2023/2.

[40] *Mega Man / Rockman*. [NES]. Capcom, 1987.

[41] *Mighty Morphin' Power Rangers: The Movie*. Saban Entertainment, Toei Company, 20th Century Fox, 1995.

[42] Michael A. Miljanovic and Jeremy S. Bradbury. "A Review of Serious Games for Programming". In: *Serious Games*. Ed. by Stefan Göbel et al. Cham: Springer International Publishing, 2018, pp. 204–216. ISBN: 978-3-030-02762-9.

[43] Jakob Nielsen. "Enhancing the explanatory power of usability heuristics". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '94. Boston, Massachusetts, USA: Association for Computing Machinery, 1994, pp. 152–158. ISBN: 0897916506. DOI: 10.1145/191666.191729. URL: https://doi.org/10.1145/191666.191729.

[44] *Online compiler API - Sphere Engine Compilers*. URL: https://sphere-engine.com/compilers.

[45] *Online Haskell Compiler - TutorialsPoint*. URL: https://www.tutorialspoint.com/compile_haskell_online.php.

[46] *Online Haskell Compiler and Interpreter - Replit*. URL: https://replit.com/languages/haskell.

[47] *Process class (System.Diagnostics) - Microsoft Learn*. URL: https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.process.

[48] Anthony Quinn. *Wadler follows the giants of science - IOHK Blog*. July 2023. URL: https://iohk.io/en/blog/posts/2023/07/06/professor-philip-wadler-follows-the-giants-of-science-at-the-royal-society/.

[49] *Rich Text: TextMeshPro | 4.0.0-pre.2*. URL: https://docs.unity3d.com/Packages/com.unity.textmeshpro@4.0/manual/RichText.html.

[50] Sandro Ropelato. *In-Game Text Editor | GUI Tools | Unity Asset Store*. URL: https://assetstore.unity.com/packages/tools/gui/in-game-text-editor-199113.

[51] samoliver. *Missile | OpenGameArt.org*. URL: https://opengameart.org/content/missile-0.

[52] SFBGames. *ChipTone*. URL: https://sfbgames.itch.io/chiptone.

[53] SimulaVR. *godot-haskell*. URL: https://github.com/SimulaVR/godot-haskell.

[54] Sogomn. *Explosion | OpenGameArt.org*. URL: https://opengameart.org/content/explosion-3.

[55] D. A. Turner. "Miranda: A non-strict functional language with polymorphic types". In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 1–16. ISBN: 978-3-540-39677-2.

[56] *Unity - Manual: Coroutines*. URL: https://docs.unity3d.com/Manual/Coroutines.html.

[57] *Unity - Manual: Input Field*. URL: https://docs.unity3d.com/2022.3/Documentation/Manual/script-InputField.html.

[58] *Unity Engine*. Unity Technologies. URL: https://unity.com/.

[59] *Unreal Engine*. Epic Games. URL: https://www.unrealengine.com/en-US.

[60] Wikipedia contributors. *Bullet hell — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-March-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Bullet_hell&oldid=1213319327.

[61] Wikipedia contributors. *Lizard (character) — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-April-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Lizard_(character)&oldid=1216863865.

[62] Wikipedia contributors. *Netcode — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-April-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Netcode&oldid=1187616649.

[63] Zehui Zhan et al. "The effectiveness of gamification in programming education: Evidence from a meta-analysis". In: *Computers and Education: Artificial Intelligence* 3 (2022), p. 100096. ISSN: 2666-920X. DOI: https://doi.org/10.1016/j.caeai.2022.100096. URL: https://www.sciencedirect.com/science/article/pii/S2666920X22000510.

Appendix A

User testing

A.1 Evaluation form

1. I allow my data to be used in future ethically approved research.

More Details Insights

Yes	17
No	0

2. I agree to take part in this study.

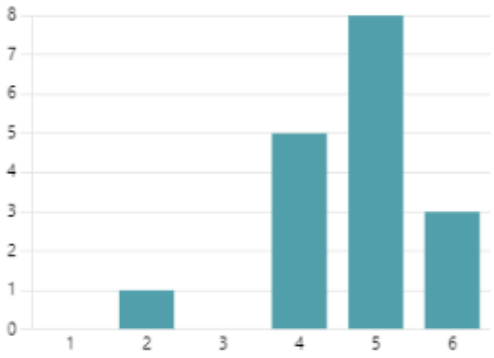
More Details

Yes	17
No	0

3. How **enjoyable** did you find the game?
(1 - It was unsatisfactory, 6 - I loved it)

More Details Insights

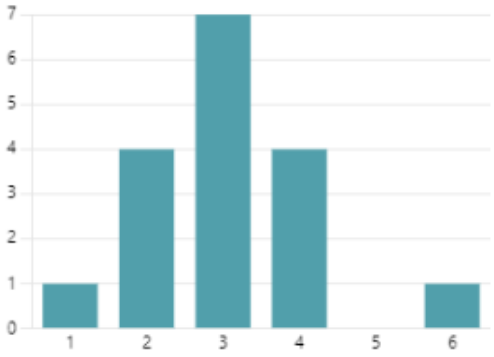
4.71
Average Rating



4. Prior to playing the game, how much **Haskell experience** would you say you had?
(1 - Complete novice, 6 - I'm an experienced Haskell programmer)

More Details Insights

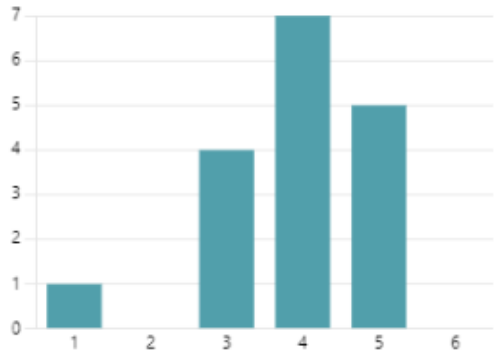
3.06
Average Rating



5. How difficult did you think the **most difficult** challenge was in this game?
(1 - Incredibly easy, 6 - Basically unfair)

More Details Insights

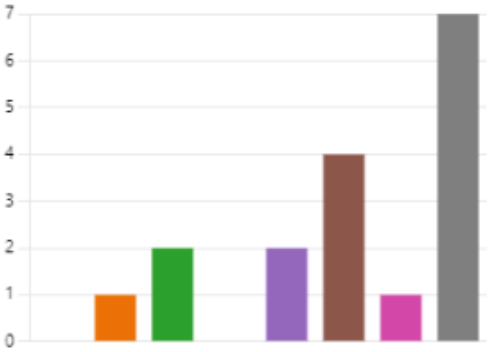
3.88
Average Rating



6. What challenge did you **last beat** in the game?

More Details Insights

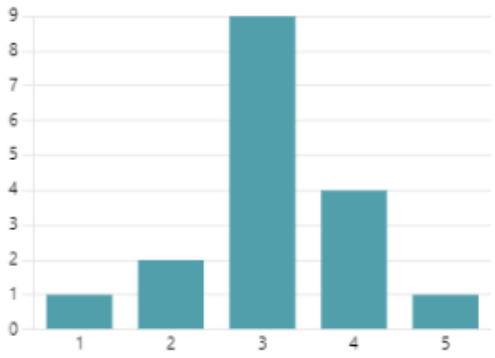
Tutorial Phase 1 - Intro to Functi...	0
Tutorial Phase 2 - Intro to Lists a...	1
Tutorial Phase 3 - Intro to Recur...	2
Colonel Phase 1 - Homing Missi...	0
Colonel Phase 2 - Missile Volley	2
Colonel Phase 3 - Combo Attack	4
Fractal Phase 1 - Splitting Fractal	1
Fractal Phase 2 - Saving Kowale...	7



9. How would you rate the **tutorial dialogue** in the first battles i.e. versus the Hologram?
(1 - Didn't cover enough at all, **3** - **Just right**, 5 - Too excessive even for a beginner)

More Details Insights

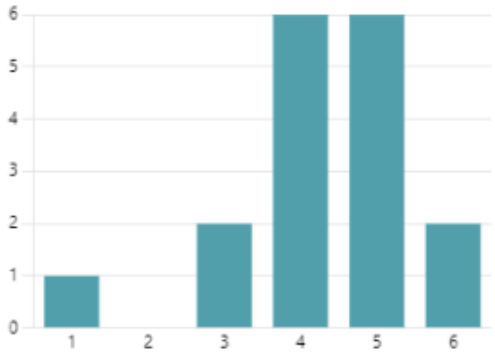
3.12
Average Rating



10. Overall, how effective do you think this game would be at **introducing people to Haskell**?
(1 - Ineffective, 6 - Very effective)

More Details Insights

4.29
Average Rating



Full list of responses for Q7: "What did you like most about the game?"

- I loved the pixel art of the bosses, the hologram and the backgrounds. Great art style.
- The art, music. I liked how the solutions affected the gameplay. It was creative. I liked being in the credits
- I liked the old school game vibe, the music, sfx, art, all of it just fit together well. The story was really cool too, and the challenges fit the theme really well, so the immersion was great!
- the visuals, music and overall concept
- The graphics and dialogue were really cool
- The interface was really cool
- The art style, followed by the Undertale inspired gameplay.
- The interface where you wrote the code was very well done.
- Honestly, I did not manage to pass past the tutorial because I'm a total dunce (I'll get my friend to play later with me to see if we can get better) but I loved the aesthetic of it. The game was also really informative, I think it covered enough information to get people into it and provide the correct information, and it was very enjoyable to play. Well done!
- Love the art style, music and overall feel. Best part was the fun of moving around a dodging things once you figure out the solution
- The battle system makes a clever use of haskell as well as some regular game skills. The tasks are also clearly defined so at no point did I feel lost. It offers a lot of potential for future fights, although it would be nice to also have some degree of freedom once the basics are out of the way where the challenge is to figure out what to actually do. In that regard there could be parts of unchangeable code that you have to work around as well.

  The graphics and overall presentation was good too.
- The execution of the visual and auditory aspects is top notch, which helps to engagement users to learn
- I think the design, visuals, music and sound effects are all amazing. Clearly a lot of time has been put into these. I love how you tell a story with it too. You slowly introduce concepts and give simple explanations on how each of them work.
- Music, graphics, creativity of challenges
- The graphics and sound.
- It had personality. I especially liked seeing all the functional programming references
- The art style, as well as how the game teaches you Haskell.

Full list of responses for Q8: "What would you change about the game?"

- I think that all of the explanation being right at the beginning would make it hard to remember by the last challenge. If I was a beginner I think that I would have struggled because the tutorial was too long ago. It might be better to have a more gradual introduction to Haskell's concepts throughout the game.
- Hints, ability to go back through dialogue after, way too much text, ability to undo, error checking (basically stuff a text editor would do), Explain why solution was wrong
- I would maybe have some option for hints, or some reminders of syntax, because by the time I went through the dialogue where some syntax was explained, and arrived at the exercise, I did not 100 % remember how guards look like. I would also keep the error visible after you fail the test, so that you can be reminded by what went wrong.
- The lack of hints when stuck, maybe with a cheatsheet of Haskell syntax.
- Have an option for hints, maybe paying through lives
- The programming language being taught (but that's just my dislike for Haskell), other than that it was cool.
- Intro forces you to understand how zip works for later stages.
- Add some more RPG elements, but this is more on the game side than the Haskell side. Maybe have a "weapons dictionary" which is just a Haskell function lookup table or something.
- Ok so as I said before, I'm a total dunce. I did not get past the tutorial, and I'm pretty sure that's because I somehow cheated my way through it (sorry about that). I would have fully appreciated a hint or a help button in the next stages, that is like, if you take an attack three times in a row you have the option to receive help. Whilst I am aware that some people might use this to cheat, some people (aka me) would appreciate some hints to get through it, because I could not get past it because I simply had no clue how to get better.
  This is simply a pet peeve of mine, but I would have appreciated a menu button with different options for like music, a little diagram of the progression of the story, and the ability to go back on the text because I accidentally clicked the forward button too many times and missed some important text.
- The intro dialogue is fun, useful and a good intro to the story, but quite long, and once you've read it once its gone, then if you forget something important about haskell you have to restart to read it again (or google it). Maybe some kind of super short summary or example screen for the important parts of haskell would help. In the final round of the colonel fight, how does he end up exploding as all i did was stop his attacks? In the tutorial, the hologram's attacks could have an effect or something to introduce the player to the movement and dodging part of the game, and to actually give a reason for taking damage if they get it wrong. Also, even if the tutorial problems are done right, the red screen flash effect still

plays which is slightly confusing. Finally, in the non tutorial fights, there are some functions which aren't defined in the visible code, and while they don't affect the gameplay and aren't needed to beat the game, they might still be nice to see somewhere like in a different file using the tab-like system at the top or in a separate window, just as more examples of proper haskell code.

- A huge infodump at the beginning is definetely unfit for the zoomer attention span. With the combination of not being able to access the information afterwards, it will inevitably lead to skipping through the dialogue, forgetting everything you read, and then having to look stuff up when you actually need them. In essence you're perpetuating the problem that educational games are trying to solve -> remembering stuff is boring.

  One simple fix that would increase the quality of the experience considerably would be to include a "journal" to which you could refer to when doing the actual fights that had all the info from the dialogue. Otherwise I'd recommend reevaluating how information is passed on to the player as well as how it is paced.

  Also the game could do with some more game juice, but that was probably not the priority.

- During the tutorial phases, it would be better to have more visual annotations of what part of the code does what, along with the narration, e.g., interactive examples showing changing the code changes code behaviour, and to what effect.

- I would potentially make a way to set your experience in Haskell, as the step up from the tutorial to even the homing missiles activity. As someone who hasn't taken Haskell in several years, I was unfamiliar with a lot of concepts. Potentially add more examples/code visuals when explaining each concept in the tutorial. Equally, having hints you could use to give some help would be great, or even revealing the answer if you're completely stuck, as I got stuck and couldn't figure out how to progress further (unless I've missed that option)!

- Better error messages. Ability to look at previous level's solutions. Ability to look at tutorial level's solutions. I played the game over a week and forgot how to Haskell. Being able to look over tutorial again would have been very helpful.

- Tutorial should be more interactive, reduce the premable before coding but have more intermediary coding steps.

- My hand holding at the start

- Some of the challenges are confusing; the game doesn't give clear directions on how you're supposed to use the function templates given to solve challenges.

Full list of responses for Q11: "Any other comments?"

- It was great, glad I could help! Could definitely help first year students with Haskell!

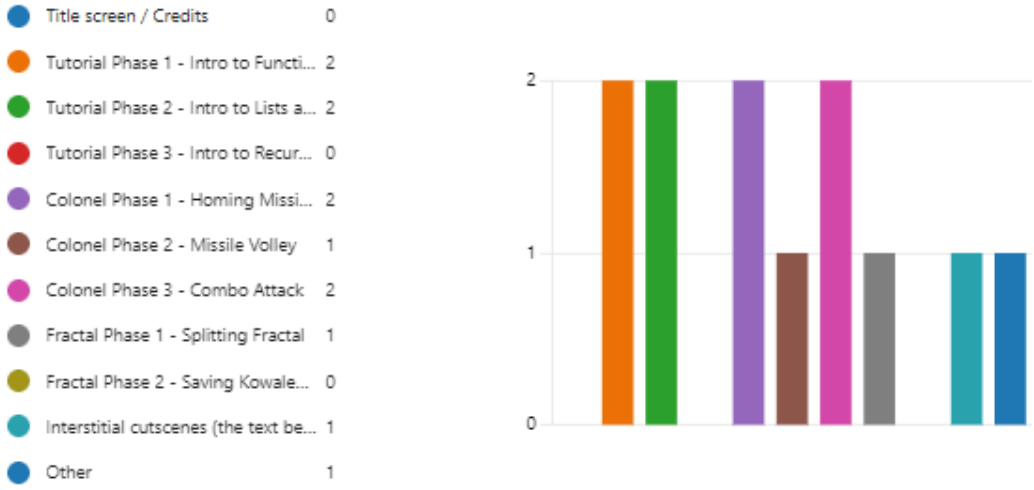
- great design

- Love the graphics
- Maybe a bit too hard, no tutorial on maps, which were useful for some challenges before they are introduced.
- Please add the hints I died so badly
- Really cool game and would love to see more with an expanded story! :)
- Note: don't take my answer for question 9, i mostly skipped through the dialogue lol. **(Author's note: uh oh)**
- I found the game extremely difficult as someone who is in 4th year who hasn't touched Haskell since 1st year. Tutorial needs to he more detailed. Should be able to open a menu and reread the tutorial during the game. Should be better error messages

A.2 Bug report form

1. Where did you encounter the bug?

More Details



Full list of responses for Q2: "Please describe the bug, as well as how to recreate it if possible"

- Even numbers list comprehension example uses '=' instead of '==' to check equality. (**Author's note: addressed by the update during testing**)
- Platform: Windows

  Also occurs on other challenges.

  If I try to delete a comment, my cursor jumps down several lines. Also when creating a new comment my cursor skips to before the start of the comment. The comment disappears if I then press backspace putting the comment on the previous line, and reappears when I press enter again. Attempting to delete a comment breaks the formatting of all code after it until I press reset. (**Author's note: addressed by the update during testing**)
- Platform: Windows

  Unable to put the caret on the last line using the mouse. Able to move onto the line using arrow keys.
- Platform: Windows

  Infix backticks are rendered above other characters rather than between them. (**Author's note: addressed by the update during testing**)
- After a gameover, beating it caused it to be stuck on the page where you are looking at the Undertale style battle, and nothing happens. Might be true on other pages but on this one it happened twice?

- Took around 5 tries following the steps to open the game. Maybe I was just too impatient. (**Context: this user was running the MacOS version, mentioned in the answer for Q1 after selecting 'Other'**)
- I think it's possible to just stand behind the attacker and never lose health while also not progressing in the game.
- Not exactly a bug but running the game on Debian Linux I cannot make the game full-screen so the bottom of the game is cut off a little bit
- I was allowed to "cheat" my way through the tutorial because I was just able to change the damage list rather than the code, which idk if is a bug but kind of defeats the purpose of the tutorial.
- Sometimes when reading the text, enter or space bar didn't work to pass the test, might just be an internal bug from me
- When making the following changes:
  Setting the list to d = [0] and the condition in the attack function to d = 0
  Made me take damage, when I don't think that should be possible
- I wrote code on the next line and when I tried to delete the enter so that the two lines would be joined, it bugged and joined the two lines of text

A.3 Participant information sheet

Participant Information Sheet

Project title:	HaskellQuest: a game for teaching functional programming in Haskell
Principal investigator:	Don Sannella
Researcher collecting data:	Neel Amonkar
Funder (if applicable):	NA

This study was certified according to the Informatics Research Ethics Process, reference number 297129. Please take time to read the following information carefully. You should keep this page for your records.

Who are the researchers?

This is an undergraduate research study, conducted by Neel Amonkar and supervised by Dr Donald Sannella.

What is the purpose of the study?

The goal of the project is to create an educational game to teach people the programming language Haskell, aimed at students/developers who have no prior experience with it. The results of this study will help be used to evaluate the effectiveness of the game, as well as potentially improve it.

Why have I been asked to take part?

The target group for this study is programmers who have never used Haskell before, or have done so but not recently.

Do I have to take part?

No – participation in this study is entirely up to you. You can withdraw from the study at any time, up until the submission of your questionnaires, without giving a reason. After this point, personal data will be deleted and anonymised data will be combined such that it is impossible to remove individual information from the analysis. Your rights will not be affected. If you wish to withdraw, contact the PI. We will keep copies of your original consent, and of your withdrawal request.

What will happen if I decide to take part?

Specify:

- Kinds of data being collected: Participants' experience with the game, previous Haskell knowledge (if any); no personal data collected
- Means of collection: questionnaire
- Duration of session: as long as the participant takes to answer the questionnaire, >5 min
- If participant audio/video is being recorded: No
- How often, where, when: Once, mid-March, participant questionnaire submitted online

Are there any risks associated with taking part?

There are no significant risks associated with participation.

Are there any benefits associated with taking part?

A 'thank you' in the credits of the game.

What will happen to the results of this study?

The results of this study may be summarised in published articles, reports and presentations. Quotes or key findings will be anonymized: We will remove any information that could, in our assessment, allow anyone to identify you. With your consent, information can also be used for future research. Your data may be archived for a maximum of 4 years. All potentially identifiable data will be deleted within this timeframe if it has not already been deleted as part of anonymization.

Data protection and confidentiality.

Your data will be processed in accordance with Data Protection Law. All information collected about you will be kept strictly confidential. Your data will be referred to by a unique participant number rather than by name. Your data will only be viewed by the researcher Neel Amonkar and the supervisor Don Sannella.

All electronic data will be stored on a password-protected encrypted computer, on the School of Informatics' secure file servers, or on the University's secure encrypted cloud storage services (DataShare, ownCloud, or Sharepoint) and all paper records

will be stored in a locked filing cabinet in the PI's office. Your consent information will be kept separately from your responses in order to minimise risk.

What are my data protection rights?

The University of Edinburgh is a Data Controller for the information you provide. You have the right to access information held about you. Your right of access can be exercised in accordance Data Protection Law. You also have other rights including rights of correction, erasure and objection. For more details, including the right to lodge a complaint with the Information Commissioner's Office, please visit www.ico.org.uk. Questions, comments and requests about your personal data can also be sent to the University Data Protection Officer at dpo@ed.ac.uk.

Who can I contact?

If you have any further questions about the study, please contact the lead researcher, Neel Amonkar, at n.amonkar@sms.ed.ac.uk.

If you wish to make a complaint about the study, please contact inf-ethics@inf.ed.ac.uk. When you contact us, please provide the study title and detail the nature of your complaint.

Updated information.

If the research project changes in any way, an updated Participant Information Sheet will be made available on http://web.inf.ed.ac.uk/infweb/research/study-updates.

Alternative formats.

To request this document in an alternative format, such as large print or on coloured paper, please contact Neel Amonkar at n.amonkar@sms.ed.ac.uk.

General information.

For general information about how we use your data, go to: edin.ac/privacy-research

A.4 Participant consent form

Participant number:_____

Participant Consent Form

Project title:	HaskellQuest: a game for teaching functional programming in Haskell
Principal investigator (PI):	Donald Sannella
Researcher:	Neel Amonkar
PI contact details:	don.sannella@ed.ac.uk

By participating in the study you agree that:

- I have read and understood the Participant Information Sheet for the above study, that I have had the opportunity to ask questions, and that any questions I had were answered to my satisfaction.
- My participation is voluntary, and that I can withdraw at any time without giving a reason. Withdrawing will not affect any of my rights.
- I consent to my anonymised data being used in academic publications and presentations.
- I understand that my anonymised data will be stored for the duration outlined in the Participant Information Sheet.

Please tick yes or no for each of these statements.

1. I allow my data to be used in future ethically approved research. ☐ **Yes** ☐ **No**

2. I agree to take part in this study. ☐ **Yes** ☐ **No**

Name of person giving consent	Date dd/mm/yy	Signature

Name of person taking consent	Date dd/mm/yy	Signature

THE UNIVERSITY of EDINBURGH informatics

Appendix B

Sprite art

Figure B.1: The game's logo.

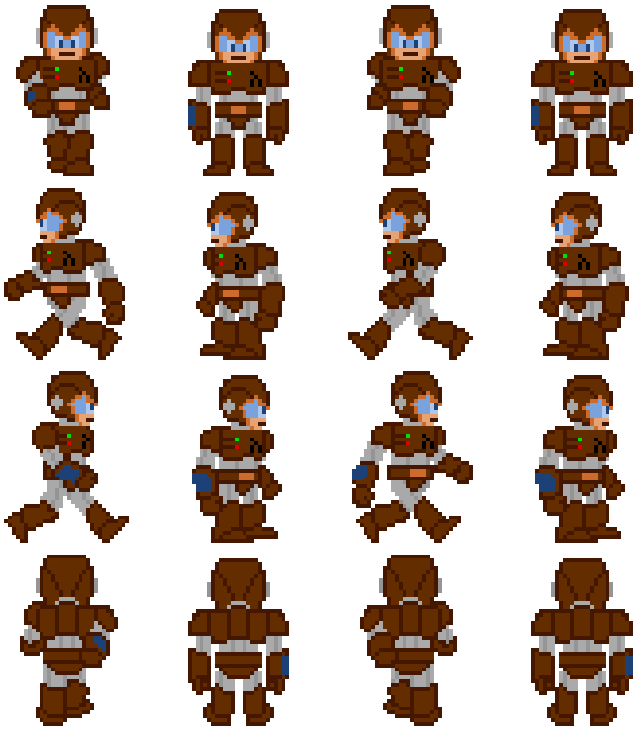


Figure B.2: The sprites for Lambda-Man moving in four directions.

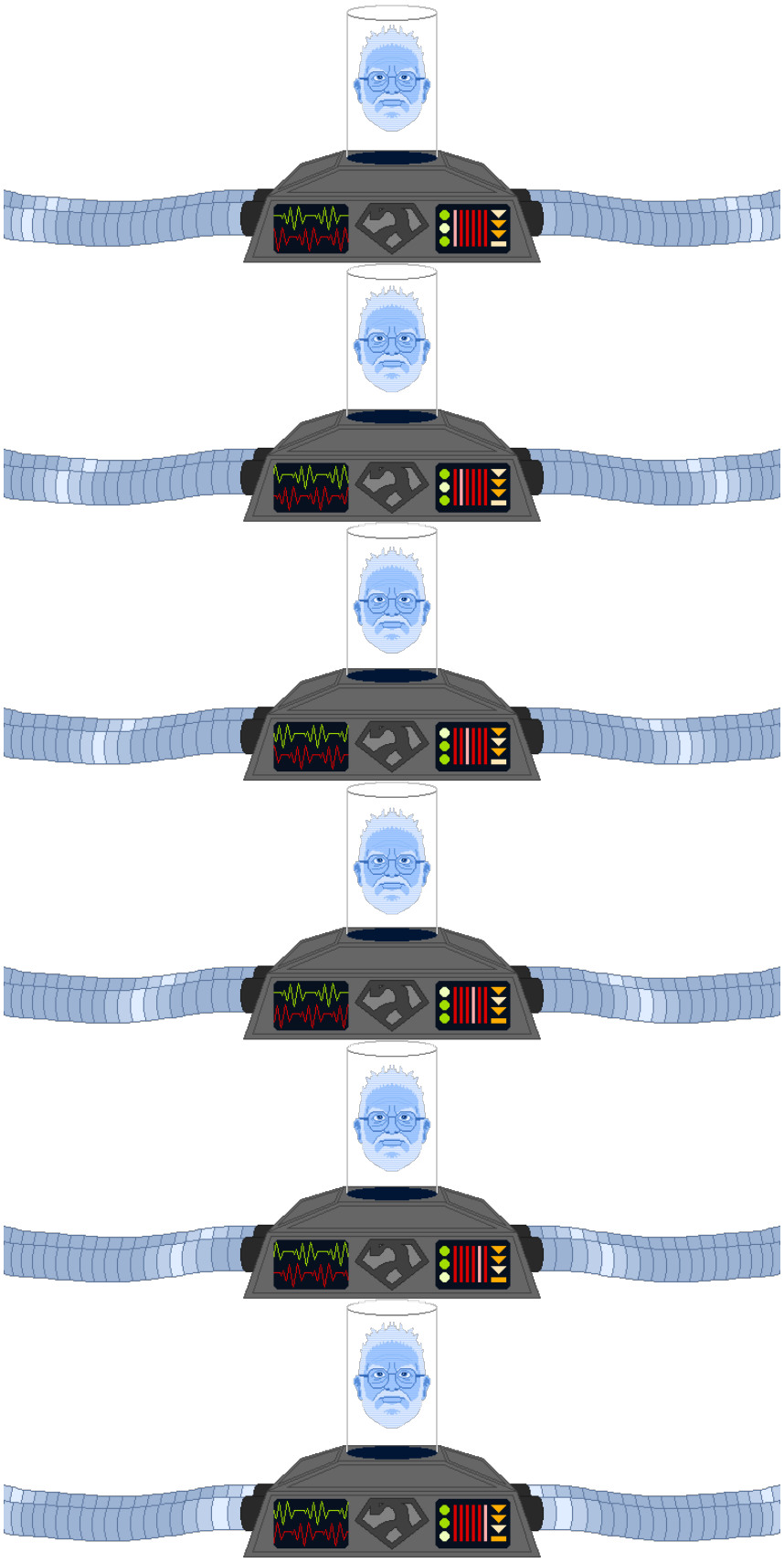


Figure B.3: All of the animation frames for the original Lambda-Man's hologram.

Figure B.4: All of the animation frames for Colonel Trigger-Finger in 3 states - normal, damaged, and exploding. The explosion animation was taken from OpenGameArt.org [54].

Figure B.5: All of the animation frames for Dr. Fractal, as well as the frames for the end of the fight when Kowalewski is saved.

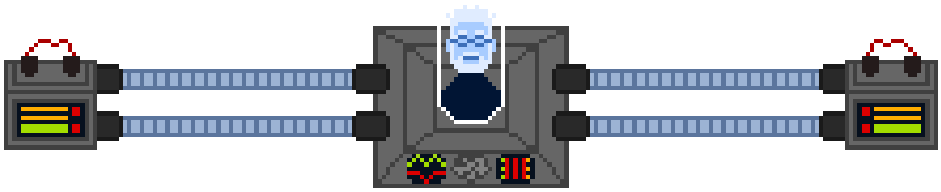


Figure B.6: The smaller 'overhead' sprite for the original Lambda-Man's hologram.

Figure B.7: The smaller 'overhead' sprite for Colonel Trigger-Finger.

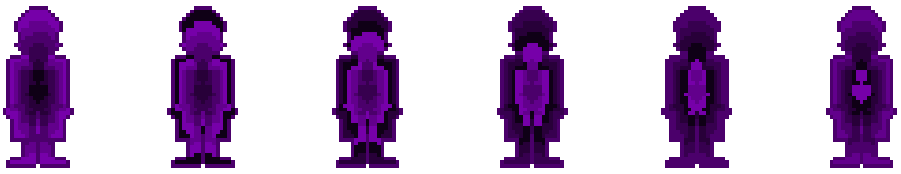


Figure B.8: All of the animation frames for the smaller 'overhead' Dr. Fractal.

Figure B.9: The 'Lambda-Cave' background for the tutorial battle.

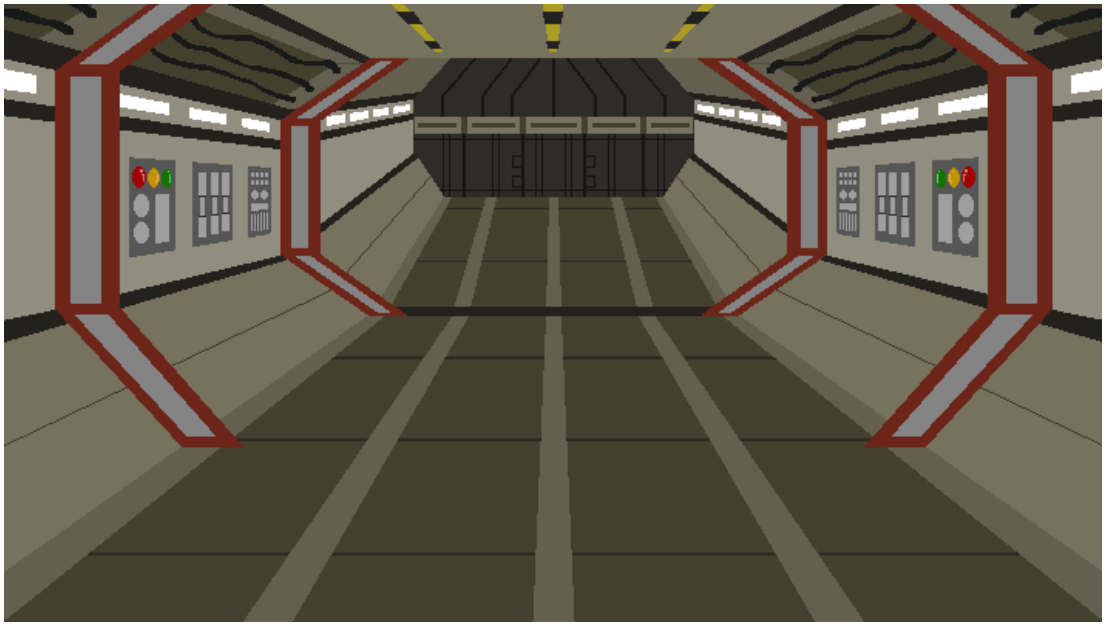


Figure B.10: The background for Col. Trigger-Finger's battle in a futuristic military airship.

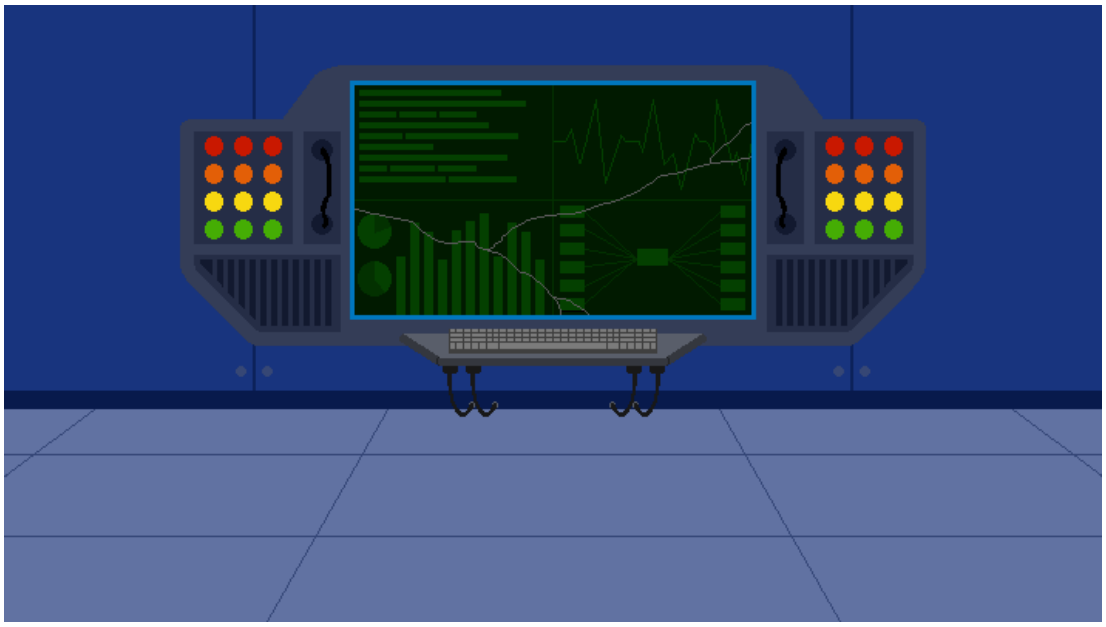


Figure B.11: The background for Dr. Fractal's battle, depicting her damaged laboratory.

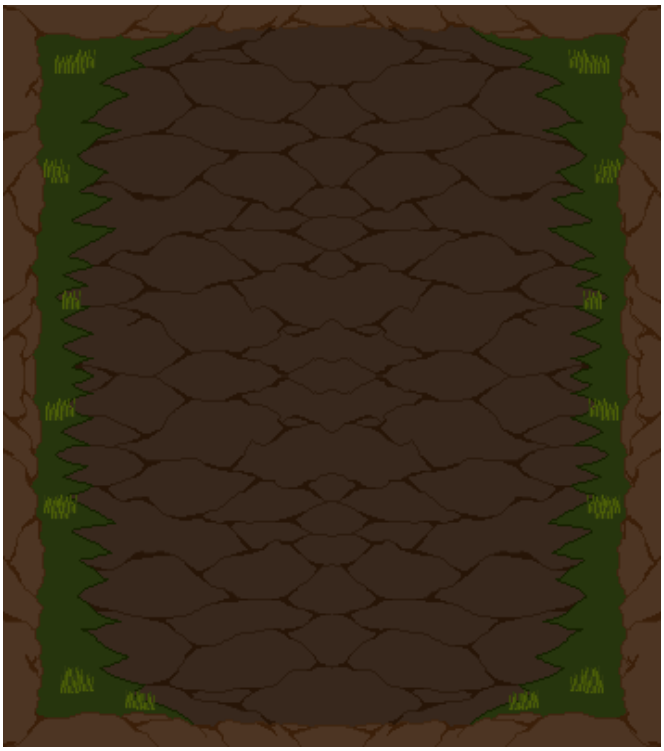


Figure B.12: The ’overhead’ background for the tutorial battle.

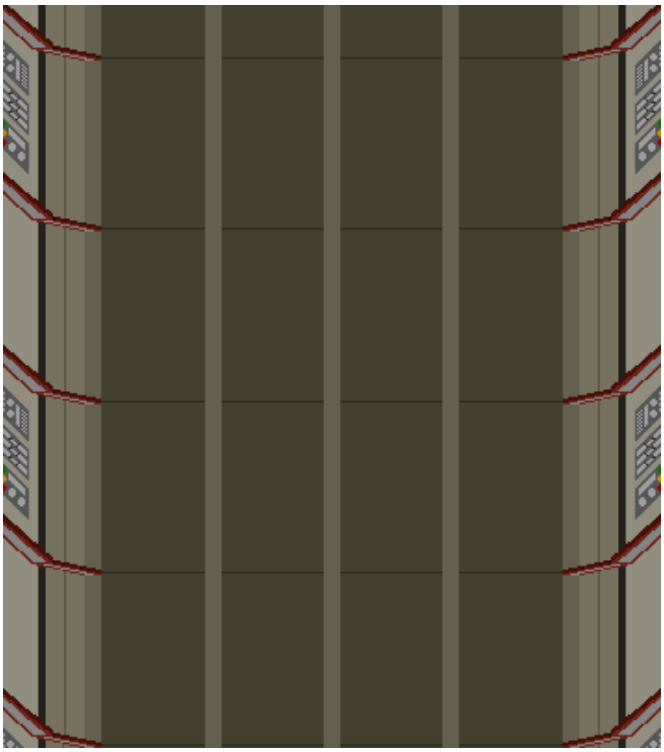


Figure B.13: The ’overhead’ background for Col. Trigger-Finger’s battle.

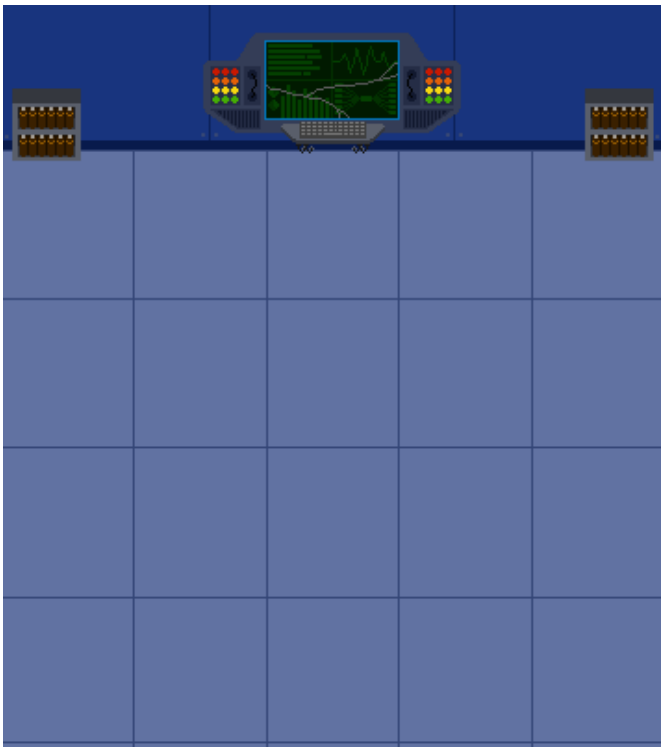


Figure B.14: The 'overhead' background for Dr. Fractal's battle.

Figure B.15: The plain code editor sprite, without any UI elements implemented.

Appendix C

Music credits

The tracks were chosen as they each pushed the limits of the Sega Mega Drive's FM sound chip, providing a synth-based score befitting the futuristic designs and retro-styled graphics of the game.

In order of first appearance in-game:

Tim Follin (`https://en.wikipedia.org/wiki/Tim_Follin`) - OST from *Time Trax*, an unreleased Sega Mega Drive port of a game based on a TV show - "Title Theme", "Mission Briefing Theme", "Stage 2, 5, 7 Theme", "Stage 4, 6 Theme"

Savaged Regime (`https://www.youtube.com/channel/UCbQQcXMh_ELHjiXY4dbRD6A`) - OST from *Life on Earth: Reimagined* by Kai Software - "Stage 4 Theme"

Note: since *Life on Earth: Reimagined* was commercially released, I erred on the side of caution and asked for permission to use the song. Savaged Regime approved (`https://twitter.com/SavagedRegime/status/1762159627376955438`), but an email to Kai Software received no response.