Expanding HaskellQuest: a collaborative multiplayer game for teaching Haskell

Neel Amonkar

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2025

Abstract

This report is the second part of the HaskellQuest project, concerning the development of an educational game to teach the functional programming language Haskell. HaskellQuest puts specific emphasis on its storyline and presentation, and covers the concepts of types, expressions, functions, case expressions and pattern-matching, tuples, lists and list comprehensions, and recursion.

The primary focus this year is adapting the existing framework of HaskellQuest to support a collaborative multiplayer campaign, a novel approach to teaching functional programming. In this campaign, two players progress through the story simultaneously - this can be accessed alongside the original single-player campaign. Additionally, improvements were made to the structure, interface and design of the game across both campaigns based on the feedback received from last year's evaluation phase.

The report also outlines the evaluation carried out this year - it compares the current versions of the single-player and multiplayer campaigns with one another, as well as the current version of the game as a whole to the one released last year. However, the results were unable to prove any significant change in either comparison; the report subsequently discusses the fundamental limitations of the design, as well as avenues for future work to push past these limitations.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: 388105
Date when approval was obtained: 2025-01-17

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'd like to thank my supervisor Don Sannella; your insightful and prompt (to a fault!) guidance and feedback on my work was absolutely invaluable. (No, seriously, on the way to InfBall?! I can't thank you enough.)

I'd also like to thank Cenk Koknar and the team on the Game Design Studio course; learning more about formal game design principles helped me push the project further than I could have done before.

Finally, I'd like to thank my parents, without whom I wouldn't be here (both figuratively and literally), and my friends, for the encouragement, critique, jokes, willingness to put up with my panicked barrage of texts about HaskellQuest evaluation, and (in one case) the free pho.

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 superhero-themed battle segments. It can often be hard for people to understand Haskell and other functional programming (FP) languages, especially if coming from an imperative programming background: this project's aim is to see if a game-based edutainment approach can effectively bridge the gap.

The major addition this year is a brand new multiplayer campaign: two players can connect to one another over the Internet and play through more complex versions of the game's levels, where each player is tasked with completing half of the solution.

1.1 Previous work

This section describes the fundamentals of the game established last year, that largely go unchanged for this year - the specifics of design and implementation are elaborated on in later chapters.

HaskellQuest goes for a linear, structured progression. Additionally, the game puts special emphasis on having an engaging presentation through colourful, varied environments and a tongue-in-cheek superhero story, in order to motivate the player to progress.

Each level comprises multiple phases that test the player's understanding of a single Haskell concept while increasing in difficulty. The challenges themselves are based around bespoke gameplay scenarios that are then expressed in terms of Haskell - the player is tasked with writing Haskell functions that behave a certain way to escape or divert the enemy's attacks.

The game covers the following topics:

- Types, including simple algebraic data types and type aliases
- Expressions and functions

- Case expressions and pattern-matching
- Tuples, lists and list comprehensions
- Recursion

Last year, testers gave it an average rating of 4.71 out of 6 for how enjoyable they found it. In particular, the presentation was universally praised, as well as how the story informed the challenges presented. However, there was room for improvement: testers took issue with how the game introduced its Haskell concepts, as well as the lack of detailed feedback for code errors.

1.2 Aims and approach for this year

The primary goal for this year is to adapt HaskellQuest's existing design into a multiplayer, collaborative learning context - while there is more than enough precedent for such an approach, few educational games have taken this route, and none of those that have are about teaching functional programming.

This does not replace the original single-player experience. The existence of both the single-player and multiplayer campaigns allows for a direct comparison on what method works best for HaskellQuest - individualistic versus collaborative learning.

Additionally, the feedback from last year was used to improve aspects of the game across both campaigns, such as the challenges in the recursion-themed level, the method of displaying error messages to the player, the pacing of the tutorial, and the introduction of a 'help screen' to view Haskell information.

1.3 Dissertation structure

- **Background**: Chapter 2 introduces functional programming and the Haskell language, and explores prior work in the fields of educational games and collaborative learning, including the work done last year.
- **Game design**: Chapter 3 recaps the established design framework for HaskellQuest, describes the decisions made in adapting the Haskell challenges for multiplayer, and discusses scrapped ideas for expanding the game's scope.
- **Implementation**: Chapter 4 recaps the fundamentals of HaskellQuest's existing implementation, and describes the implementation of the systems for the multiplayer campaign, as well as quality-of-life changes made to the in-game UI based on tester feedback from last year.
- **Evaluation**: Chapter 5 explains the method of evaluating HaskellQuest, explores the results, and discusses the issues testers had with the game.
- **Conclusion**: Finally, Chapter 6 summarises the achievements of the work done this year and discusses avenues for future work.

Chapter 2

Background

2.1 Functional programming and Haskell

2.1.1 Introduction to functional programming (FP)

Functional programming is wholly separate from the imperative paradigm which most programmers would be familiar with. While the latter is based on sequences of statements that update a program state, FP involves applying and composing functions – expressions that map values to other values, similar to the mathematical definition. Most commonly-known FP languages are in fact **pure** functional programming languages, where functions are reliant on their arguments only and do not consider any global or local state.

2.1.2 Haskell

Haskell [32] is one of the most popular functional programming languages - on a list of the most commonly-used languages on GitHub in 2024, ranked by number of pull requests, it placed 27th [7]. Additionally, Haskell has a variety of uses in industry applications; it has been used by companies such as Meta, Microsoft, NVIDIA, and Qualcomm, among others [29]. Given all of this, it is sensible to use Haskell for the purposes of teaching functional programming.

2.1.3 Why teach FP?

FP languages are already taught in many Computer Science university programs as part of an introductory course; for instance, at Oxford [26], Imperial [1] and of course here at Edinburgh [34].

In 2004, Chakravarty and Keller [10] argued that FP languages were ideal for introductory programming courses in CS degrees. As such, their paper outlines some technical advantages of FP languages over traditional imperative ones:

- FP languages like ML and Haskell have a “rigorous static type discipline” - the paper argues that this helps structure problem solving more easily.

- The “lightweight and orthogonal syntax” and high level of abstraction allows the elision of syntactical issues and implementation details like memory allocation.
- FP language features such as algebraic data types and pattern-matching “put complex structures, such as expression trees and memory tries, within reach of beginners”.

2.2 Serious games

2.2.1 Introducing the term

HaskellQuest, being a game intended to teach Haskell, falls under the umbrella of ’serious games’. The term’s current meaning is attributed to Clark C Abt in his 1970 book on the subject [3]: games having “an explicit and carefully thought-out educational purpose” and “not intended to be played primarily for amusement”, though he stressed that this “does not mean that serious games are not, or should not be, entertaining.”

2.2.2 Video games as educational tools

In his 2003 book [28], Gee argues that learning, like thinking or reading, is “not just a matter of what goes on inside people’s heads, but is fully embedded in (situated within) a material, social and cultural world”. In particular, learning in any **semiotic domain** (defined as “an area or set of activities where people think, act and value in certain ways”) requires people to “take on and play with new identities – forming bridges from one’s old identities to the new one”. For instance, when learning particle physics, one is tasked with thinking, acting, and assigning value as a particle physicist (the new identity in question) would.

Additionally, Gee is a proponent of the learning theory of **connectionism** – essentially, people don’t think or learn best when presented with abstract principles out of context, but rather “on the basis of patterns they have picked up through actual experiences.”

The book then argues, on the basis of these tenets, that well-designed video games can serve as good learning tools, regardless of their educational intent:

- Games offer strong identities for players to take on – either specific characters with distinct viewpoints, or blank slates that the players can build from the ground up (both figuratively and literally).
- Understanding a game’s mechanics is akin to the scientific method - players “hypothesize, probe the world, get a reaction, reflect on the results, reprobe to get better results”.
- Games generally have a cycle of “consolidation and challenge” – players can practice challenging problems until they have “routinized their mastery”, at which point the game introduces a new type or difficulty of problem, causing them to “rethink their taken-for-granted mastery”.

- Games lower the consequences of failure with savegames and retries, and allow customization to fit different players' learning styles i.e. adjustable difficulty levels, multiple possible solutions, and so on.
- Games are designed to "feel 'doable' but challenging", serving to motivate players into a "flow state" - if a game is too easy or too difficult, the player would feel no impetus to continue.
- Games present information just as it becomes relevant to some experience within the game, or upon the player's request. Thus, players have an immediate opportunity to **apply** that information, helping them understand it in a deeper manner, in accordance with the theory of connectionism.
- Games encourage a sense of community – in learning to play a game, players often "(draw) on resources that reside in other gamers and their associated websites and social interactions". The book makes note of the fact that even single-player games are played in multiplayer settings by taking turns. This social aspect can serve as a powerful motivator and tool for learning.

2.2.3 The effectiveness of video games in education

Supporting the book's argument, previous analyses have concluded that serious games are an effective teaching method - a 2013 literature review, covering use in elementary school up to higher education, concluded that "serious games can be effective learning materials in their own right" [6], while a 2019 meta-analysis covering use in science education found that "serious games were... more beneficial than conventional instructional methods" [55].

Additionally, a 2022 review [43] concluded that even commercial games intended for entertainment purposes (such as those cited by Gee) had positive effects on students' academic performance. A 2020 study cited in the review [11] concluded that students who played games of various genres (particularly action games) showed improvements in cognitive functions like attention and working memory, while other cited studies showed improvements in reading [24] and mathematical [49] ability.

2.2.4 Using games to teach programming

There have been prior attempts to use games to teach programming. One of the most popular is the CodinGame online platform [14], which has multiple 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 objects within the game (e.g. a turret attacking oncoming enemies, or a car moving between checkpoints) are controlled by the code that the player submits. Crucially, however, the player is not simply implementing a game object from scratch: a level of abstraction is provided to make the experience accessible to newcomers.

Other examples of games teaching programming are:

- **Code Hunt by Microsoft Research** [8]: A game aimed at AP Computer Science

students, where the player is tasked with writing functions in C# / Java, based on input-output pairs provided to the player.

- **Human Resource Machine by Tomorrow Corporation** [33]: A commercially-released puzzle game, where the player controls their avatar through an in-game visual scripting tool – the game's tasks teach the player the principles of assembly programming through the abstraction of a menial office environment.
- **ScalaQuest by Alejandro Lujan and Christian Leger** [39]: The original inspiration behind the HaskellQuest project – the player is tasked with navigating through a fantasy-themed world, encountering obstacles that require the player to complete Scala challenges (e.g. write a specific function, or define a specific class).

A 2018 review of existing literature about serious games in programming [44] found that most games surveyed were effective at teaching introductory programming concepts, but there were some deficiencies in teaching more advanced software development, and there was a lack of consensus in evaluating their effectiveness. The former issue is not a concern for HaskellQuest, as it falls outside the scope of the project.

In general, such games focus primarily on imperative programming – even Scala, which can be used as an FP language, is used in an imperative fashion in ScalaQuest.

2.3 Teaching FP through gaming

2.3.1 Learning FP through game development

Hudak [31] argues that multimedia development is a field where Haskell's strengths shine, and the rest of the book demonstrates this by providing exercises that range from implementing basic shapes to writing domain-specific languages for reactive animations and MIDI music.

Following his lead, some universities took to using game development as a method to teach functional programming. For instance, the Radboud University in the Netherlands ran the Soccer-Fun project [4] in 2011 – the course provided a framework for a football simulation in the FP language Clean, using its GUI library for rendering, and students were tasked with implementing the decision-making processes of a virtual football player. Students could then pit their players against one another – the paper concluded that this competitive element served as the prime motivating factor for students to learn Clean.

Another example is 'Haskell in Space' [40], a 2003 project at the University of Bremen in Germany. Second-year students were tasked with implementing a version of the arcade game *Asteroids* – specifically, their assignment was to write Haskell functions to model the in-game geometry and the physics-based behaviour of the ship. Here, the game's animated graphics proved to be the main appeal – the paper notes that students embellished their submissions with fancy graphics even though they weren't required to.

None of the above have much to do with serious games on their own. However, Hudak's 'show-by-example, learn-by-doing' approach, maintained in the two FP courses, is fundamentally compatible with Gee's argument about games as learning tools. Additionally, students were motivated to learn by being able to see their progress play out in-game – there was a built-in mechanism for immediate feedback in the assignments, which would also be advantageous for HaskellQuest. It should be noted that this is also the fundamental principle of the exercises on CodinGame – the player's code affects the game's world.

However, this cannot be applied to HaskellQuest directly. Obviously, if the game were to be made in Haskell like the two projects, the player wouldn't be able to modify the game's scripts at runtime. Additionally, Haskell is fundamentally *stateless*, while games are all about manipulating a global state. Soccer-fun and Haskell in Space define a custom 'State' datatype to describe the **entirety** of the current game state and functions that take a State as an input and return a State as an output. With this approach, the complexity of this state type increases drastically alongside the complexity of the game itself. If HaskellQuest players were shown a type describing the entirety of the game state, it could lead to them being overwhelmed with potentially unnecessary information.

2.3.2 Games teaching FP - prior HaskellQuests

For a solution, we can look at previous HaskellQuest projects, particularly the 2018 game by Karolina Drobnik [20] and the 2023 game by Eve Bogomil [9] (as seen in Figure 2.1). Both were developed using the Unity engine [66], meaning the game state is kept track of with a standard imperative approach. The games present puzzles/obstacles which involve writing Haskell functions; crucially, when custom states are defined here, they are **situational**. In Drobnik's HaskellQuest, for instance, players are tasked with rotating a bridge to progress, and the Haskell challenge involves writing a function that modifies a value of type Bridge.

Additionally, the interaction between said functions and the game state itself is indirect: in Bogomil's HaskellQuest, 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. This approach allows for a level of abstraction similar to the CodinGame exercises; the player is only presented with the information relevant to the challenge at hand, in line with Gee's principles.

The work done in year 1, subsequently, used the same technical approach - this is explored in further depth in Chapter 4.

2.4 Collaboration in serious games

2.4.1 Defining collaborative learning

In their 1994 book [37], the Johnsons define cooperative learning as a setting in which students "work cooperatively in small groups, ensuring that all members master the

Figure 2.1: The introductory battle in E. Bogomil's HaskellQuest [9]: 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.

assigned material". This is contrasted with competitive learning, where only one student can succeed, and individualistic learning, where students work independently with no regard to what the others are doing. (Here, 'cooperative' and 'collaborative' learning are treated as the same thing.)

The book's primary argument is that while an ideal teaching curriculum incorporates all three learning paradigms in appropriate amounts, cooperative learning should be the main focus, as cooperation itself is fundamental to human existence: from human biology to economic and legal systems to evolution.

It notes that structuring lessons individualistically is ideal for situations where students need to complete "unitary, nondivisible, simple tasks", and can benefit from self-pacing and isolation. The aim of this year's multiplayer campaign, then, is in part to determine which approach works better for learning Haskell.

The book's focus is on collaborative learning in the classroom. As such, it outlines five essential elements for teachers to structure lessons around:

1. **Positive interdependence**: The task needs to be designed in such a way that a student cannot succeed individually. This incentivizes them to care about their group members' success.
2. **Promotive interaction**: Subsequently, the opportunity for students to encourage one another's efforts and assist each other should be maximised.
3. **Individual accountability**: Students should be assessed individually as well, to hold them accountable to do their share. These results should be given back to both the individual and the group, so that they can know which member needs more assistance or support.

4. **Interpersonal and small-group skills**: Students must be specifically taught social skills like "leadership, decision-making, trust-building, communication, and conflict-management".

5. **Group processing**: The group needs to be given ample opportunity to reflect on their work, allowing them to decide how best to proceed.

2.4.2 Collaborative games for education

While the Johnsons focus primarily on collaborative learning within the classroom, gaming is a natural avenue for enabling collaborative learning, given the prominent social component mentioned in section 2.2.2. There are plenty of 'entertainment' games focused on collaboration and player interaction - however, there are considerably fewer intended for an educational or academic context.

Existing research into multiplayer games as educational tools has mostly focused on massively-multiplayer online games (MMOs) - for instance, a 2004 article focusing on the MMO role-playing game *Lineage* [61], or a 2010 article using multiple MMOs as a basis for design principles for future serious games [50]. While games such as these exemplify Gee's principles well, engineering a persistent online world is rather dramatically outside HaskellQuest's scope, especially when considering the original game's design.

Indeed, few (if any) research projects actually attempt to **implement** an MMO, instead opting for smaller-scope games with a set end goal. A 2013 project, 'Escape from Wilson Island' [69], uses the Johnsons' principles of collaborative learning as a basis for its game design:

- The game grants each player one unique tool. Thus, each player has unique capabilities and tasks, but all tasks need to be completed to successfully accomplish the end goal. Additionally, some tasks require multiple players to coordinate; for instance, players are tasked with steering a raft across shifting currents, and each player can affect the movement of the boat a little.
- The game facilitates promotive interaction by including an in-game chat window - players can choose who to talk to and can communicate with multiple other players simultaneously.
- At the end of the game, players are ranked on their individual performances; scores are determined by the number of helpful tasks completed during the game.
- The game offers personal inventories of items that can be traded between players, such as food items to refill health and wood for fires, rafts, and so on. This introduces an opportunity for player interaction and discussion, as there is a tension between personal benefit and group success.

While the 'Wilson Island' project did not have an educational intent beyond facilitating and studying player collaboration, the developers found that designing a game specifically along the Johnsons' principles led to an entertaining experience for their testers, concluding that the approach served as fertile ground for future serious games.

2.4.3 Multiplayer games for teaching programming

There is precedent that suggests a collaborative approach may be fruitful for programming education – specifically, the prominence of pair programming as a teaching method. The most common approach to this is the 'driver-navigator' approach: one person is in charge of writing the code directly (the driver), while the other observes and provides suggestions and corrections (the navigator). A 2010 meta-analysis of prior studies focusing on the impact of pair programming [58] showed that "almost all studies' findings reported that students' satisfaction was higher when using PP compared with working individually", though there were mixed results in terms of academic improvement - the use of pair programming improved students' scores for assignments, but not those for their final exams.

However, this does not seem to have substantially influenced prior serious games that focus on teaching programming. Most of those that do offer multiplayer take a competitive approach, such as the minigames available on CodinGame [14] or a 2009 project in the real-time strategy (RTS) genre [46], or an MMO / 'persistent world' approach, such as a 2016 project aiming to teach C [41] or a 2007 project covering introductory programming concepts [42].

There are **two** instances of collaborative programming games, however; a 2022 paper outlines the addition of a two-player collaborative mode to an online game covering Python [62], while a 2025 paper focuses on the implementation of a two-player game covering language-agnostic imperative fundamentals [63]. Still, though, no such project exists for teaching functional programming; certainly no prior HaskellQuest project has attempted it.

Chapter 3

Game design

3.1 Summary of established design

As mentioned before in section 1.1, HaskellQuest's approach to engaging players relies on its art style, presentation and storyline. The point of the storyline is not to offer a cohesive narrative – instead, it serves as a basis for entertaining level settings and scenarios. The core game loop of HaskellQuest was based around this fact – the player's motivation to progress would be "what am I going to see next?"

Players play as a new incarnation of Lambda-Man, a superhero who defeats his enemies by writing functions that divert / defeat their attacks. In this way, the game's design attempts to follow Gee's principle of offering a strong identity for players (mentioned in section 2.2.2) – Lambda-Man serves as a player stand-in as someone being introduced to Haskell through the game's events.

The use of Lambda-Man as the protagonist was an in-joke specifically intended for Informatics students at the University of Edinburgh (the primary demographic last year). Professor Philip Wadler, former lecturer on the Introduction to Computation course, dresses up as the Superman-themed 'Lambda-Man' at conferences as a joke - he appears in the game's tutorials as a hologram mentoring the player character.

The story's second function was to contextualise the Haskell challenges. Within each challenge, the code defines custom data types that mimic the gameplay scenario (similar to the CodinGame challenges mentioned in section 2.2.4). The player is then tasked to write one particular function that interacts with the data types in a specific way. This story-challenge integration serves to stop the challenges from feeling overly didactic.

The gameplay scenarios themselves take the form of fast 'bullet hell'-esque segments inspired by the game Undertale [23] - players are tasked with dodging various projectiles if their Haskell function fails the test. If the function works as expected, however, the player can often avoid taking any damage altogether. Since progressing through the storyline is intended to be the player's reward, these attacks serve as the main obstacle – if the player's health reaches zero, they have to replay the level from the start.

As mentioned before, each level consists of multiple challenges – these are intended

to increase in difficulty to offer a clear progression for the player in understanding a given Haskell concept. This was a response to playing E. Bogomil's HaskellQuest [9] before beginning to design my own – there, players can use Haskell to freely implement RPG actions such as attacking or healing, but as someone who hadn't written Haskell in years I felt deeply lost and unable to progress.

To provide an example, the second level covers list comprehension. The story for that level is that Lambda-Man is up against a militaristic villain called Colonel Trigger-Finger. As such, the first segment of the battle involves the Colonel launching missiles that home into the player - the corresponding Haskell challenge defines a Missile data type, and the player has to write a function that, given a list of Missiles and a Target, changes all the Missiles to point to the new Target. If this function works as expected, the missiles will avoid the player entirely and collide with the Colonel, thus allowing the player to progress to the next segment. The single-player version of this battle from this year's release can be seen in Figure 3.1.

One additional element last year was a scoring system for the levels - the player started with a 'code score' and a 'damage score' of 5000 each, which would decrease by a set amount every time the player submitted non-functional code or took a hit from a projectile. However, this system wasn't meaningfully related to the player's motivation, nor did it serve to make the challenges more replayable - the puzzles are only solvable once anyway, as players would already know the answer on repeat playthroughs. The scores were never mentioned in the tester feedback from last year - as such, they were removed entirely from this year's release.

Figure 3.1: The first phase of the Colonel Trigger-Finger battle in the single-player campaign.

3.2 Approach to multiplayer challenge design

The fundamental idea for the multiplayer expansion was already in place. To quote last year's report [5], "two players could go through the game simultaneously, working on

Haskell code together and dodging more complex projectile patterns from the enemy" - the increased difficulty would be a counterweight to the fact that two players would be working together to solve the puzzles. Additionally, the core design approach as described in the previous section would remain largely unchanged – this decision was made to allow closer comparison between the evaluation results for the single-player and multiplayer campaigns.

However, the actual execution of this was up in the air for quite a while.

3.2.1 Discarded initial approach - multiplayer code editing

The first idea was that both players could simultaneously edit the same text at once, akin to Google Docs - this is a common feature for online code editors such as Codeshare [13] and Replit [15]. Thus, both players would be able to see the results of the evaluation at the same time, as well as view any error messages. Indeed, this got as far as a basic implementation, as described in section 4.5.1.

The problem here was that designing the game this way would violate the very first principle for structuring collaborative learning, as mentioned in section 2.4.1; if both members were tasked with editing the same code, then it would theoretically be possible for one player to complete the challenge by themselves, even if the Haskell challenge itself was more difficult to compensate.

Additionally, the idea of "more difficult challenges" was too vague - it was a struggle to conceptualise new challenges and gameplay scenarios off of that alone.

3.2.2 Final approach - 'simultaneous pair programming'

In rethinking the design approach, prior collaborative serious games such as 'Escape from Wilson Island' served as an inspiration - namely, the principle of each player having unique capabilities, all of which are required to progress (as mentioned in section 2.4.2). Applying this to the current structure of HaskellQuest wasn't straightforward, however – the game is built on players writing code, which would have had to apply for both players in the multiplayer scenario, so they could not have different **abilities** as such.

The solution was imposing an artificial boundary. The multiplayer challenges would consist of two separate yet interconnected problems requiring two functions to be written/modified, where progress would only be possible if both functions were correct – thus, each player would straightforwardly be responsible for one function. Pair programming served as an inspiration for designing the main method of player collaboration (its effectiveness is elaborated on in section 2.4.3) – each player can freely view the code that the other player is currently writing in real time, but they cannot **edit** that code. Thus, 'simultaneous pair programming' – each player is the driver for their own function and the navigator for the other at the same time.

3.2.3 Handling health for two players

As mentioned before, since progress through the story is intended to be the player's main motivation, getting hit by projectiles serves as the main obstacle, as the player's progress is reset to the start of the level if their health hits 0. For the multiplayer campaign, two key decisions were made:

- The players taking damage is meant to be a form of feedback, as writing successful Haskell functions often circumvents damage entirely. As such, players would have separate health bars in an attempt to provide individual feedback, in accordance with the Johnsons' principles in section 2.4.1. However, there is a flaw here - as any projectile could hit any player, one player may end up losing health because of the other player's function not working.
- It was also decided that if one player's health hit 0, both players would have to restart the level. This further ties into the idea that one player cannot progress without the other, though it does make the multiplayer campaign even more difficult by comparison.

3.3 List comprehension level - Colonel Trigger-Finger

Since the decision had been made to have two functions for each Haskell challenge in the multiplayer campaign, I only had to determine what the function for player 2 to write would be, as player 1's functions could be the same as the single-player campaign. Additionally, the list comprehension level was the second level in the game – players would still be getting used to the split challenges and collaborating with another player. As such, I decided that the functions themselves wouldn't be dependent on one another - rather, the connection between them would be through the attack scenario. The approach was, therefore, to come up with a second obstacle that would make the existing one more difficult, then express it as a Haskell challenge that would require a list comprehension.

3.3.1 Phase 1 - basic list comprehension

For instance, the first phase in the single-player version, as mentioned beforehand, was based on missiles that would home in on the player; in the 'battle' segments, if the player hadn't succeeded in writing the Haskell function, they could attempt to dodge the projectiles by moving around.

For the multiplayer version, the second obstacle here would be a set of 'freeze missiles' that would stop the player in-place for a set duration, thereby making them more vulnerable to hits from the homing missiles. Player 2 would also be tasked with writing a function to deactivate a list of FreezeMissiles – rather than changing the target of the missile, however, this function would have to disable them by changing a Boolean value from True to False.

3.3.2 Phase 2 - list comprehension with a guard

In the same vein, the original second phase had a barrage of missiles bearing down on the player - one of these would be inactive, however (as represented by a Boolean). Thus, the player's challenge is to write a list comprehension function using a guard to filter a list to only contain inactive Missiles.

For the multiplayer version, the second obstacle would be a set of randomly-positioned landmines - the missile barrages moving down towards the player served to restrict their range of movement, and the landmines would emphasise this challenge while keeping the militaristic theme in place. Once again, though, one of the landmines would be inactive - the second player would also have to write a function isolating the inactive Landmine from a list of Landmines.

3.3.3 Phase 3 - parallel list comprehension

The final segment of this level concerns parallel list comprehension, a Haskell language extension introduced in more recent versions of the Haskell compiler. The original third phase was, in the storyline, Colonel Trigger-Finger going all out in an attempt to finish off Lambda-Man; as such, the player would have to navigate a combination of both prior attacks i.e. the homing missiles and the missile barrage. The intended solution was for the player to write a function that would, given a list of Missiles and a list of HomingMissiles, make the HomingMissiles target the regular ones using parallel list comprehension - the two threats would take each other out, leaving the player unharmed.

The concept of an 'all-out attack' would be retained for the multiplayer version – thus, the freeze missiles and landmines would have to be incorporated in some way. It seemed natural to have player 2 have a function to redirect the freeze missiles into the landmines to disable them. However, the FreezeMissile type did not actually have a Target field - adding one would make it even more similar to the HomingMissile type, meaning the problem from the previous phase would persist.

In the end, the behaviour of the freeze missiles had to be modified somewhat. The Landmine type had been defined with a field of type Coords (an alias for an (Int, Int) tuple) - the FreezeMissile type was then modified to have a field of this type as well. This was reflected in the attack segments; where previously the freeze missiles would track the player's position like the homing missiles, now they would home into a fixed position - either the player's position at the time of launch if player 2's function was unsuccessful, or the position of a landmine otherwise. This also had the effect of adjusting the difficulty of the final attack phase to be less frustrating for players.

3.4 Tutorial levels - Lambda-Man Hologram

The idea behind the tutorial challenges was that they would serve as a gentle introduction to the game's concept, as well as provide players with an early example of Haskell's syntax for functions - players are tasked to **modify** a function, rather than write one. Thus, the concept for the Haskell challenges was as simple as possible: players are

presented with a function that reduces their health, and they have to modify it to avoid taking damage.

Because of this simplicity, however, it was difficult to come up with something for player 2 to do. In the end, I came up with a slight variation on the theme. While player 1's function would reduce the players' health by a constant amount, player 2's function would multiply the players' health by some Float less than 1 e.g. 0.9. Again, player 2 has to modify the function so that the change in health is either zero or positive. This does make player 2's challenge harder than player 1's; the provided function involves conversion from an Int to a Float and back, which may be harder to parse for a beginner.

Additionally, the game last year suffered from a pacing issue. All of the tutorial challenges happened at the beginning of the game - this overwhelmed the player with information that, in the case of recursion, wouldn't actually be required until later in the game. As such, this year's release splits the tutorial level into two - basic functions and list comprehensions are covered at the beginning i.e. right before the Colonel Trigger-Finger fight, and recursion is covered right before the Dr. Fractal fight.

In the single-player campaign, the second phase involves a function that takes a list of Ints and reduces the player's health by the sum of the positive integers:

```
hurtPlayer :: Player -> [Int] -> Player
hurtPlayer (Player health) dList = Player (health - damage)
    where damage = sum [d | d <- dList, d >= 0]
```

...while the third phase implements the same function recursively by pattern-matching against the head and tail of the list.

```
hurtPlayer :: Player -> [Int] -> Player
hurtPlayer player [] = player
hurtPlayer (Player h) (d:ds)
    | d >= 0    = hurtPlayer (Player (h - d)) ds
    | otherwise = hurtPlayer (Player h) ds
```

For the multiplayer campaign, player 2 is given functions that take a list of Floats and multiply the players' health by the elements that are less than 1. The player is similarly tasked with modifying these so there is no loss in health.

```
-- the version shown in phase 2
weakenPlayer :: Player -> [Float] -> Player
weakenPlayer (Player health) dList = Player newHealth
    where
        newHealth = round (fromIntegral health * damageFloat)
        damageFloat = product [d | d <- dList, d <= 1]

-- the version shown in phase 3
weakenPlayer :: Player -> [Float] -> Player
weakenPlayer player [] = player
weakenPlayer (Player h) (d:ds)
    | d <= 1 =
        weakenPlayer (Player (round (fromIntegral h * d))) ds
    | otherwise =
        weakenPlayer (Player h) ds
```

3.5 Recursion level - Dr. Fractal

The recursion level was different in that the single-player version would require substantial improvement. In last year's release, this level suffered from being rushed - I was unable to come up with a conceptual progression for recursion. As such, the level only had two phases, and the second phase was based around a story element rather than an elaboration on the concept of a recursive function.

In subsequent feedback, my supervisor suggested a suitable progression:

- A function with one base case and one recursive case
- A function with multiple base cases
- A function with multiple recursive cases

3.5.1 Phase 1 - one base case and one recursive case

For the single-player campaign, the first part was already in place, so the challenge there remained largely unchanged from last year. There, the challenge presented a Fractal data type with a Size component (an alias for Int), representing Dr. Fractal herself; the player would write a function such that, if the Fractal's size was even, it would halve the Fractal and call the function on the two resulting Fractals (the recursive case); otherwise, it would just return a list containing the odd-sized Fractal itself (the base case).

In terms of the attack, splitting the Fractal would make the resulting projectiles cause less damage, though there'd be more of them; this was balanced so that even if the player got hit by every single projectile after passing the Haskell challenge, they would still have taken less damage than if they'd failed the challenge.

For the multiplayer version, I decided to have player 2 write a function to recursively halve the laser in the same way – it had already been established that the damage of the projectiles was based on the size of the missile, so the correlation between the two players' functions was in place.

3.5.2 Phase 2 - multiple base cases

For this phase, the single-player version would need designing as well. However, I initially struggled to come up with a gameplay concept that would require a function with multiple base cases. Eventually, I got the suggestion of a projectile that splits into two different kinds of projectiles that need to be dealt with differently; this was adapted into the recursively-defined Bolt type for the Haskell challenge.

```
data Bolt = IntBolt Int |
            FloatBolt Float |
            NodeBolt (Bolt, Bolt)
```

Here, the IntBolt would deal a set amount of damage to the player, while the FloatBolt would multiply the player's health value by some Float less than 1 - this is similar to the two functions in the tutorial challenges.

The player would then need to write a function to traverse a tree of Bolts, given the root node - the NodeBolt case would be the single recursive case, while the IntBolt and FloatBolt cases would be the multiple base cases. The player would have to modify the IntBolt and FloatBolt values directly to negate damage – just as in the tutorial exercises, they could modify these however they wanted.

For the multiplayer version, I decided to 'up the stakes' – where previously the functions did not need to rely on each other to be correct, this time player 2's function would need to call player 1's. In this instance, Dr. Fractal herself would turn out to be recursively defined - the Fractal data type for this challenge was thus defined as follows.

```
data Fractal = DecoyFractal |
               BoltFractal Bolt |
               NodeFractal (Fractal, Fractal)
```

Player 2 would need to write a function to traverse a Fractal tree, given the root node: while the DecoyFractal base case would change nothing, the BoltFractal base case would need to call player 1's 'disableBolts' function. There would be multiple BoltFractals in a given tree - thus, the number of projectiles on screen would be increased as well.

3.5.3 Phase 3 - multiple recursive cases

The storyline for this level was that Dr. Fractal started as a Haskell scientist who had an accident while attempting to recreate Lambda-Man's powers and studying recursion - as such, the last phase of last year's version revolved around rescuing Dr. Kowalewski from the Fractal structure. I decided to retain it for the final phase here, as I felt it was entertaining. Using a tree structure for the data types in the previous phase proved to be intuitive, so I continued with the approach for this phase. The Fractal type for this challenge was defined as follows:

```
type Depth = Int

data Stability = Inert | Unstable | Healed
data Fractal = NodeFractal Depth Stability (Fractal, Fractal)
                | LeafFractal
```

The player would need to write a function to traverse a Fractal tree: the LeafFractal base case would be trivial, while the NodeFractal recursive case would need to behave differently depending on the node's Stability. If it was Inert, it would need to stay that way; if it was Unstable, the Stability would need to be changed to Healed.

For the multiplayer version, I similarly used the 'tree nested within another tree' approach from the previous phase. This time, player **2**'s function would be identical to that of the single-player version, while player 1 would need to write a function calling player 2's. As such, I defined a 'MetaFractal' type:

```
data MetaFractal = LeafMeta
                 | StableMeta (MetaFractal, MetaFractal)
                 | UnstableMeta Fractal (MetaFractal, MetaFractal)
```

Player 1's function would need to traverse a MetaFractal tree and call player 2's function on the contained Fractal for any UnstableMetas.

While the Haskell challenge here is still related to the storyline, it is not related to the actual **attack**. This phase reuses the attack from last year's final phase, which was a flurry of high-damage projectiles fired in random directions - the intention was to show Dr. Fractal's instability as a 'panicked' reaction. Still, it is a notable change from the norm for the game, which may throw players off-balance.

3.6 Unused expansion ideas from last year's report

There were a few more ideas for expanding on HaskellQuest that were brought up in last year's report; this section discusses why they weren't implemented this year.

3.6.1 Overworld segments and random encounters

As mentioned in section 3.1, the primary source of inspiration for HaskellQuest's battles was the game Undertale. Originally, this inspiration would have been taken further - like most role-playing games (RPGs), Undertale includes overworld segments where the player character can talk to non-player characters (NPCs) and trigger random encounters with enemies. This was also part of Bogomil's HaskellQuest [9] - there, the NPCs served to introduce Haskell concepts to the player.

The levels described so far were originally intended as boss encounters to cap off a larger segment of the game. The cutscene text before the Colonel Trigger-Finger fight describes Lambda-Man "handily defeating most of the rank-and-file soldiers" aboard the airship - in this conception, the player would have actually played through this segment. Segments like these could have served as a more gradual introduction to Haskell's concepts and the gameplay mechanics.

However, time constraints aside, there is a fundamental issue in how such encounters would be incorporated into the current HaskellQuest framework. In RPGs, putting aside the player's expanding capabilities, the basic battle mechanics are essentially constant, and there are usually broad categories of 'generic' enemies whose stats are randomly-generated within defined ranges. This makes the game more scalable - variety can be created from putting different kinds of enemies together, and enemies of the same kind are not homogenous due to the randomness.

Meanwhile, in HaskellQuest, while there is an element of randomness in some attack patterns, the Haskell challenges relating to them are bespoke, and the game has a very strict progression. Defining 'generic' enemy types would therefore lead to a lot of repetition for the player. Each type would have a finite number of associated Haskell challenges - given time, the player would eventually encounter each possible challenge multiple times, which would just be tedious as the challenges are not designed to be replayable, as mentioned before.

Additionally, the overworld would make less sense within a two-player cooperative framework; are the players forced to accompany each other at all times? If not, what happens if one player gets into an enemy encounter while the other is elsewhere?

As such, I decided not to incorporate this element into HaskellQuest - the game would

require a substantial rework to accommodate random enemy encounters, and if those are absent the overworld serves less of a point.

3.6.2 Items

Another feature that could have been taken from traditional RPGs are the use of items - during a player's turn, they would be able to access an inventory and use consumable items to offer advantages in battle. A remnant of this idea is still in the final game: in battles, 'HACK' is the only option offered to the player, yet after a failed attempt at submitting code the editor still slides back to offer the player the option to 'HACK' again. Originally, there would have been the option to use 'ITEMS' as well - these could allow the player to regenerate health, or move faster during attack segments, or offer hints for the current phase of the battle.

Additionally, unlike the overworld, this could work in a two-player cooperative context. As mentioned in section 2.4.2, the collaborative game 'Escape from Wilson Island' offers personal inventories of items - the implementation there facilitates player discussion and interaction.

However, the issue that prevented me from implementing items last year persists - how would the player obtain them? The original idea, after nixing the overworld, was to include an 'item shop' screen in between battles, but the currency design was problematic. If well-performing players gained more in-game money after beating a battle, they would be able to stock more items, but they'd need them less than poorly-performing players.

3.6.3 Save feature

The utility of a save feature is obvious - more than being a convenience feature, in the context of HaskellQuest it could also allow learners to pace their own learning (as noted in section 2.2.2).

However, such a feature wouldn't be possible for multiplayer - the method used for handling player connections obfuscates any information that could serve as a unique identifier (more in section 4.3), so it would be difficult to save progress for a particular pair of players. Additionally, for the purposes of evaluation, the single-player campaign couldn't have a feature that the multiplayer campaign didn't, so this was nixed as well.

Chapter 4

Implementation

4.1 Summary of existing implementation

HaskellQuest was developed using the Unity game engine [66] - there are a few rationales for this choice. First, it was one of two game engines I had prior experience with at the start of development last year, the other being GameMaker [27]. Secondly, unlike GameMaker, Unity has a robust library for UI elements, which is an absolute necessity – the player's main interaction with the game is writing code.

Additionally, this choice was extremely beneficial for the development of the multiplayer campaign; Unity offers multiple frameworks and services for this purpose.

4.1.1 In-game code editor

The in-game code editor (last year's implementation of which can be seen in Figure 4.2a) is the main in-game object that the player interacts with - as such, it had to be immediately understandable and usable by the target demographic. The visual is explicitly modeled after actual IDEs and text editors, in accordance with Principle 2 of Nielsen's Usability Heuristics for interface design [48]: "Interfaces should not contain information that is irrelevant or rarely needed."

Additionally, it offers real-time comment highlighting - I felt this was of greater importance than other IDE features like syntax highlighting and automatic indentation, as the Haskell challenges frequently use comments to provide instructions and context to the player. Unity's TextMeshPro system allows the use of rich text tags [54] - '<color>' tags, which can change the text colour of a substring, are inserted when a '--' is detected, and removed when they aren't.

4.1.2 Haskell-game interface

Each Haskell challenge comprises two sections - the code that is shown to the player, and hidden test code that is appended to the code text when making a compilation request. This section of the code involves checking the player's written function against a sample implementation and using an IO monad to print the output - additionally, it can

print custom error messages to provide extra information to the player on what went wrong.

Meanwhile, each attack phase in the game corresponds to one behaviour script inheriting from the abstract class AttackController. The class defines a Trigger() method that takes a Boolean argument corresponding to the result of the associated Haskell challenge's test code. This is then overridden by concrete implementations in the individual attack scripts.

In last year's implementation, most challenges' test code output a simple Boolean value. After the player presses the 'Submit' button, the Boolean value is parsed from the compiler output (after the check for error messages) and the Trigger function is called on the current phase's attack script.

The exception to this was the challenges in the tutorial section; since the player there is interacting a function that is supposed to directly modify their health, I decided to account for the case where the player **increases** their own health. To this end, the Trigger function is overloaded with a version that accepts an 'additionalConditions' string: the Haskell test code can therefore output any additional information required.

4.1.3 Online Haskell compiler

Similar to Bogomil's HaskellQuest [9], the game uses the JDoodle online compiler [36] to handle Haskell compilation. JDoodle provides a compilation API - the service allows a specific number of requests per day, with the limit being larger for accounts with a monthly subscription. (All of the HaskellQuest projects this year share a JDoodle account).

The use of an online compiler sidesteps any issues with needing to have a local version of GHC installed - the installation process, compiler version inconsistencies, handling infinite loops, and so on. Additionally, the fact that JDoodle's compilation API is a paid service allows an expectation of input sanitation for security purposes. Using JDoodle does mean that even the single-player version requires an internet connection, but this feels like a reasonable expectation for the target demographic of the game.

However, the use of JDoodle introduces some issues - these are discussed in section 4.7

4.2 Network architecture for multiplayer

Unity's proprietary solution for networking is called Netcode for GameObjects (NGO) [2]. It supports two types of network architecture by default; a central server-based setup and a host-client setup [12].

4.2.1 Central server architecture

Here, a dedicated central server is used to keep track of the game state (i.e. it is **authoritative**) – the players' individual game instances communicate with the server to receive the latest state updates.

With NGO, whichever instance is authoritative is also the only one actually allowed to modify the game state – the other instances send requests to the authority to modify the state in specific ways. To use an example related to HaskellQuest, any time a player wished to move their player avatar, their game instance would need to send a request to the central server. The server would then make the change and reflect this to the game instances - thus, the players would see any positional change take effect with a delay. Commercially-released multiplayer games use techniques such as client-side prediction and rollback to lessen the impact of this. However, I had no experience in implementing these (or in hosting a server), so HaskellQuest opts for the host-client approach.

4.2.2 Host-client architecture

Here, one player acts as a host; its game instance acts as both a client and a server simultaneously, meaning it is authoritative. In the case of HaskellQuest, player 1 is the host, while player 2 is the client – the game state therefore 'exists' on player 1's instance. This provides player 1 with a massive latency advantage; in a given attack phase, the projectiles' positions and destruction are reported to player 2's instance with a time delay, and by default player 2's directional inputs would also register with a delay.

To ameliorate this somewhat, the player avatars are set to be **client-authoritative**. While this makes no difference for player 1, it changes the behaviour for player 2: instead of only sending requests to update the avatar's position, player 2's client instance updates the position itself and then informs player 1's host instance of the change. Thus, player 2's avatar 'exists' on player 2's instance - this allows player 2 to see their directional inputs take effect immediately.

However, spawning and destroying NetworkObjects cannot be set to be client-authoritative; thus, every single projectile object still 'exists' on the host instance. This means that projectile collisions with player 2's avatar are detected using a slightly-out-of-date position. Thus, in a measurable sense, dodging projectiles is harder for player 2 than it is for player 1.

Thankfully, this only applies to objects that are dynamically spawned during runtime; objects that are present from the beginning of the scene, like each player's code editor, communicate on a more equal basis using Remote Procedure Calls (RPCs).

Another issue with this network setup is that if the host stops their instance, the game session is over. However, this isn't a problem in the context of HaskellQuest. Both players have to progress through the game from beginning to end at the same time, so the game session **should** end if either player disconnects - HaskellQuest's handling of this is described in section 4.3.

4.2.3 Potential improvement - Distributed Authority

Unity has begun supporting an alternative topology for NGO known as Distributed Authority [18]. Here, there is no 'server'; each client assumes authority for a **subset** of objects in a scene - the state updates for objects not owned by a client are communicated

by other instances. Authority is transferrable between clients - however, this leaves the game state open to race conditions which need to be explicitly handled if multiple clients attempt to obtain authority of the same object at once. This method is distinct from a true peer-to-peer implementation, as it requires a central service to establish client communication.

While its implementation in HaskellQuest may solve the issue of player 1 having a latency advantage during the attack phases, the question of 'which objects are owned by which client' is far from straightforward - a key part of the gameplay segments is that any projectile can hit any player, which may increase the chance of race conditions if a projectile and player avatar are owned by separate clients. Additionally, Distributed Authority is still in its beta phase, which could have made implementation problematic.

The reason I didn't investigate it further at the beginning of development, however, was that I was under the impression that Distributed Authority was a fully-paid service - in fact, it is free up to a monthly limit of 6000 connectivity hours / 100 GB of bandwidth [64], which would have easily covered HaskellQuest's needs. By the time I realised, however, too much of the game's multiplayer architecture had already been implemented.

4.3 Handling connection and disconnects

Direct connections between game instances based on IP address are not possible without port forwarding or NAT punch-through [16] - the former requires access to router settings (which would be problematic if, say, players were to use a public WiFi like Eduroam) while the latter is known to be unreliable. Therefore, to handle connections, the game uses Unity's Relay service [53].

The service allows hosts to request allocations on a relay server, a public endpoint that is open to all players and exists independent of the session lifespan - it acts as a go-between to enable player connections [52]. Once an allocation is in place, the service returns a room code to the host: a six-character string used as a unique identifier. Clients can then enter this code to join the session. Since HaskellQuest requires only 2 players, the game requires connecting clients to receive approval from the host - if a client has already joined, the host automatically denies any others.

As with Distributed Authority, use of the Relay service is free up to a point - in this case, up to 50 concurrent users monthly, and up to 3 GB of bandwidth per concurrent user [64]. Again, this is more than enough for the needs of HaskellQuest's multiplayer campaign.

NGO defines a NetworkManager singleton that contains networking information and netcode-related settings [47]; in HaskellQuest, the singleton is created at the start screen and persists across the remaining scenes. The same scenes are used for both the single-player and multiplayer campaigns; different dialogue text, cutscene text, Haskell challenges and attack controller scripts are enabled depending on the status of the NetworkManager. (Thus, during the single-player campaign, the player's instance is still a host - there are just no connected clients and no Relay allocation.)

The NetworkManager has functionality that allows for game code to be executed when a disconnect is detected. While other games allow client instances to 'catch up' on the game state if they reconnect after a period of brief disconnection, HaskellQuest opts to instantly end the session - the player still in the session is informed of the disconnection and shown a prompt to return to the main menu, with there being no option to continue. This is because the design of the game does not allow for much desynchronization between players: they are meant to go through the same segments of the game at the same time.

However, NGO's detection of disconnects is nuanced. By default, game instances send heartbeat packets to one another at constant intervals in periods of inactivity to maintain the connection. If one instance hasn't received a packet from another in some time, it begins sending a set number of connect requests at constant intervals - if all of these go unanswered, the disconnect event is triggered.

While a player outright terminating the connection (by, for instance, closing the application) is detected instantaneously, disconnects in the face of network issues are detected with a delay due to the above behaviour - additionally, in situations where the connection quality is variable, this delay may be quite large.

As such, the multiplayer campaign's behaviour with unstable Internet connections is untested and may potentially lead to significant technical issues - thankfully, none of these problems were encountered during evaluation.

4.4 Adaptations to game systems for multiplayer

4.4.1 Attack controllers

In the multiplayer campaign, given that there are two functions per Haskell challenge which are usually independent of one another, the test code usually needs to output one result for each. As such, the version of the Trigger() function with the single Boolean is not sufficient; all of the AttackController scripts for the multiplayer campaign use the version with the 'additionalConditions' argument. (The Boolean version is used as a fallback in case of syntax / compilation errors.)

Additionally, section 4.2.2 mentions that NetworkObjects such as the projectiles can only be spawned by the authoritative instance, i.e. the host. Thus, while either player can use the 'Submit' button to send their code for compilation, only player 1's instance can actually call the AttackController script; thus, if player 2 has submitted, an RPC is used to notify player 1's instance of the result.

This works fine on its own in cases where all the AttackController is doing is spawning objects. However, the scripts for the tutorial and Dr. Fractal fights also involve visual effects like screen flashes - thus, in those cases, the requisite AttackController script on player 1's instance uses an RPC to tell the script on player 2's instance to execute the effect code.

4.4.2 Dialogue / cutscene text advancement

In last year's implementation, each new phase of the battle starts with a brief dialogue segment that advances the story. Within the dialogue box, each line appears character by character, accompanied by a 'speech' sound effect. The dialogue box is accompanied by an invisible button taking up the whole screen, which serves the following function: if the current line has finished, move to the next line of dialogue. Otherwise, the dialogue box auto-completes the line. The cutscenes in between levels use the same basic system.

For the multiplayer campaign, this would also need to be synchronized; the end of the dialogue is when players are allowed to use the code editor, while the end of the cutscene governs when the next scene is loaded. As such, RPCs are used; if one player clicks the screen, the relevant function is executed locally and also remotely executed on the other player's instance.

However, the implementation is flawed; the advance / autocomplete decision is still made based on local state, rather than through the RPC call itself. Thus, if player 1 clicks the screen when the line is complete for them, but still going for player 2, player 1 advances to the next line while player 2 does not. This causes desynchronization, which can lead to bugs for one player like the code editor being able to be triggered while the dialogue box is still on screen, or the next scene being loaded while the cutscene text hasn't finished.

4.5 Synchronizing code between players

4.5.1 Discarded initial implementation

Since the initial idea for the Haskell challenge design involved real-time text synchronization a la Google Docs, the obvious first step was to find out how Google Docs implemented it. This led to considering two approaches:

- **Operational Transform (OT)** [22]: This relies on a given document being described as a sequence of edit operations. Each instance of the document, when receiving edits from other instances, transforms them relative to any local edits e.g. adjusting the position of an insertion. Google Docs uses a version of the Jupiter OT algorithm for multi-user collaboration [67].
- **Differential Synchronization (DS)** [25]: Here, each instance maintains its own copy of the text upon which edits are performed, as well as an additional copy known as a 'shadow' that cannot be edited by the user. These copies are used to calculate a list of deltas ('diffs') i.e. the user's edits to the text, which are then sent over to the other instance and applied to the text and shadow there ('patches'). The algorithm is symmetric; each instance receives diffs, applies them, and sends a list of edits. The DS algorithm was used in Google's now-defunct MobWrite service [45], enabling multi-user collaboration for JavaScript applications.

The latter was picked largely because the DiffMatchPatch library [17], providing operations for calculating diffs and applying patches, was available in C# - it would therefore be very easy to use with Unity. However, some wrangling was required -

the Diff class provided by the library could not be used with Unity NGO's RPC calls, so I had to add a functionally-identical class implementing the INetworkSerializable interface. Additionally, when retrieving the players' code to calculate Diffs, the color tags for comment highlighting would have to be removed to avoid errors – these would then need to be re-added on the other instance when applying the Patches.

Differential Synchronization assumes a client-server network setup: this straightforwardly mapped to the client-host setup used for HaskellQuest. The implementation uses the Dual Shadow method described in the original paper, as seen in Figure 4.1. The paper outlines a more advanced implementation known as the Guaranteed Delivery Method. However, this was unnecessary - Unity NGO guarantees eventual ordered delivery for RPCs, so there would be no need to consider the scenario of a packet being outright lost.

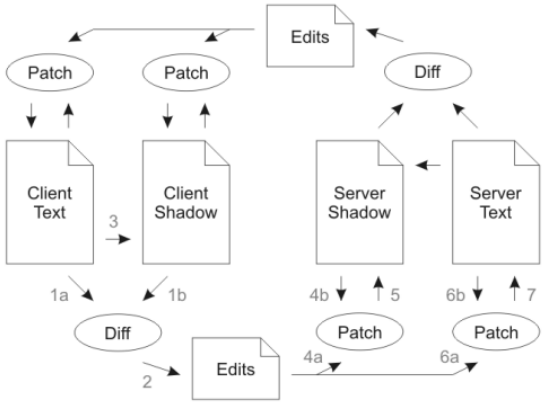


Figure 4.1: An illustration of the Dual Shadow variation of Differential Synchronization, taken from the original paper [25]

4.5.2 Final implementation

Since the two players are no longer working on the same code for the reasons described in section 3.2.1, a Differential Synchronization implementation is no longer needed. Instead, at fixed time intervals, the client instance sends an RPC to the host instance with the string contents of the InputField, i.e. player 2's code; upon receiving this, the host instance updates the code block for the other player's code (as seen in Figure 4.2) and sends back the contents of its own InputField i.e. player 1's code. As such, while the symmetric nature of DS is maintained, as well as the detail of the client initiating the connection, most of the actual complexity is stripped out.

4.6 In-game code editor improvements

4.6.1 Code blocks

In last year's release, the code editor only consisted of one InputField, i.e. an editable text box, and a small display to show the result of compilation as well as any errors (which had its own issues - more in section 4.6.3). However, the Haskell challenges

were specifically designed with the idea that players would only modify a specific section of code;

This led to unintended exploits. For instance, the tutorial challenges for list comprehension and recursion defined a list to be used as an argument to the function that the player was supposed to modify; however, the challenge could be passed by just modifying the argument list itself.

As such, for this year, the code editor 'window' is split into two sections; the InputField contains the code that the player is supposed to modify, while a Scroll Rect [59] containing an ordinary text box is used to display the code that the player cannot modify, like type definitions and helper functions. Additionally, for the multiplayer campaign, players needed to be able to view the other player's code being edited in real-time - an additional block was introduced for the two-player campaign, which can be toggled with the '1P / 2P Code' button.

The new editor can be seen in Figure 4.2b.

(a) (b)

Figure 4.2: a) The code editor as it looked last year. b) The expanded code editor, showing all code blocks, taken from player 1's instance in the two-player campaign.

4.6.2 Haskell help text

In last year's release, the dialogue in the tutorial levels was used to introduce Haskell concepts to the player - the thought was that structuring it as a back-and-forth dialogue between Lambda-Man and the Hologram would engage the player. However, testers universally took issue - the problem was that all of the dialogue was skippable, and there was no way to revisit the information later.

Testers suggested a kind of Haskell 'cheatsheet' - such a feature already existed in Bogomil's HaskellQuest [9] as the 'Grimoire', which provided brief explanations for Haskell topics that could be used as a reference. Since this fit Gee's maxim of allowing

the player to view information upon request (see section 2.2.2), it was included in this year's release.

The actual text was based on 'Learn You A Haskell For Great Good' [38], just as the tutorial dialogue from last year was. The text is split based on topics that players can choose between using an initial menu.

To avoid overwhelming the player with information, the help topics are unlocked as the game progresses. The player can only access the first 3 topics to start with, i.e. types, expressions and functions, and case expressions and pattern matching. The information on lists, tuples and list comprehension is unlocked upon progressing to the second phase of the tutorial i.e. right before the Colonel Trigger-Finger battle, while the information on recursion is unlocked upon reaching the third tutorial phase i.e. right before the Dr. Fractal battle. This attempts to address the other half of Gee's principle: that information should be presented just as it becomes relevant to some actual experience.

I also had to decide how the help screen would be incorporated alongside the existing UI. As can be seen in Figure 3.1, the code editor already takes up half of the screen when active. The help screen would need to be usable at the same time as the editor, but it couldn't take up too much space on the other half of the screen as it would cover up the level graphics. Additionally, the size of the help screen would help determine the length of each topic's text. While the use of Scroll Rect would ensure that the text could extend past the visible boundaries, it still couldn't be too lengthy - otherwise, it could frustrate the player reading it.

The final implementation of the help text can be seen in Figure 4.3. It can be toggled with the use of the 'Help' button.

Figure 4.3: The help screen showing information about types in Haskell, taken from the tutorial in the single-player campaign.

4.6.3 Error messages

Another major criticism from last year's testing was the fact that the error messages were inadequate. The implementation there only kept the first line to show the error type, as I was worried that GHC's multi-line errors would inadvertently reveal the test code to the player. However, this impeded the game from providing valuable feedback on the player's attempts - this was a frequent enough complaint that I decided to disregard my concerns about the test code entirely.

As such, this year, the game displays all of the error text to the player - the text refreshes every time the player submits, so they have access to the error message for the **previous** execution. Similarly to the help screen, Unity's Scroll Rect is used to handle text that goes past the viewport boundaries.

The error screen is positioned in the same way as the help screen - as such, only one of them can be active at a time. Additionally, in the multiplayer campaign, any error messages returned by JDoodle's compiler API are sent to all players using RPCs, regardless of which player submitted the code.

The final implementation of the error display can be seen in Figure 4.4. It can be toggled with the use of the 'Errors' button.

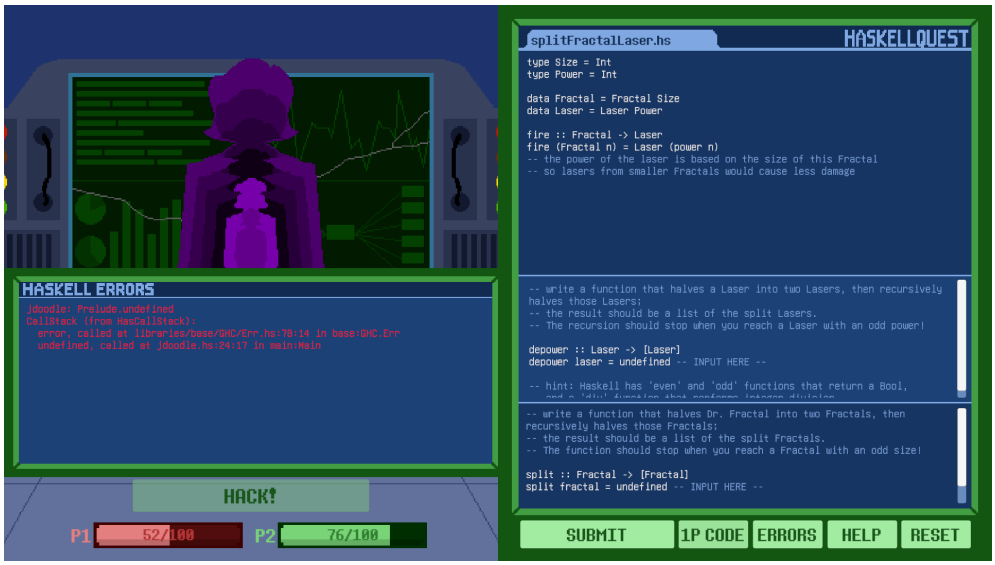


Figure 4.4: The error display showing a compilation error caused by the 'undefined' keyword, taken from the Dr. Fractal battle in the multiplayer campaign.

4.7 Issues with JDoodle

As mentioned in section 4.1.3, the game uses JDoodle, an online compiler service, to handle compilation requests - this sidesteps any issues with having to install GHC. However, it comes with its own issues, which were encountered more frequently this year.

4.7.1 Service unreliability

The time taken for JDoodle's compilation endpoint to respond to requests is immensely variable; occasionally, it will time out entirely (in the absence of a problem with the code itself, like an infinite loop), with the endpoint either returning an error or a truncated compilation output. Since these errors are sporadic but persistent, the game code for parsing the compiler output includes a special case for handling them.

At one point in March, the JDoodle compilation endpoint was inaccessible, returning repeated 500 (Internal Server) errors. A colleague working on his own HaskellQuest project, Ol Rushton [57], was able to quickly release a workaround - first, by exposing a port on the machine to allow the game to connect to a local instance of GHC (this took some modification to get working on Windows) and second, an alternative compiler endpoint running on Tardis Project servers.

While the JDoodle service came back up online in time for HaskellQuest's testing sessions, this illustrates one of the major downsides of relying on an external compiler service – the game's basic playability is entirely at the whims of said service.

4.7.2 Restrictions on target platform

The previous year's release was available for Windows, MacOS and Linux, to increase the potential demographic for testing. While Unity's support for multi-platform exports made this easy, the latter two versions suffered from a few compatibility issues. The Linux version opened in the wrong aspect ratio on DICE machines, while the MacOS version requires a complicated process to start the game for the first time – Apple requires executables to be notarized [65], and the certificate to do so can only be generated by members of the Apple Developer Program, so MacOS by default designates HaskellQuest as potential malware.

While this year's release ultimately ended up being available on the same platforms with the same compatibility issues, an alternative was initially considered. My colleague Dylan Drucker, also working on an educational game [21], released the test build on the platform Itch.io [19]; the site is intended for smaller independent game developers to showcase their work (among other things), and therefore allows WebGL builds to be run in a browser window. Since the game is running on Itch.io's servers and not natively on testers' machines, this would allow Mac and Linux users to play the game without encountering any issues.

However, when testing an early build of HaskellQuest on Itch.io, I was unable to get past the title screen – the game checks if it can connect to JDoodle's servers before progressing, but these checks were failing. JDoodle's servers do not allow CORS (Cross-Origin Resource Sharing) requests - thus, the WebGL build hosted on Itch.io's servers wouldn't be able to make any compilation requests. If HaskellQuest were to use its own compilation server instead, the server could be configured to allow CORS requests.

Chapter 5

Evaluation

5.1 Method of evaluation

5.1.1 Tester demographic

The game's target demographic was intended to be people who had prior experience with imperative programming, but none with Haskell. Last year, the game was sent out to undergraduate Informatics students in years 1-4; however, the problem was that everyone would have been taught Haskell in Introduction to Computation in year 1. While students in later years would have spent the vast majority of their time with imperative languages, and might therefore be 'rusty' with Haskell, this was still a mismatch.

To remedy this, invitations for testing were sent to **all** Informatics students this year; this would include Master's students who hadn't done their Bachelor's degrees at Edinburgh, who would therefore be more likely to have zero familiarity with Haskell. In addition, I enlisted the help of some of my friends from the Edinburgh College of Art (ECA)'s Game Design Studio course; the course involves the development of a game with Unity, meaning they'd have some prior programming experience as well.

5.1.2 Testing groups and questionnaire design

Part of the aim of this project is to assess whether collaborative learning has an impact on teaching Haskell. To this end, there would be two groups of testers, one for each campaign - the single-player group serves as a control to compensate for the improvements applying to **both** campaigns. Last year, links to the game and the evaluation forms were simply sent out to the people who registered interest; thus, they were allowed to play the game in their own time. However, this approach wouldn't work for the multiplayer testing - players would be required to pair up, which would shift substantially more responsibility onto the tester to find a partner and schedule a session.

Thus, it was decided to have the multiplayer sessions in person; since the single-player group was a control, that session would have to be in person too. However, this led to scheduling issues, as I predicted in last year's report; ultimately, I had to organise

3 separate sessions of 1.5 hours each, one for single-player and two for multiplayer testing. Testers were split in a 1:2 ratio, such that there would be the same number of unique playthroughs per group. In the end, there were 5 responses for the single-player evaluation and 10 responses for the multiplayer evaluation, as compared to 17 responses from last year's evaluation.

Given the increased variability of testers' prior programming and Haskell experience, I decided to include fields for testers to self-assess these attributes in a pre-game section of the questionnaire. While a more formal assessment (e.g. asking questions about programming concepts and Haskell syntax) would provide a more reliable estimate, it would also require more of a time commitment from the tester, and the in-person sessions were already a lot to ask. Additionally, last year's questionnaire already included a field similarly asking about the testers' prior Haskell experience - keeping this consistent would allow for easier comparisons.

The rest of the form is largely identical to that of last year; testers are asked to rate the game's entertainment value and suitability for newcomers, as well as the difficulty of the game and the usefulness of the help text. One new question asks the tester to rate the difficulty **progression**, to account for any sudden jumps in difficulty. The standard scale is from 1 to 6; the help text rating uses a different scale where 3 is the ideal score, 1 is 'didn't cover enough at all', and 5 is 'too excessive even for a beginner'.

Additionally, given the in-person sessions, I was able to directly make observations on the testers' experience with the game. Indeed, while there was a bug report form this year, it served little purpose - testers could just tell me about the issues they encountered.

5.2 Analysis of game evaluation metrics

This section uses the results from the previous report as well, to provide a point of comparison [5].

The mean, median and mode of all the evaluation metrics, across all three rounds of testing, can be seen in Table 5.1. From this, we see that the only score that shows a significant difference is the difficulty rating for the hardest challenge for the single-player group. As well, this seems to have changed for the **worse**, as the rating is higher - the potential reasons for this are explored in section 5.3.1.

However, any differences in these values can simply be explained by smaller sample sizes for this year's groups, as well as the change in tester demographic. As such, I used a one-tailed Welch's t-test [68] to assess any change in scores; this test assumes that the population variances are unequal, which hopefully compensates for the change in tester demographic. In addition, t-tests do not require the sample sizes to be equal.

As with any t-test, the null hypothesis is that the population means are equal. The alternative hypothesis for these tests is that the population mean for the first sample is less than for the second sample, i.e. the scores increased between year 1 and year 2, and between the single-player and multiplayer campaigns. The results can be seen in Table 5.2. However, most of the p-values do not fall below a typical significance threshold of 0.05 - the sole exceptions are a change in the highest challenge difficulty

	Mean			Median			Mode		
	Prev.	1P	2P	Prev.	1P	2P	Prev.	1P	2P
Enjoyment	4.71	4.20	4.60	5	4	5	5	4 / 5	5
Intro effectiveness	4.29	4.20	4.30	4	4	5	4 / 5	4 / 5	5
Peak difficulty	3.88	5.20	4.00	4	5	4	4	5 / 6	4
Diff. progression	-	3.00	3.10	-	3	3	-	2 / 4	3
Haskell info text	3.20	2.00	2.90	3	2	3	3	2	2 / 3

Table 5.1: The mean, median and mode for all metrics across all 3 rounds of testing.

between year 1 and year 2's single-player campaigns, and a change in the usefulness of the Haskell help text between this year's single-player and multiplayer campaigns. Both t-values are negative, meaning the scores improved - while this is meaningless for the latter score, seeing as the help text was identical, it is encouraging for the former score.

The fact still remains, though; we cannot conclude that the multiplayer campaign, or indeed any of the changes made this year, had a statistically-significant impact on most of the scores given by players. (The radically-different testing environment would also have logically had an impact on scores, but I couldn't compensate for that.)

Additionally, to further evaluate the game's suitability for newcomers, I wanted to understand the correlation between testers' prior experience with programming and Haskell and how they evaluated the game. As such, Pearson correlation coefficients [51] were calculated for each group - the results can be seen in Table 5.3. However, most of these results are not statistically-significant; the exceptions are the relationships between prior experience and the game's difficulty, which is obvious.

From these results, we can infer two things: testers' issues with the game are deeper than what was addressed in this year's development, and their impact was felt regardless of their prior experience with programming languages or Haskell.

Players were also asked to select the last Haskell challenge they were able to clear - this was also a part of last year's evaluation. The idea was that in place of a post-game Haskell evaluation, it would be a metric for what skills the player was able to learn, under the assumption that a player could not beat a particular challenge without understanding its Haskell concept. (This is a flawed assumption - the reasons are explored in the next section.)

Last year, the most common choice was the final phase of the recursion level i.e. beating the game as a whole. In this year's evaluation, the most common choice for the was the final phase of the list comprehension level for the single-player group, and the tutorial on recursion for the multiplayer group. While this could be explained by the time limit imposed on this year's testing sessions, this does also suggest that the list comprehension level is something of a bottleneck – this is borne out by observations, tester comments, and the fact that the **second** most common category last year was also the final phase of the list comprehension level.

Also, we can conclude that the improvements to the Dr. Fractal fight, detailed in section 3.5, had little effect on scores, as most players this year didn't get to that level.

	Test groups being compared		
	Prev ->1P	Prev ->2P	1P ->2P
Enjoyment	1.1, 0.86	0.27, 0.61	-0.83, 0.21
Intro effectiveness	0.20, 0.58	-0.014, 0.49	-0.21, 0.42
Peak difficulty	**-2.9, 0.0096**	-0.20, 0.42	1.9, 0.96
Diff. progression	-	-	-0.17, 0.43
Haskell intro text	2.9, 0.99	0.56, 0.71	**-2.0, 0.034**

Table 5.2: The t-values obtained by comparing group results, alongside the associated p-values. The results that show a significant correlation ($p < 0.05$) are in **bold**.

	Prior Haskell			Prior programming	
	Prev.	1P	2P	1P	2P
Enjoyment	0.52, 0.034	0.33, 0.59	0.059, 0.87	-0.25, 0.69	0.32, 0.37
Intro effectiveness	0.35, 0.17	0.87, 0.053	0.48, 0.16	0.83, 0.084	0.34, 0.33
Peak difficulty	**-0.56, 0.018**	-0.76, 0.13	**-0.75, 0.012**	-0.40, 0.51	-0.25, 0.49
Diff. progression	-	0.46, 0.44	-0.19, 0.60	**0.89, 0.039**	0.15, 0.69
Haskell info text	0.35, 0.17	0.65, 0.24	-0.44, 0.21	0.73, 0.17	-0.38, 0.28

Table 5.3: The Pearson correlation coefficients between the testers' prior knowledge and their evaluation scores, alongside the associated p-values. The results that show a significant correlation ($p < 0.05$) are in **bold**.

5.3 Areas of improvement

This section discusses areas of the game that testers took issue with - these were either expressed by testers or observed by me during the testing sessions. These factors may be the reason for the lack of significant improvement in the scores - essentially, the approach of this year's project was to build on the foundations from last year, but those foundations were flawed to start with.

5.3.1 Difficulty progression

Testers felt that the tutorials were far too easy to serve as an effective introduction to the game. Specifically, solving the challenges involve very minor alterations (e.g. removing a '- 10' from an expression, or replacing a '<=' with a '>='), meaning players didn't really need to engage with the Haskell syntax. Consequently, the transition to the Colonel Trigger-Finger fight, where players have to write a function themselves requiring more syntax knowledge than before, was felt to be too steep - the ECA students (the closest to complete beginners) couldn't progress past that point.

In terms of the attack segments, there were a few balancing issues. For instance, the attacks in the multiplayer campaign's Colonel Trigger-Finger level had too many projectiles, furthering the sense of that fight being an unfair escalation in difficulty. In the opposite direction, the attack in phase 2 of the Dr. Fractal fight was too easy to dodge, as the projectiles don't follow the player's position.

These issues break both the "consolidation and challenge" progression principle and the "'doable' but challenging" difficulty principle mentioned in section 2.2.2.

5.3.2 Utility of help screen and comment hints

HaskellQuest, in its current state, relies on explanatory text, from the Haskell info text in the help screen to the dialogue establishing story context to the comments in Haskell challenges letting the player know what to do.

While the help screen was an improvement from last year's tutorial dialogue (specifically called out as such by one tester who had taken part last year), the difference in the actual score was marginal, and there were still some issues.

For instance, there was a notable omission from the information covered in the text - how pattern-matching can be incorporated in the generators in a list comprehension. I excluded this on the grounds that I thought it would make the Colonel Trigger-Finger list comprehension challenges too easy; however, testers were consistently unable to figure the feature out based on the other information provided. Indeed, the testers in the single-player group were completely stuck - I had to provide the same hint to everyone so that they could progress past the Colonel Trigger-Finger level.

As well, though testers were seen using the help panel when attempting the Haskell challenges, they also showed an inclination to disregard it unless absolutely forced to - this applied to the explanations given in comments in the tutorial challenges as well, meaning testers were frequently unaware of the 'Error' and 'Help' buttons, or the fact that they could move during the attack segments, until these were pointed out to them.

The latter point gets at a more fundamental problem. HaskellQuest's challenges take inspiration from the tutorials in the Inf1A course; consequently, the instructional text is not an incorporated part of the experience, but an obstacle to get through in order to actually get to the gameplay. Additionally, as mentioned in section 4.6.2, the text was explicitly based on a Haskell textbook - however, an actual textbook is necessarily going to do a better job of explaining Haskell concepts than a help panel in a video game ever could. To further the point, testers in the first multiplayer testing session were seen using supplemental resources like StackOverflow or ChatGPT.

As such, HaskellQuest's current design framework may be better suited to being a tool used to **reinforce** Haskell concepts alongside some other method dedicated to **introducing** said concepts, rather than being an all-in-one learning solution – the design may be better suited for a different demographic than the one it is nominally aimed at. Indeed, Rushton's HaskellQuest [57] seems to take this approach - the game places emphasis on 'comprehension checks' and allows limited and situational modification of code, but at no point does it actually **introduce** Haskell's core concepts to the player.

5.3.3 Haskell challenge design

The restrictive nature of HaskellQuest's challenge design was intended to make the game more beginner-friendly, but there are some key flaws. While each challenge is **meant** to focus on one Haskell concept, the game only requires that the function behaves in a certain way, with no constraints on **implementation**. For instance, any list comprehension challenge can instead be solved by pattern-matching against the head and tail of the list, modifying the head, and calling the function recursively on the tail.

Indeed, the Inf1A tutorial exercises rely on this concept, which was briefly considered as the design approach last year - it was nixed because parsing the player's function to figure out how they'd implemented it would be difficult. The game currently attempts to mitigate this by only offering Haskell information prior to the level intended to use it, as mentioned in section 4.6.2. Even so, the challenges have unintentional ambiguities, making the approach less beginner-friendly.

The restrictiveness also negatively affects the gameplay - players may be punished for taking actions that make sense within the story, but aren't the intended answer. For instance, in phase 2 of the Colonel Trigger-Finger fight, rather than filter an input list to find one inactive Missile, the player could instead opt to deactivate **all** the Missiles - however, the output list would be longer than expected, leading to the test failing.

It is instructive here to look at two other HaskellQuest projects. Segboer's HaskellQuest [60] eschews writing syntax in favour of an approach where values and functions are physical objects in-game that can affect one another; the game thus prioritizes teaching the logic (rather than syntax) of functional programming to newcomers through puzzles with only one solution, as players aren't allowed to introduce new functions.

On the other side of the spectrum, Bogomil's HaskellQuest [9] seems almost to define a domain-specific language for the player's actions in battle. This approach provides a good basis for emergent gameplay [70], which can serve as a particularly effective learning tool - this is Gee's fundamental thesis in section 2.2.2.

HaskellQuest's current design philosophy falls into an unfortunate middle ground - it isn't as accessible and intuitive to beginners as Segboer's HaskellQuest, but doesn't allow as much freedom for experienced Haskell users as Bogomil's HaskellQuest. Additionally, the restrictive conceptual progression poses a problem for the design of the multiplayer challenges. Since each half of the challenge needs to cover the same Haskell concept and story context, they can end up being quite similar. In the worst cases, if one player has a solution, the other can simply copy it and change a few words, like phase 2 of the Colonel Trigger-Finger level:

```
filterMissiles missiles = [Missile pos active | (Missile pos
active) <- missiles, not active] -- PLAYER 1

filterLandmines mines = [Landmine coords active | (Landmine
coords active) <- mines, not active] -- PLAYER 2
```

5.4 Technical issues

5.4.1 UI shortcomings

Testers reported issues with the in-game code editor. These were mostly problems that were observed last year, as well as during development: for instance, lines overlapping each other if you backspace at the beginning of a line, or being unable to select text. These occur sporadically, and are to do with Unity's InputField implementation itself - they may be fixed in more recent versions of the Unity Editor.

Testers also requested features like syntax / error highlighting and automatic indentation - this is consistent with last year's feedback. There are in-game code editor assets available on Unity's Asset Store that offer these features [56][35] - however, they are all paid. Also, using these assets with HaskellQuest would have required more work than reusing the previous implementation, especially with the UI enhancements described in section 4.6. Thus, while these assets could potentially solve the other InputField issues as well, I decided not to, as I believed the missing features were simply inconveniences.

However, in one of the testing sessions this time around, two testers encountered a strange issue to do with newline characters. Copying and pasting a line of code onto a new line introduced some problematic characters that resulted in a parse error, even though the code itself was functionally correct – the testers got penalized in-game, and were ultimately unable to progress. While I was unable to recreate this myself, I suspect the root cause is the real-time commenting, along with the InputField's issues.

Additionally, Unity's Scroll Rect implementation, used for the non-editable code blocks and the help and error screens, has a bug - occasionally, the scrollbar will fail to change length if the viewport contents increase in size. For the code blocks, I specifically had to force the game to rebuild the objects' UI layouts to fix the issue - the help screen, meanwhile, seemed mysteriously unaffected in testing. However, I forgot to fix this for the **error** screen, so on occasion players will be unable to read all of the error text.

5.4.2 Game not running in background

By default, Unity does not allow games to run in the background (i.e. not the primary window) - this is governed by one setting in the Project Preferences [30]. Because of this, if a player switches windows while HaskellQuest is open, the game pauses execution. This didn't pose a problem last year; however, in this year's release the NetworkManager registers a disconnect. Thus, if the player switches windows while in multiplayer, the session immediately halts, forcing testers to start from the beginning. Again, though, the fix for this is trivial - the bug just went uncaught during testing.

5.4.3 Platform compatibility issues

These were described in section 4.7.2. Additionally, at the start of evaluation, I'd provided incorrect setup instructions for MacOS. I'd assumed the process would be the same as last year; however, with more recent versions of MacOS, the setup had become even more complex. This meant that in all sessions, I had to offer my own laptop to one tester so they could take part. The problem of the game not running in the background seems to be worse on MacOS as well; testers reported the game crashing upon switching windows, even when in single-player.

5.4.4 Desynchronization issues

These were described in section 4.4.2. While they didn't significantly impact testers, they were consistently observed - the specific instances mentioned previously were flagged by testers.

Chapter 6

Conclusion

6.1 Achievements

The main achievement of this year's work was the successful design and implementation of the collaborative multiplayer campaign, a completely novel approach to teaching functional programming. While the evaluation failed to show a significant change in scores, testers' responses to the questionnaire showed they really enjoyed the experience and wanted to see it further fleshed out. Anecdotal evidence from the testing sessions also proves the basic validity of the collaborative approach; even with the single-player campaign, the testers' first instinct was to collaborate with one another on problems too difficult for them to solve on their own.

Additionally, though the improvements made to both campaigns didn't address the more fundamental limitations of the game, they did improve the experience - the help text was used quite frequently by testers (and specifically called out as an improvement), as was the multi-line error display. The core emphasis on a fun storyline and colourful, detailed presentation continues to be a successful approach - as with last year, testers consistently praised the attention paid to the game's artwork, sound effects and creative gameplay concepts.

Once again, working on HaskellQuest has taught me a lot about game development. On a baseline level, I now know more about designing and implementing networked games. Additionally, reading through the existing literature on educational game design and collaborative learning in order to justify the game's design choices has helped strengthen my understanding of game design principles in general.

6.2 Future work

The evaluation this year was extremely valuable; outside of the insights on the multiplayer campaign itself, it revealed that a lot of the fundamental systems of HaskellQuest would have to be reconceptualized in order to push it to its full potential as an educational tool.

As discussed in section 5.3.3, the current design seems to be pulled in two directions. Lambda-Man's stated powers of 'altering reality by writing Haskell', as well as the game's basic premise of mirroring the gameplay scenarios in Haskell challenges, lean towards player inventiveness and interaction, but this would leave newcomers incredibly lost - as such, the game has to offer the **illusion** of freedom through challenges that each allow completely unique, but limited, interactions.

However, this approach requires newcomers to follow the game's exact 'train of thought', so to speak, which is problematic if the jump from one challenge to the next is too big, or the player feels they don't have enough information to figure out what the game wants them to do. Additionally, such an approach also limits the game's replayability, as well as its potential for expansion (as discussed in section 3.6).

It would therefore be beneficial to reconsider the challenge design approach and target demographic in tandem, i.e. to more intentionally alter the design towards newcomers or experienced programmers. A change that could make the adjustment in either direction easier would be to commit to a consistent set of game mechanics from beginning to end, rather than introducing a new kind of interaction per challenge. This would also allow the tutorial to be more effective – the mechanics introduced would translate directly to the more difficult battles, rather than running the risk of being too dissimilar.

The downside such an approach has over the current design is that it isn't intuitively structured around the introduction and testing of Haskell concepts - however, the game as it stands has problems around those two areas anyway.

A reconceptualization of the design would also benefit the multiplayer campaign, in that the collaborative aspect could be considered from the very beginning. A lot of the difficulty I experienced in translating HaskellQuest to multiplayer, aside from the basic novelty, was because the original framework was never really intended to support a collaborative experience.

This wouldn't quite be starting from scratch; the core ideas could be preserved, as well as the entertaining aesthetic approach, but without the accrued design and technical debt of the current version of HaskellQuest.

Essentially, the ideas and lessons across both years of this project can serve as a strong foundation for future work on HaskellQuest.

Bibliography

[1] *40009 (Computing Practical 1) — Faculty of Engineering*. Imperial College London. URL: https://www.imperial.ac.uk/computing/current-students/courses/40009/.

[2] *About Netcode for GameObjects — Unity Multiplayer*. Unity Technologies. URL: https://docs-multiplayer.unity3d.com/netcode/current/about/.

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

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

[5] Neel Amonkar. "HaskellQuest: a game for teaching functional programming in Haskell". Master's Thesis, Part 1. School of Informatics, University of Edinburgh, 2024.

[6] 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.

[7] Fabian Beuke. *GitHut 2.0 - GitHub language stats*. URL: https://madnight.github.io/githut/#/pull_requests/2024/1.

[8] 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.

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

[10] Manuel M. T. Chakravarty and Gabriele Keller. "The risks and benefits of teaching purely functional programming in first year". In: *Journal of Functional Programming* 14.1 (2004), pp. 113–123. DOI: 10.1017/S0956796803004805.

[11] Eunhye Choi et al. "Commercial video games and cognitive functions: Video game genres and modulating factors of cognitive enhancement". In: *Behavioral and Brain Functions* 16 (Dec. 2020). DOI: 10.1186/s12993-020-0165-z.

[12] *Client-server topologies — Unity Multiplayer*. Unity Technologies. URL: https://docs-multiplayer.unity3d.com/netcode/current/terms-concepts/client-server/.

[13] *CodeShare*. URL: https://codeshare.io/.

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

[15] *Collaboration - Replit*. URL: https://replit.com/collaboration.

[16] *Create a game with a listen server and host architecture — Unity Multiplayer*. Unity Technologies. URL: https://docs-multiplayer.unity3d.com/netcode/current/learn/listen-server-host-architecture/.

[17] *DiffMatchPatch — GitHub*. Google. URL: https://github.com/google/diff-match-patch.

[18] *Distributed authority topologies — Unity Multiplayer*. Unity Technologies. URL: https://docs-multiplayer.unity3d.com/netcode/current/terms-concepts/distributed-authority/.

[19] *Download the latest indie games - Itch.io*. URL: https://itch.io/.

[20] Karolina Drobnik. "HaskellQuest: a game for teaching functional programming in Haskell". Master's Thesis. School of Informatics, University of Edinburgh, 2018.

[21] Dylan Drucker. "Let's Get Cracking: Leveraging Gameplay from an Adversarial Perspective to Teach Password Security Concepts". Master's Thesis, Part 2. School of Informatics, University of Edinburgh, 2025.

[22] C. A. Ellis and S. J. Gibbs. "Concurrency control in groupware systems". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. Portland, Oregon, USA: Association for Computing Machinery, 1989, pp. 399–407. ISBN: 0897913175. DOI: 10.1145/67544.66963. URL: https://doi.org/10.1145/67544.66963.

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

[24] Sandro Franceschini et al. "Action video games improve reading abilities and visual-to-auditory attentional shifting in English-speaking children with dyslexia". In: *Scientific reports* 7.1 (2017), p. 5863.

[25] Neil Fraser. "Differential synchronization". In: *Proceedings of the 9th ACM symposium on Document engineering*. 2009, pp. 13–20.

[26] *Functional Programming (2024-25)*. University of Oxford. URL: https://www.cs.ox.ac.uk/teaching/courses/2024-2025/fp/.

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

[28] James Paul Gee. *What Video Games Have To Teach Us About Learning and Literacy*. Revised and Updated edition. New York: St. Martin's Griffin, 2007 - 2003. ISBN: 9781403984531.

[29] *Haskell in Industry - HaskellWiki*. HaskellWiki. URL: https://wiki.haskell.org/index.php?title=Haskell_in_industry.

[30] *How do you keep your game running even when you switch out of it? - Questions & Answers - Unity Discussions*. URL: https://discussions.unity.com/t/how-do-you-keep-your-game-running-even-when-you-switch-out-of-it/928.

[31] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, 2000.

[32] 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).

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

[34] *Informatics 1 - Introduction to Computation*. University of Edinburgh. URL: http://www.drps.ed.ac.uk/24-25/dpt/cxinfr08025.htm.

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

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

[37] David W. Johnson and Roger T. Johnson. *Learning Together and Alone: Cooperative, Competitive, and Individualistic Learning*. Fourth edition. Needham Heights, Mass., London: Allyn and Bacon, 1994.

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

[39] Alejandro Lujan and Christian Leger. *ScalaQuest*. URL: https://www.kickstarter.com/projects/andanthor/scalaquest-a-game-to-learn-scala/.

[40] 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.

[41] Christos Malliarakis, Maya Satratzemi, and Stelios Xinogalos. "CMX: The Effects of an Educational MMORPG on Learning and Teaching Computer Programming". In: *IEEE Transactions on Learning Technologies* 10.2 (2017), pp. 219–235. DOI: 10.1109/TLT.2016.2556666.

[42] Konstantinos Maragos and Maria Grigoriadou. "Designing an Educational Online Multiplayer Game for learning Programming". In: *Proceedings of the Informatics Education Europe II Conference*. 2007, pp. 29–30.

[43] Léa Martinez, Manuel Gimenes, and Eric Lambert. "Entertainment Video Games for Academic Learning: A Systematic Review". In: *Journal of Educational Computing Research* 60.5 (2022), pp. 1083–1109. DOI: 10.1177/07356331211053848. eprint: https://doi.org/10.1177/07356331211053848. URL: https://doi.org/10.1177/07356331211053848.

[44] 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.

[45] *MobWrite — Google Code Archive*. Google. URL: https://code.google.com/archive/p/google-mobwrite/.

[46] Mathieu Muratet et al. "Towards a Serious Game to Help Students Learn Computer Programming". In: *International Journal of Computer Games Technology* 2009.1 (2009), pp. 1–12.

[47] *NetworkManager — Unity Multiplayer*. Unity Technologies. URL: https://docs-multiplayer.unity3d.com/netcode/current/components/networkmanager/.

[48] 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.

[49] Elena Novak and Janet Tassell. "Using video game play to improve education-majors' mathematical performance: An experimental study". In: *Computers in*

Human Behavior 53 (2015), pp. 124–130. ISSN: 0747-5632. DOI: https://doi.org/10.1016/j.chb.2015.07.001. URL: https://www.sciencedirect.com/science/article/pii/S0747563215300121.

[50] Fotini Paraskeva, Sofia Mysirlaki, and Aikaterini Papagianni. "Multiplayer online games as educational tools: Facing new challenges in learning". In: *Computers & Education* 54.2 (2010), pp. 498–505. ISSN: 0360-1315. DOI: https://doi.org/10.1016/j.compedu.2009.09.001. URL: https://www.sciencedirect.com/science/article/pii/S0360131509002413.

[51] Karl Pearson. "VII. Note on regression and inheritance in the case of two parents". In: *Proceedings of the Royal Society of London* 58.347-352 (1895), pp. 240–242.

[52] *Relay servers*. Unity Technologies. URL: https://docs.unity.com/ugs/en-us/manual/relay/manual/relay-servers.

[53] *Relay: Free P2P Networking & Connection Solution — Unity*. Unity Technologies. URL: https://unity.com/products/relay.

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

[55] Martin Riopel et al. "Impact of serious games on science learning achievement compared with more conventional instruction: an overview and a meta-analysis". In: *Studies in Science Education* 55.2 (2019), pp. 169–214. DOI: 10.1080/03057267.2019.1722420. eprint: https://doi.org/10.1080/03057267.2019.1722420. URL: https://doi.org/10.1080/03057267.2019.1722420.

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

[57] Ol Rushton. "HaskellQuest: a game for teaching functional programming in Haskell". Bachelor's Thesis. School of Informatics, University of Edinburgh, 2025.

[58] Norsaremah Salleh, Emilia Mendes, and John Grundy. "Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review". In: *IEEE Transactions on Software Engineering* 37.4 (2010), pp. 509–525.

[59] *Scroll Rect — Unity UI — 2.0.0*. Unity Technologies. URL: https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/script-ScrollRect.html.

[60] Daniel Segboer. "HaskellQuest: Can a Fun Game be Educational?" Bachelor's Thesis. School of Informatics, University of Edinburgh, 2024.

[61] Constance A Steinkuehler. "Learning in Massively Multiplayer Online Games". In: *International Conference of the Learning Sciences 2004: Embracing Diversity in the Learning Sciences* (2004), pp. 521–528.

[62] Alexander Steinmaurer et al. "Developing and Evaluating a Multiplayer Game Mode in a Programming Learning Environment". In: *2022 8th International Conference of the Immersive Learning Research Network (iLRN)*. 2022, pp. 1–8. DOI: 10.23919/iLRN55037.2022.9815973.

[63] Pavlos Toukiloglou and Stelios Xinogalos. “Effects of collaborative support on learning in serious games for programming”. In: *Journal of Educational Computing Research* 63.1 (2025), pp. 126–146.

[64] *UGS Pricing*. Unity Technologies. URL: https://unity.com/products/gaming-services/pricing.

[65] *Unity - Manual: Code sign your application*. Unity Technologies. URL: https://docs.unity3d.com/2022.3/Documentation/Manual/macoscodesigning.html.

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

[67] Matthew Weidner and Martin Kleppmann. “The art of the fugue: Minimizing interleaving in collaborative text editing”. In: *arXiv preprint arXiv:2305.00583* (2023).

[68] Bernard L Welch. “The generalization of ‘STUDENT’S’problem when several different population varlances are involved”. In: *Biometrika* 34.1-2 (1947), pp. 28–35.

[69] Viktor Wendel et al. “Designing collaborative multiplayer serious games: Escape from Wilson Island — A multiplayer 3D serious game for collaborative learning in teams”. In: *Education and Information Technologies* 18 (2013), pp. 287–308.

[70] Mark Wilson. *Emergent Gameplay - Bumbling Through Dungeons*. URL: https://bumblingthroughdungeons.com/emergent-gameplay/.

Appendix A

User testing

The results from these questionnaires have been submitted as part of the Project Materials.

A.1 Single-player evaluation questionnaire

Participant Consent

Researcher: Neel Amonkar (s2030247@ed.ac.uk)
Supervisor: Don Sannella (don.sannella@ed.ac.uk)
Informatics Ethics Approval: 388105

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 the following 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

Prior to playing the game

3. How much **general programming experience** would you say you have?
(1 - Complete novice, 6 - I'm an experienced programmer) *

1	2	3	4	5	6

4. How much **knowledge of Haskell**, or functional programming in general, would you say you have?
(1 - Complete novice, 6 - I'm an experienced Haskell programmer) *

1	2	3	4	5	6

Evaluating the game

Start this section after playing HaskellQuest

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

1	2	3	4	5	6

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

- ◯ Tutorial Part 1 - Introduction to Functions
- ◯ Tutorial Part 2 - Introduction to Lists and Tuples
- ◯ Colonel Part 1 - Homing Missiles
- ◯ Colonel Part 2 - Missile Volley
- ◯ Colonel Part 3 - Combo Attack
- ◯ Tutorial Part 3 - Introduction to Recursion
- ◯ Fractal Part 1 - Splitting Dr. Fractal
- ◯ Fractal Part 2 - Traversing the Bolt Tree
- ◯ Fractal Part 3 - Healing Dr. Fractal

7. Of the challenges you completed, how **difficult** did you find the most difficult one?
(1 - Incredibly easy, 6 - Basically unfair) *

1	2	3	4	5	6

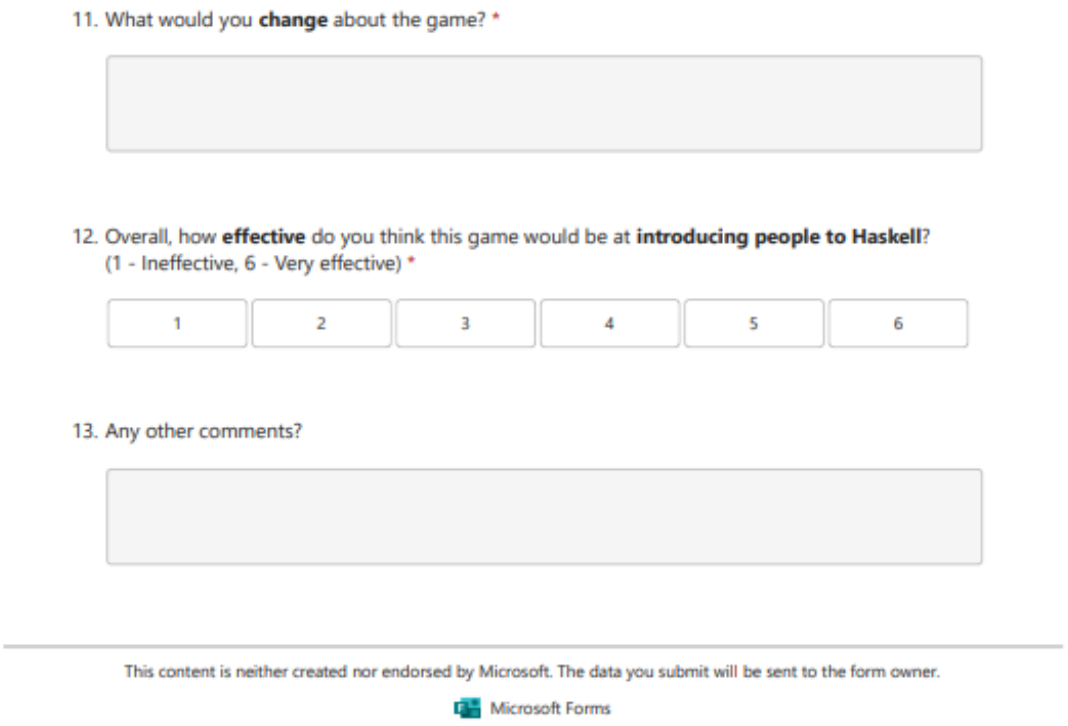
8. What did you think of the game's **difficulty progression**?
(1 - Poor e.g. unreasonable jumps in difficulty, 6 - Natural progression) *

1	2	3	4	5	6

9. What did you think of the **Haskell information text** available from the 'Help' panel?
(1 - Didn't cover enough at all, **3 - Just right**, 5 - Too excessive even for a beginner) *

1	2	3	4	5

10. What did you **like most** about the game? *


11. What would you **change** about the game? *

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

1 2 3 4 5 6

13. Any other comments?

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.

Microsoft Forms

Full list of responses to question 10: "What did you like most about the game?"

- The art and creativity is great. I love the aha moment of getting a puzzle right. The haskell information text is a BIG help and a huge improvement to the game and tutorial.
- I liked the way it looks- it is very engaging and fun to play. It was very user-friendly and I appreciated the "help" and "error" options.
- I enjoyed how the game poses challenges for the player to solve as it makes the learning curve far more memorable. I also liked how the challenges fit into a wider agenda.
- I liked the interactivity of the combat sequence - my code actually impact the behaviour of the missiles

  it took me a moment to realise what was happenning though when my missiles were successfuly redirected
- graphics, design, sound, story - it's lovely!!

Full list of responses to question 11: "What would you change about the game?"

- More help for the player. Perhaps a more tailored hint system. Syntax error information would be very helpful. The challenge of the airship level was much higher than the others imo
- I think examples of inputs and outputs would be helpful and help users with problem solving their answers. Also, the exercises at the start are a bit too simple

compared to the questions later- maybe having a step-by-step walkthrough of how to answer a question would be helpful. Another thing to add is that there was a lot of text during the intro to the game- maybe shortening it would be better.

- I would include a bank of some basic examples, not everything in the game may be easily deduced from the help panel, but the game should still maintain the puzzle element. I was particularly confused as to how a recursive function could map from a type to a list of the same type, and the process of extracting attributes from types using pattern matching. Although I have very little knowledge of Haskell or type theory, I have had a lot of experience with scientific computing in Python before, which did help out a bit. As such, I think the game is really aimed at those who are familiar with other high level languages like Python or Java, but are unfamiliar with the nuisances of type theory and Haskell.
- I would give the player way more context at the beginning - with examples etc.

  For the most part I felt like I was fighting a new unfamiliar syntax without enough resources to understand it - i coded in 5 different languages in my software engineering job but had zero idea about Haskell and its syntax is quite distinct

  That made the first few lever feel way harder then they actually were.

  i would also consider showing the errors before/during the combat to make it seem more like a coding experience - the errors were an afterthought and i forgot to look at them which probably wasn't the intended outcome
- it's quite hard - less steep learning curve, more explanations (maybe tips after game over) or more examples that resemble the actual challenges

Full list of responses for question 13: "Any other comments?"

- Great work! Thank you for the cake
- This was a fun game to play. I really enjoyed the characters and artistic elements of it as well as the quality of the game in terms of how it would help teach Haskell from a technical standpoint. I wish I had this in my first year when I was learning Haskell- this would have been a great thing to have available as part of the learning materials!
- it could be very effective and fun, just a little more help for beginners ;)

A.2 Multiplayer evaluation questionnaire

Participant Consent

Researcher: Neel Amonkar (s2030247@ed.ac.uk)
Supervisor: Don Sannella (don.sannella@ed.ac.uk)
Informatics Ethics Approval: 388105

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 the following 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

Prior to playing the game

3. How much **general programming experience** would you say you have?
(1 - Complete novice, 6 - I'm an experienced programmer) *

1	2	3	4	5	6

4. How much **knowledge of Haskell**, or functional programming in general, would you say you have?
(1 - Complete novice, 6 - I'm an experienced Haskell programmer) *

1	2	3	4	5	6

Evaluating the game

Start this section after playing HaskellQuest

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

1	2	3	4	5	6

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

- ◯ Tutorial Part 1 - Introduction to Functions
- ◯ Tutorial Part 2 - Introduction to Lists and Tuples
- ◯ Colonel Part 1 - Homing & Freeze Missiles
- ◯ Colonel Part 2 - Missiles & Landmines
- ◯ Colonel Part 3 - All-Out Attack
- ◯ Tutorial Part 3 - Introduction to Recursion
- ◯ Fractal Part 1 - Splitting Dr. Fractal and Lasers
- ◯ Fractal Part 2 - Traversing the Fractal and Bolt Trees
- ◯ Fractal Part 3 - Healing Dr. Fractal

7. Of the challenges you completed, how **difficult** did you find the most difficult one?
(1 - Incredibly easy, 6 - Basically unfair) *

1	2	3	4	5	6

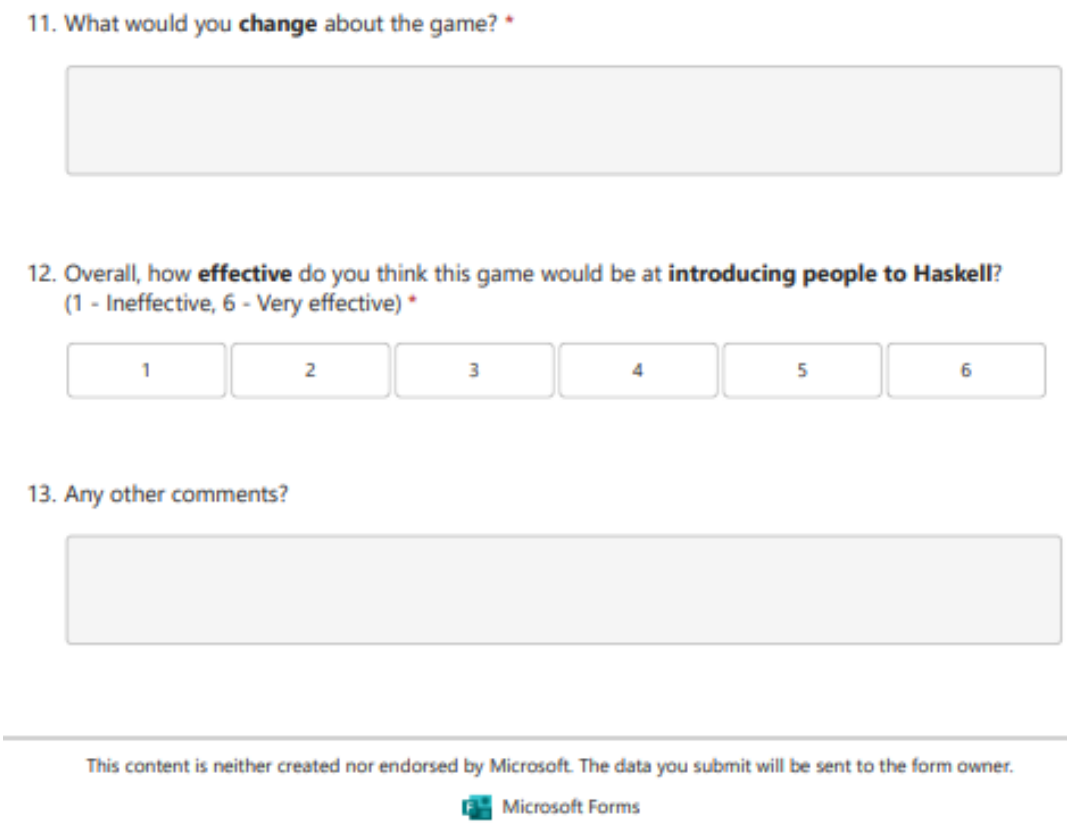
8. What did you think of the game's **difficulty progression**?
(1 - Poor e.g. unreasonable jumps in difficulty, 6 - Natural progression) *

1	2	3	4	5	6

9. What did you think of the **Haskell information text** available from the 'Help' panel?
(1 - Didn't cover enough at all, **3 - Just right**, 5 - Too excessive even for a beginner) *

1	2	3	4	5

10. What did you **like most** about the game? *


11. What would you **change** about the game? *

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

1	2	3	4	5	6

13. Any other comments?

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.

Microsoft Forms

Full list of responses to question 10: “What did you like most about the game?”

- You can see the other player’s progress in real time
- Great character names, good concept in collaboration. Think it could be a great tool, I could see it in schools
- I loved your art, aesthetic and music! The idea of a multiplayer educational game is also fun and could be really useful if implemented in an educational setting.
- - Multiplayer made it more fun. - the visual design was great as well. - the “”help“” and “”error“” tabs were useful as well.
- Both art and audio followed an retro style that was appealing.
- This is probably the most interactive and somewhat satisfying haskell quest I have played. The fact that you can see your code working and changing their attributes are actually quite cool, combined with the retro graphics, its actually quite an enjoyable experience imo. Other HaskellQuests that I have played before is rather static and passive, which is sometimes slows down the pacing of the game, which is too boring for kid like me. And this game doesn’t need you to go around solving puzzles and stuff is a plus for me coz puzzle rpgs need a lot of polishing to make it engaging. The puzzle sizes here are basically just right, which doesn’t steal too much of your time during coding and can actually enjoy some of the more “”kinetic“” parts of the game.

The graphics are good for me coz I like this style, as an FTL fan. quite retro. some of the maps could be more polished such as the turorial cave but mostly good. Could probably add some more spirites for damaged enemies to indicate health?

- The gamification of learning, getting to play mini game while also reviewing your code to see your mistakes
- I enjoyed having to dodge the bullets and being able to come back from failure. I also enjoyed the art style and music. More bullet patterns would always be appreciated.
- graphics were really cool and I enjoyed the shorter challenges
- Progression, graphics, and making learning Haskell enjoyable

Full list of responses to question 11: "What would you change about the game?"

- Add ctrl+z shortcut Add stage selection for stages I've passed Add a hotkey to skip dialogues Add some instructions on how to access the fileds in the data
- just felt a bit too hard or that some concepts weren't taught the best, meant I got a bit frustrated Would be cool to be able to see what the other player has highlighted so you can talk about the same section of code
- I think consideration of target audience would be helpful, if you haven't already defined this. As an absolute beginner I found a lot of this to be over my head, and would have found it helpful to watch a demo of a similar solution before solving on my own.
- - make the player names more interesting than ""Lambda-Man-1"" and ""Lambda-Man-2"" - Mac had crashes if I had the game in fullscreen and then switched to Google Chrome (network stuff?) - maybe make slight variations in the text screens as the dialogue progresses."
- The debugging feature should be activated using a button instead of activating it before the battle. UI could be improved: - Highlighting the words could show definition
- I think first of all, certainly fix bugs.

  Secondly, I think this may introduce a lot of complexity for both the dev and the players (especially when the target audience is new to haskell), but I think you could try adding more attributes to the enemies so that you have multiple ways you can tweak around and beat the boss; perhaps even different dialogue options/weaknesses exposed if you poke around in the code, which rewards the player for actually being exploring the the boundaries of Haskell and instantly applying knowledge that you just learnt. This could also be setting some optional things that are coded in some relatively difficult/unfamiliar/untaught Haskell mechanics, but the pattern might just be obvious enough that your player could figure it out by intuition, it could be a huge dopamine hit I am thinking. (And as yoi have noted, these attributes could also add a lot of PVE, PVP elements

especially for multiplayer options, e.g. sabotage teammate, COD zombies but Haskell LMAO)

Thirdly, although myself is fine with how things are coded, I think somethings still need better explanation, especially when the same weapon could have inconsistencies across puzzles. For example, freeze missiles had angle as an attribute at first, but in later stages it has the coord attribute.

Probably a final minor point is I think the hit boxes of the missiles are HUGE. It may be good to allow some dodging mechanics, but it doesnt need to be much tbh.

(Additional resources, games from steam: TIS-500, EXAPUNK. They are different from your game but certainly there has to be something that you can take away from; I liked EXAPUNK quite a lot.)"

- Syntax highlighting most of our errors were misspelt variable names or types
- More difficulty options, ability to adjust the size of the text editor space.
- probably have syntax errors underlined and show error for last failed task when you run out of lives
- Start even easier, syntax highlighting, prompt both players to submt before submitting, add some kind of optional reading before progressing to next level (a newbie mode)

Full list of responses to question 13: "Any other comments?"

- In the first two challenges all I needed to do was to change a minus to a plus, or just get rid of the code completely. It would be better if it cannot be cheesed in this way, or if you could show the 'correct'/suggested solution, so I know what I'm supposed to learn from solving it. In the third challenge (missle one) I'm a bit confused about what I need to do. Maybe be more explicit what the function that needs writing should be able to do? And maybe also show how the function would be called. (e.g. the retarget_missle function would be called with a new given target)"
- Great overall execution, I think with more playtesting this could be so interesting to publish!! Keep going!! :)
- Great game!
- Might be too hard for introducing people to Haskell.
- Uhh ... in this 15 minutes of constant texting I might have emptied my reservoir of ideas. If I have more, I will certainly talk to you more on this topic.

  I know you absolutely have no time to make tweaks to this game, but after your PHD you could probably dev this game further :3 publish on STEAM perhaps? GameDevSIG might be getting a STEAM publisher account soon, we could borrow you that."

- I think getting set back too far can be frustrating especially as you dont get to see the final set of errors when you die
- Very fun. More bullet patterns, and more Ide type functions like syntax highlighting would be great.
- collaborative multiplayer makes it fun to play and motivates you to get further in the game!
- Answer to 12 can be 6 with introducing the changes in 11

A.3 Bug report form

* Required

1. Which campaign were you playing through? *

- ◯ Single-player
- ◯ Multiplayer

2. Which part of the game did you encounter the bug in? *

- ◯ Tutorial Part 1 - Intro to Functions
- ◯ Tutorial Part 2 - Intro to Lists and Tuples
- ◯ Colonel Part 1 - Homing Missiles / Homing & Freeze Missiles
- ◯ Colonel Part 2 - Missile Volley / Missiles & Landmines
- ◯ Colonel Part 3 - Combo Attack / All-Out Attack
- ◯ Tutorial Part 3 - Intro to Recursion
- ◯ Fractal Part 1 - Splitting Dr. Fractal / Splitting Dr. Fractal and Lasers
- ◯ Fractal Part 2 - Traversing the Bolt Tree / Traversing the Fractal and Bolt Trees
- ◯ Fractal Part 3 - Healing Dr. Fractal

3. Please describe the bug, as well as how to recreate it if possible *

4. Please upload any screenshots / video of the bug if possible

Upload file

File number limit: 10 Single file size limit: 10MB Allowed file types: Word, Excel, PPT, PDF, Image, Video, Audio

Both recorded responses were from the single-player group.

- Tutorial Part 1 - Intro to Functions

Whenever i try to switch to another window from inside the game, the entire game freezes and has to be force quit to do anything

I don’t know if my case is special since i use a custom tiling window manager.

MacOS

- Tutorial Part 2 - Intro to Lists and Tuples

  Right after finishing a level I can see the robot guy from the next one for like quarter of a second (I suppose I’m not supposed to see it) - then it goes to the hologram guy saying I did a good job like it should

A.4 Participant information sheet

Participant Information Sheet

Project title:	Expanding HaskellQuest: a collaborative multiplayer game for teaching Haskell
Principal investigator:	Don Sannella
Researcher collecting data:	Neel Amonkar
Funder (if applicable):	N/A

This study was certified according to the Informatics Research Ethics Process, reference number 388105. 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 Don 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 novice programmers 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 the questionnaire, without giving a reason. After this point, it will no longer be possible to withdraw because we are not collecting any data that would allow us to identify you.

What will happen if I decide to take part?

- Kinds of data being collected: Participants' self-assessment of Haskell knowledge before and after playing the game, their opinions on various areas of the game (difficulty, presentation, etc.)
- Means of collection: Questionnaire after in-person playtesting session
- Duration of session: The time taken to play through the game, <1 hour, and the time taken to answer the questionnaire, ~5 minutes
- How often, where, when: Once, mid-March, through an online questionnaire

Are there any risks associated with taking part?

There are no significant risks associated with participation.

Are there any benefits associated with taking part?

Participants will be thanked by name in the game's credits. Additionally, at the in-person playtesting sessions, there will be snacks and refreshments for participants.

What data are you collecting about me?

The data we collect for our research is completely anonymous: We are not collecting any information that could, in our assessment, allow anyone to identify you. Your signed participant consent form will be kept separately from your responses and destroyed after **30 June 2025**.

What will happen to the results of this study?

The results of this study may be summarised in published articles, reports and presentations. Your anonymised data may be published and can also be used for future research.

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.

THE UNIVERSITY of EDINBURGH informatics

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.

A.5 Participant consent form

Participant number:_____

Participant Consent Form

Project title:	Expanding HaskellQuest: a collaborative multiplayer game for teaching Haskell
Principal investigator (PI):	Don Sannella
Researcher:	Neel Amonkar
PI contact details:	

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

Appendix B

Custom artwork

Figure B.1: The game's logo, as seen on the title screen.

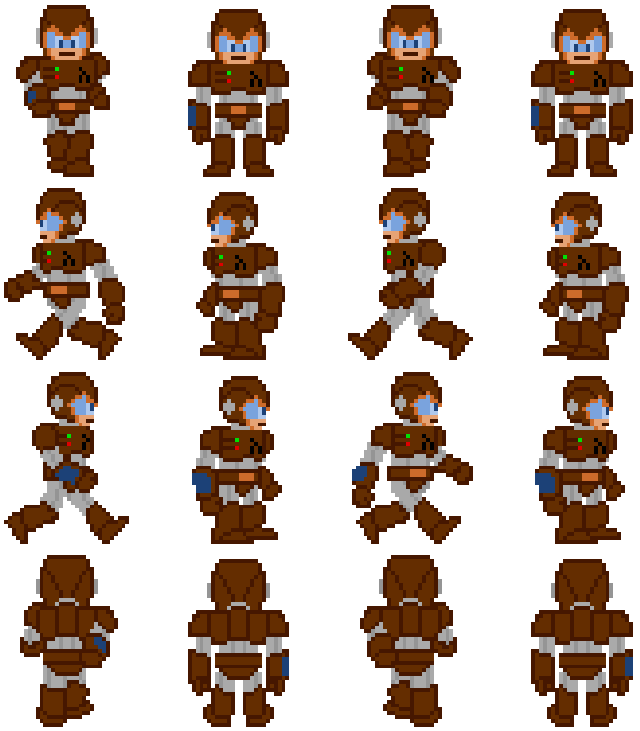


Figure B.2: The sprites for Lambda-Man moving in four directions - the sprites for the player avatars in the multiplayer campaign are just these sprites recolored.

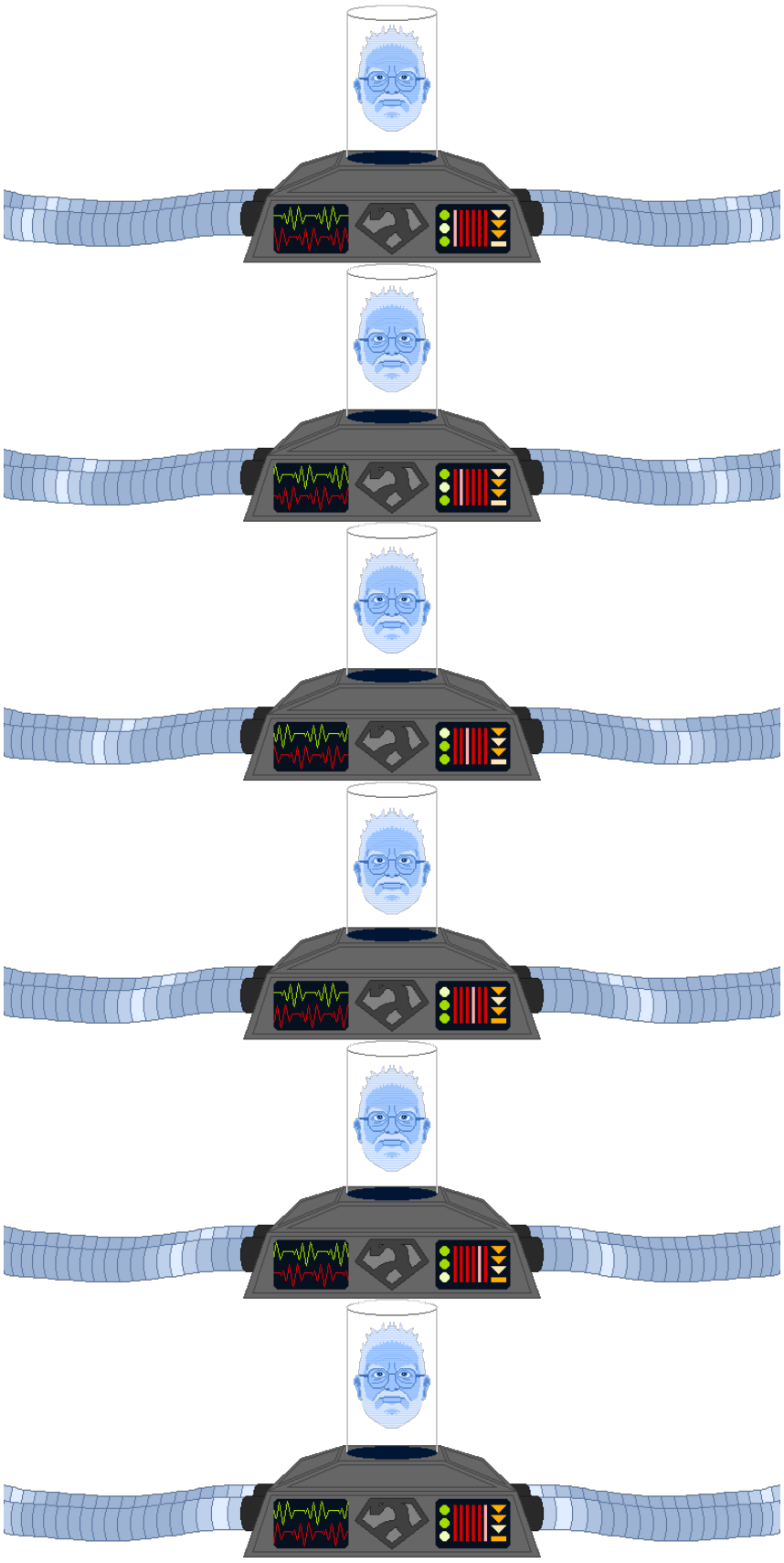


Figure B.3: All of the animation frames for the original Lambda-Man's hologram.

Figure B.4: All of the animation frames for Colonel Trigger-Finger in 3 states - normal, damaged, and exploding. The explosion sprites were taken from OpenGameArt.org - the original is credited in-game.

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.

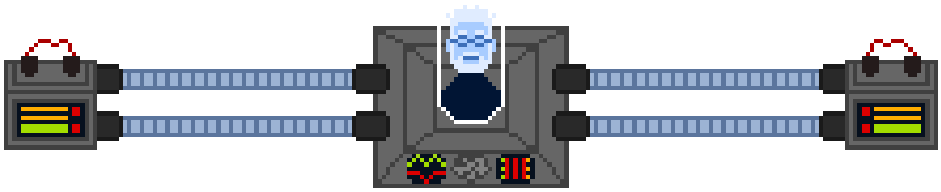


Figure B.6: The smaller 'overhead' sprite for the original Lambda-Man's hologram.

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

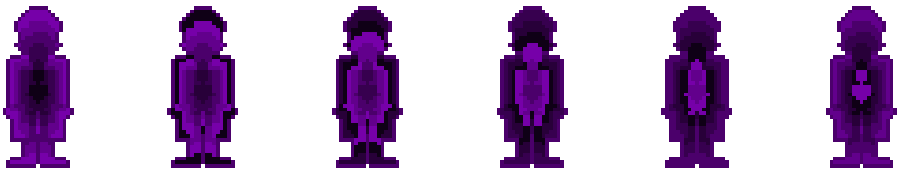


Figure B.8: All of the animation frames for the smaller 'overhead' Dr. Fractal.

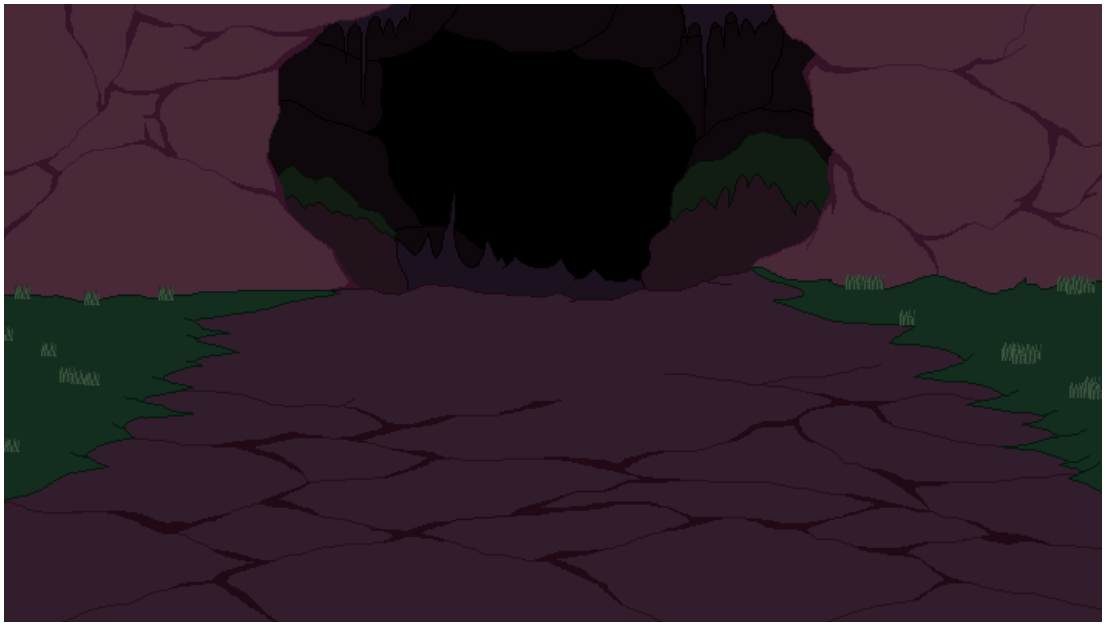


Figure B.9: The 'Lambda-Cave' background for the tutorial battle.

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

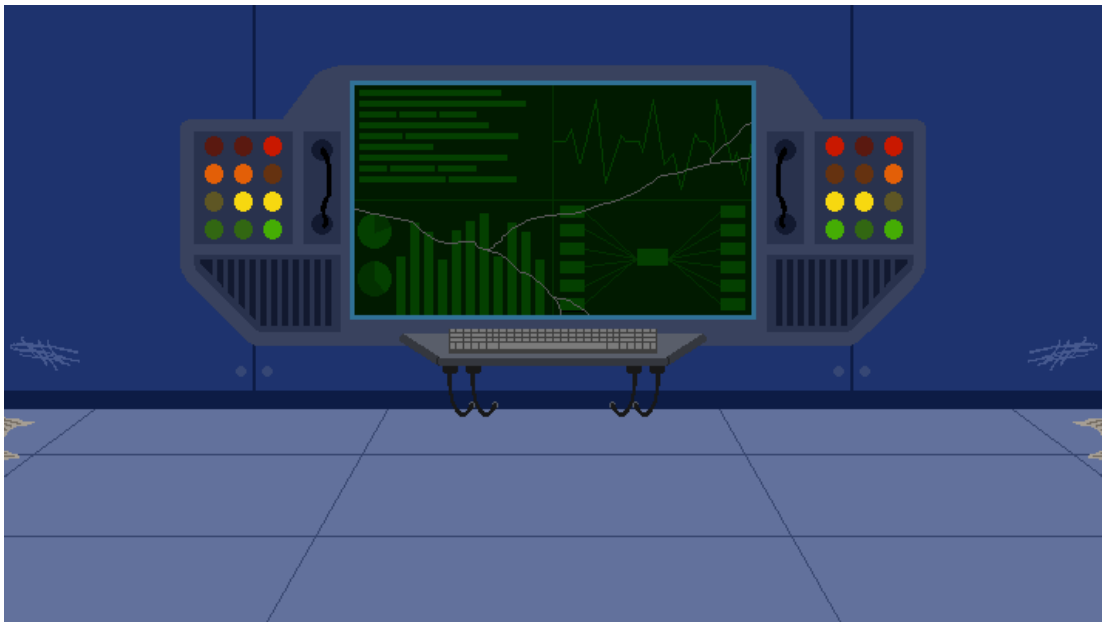


Figure B.11: The background for Dr. Fractal's battle, depicting her ruined laboratory.

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

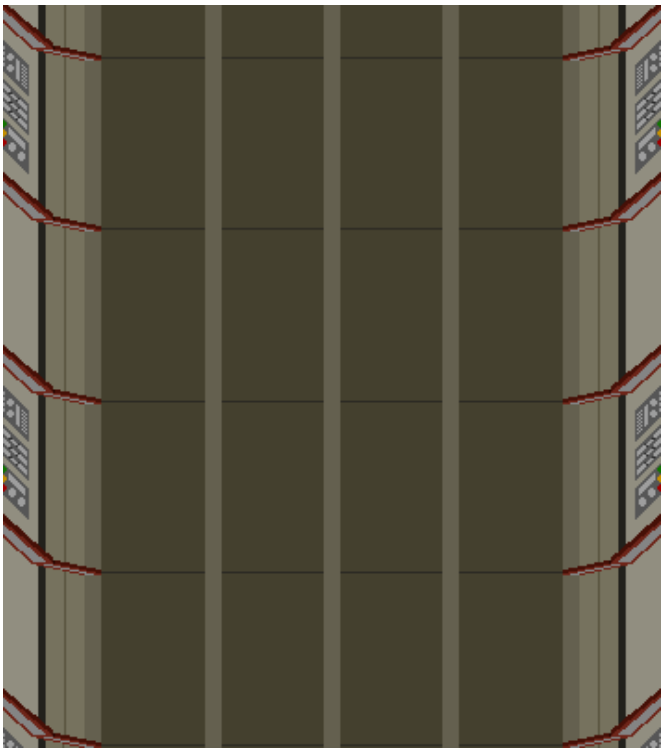


Figure B.13: The ’overhead’ background for Col. Trigger-Finger’s battle.

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

Figure B.15: The plain code editor background, without any UI elements implemented. The UI for player 1 and player 2 in the multiplayer campaign use recolored versions of this sprite.

Figure B.16: The plain help screen background, without any UI elements implemented. The UI for player 1 and player 2 in the multiplayer campaign use recolored versions of this sprite.

Figure B.17: The plain error screen background, without any UI elements implemented. The UI for player 1 and player 2 in the multiplayer campaign use recolored versions of this sprite.

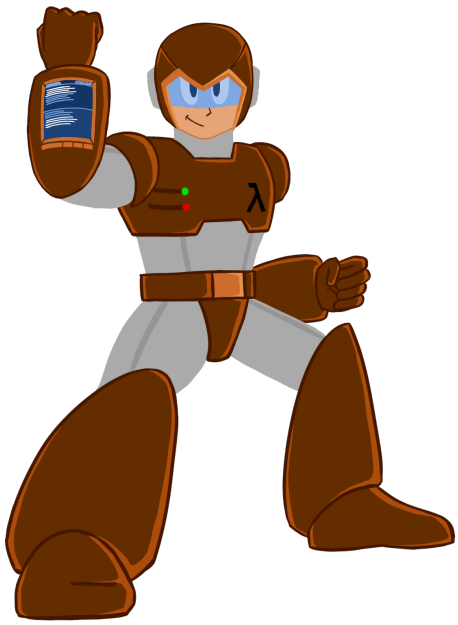


Figure B.18: Character art for Lambda-Man, used on the Project Day poster.

Figure B.19: Character art for Lambda-Man 1, used on the Project Day poster.

Figure B.20: Character art for Lambda-Man 2, with his two canonical right hands, used on the Project Day poster.

Appendix C

Music credits

This section is identical to the one in last year's report.

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.