University of Glasgow | School of Computing Science

Gamifying Concurrency Teaching

Jack Parkinson

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — July 2, 2018

Abstract

Concurrent programming is one of the hardest concepts in computing to learn and to teach. Often computing science students will grasp the basic principle of parallelism and concurrent programs, but will not be able to spend as much time as they need to learn the more complicated concepts, and will struggle to apply these ideas practically. This project aims to investigate the potential for concurrency to be taught in a game format, which will theoretically be enjoyable to engage with in the students' free time and will push them to solve challenges in the same way they would in concurrent programming.

Acknowledgements

I would like to thank Dr John Williamson for his constant support, guidance and patience throughout this project. I would also like to extend a special thanks to Kate, Liam and Marcus, who went so far out of their comfort zones to help me.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

The following chapter will give a brief introduction to the project, outlining its background, key motivations and main aims.

1.1 Motivation

Concurrency is typically taught to undergraduate students of computer science at some point. One field in which it appears with a particularly high frequency is that of operating systems [4], which most students on a typical course will have been taught (at least at an abstract level) at some point, and concurrent programming will have to be comprehended to some degree if operating systems are to be understood. Concurrency also appears in other forms of programming beyond the scope of operating systems, like C and Java threaded programs, but again these courses spend more time on the semantics and execution of parallelism in the given language. Despite the fact that it is often relegated to a fraction of the time required to fully understand it, concurrent programming is a very important concept to grasp to have a greater understanding of the field of computing science as a whole.

Concurrency is often taught in a typical classroom setting with little to no focus on the practical engagement with concurrent problems. The standard approach for years has been to convey concurrency theory in lectures and expect students to be able to apply these theoretical concepts in a real world setting without requiring additional practice. The outcome is, however, that students in general struggle to make connections between the abstract points covered by someone at the front of the class and practical solutions to problems.

M. Carro et al identify in their paper entitled *A model-driven approach to teaching concurrency* that students struggle with visualising how concurrency really works [1]. Their approach to solving these issues involves using formal models; this project is motivated by the lack of visualisation tools to attempt to recreate the running of a concurrent program with a purely visual, interactive interface.

The project is also motivated by an issue faced by many syllabi of computing courses involving concurrency: understanding of concurrent programming concepts is a requirement and is taught to some degree, but it is not the underlying core of the syllabus and therefore not enough time is spent on the practical application of the concurrency theories learned in class. With the possible exception of any course directly focusing on concurrent programming, courses involving concurrency do not go into enough depth on the matter or promote engagement with practical problems [4].

1.2 Background

When beginning to investigate concurrency teaching, it is important to make a distinction between parallelism and concurrency. Parallelism refers to the idea of running multiple processes at the same time. Concurrency, in the context of parallelism, refers to structuring parallel processes to ensure that they all run appropriately and without any issues, which in a practical context would manifest as collisions or deadlocks. Rob Pike breaks down the relationship as follows:

> "If you think about it in a sort of general, hand-wavy way, concurrency is about *dealing* with a lot of things at once, and parallelism is about *doing* a lot of things at once." [12]

But concurrent programming is not an easy aspect of computing to learn nor to teach - it is struggled with by students and teaching staff alike [1]. Not only is the content more complicated and further afield than other areas of the standard sequential nature of computing, but to fully grasp its use usually takes more time and effort spent engaging with realistic, practical tasks beyond theoretical discussion of conceptual problems. Engaging with concurrency problems is generally only available to students in abstract, theoretical formats - such as the dining philosophers problem - and rarely is much time spent engaging with actual, working solutions [4].

This project has been heavily influenced by *The Little Book of Semaphores* [4], a free book by Allen B. Downey, which aims to present synchronisation solutions for common concurrency problems. The book promotes active engagement by encouraging users to write practical solutions to coding problems to instill a level of understating in terms of what would be required if they faced such a problem in a coding environment.

1.3 Aims

This project aims to teach concurrent programming concepts to the user in a game format. Specifically, the game needs to be a platform for students to engage with practical problems in a format that is different from lectures and labs, and is entertaining and enjoyable enough to be used in their free time rather than as a class exercise.

The project also aims to be a useful teaching platform for concurrency - that is, the concepts relayed must be firmly grounded in concurrent teaching theory and encourage the users to approach problems with these concepts in mind. The problems must also be practical in nature, representing real issues faced in concurrent programming with solutions that are likewise dependent on a practical approach.

Finally, this project also aims to evaluate how effective the developed platform is at teaching concurrency to users. It will have to record the users' understanding of concurrent programming prior to playing the game and investigate any changes to their understanding afterwards, to establish whether or not the game has been successful and to what degree.

1.4 Document Outline

The remainder of this document is structured as follows:

- Chapter 2 details the requirements identified for this project
- Chapter 3 explains the design stage of the project

- Chapter 4 explains the implementation stage of the project
- Chapter 5 outlines the evaluation of the final product
- Chapter 6 concludes the findings of this report

Chapter 2

Requirements

This chapter provides a thorough breakdown of the requirements, both functional and non functional, as well as a detailed problem analysis.

2.1 Problem Analysis

As previously mentioned, the project was heavily influenced by *The Little Book of Semaphores* [4], which aims to address particular issues faced by students undergoing concurrency teaching. The author identified concurrency as an area which was more or less understood by students, but not to the point that they could effectively reproduce solutions to standard problems. He notes that this could be due to the relatively short amount of time allocated to teaching concurrency - grappling with even the fundamentals of concurrency takes more time and practical application to instill understanding than most courses are prepared to allocate.

It was decided that a game would be an appropriate platform to attempt to solve these problems. Games can be played out-of-hours in the students' own time, and if they are engaging enough the student will be learning but won't feel like they are in a class. It shouldn't feel like a lesson, but entertainment, and if the student learns anything then it will be in addition to whatever time has been spent on the subject in class.

Additionally, a game that requires user interaction could, given appropriate problem modelling, act as a platform for users to gain practical experience rather than just reading from a textbook or listening to the lecturer speak. By engaging with the user and requiring them to think about the approach they are going to take to solve a problem which is based on a genuine issue faced in concurrent programming, they can be equipped with tools which can be used practically in the real world.

This does, however, rely heavily on the problems being an accurate representation of real world problems and their solutions being modelled on proper approaches. This was something that had to be considered at each stage of the game's development.

2.2 Functional Requirements

While the motivations and aims of the project were clear, the specific requirements of the project depended primarily on decisions made after examining existing systems and deciding on a more definite structure. There

were three specific areas that were felt completely necessary for the project, each derived from the project name: Gamifying Concurrency Teaching.

Firstly, gamifying: the project needed to produce a game, in the sense that it was enjoyable and fun to play. The Oxford Dictionary defines a game as:

> "An activity that one engages in for amusement or fun." [17]

It was felt that for the project to be classified as a 'game', these requirements had to be met. Additionally, it was felt that the game needed to provide not only a 'fun' experience, but one that was easy to get into quickly - it shouldn't require extensive set up, waiting times or resources on the users part. Ideally, they should be able to open it up and begin playing immediately.

The next word: concurrency. The game had to have a strong connection with concepts of concurrent programming. This almost goes without saying, but the connection was important to make when defining the project's requirements.

And finally, teaching. This requirement was more difficult to nail down because it is difficult to gauge exactly when someone has learned something, particularly when that something is a concept which should be applied practically in real world problems. Nevertheless, if the project had some noticeable difference on the users' understanding it would have to record it, so the final requirement was to establish a form of quantifiable measure of the users' learning.

Another requirement emerged in the examination of the 'game' and 'teaching' aspects of the specification. Typically, games provide functionality to be returned to after playing for a time, especially if they are long and challenging, to allow users to disconnect, unwind and continue playing the game later. The game shouldn't be a chore to be worked through, but something that students could use flexibly in their free time of their own volition. As such, a save state was deemed necessary, which would also benefit the evaluation of the teaching if the users' data could be recorded. Ideally it would then be possible to issue the game to a number of users and review their state data once they were all done, to examine how much of the game they played and how well they performed.

With these issues to be addressed in mind, the following points were considered the project's main functional requirements:

- Must be an enjoyable and entertaining experience
- Must require minimal setup and have low waiting times
- Must be directly related to aspects of concurrent programming
- Must have the capacity to record the users' engagement to evaluate their learning
- Must allow session state saving

2.3 Non Functional Requirements

Once the core functional requirements were established, some non functional requirements could be extracted. These reflected more on the ways in which the system would be perceived by users and less on its actual behaviour. The following non functional requirements were identified:

- The game should be easily accessible to users in their own homes - this is a comfortable environment which is most likely to be where the users would be spending their free time.
- The game instances should be accessible by multiple users at the same time - when the service is distributed to users, they should all be able to access it whenever they want, even if others are playing it.
- The game should be robust - a crash while playing would be extremely frustrating and could give false user data for evaluation.
- The game must be relatively secure - users shouldn't be able to access each others' accounts, as this will give incorrect data for evaluation.
- Users' game data should be easy to extract - this will streamline the evaluation process and reduce the risk of error in examining a database or other local storage.

Chapter 3

Design

This chapter details the design stage of the project. It includes the impact of external influences, initial exploration of potential design models and the evolution of the final design of the game.

3.1 External Influences

Upon beginning the design stage, it was deemed prudent to collate a collection of resources which could influence the game's design. These included, in broad categories:

- The Little Book of Semaphores
- Existing games to teach concurrency
- Existing puzzle games

3.1.1 The Little Book of Semaphores

The Little Book of Semaphores influenced the design of the game a great deal [4]. The book's approach to teaching concurrency involves introducing simple primitives one at a time and providing multiple puzzles for each one before moving onto the next. Often, when a structure was introduced, it would require an understanding of or even be directly connected to a previously introduced primitive. Each chapter built on the previously learned concepts and had a very strong focus on the practical challenges involved, often encouraging the reader to write out sections of code before moving on.

The project drew several elements from these features. It was decided that it would be useful to introduce very simple primitives initially and add more as the game progressed, constantly requiring the player to recall how the basics worked and apply them to the next stage. It was also seen as useful to introduce a significant amount of repetition in different problem domains to ensure that the users were clearly understanding the process and that it was being properly instilled.

Some consideration was put into having the users either writing small sections of code or rearranging code blocks for them to formulate a solution, but it was also considered potentially detrimental to include large amounts of code in the game, as this might be seen as being too much like a classroom tool rather than a game. Also, the project aimed to have a strong focus on visualising concurrency problems, which was identified as something students struggle with [1], and this is not as easy when code is being used.

3.1.2 Concurrency Teaching Games

Games which aim to teach concurrency are not common and have their flaws, and only one was really discovered in this project's research, in addition to a paper investigating the potential for a game.

The Deadlock Empire

The game discovered online which aimed to teach concurrency was called The Deadlock Empire [18]. The game had an attempt at a fantasy motif with references to wizards, castles and dragons, however there was little in the game itself which actually reflected this at all. The user interaction involved placing them in the role of the scheduler and stepping threads through processes to reach critical areas. There were large amounts of code, but the user was not actually required to write any of it, just understand its function.

Overall, it was clear what The Deadlock Empire game was aiming to achieve, but there were some major flaws in its execution and motivation. Firstly, it attempted to integrate an immersive theme, but it was entirely at odds with the actual game's content. The code had little connection with anything related to the theme, so the fantasy element felt 'bolted-on' and irrelevant. Secondly, it did not 'feel' like a game. The code was presented in a very simple format with minimal interaction - it was akin to studying code on a screen in a lab rather than being an entertaining experience.

An Educational Game for Teaching and Learning Concurrency - Paper

Naoki Akimoto and Jingde Cheng of Saitama University wrote a paper on a game they had designed to teach concurrency [10]. Their main goals were to present a fun, learning environment to potential beginners, which are partially include the requirements of this project. The game involved multiple robots which had to reach tokens to pick them up, navigate around obstacles and place them on a particular tile. The user interaction involved placing instructions on tiles in the robots' paths (such as 'turn right' or 'drop cargo') which the robots would reach and react to.

This game was an interesting approach to concurrent teaching. One of the things taken from the paper as inspiration was the concept of not having direct control over the processes, but manipulating them with external factors. It was felt that this was a fairly accurate representation of the behaviour of processes and threads in concurrent programming, so was something that was considered to be useful in the game.

3.1.3 Existing Puzzle Games

It was decided that for a game to be successful in the sense that it was entertaining and enjoyable, it would be a good idea to examine some existing puzzle games that had some degree of interaction similar to what had already been thought about.

The Incredible Machine

The Incredible Machine [19] was a game that was looked at because of its vast environment manipulating capabilities. The goal is to set up the environment and have everything in place before running a process, and ideally having the process run correctly. While the degree of control over the environment was very impressive, it was deemed quite complicated for the desired level of complexity.

Flockers and Lemmings

Flockers [7] and Lemmings [9] are fairly similar games, in that they both had a set of game characters which would walk in a straight line until they hit an obstacle or reached the end-point, the goal being to place navigation aids along the way to guide them in the correct direction. The environment manipulation aspect was a little more rigid than The Incredible Machine, which was more in sync with the project goals, but both games also included the ability manipulate the characters directly. This was something that the concurrency game would not have: threads should be capable of being manipulated, but not via direct interaction by the user.

Trainyard

Trainyard [21] was a particularly interesting game that was investigated. The setup involves sources and end-points, and the user has to connect the sources with the endpoints and engage in various problem solving elements to overcome certain obstacles. Again, the design involved coming up with a solution before running the process, and while timing itself wasn't a core factor in solving the problems, the rate at which processes completed certain actions had to be taken into account to ensure they were in the right place at the right time. Elements taken from this design included a novel way to have threads running on a set course without any direct manipulation and relatively basic user interaction with the environment - a simple touch and drag system for connecting up nodes.

Figure 3.1: Screen grabs of all existing puzzle games investigated, with important features highlighted

3.2 Initial Exploration into Problem Domain Representations

With some essential game features established, the next stage was to begin thinking about how to represent the problem in a game domain to the player. Several ideas were pooled and evaluated:

- **Pipes and running liquids.** This idea involved representing concurrent processes as channels of coloured liquids which would flow in parallel, and the player would have to place a series of pipes in their path to lead them to the right endpoint. The liquids were intended to be representative of processes which would run in sync, sometimes having to be stopped to wait for each other, sometimes running into the same vessel. The user interaction would involve placing appropriate pipes in the path of the liquids to force them into the right channels to reach the areas that are required.
- **Scientists in a lab.** Another model involved a number of scientists which would collect flasks of chemicals and prepare them at various workstations. Some workstations would require multiple scientists and chemicals to be used. In this scenario, the scientists represent threads, the chemicals represent resources that they need and the workstations represent the lock structures which enforce concurrent operation. This idea had the benefits of being able to express several different kinds of locks in a variety of interesting formats. A potential risk with this format is failing to make the link between the in-game elements and real concurrency concepts clear, which could cause the game to be ineffective in teaching concurrency concepts.
- **Zombies or robots in a factory.** This idea was similar to the scientists scenario. Zombies or robots were chosen to represent threads to convey the inherent unpredictability in their behaviour. The workers would operate on a production line, needing to work together to bring items to the required locations to be able to have the processes proceed. Some workers would have to wait for another to arrive at a point to be able to proceed, which would factor in some of the concurrency concepts for users to learn. It faced the same problem as the scientist scenario in that there could be a disconnect between the represented elements and their concurrency counterparts. It was also unclear how to integrate user interaction in such a way that they were in direct control of the concurrency tools - it was not decided exactly how these would be enforced or how the user would execute them.

None of these ideas were perceived as being the definitive one that would be a suitable representation of the game, so some further evaluation of all the ideas was required.

3.3 Final Game Design

Eventually, a design which took elements from all the previous ideas was settled upon. Rather than having a particular theme to the game, it was decided that it would be better to abstract out the idea and represent it as simple shapes. The reason for this was, although there might be some immersion or player attraction lost, The Deadlock Empire game attempted to generate attraction with a thematic environment and the difference between the game's theme and the content was especially jarring. In addition to this, with abstract shapes there would be no need for additional mapping of themed elements onto their concurrent programming counterparts. Take for example the connection of zombies with threads or workstations with resources - the shapes could simply be called what they would be called in concurrent programming, meaning that there was less potential for a disconnect of the player's understanding of the presented scenario and the real world counterpart.

The threads would be represented by circles or orbs, which would move around an area until they reached their destination. It was thought that this was a good representation of the 'pseudo-random' nature of threads - while not inherently unreliable, threads perform actions at different rates and cannot be guaranteed to be exactly

where they're needed at a given time. The purpose of the game was to demonstrate to users that they needed to perform certain actions and checks to ensure that the threads were arriving where they needed to be, which was considered as being a good representation of how concurrent programming is carried out.

The specifics of the exact nature of the threads movement were also considered. A few ideas were brought to the table: random walking, flocking and a physics based system. Random walking was an idea considered from Daniel Shiffman's *The Nature of Code* [15], which involved a thread picking a random angle and moving in that direction for a set (or sometimes also random, depending on the degree of freedom specified) distance or time, causing the thread to travel around the screen in a meandering manner. The flocking movement pattern was also drawn from *The Nature of Code*, and involved threads encountering other nearby threads and manipulating their movement, forming 'flocks'. The physics based system would involve the threads having set initial travel paths which would only change depending upon the physical entities they encountered, such as bouncing off walls or other static objects.

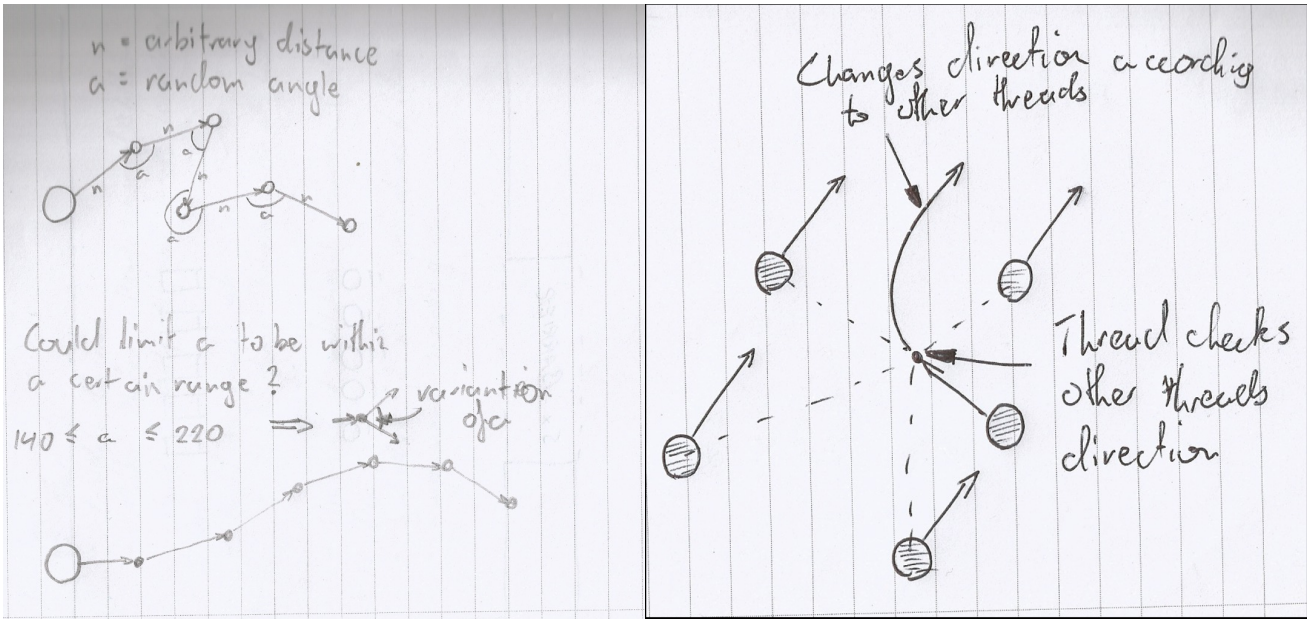

Figure 3.2: Two hand-drawn diagrams indicating the nature of the random walker model and the flocking model respectively.

The random walker system was deemed as not being a particularly accurate representation of real thread behaviour: threads have set instructions which they are to perform, and introducing too much of a random element could cause users to misunderstand their nature. While the game should reflect that threads cannot be relied upon to behave exactly as we expect, they do not behave completely of their own volition. The flocking model was also deemed as not being a good representation of thread behaviour, as it suggests that threads are not independent of one another and will manipulate each others' operation. Eventually, the physics model was decided as the system that the threads would follow, because even though threads cannot be directly manipulated, their behaviour can be predicted to some degree and the user can act appropriately. Also, many game systems and frameworks have a built in physics engine, which could be used with minimal changes to model the required behaviour.

Resources would be represented in game by squares which threads would 'slot' into when they encountered them. If two threads or an unauthorised thread tried to access an unguarded resource, they would be ejected. To communicate the 'bad' nature of collisions like this, there would have to be some negative visualisation whenever a collision occurred. At this stage of design, this was thought to be a red flash, of either the resource encountering a collision or the border of the screen. Once a thread was done with a resource, it would be ejected and considered completed, so no other threads would need to access it. Some resources would be linked and would require that each one had a thread ready to process it before being processed.

Figure 3.3: Two hand-drawn diagrams indicating some initial design plans for the game

Locks would be the aspect of the game that the user would interact with: they would be shapes which could be dragged onto resources to limit access to certain threads. Different locks would cause different behaviour, so the player would have to identify the correct one to use in different situations. It was felt that as there was a direct correlation between the game elements and the real world elements (i.e. locks in the game simply represent locks in real life) the chance of misunderstandings occurring was reduced. The required interaction was also a good representation of what a concurrent programmer would have to do to solve concurrency problems: enforcing rules on thread access, typically using standard, boiler-plate code.

3.4 Level Design

The game would be comprised of several levels, each level incorporating the elements previously discussed. Incremental levels will introduce new concurrent concepts gradually, as in *The Little Book of Semaphores*, and become progressively more challenging. There were several concepts that were specifically identified that should be incorporated into levels - each of the following concepts would have one or two levels detailing how they worked before introducing the next, with several levels in between using the previously learned concepts:

- Rendezvous structures
- Barriers
- Concurrent access to multiple resources
- Lock structures used to manipulate thread behaviour
- Multiple multi-threaded processes running concurrently

In addition to these concepts, other issues faced in concurrent programming would be potential issues in many of the levels, such as collisions and deadlocks. Later levels should begin to mirror some of the puzzles presented at the end of *The Little Book of Semaphores*, which combine many of the elements discovered in earlier stages of the game with the more complicated elements. It was decided that these fairly simple concepts should be focused on initially, which would most likely be useful to users with little to no experience in concurrent programming. To clarify the process of the level design, find three examples below.

3.4.1 Example 1

This example introduces the rendezvous structure to the user. The user is provided with three different lock structures, so they will have to choose the one which best suits the situation - as the resource can only be accessed by two threads at once, the '2 lock', which is described to the user as a rendezvous, is the best choice.

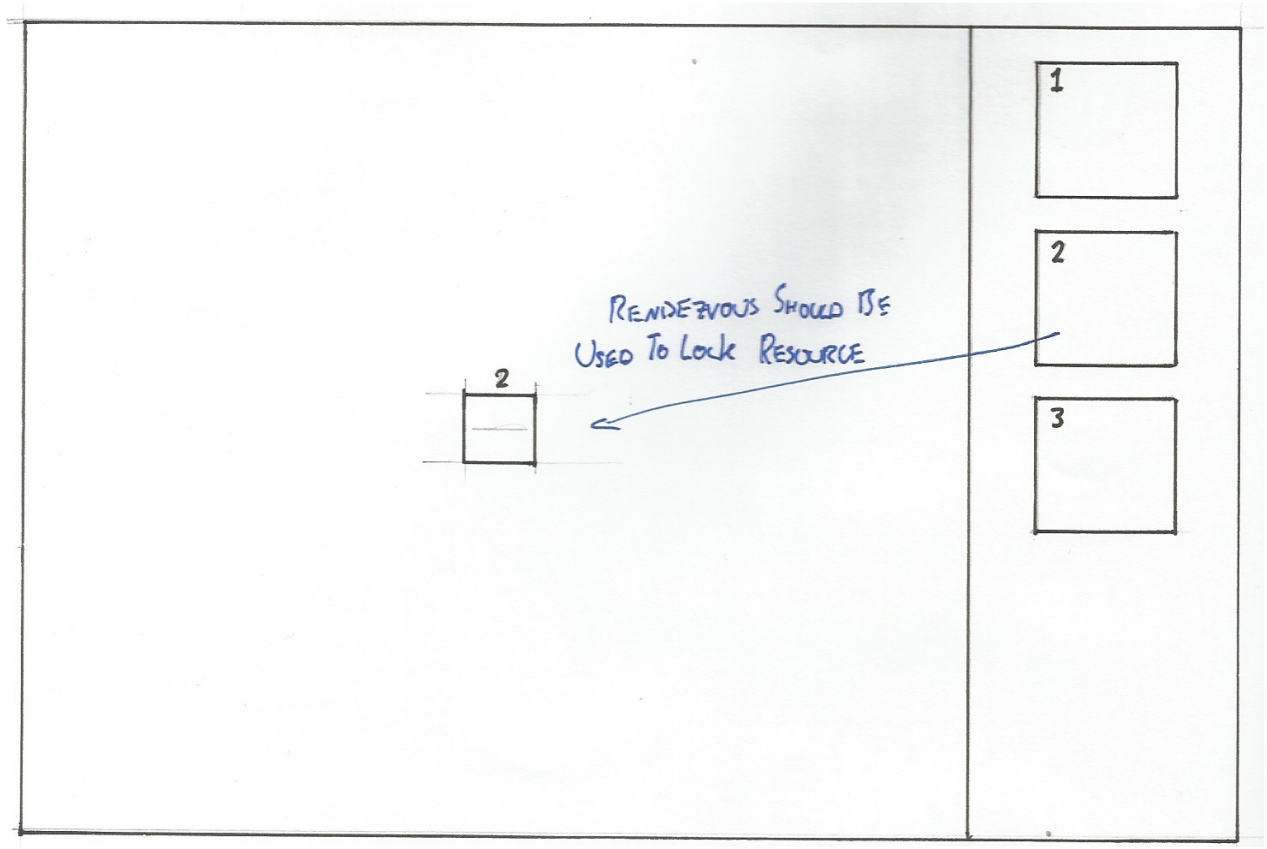

Figure 3.4: Initial concept for a level design consisting of the application of a rendezvous lock structure

This example is very simple, and would be one of the earlier levels of the game. Its purpose is not specifically to challenge the user, but rather to encourage them to think about a fairly simple problem and approach it with the basic tools provided. If this level is successfully completed, the user should now be familiar with the rendezvous structure and should be able to use it appropriately in future levels.

3.4.2 Example 2

This example represents a situation in which concurrent access is required by the process on multiple resources. This is represented in-game by the dashed line connecting the two resources. The user will have to examine both structures and pick an appropriate lock for each and ensure that they are locked simultaneously.

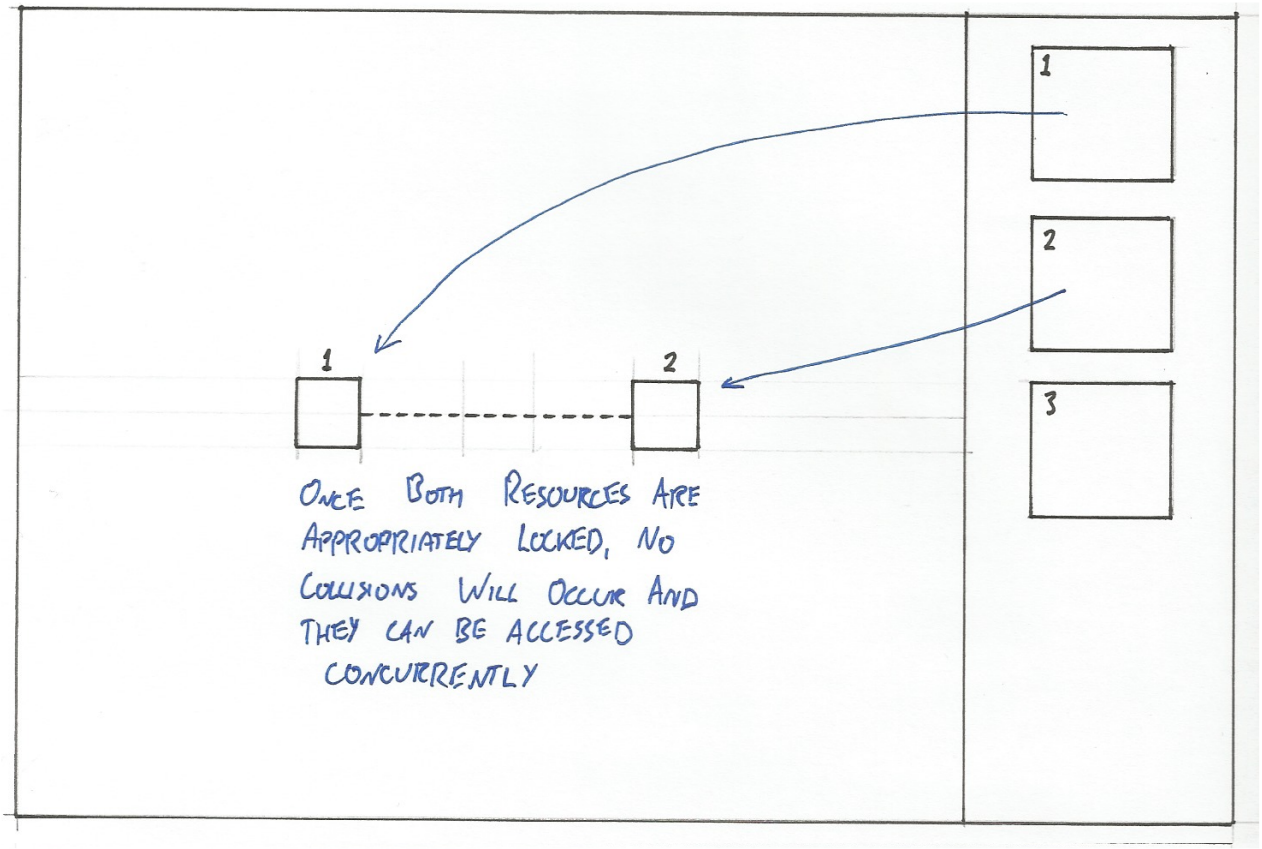

Figure 3.5: Initial concept for a level design consisting of two resources for which the process needs concurrent access

Again, the example is fairly trivial, but its purpose is not to try and catch out the user. It is supposed to introduce a new concept (concurrent access on multiple resources) which also requires knowledge of a previously encountered concept (the rendezvous structure.)

3.4.3 Example 3

This example is a little more challenging. By this point, we can assume the user has been taught how to use locks which manipulate thread behaviour (directional locks, which will fire threads in a set direction) and are aware that they can mirror other, already established lock structures (like rendezvous and barriers.) This example also has restricted areas, which threads will not spawn in and have to be specifically 'fired' into.

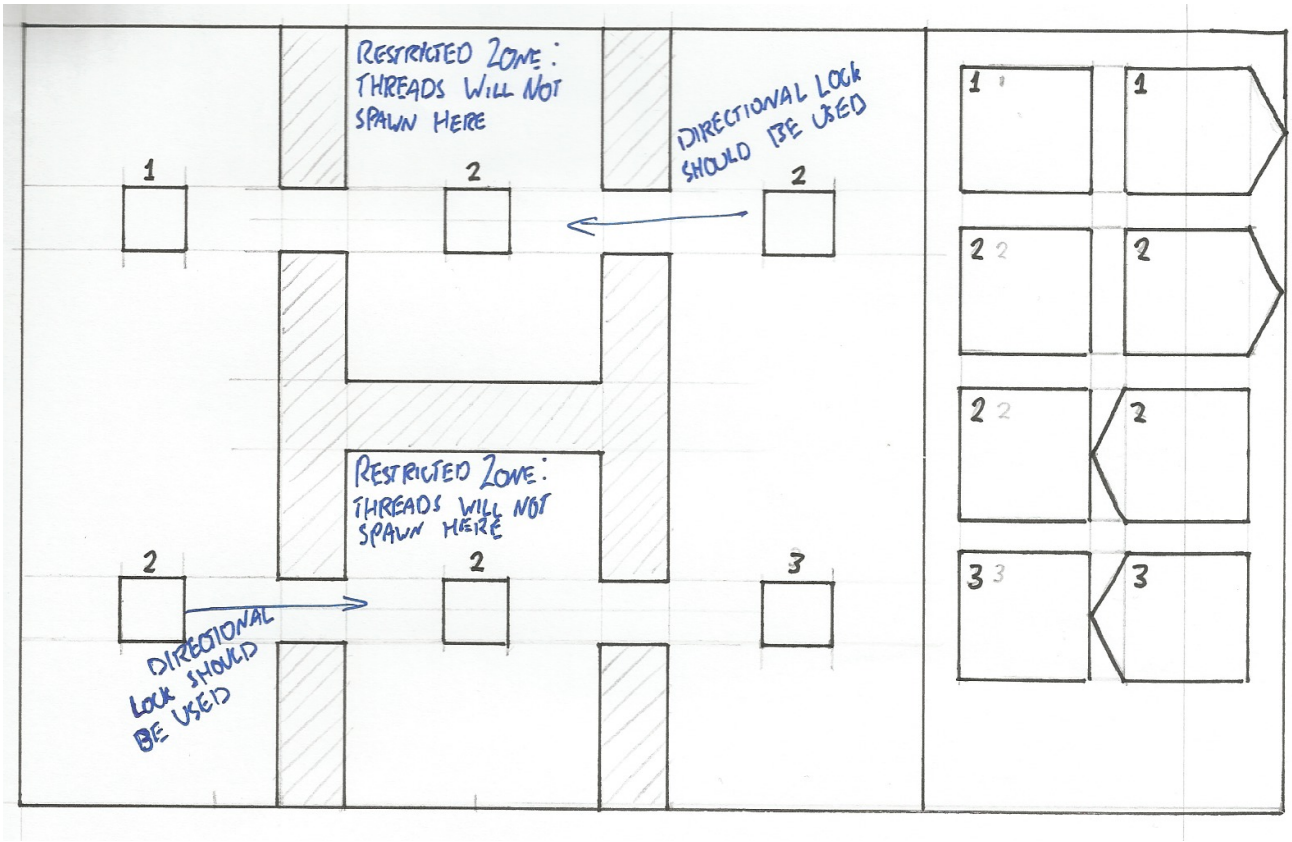

Figure 3.6: Initial concept for a more complex level design requiring the use of directional locks

The level requires some pre-planning. Not every lock, even if it appears to restrict access to a resource appropriately, will give the required outcome in the grand scheme of the level. For example, if the top right resource is locked with a non-directional rendezvous, the process will not be able to get the required number of threads into the top central restricted zone and therefore will not be able to access the resource. The purpose of levels like this are to encourage the users to think ahead when beginning levels, in case the obvious solution will not be appropriate for the end result.

3.5 Game Platform and Framework

The next important design decision that needed to be made was what platform to develop the game on. There were a huge range of options: mobile applications (such as an Android or IOS app), web applications (which would host a game framework), desktop applications (primarily executable programs, built in Unreal or Unity for example) and others.

3.5.1 Platform Requirements

To identify the best platform, a set of requirements of the platform were picked out in the game design process which it would have to fulfill. They were based on the requirements laid out at the start of the project.

- The framework must allow for a flexible visual display
- The framework must be as easily accessible as possible, to allow for easy evaluation with users and streamlined access

- The framework must for allow some form of tracking, to log users' progress

Desktop applications would provide excellent visual feedback and interaction tools, but they are not as accessible to some users (often requiring a standalone installation and potentially external requirements like DirectX Runtime.) Most tools used for building such desktop applications take a significant amount of time and effort to learn even the basics, and the engines could be considered unnecessarily complex for the relatively simple functionality and graphical feedback that the game aims to incorporate. Also, there could be potential problems with logging user data and state saving, which would require either access to the local machine or an online connection.

Mobile apps are a very popular platform for games, with many people these days spending time on mobile games. Also, most mid-range smartphones would also most likely be powerful enough to serve the fairly simple functionality and graphics of the proposed idea, and tracking user progress could be done online. However, they are not particularly accessible because installing apps from external sources requires additional user permissions, which is often seen as a security risk and is discouraged by mobile OS providers. Also, a decision would have to be made about which mobile operating system to develop for, alienating significant groups of potential users.

Selected Platform: Web Application (Flask)

It was felt that the platform that best fulfilled these requirements was a web application. There is a wide range of packages and libraries available to web developers who are specifically building games, providing a wealth of options for the visual display. Webapps are also very accessible, especially when deployed on a webserver, making them available to anyone with a link and an internet connection. Additionally, with most web frameworks having functionality to connect to an online database, user data can easily be saved and reviewed whenever required.

Specifically, Flask [5] was selected as the web framework of choice. It is lightweight, but also functional enough to handle database management and user login functionality [6]. Also, as most web framework game libraries are based on HTML5 and JavaScript, the core game functionality required by the web app was just to render the appropriate pages and sever the required files.

3.5.2 Game Frameworks

Once it had been decided that a web application was the most appropriate platform to host the game, an appropriate game framework in which to build the game had to be selected. I was particularly looking for a JavaScript framework, as I felt that I had had enough JavaScript experience to engage with the project professionally and efficiently without the need for extensive self-teaching.

Among the frameworks investigated were CreateJS [2], PlayCanvas [13], Phaser [11] and Three.js [20]. Both Three.js and PlayCanvas had excellent visual display and extensive tools for 3D rendering, but each had their downsides: PlayCanvas only provide hosting on their own service for free, with paid options allowing the developer to export code to their own platform, which would have an effect on the way the game handled user data and how we planned to present it to them. Three.js (despite having several games hosted on the website as examples) seemed like it was more of a useful tool for 3D effect rendering than game design. CreateJS also had very good visual capabilities, but was compiled of four separate libraries (EaselJS, TweenJS, SoundJS and PreloadJS) which was not a problem in and of itself, but documentation for each library was written separately and caused concern about linking up all the elements to construct a game.

Phaser was the best contender as it featured extensive and highly comprehensive documentation, dozens of examples (displaying code used and a live visualisation of its functionality), several approved from-scratch

tutorials and an active community of developers [3]. It was felt that, although Phaser's demos and examples seemed less technically impressive than some of the other frameworks investigated (particularly graphically), it would provide a suitable platform for the purposes of this project. Additionally, Phaser was one of the listed frameworks which supported a selection of physics engines, each geared towards a different style of interaction, which would be very useful in implementing the threads' physics based movement.

3.5.3 Application Framework and Interface Design

At this point, some thought was put into what the overall interface that the user would see at any given point while interacting with the game would look like. The concept was kept simple and clean to streamline the user's experience. The index page (home page) would display all available levels, with star scores visible for completed levels. The actual game page would consist of primarily a div containing the JavaScript game, some details about the level (the level number and a hint) as well as restart and home buttons.

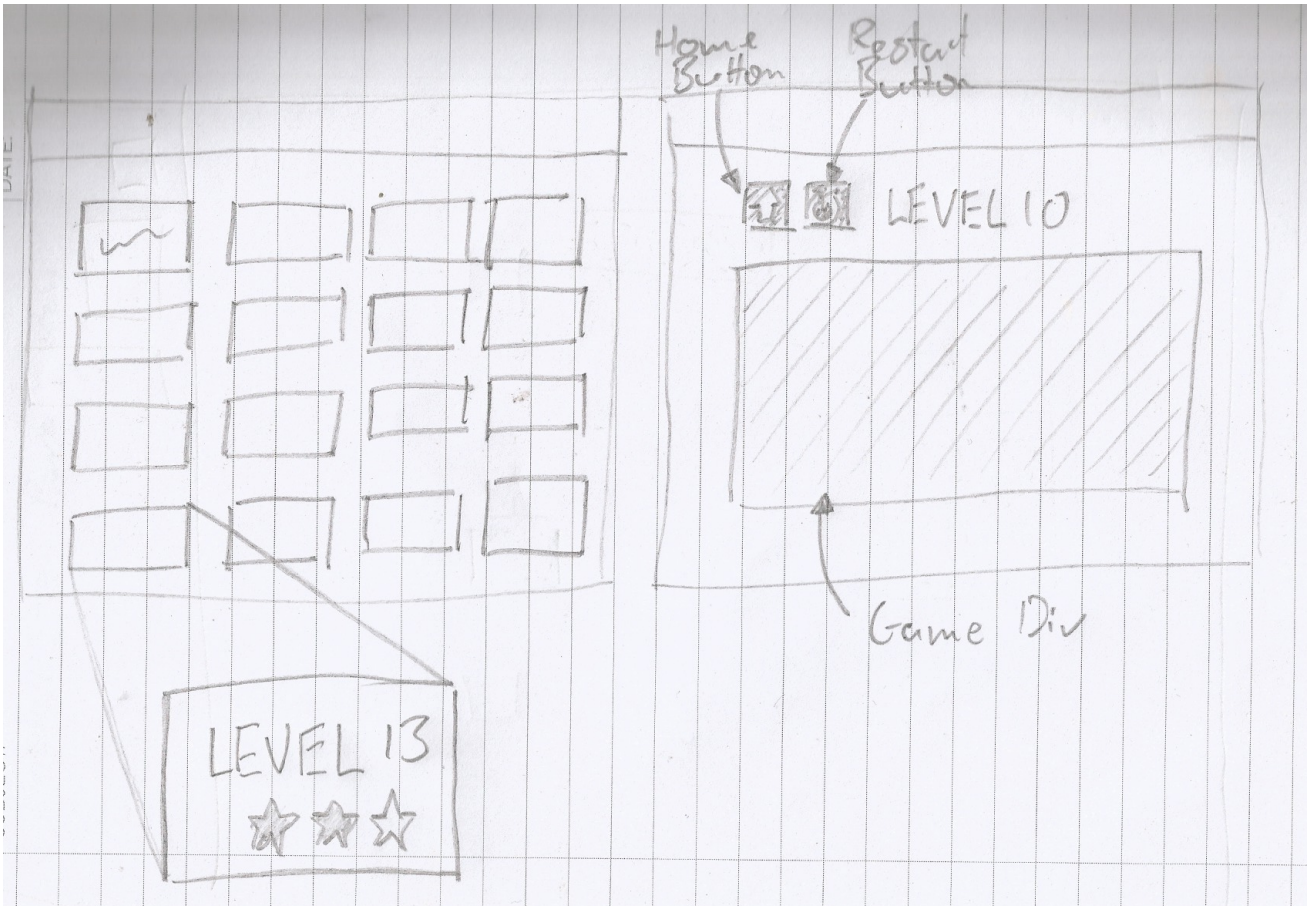

Figure 3.7: Initial, rough designs of the application page layout

Chapter 4

Implementation

The following chapter provides a step-by-step process of the implementation of the project, working from the initial core set up to the changes and deviations made in later iterations.

4.1 Architecture

With a good idea what elements needed to be presented to the user and what they were going to look like, the next step was to decide on the most appropriate structure of the application being served. Based on decisions made in the design stage of the project, it was possible to construct an architecture diagram with a clear representation of the fundamental structure.

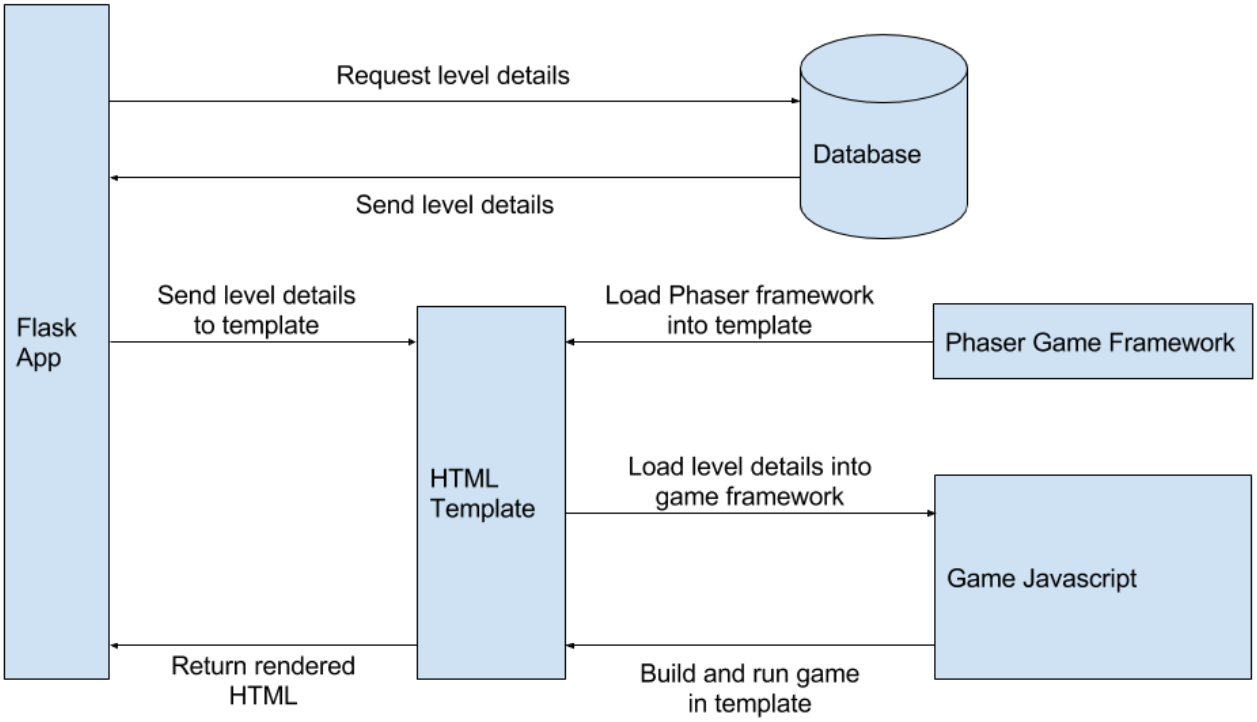

Figure 4.1: Architecture diagram

The Flask framework was to act as an interface to allow the user to login and select levels, and would also handle the database connection and page rendering. The database used was hosted on WebFaction [22] for easy access, and a Python database ORM, SQLAlchemy [16], was used to handle the database connection, as it provided a very simple yet robust format in which to interact with the SQL and provided excellent Flask integration and implementation documentation. Each time a level was loaded, the Flask app would send a request to the database for that specific level's data, then load it into the game HTML template and the JavaScript game function. The JavaScript function required the core Phaser JavaScript in order to run, which was made available by being loaded into the template. With the Phaser JavaScript, the game JavaScript and the level data, the HTML was able to render a level and display it to the user.

4.2 Phaser Game Structure

Phaser games are recommended to be structured with a main state requiring three core functions: the `preload`, `create` and `update` functions. The `preload` function runs first, and loads all the required assets into the game's main state. The `create` function uses the loaded assets to construct the game's sprites and other structures. This function is also generally used to initiate the start of the game. The `update` function runs throughout the game, constantly updating states as required. In addition to the three core functions, other functions can be added to the main state to be called within the three core functions.

4.3 Core Game Framework

The following are important features of the game that were absolutely necessary. They involve the base game layout and key elements of the game's runtime and interaction methods. They were the most important features to have setup because the rest of the game's elements would be built upon them.

4.3.1 JSON Level Details

Each level had a JSON object stored in the database with which they would be associated, which was loaded into the game space. The JSON object carried important information about the level's construction, including:

- The number of threads required
- The number and nature of the resources required
- The number and nature of the lock structures required
- A grid, representing the level layout

The grid in particular was very important: it was constructed from a two dimensional array, with each cell representing a tile of the game environment. The grid was used to dictate the placement of processes and walls, as well as which cells would allow threads to be spawned in them - this was crucial to later stages of the game when restricted areas were introduced.

With the JSON data capable of representing the required details to construct and populate a level, the next stage was to write the code to generate these details as objects in game. The three core objects to be included in the first iteration were:

- Threads
- Resources
- Locks

The construction and initialisation of each of these game elements, and how they were rendered in the game world, is detailed in the remainder of this section.

4.3.2 Threads

The first stage was to implement the threads, which needed to move randomly around the environment. This was achieved by picking at random an available cell (as dictated by the level's grid) and placing a thread in it. As previously discussed, the thread movement model that was decided upon was based around Phaser's built in physics engine. When the level started, the thread would be fired at any angle at random. The threads were designed to have no deceleration and would keep moving until they bounced off a wall or slotted into a resource.

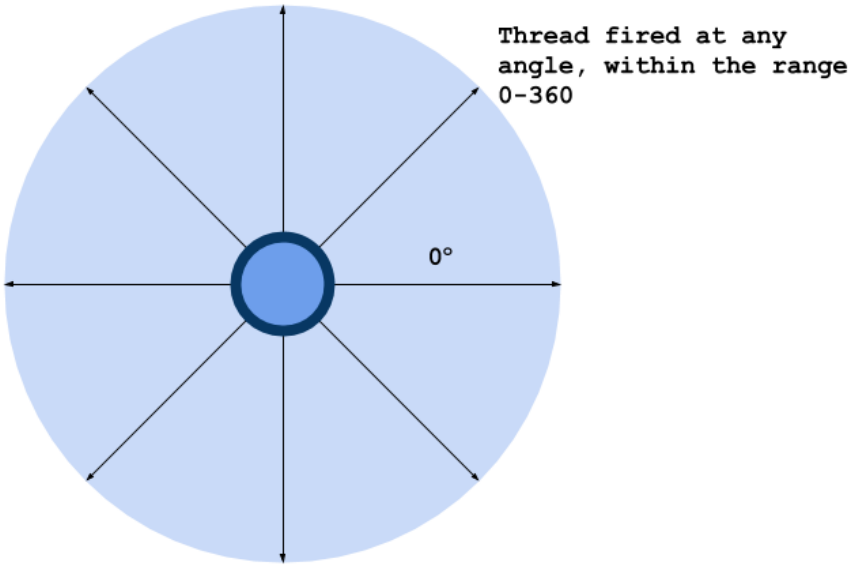

Figure 4.2: Visual representation of a thread, which is fired at any angle

At this stage walls were also implemented to contain the threads within the game environment - it was possible to force the threads to stay within the game bounds via Phaser commands, but by introducing walls as a concept early in the game it was thought that their function would be clear when they became more involved. Also, they served as an efficient barrier between the game environment and the lock tray, which meant that no additional tweaking was required to stop threads entering the tray.

4.3.3 Resources

Next, resources needed to be introduced. This process involved taking a resource tile from the grid and matching it up with the correct resource details, also given in the level JSON. The data for each resource included in the level's JSON included the number of threads required for completion and the time taken.

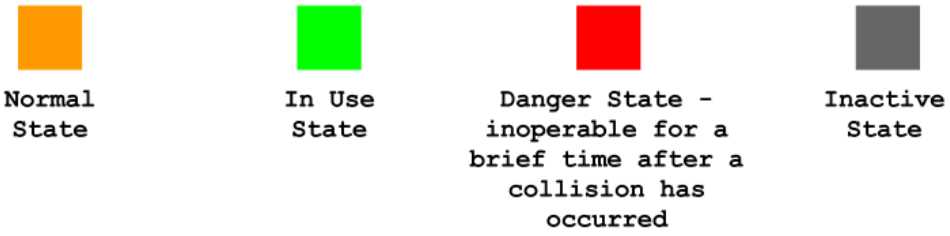

Figure 4.3: Visual representation of resources and the colours corresponding to their various states

When each resource was actually created, a few other attributes were assigned to them for the running of the game. These included:

- `contained`: an initially empty list, in which slotted threads would be stored
- `containedtext`: a text object, which would display the number of contained threads and the number of required threads

- `runTimer`: a boolean, which is initially false but is set to true when the resource contains the correct number of threads
- `resourceTimer`: acts as a timer which ticks down, initially set to be the time required for thread completion
- `timerText`: a text object displaying how much time is left
- `completed`: a boolean, initially false, which will be set to true once the process is done with the resource

When a thread overlaps a resource, its velocity is set to zero and the resource adds it to the list of contained threads. The resource then checks if the number of contained threads is equal to the required number of threads; if so, the boolean `runTimer` is set to be true. Every second, this boolean is checked and all resources with a true `runTimer` state have the `resourceTimer` decremented. If the number of contained threads is now greater than the required number of threads, this means that a collision has occurred and the resource has to eject all the contained threads. This process involves setting the velocity of each thread back to the original value at a random angle, as when starting the game. Additionally, the `resourceTimer` is set to the original value and the contained threads list is set to being empty.

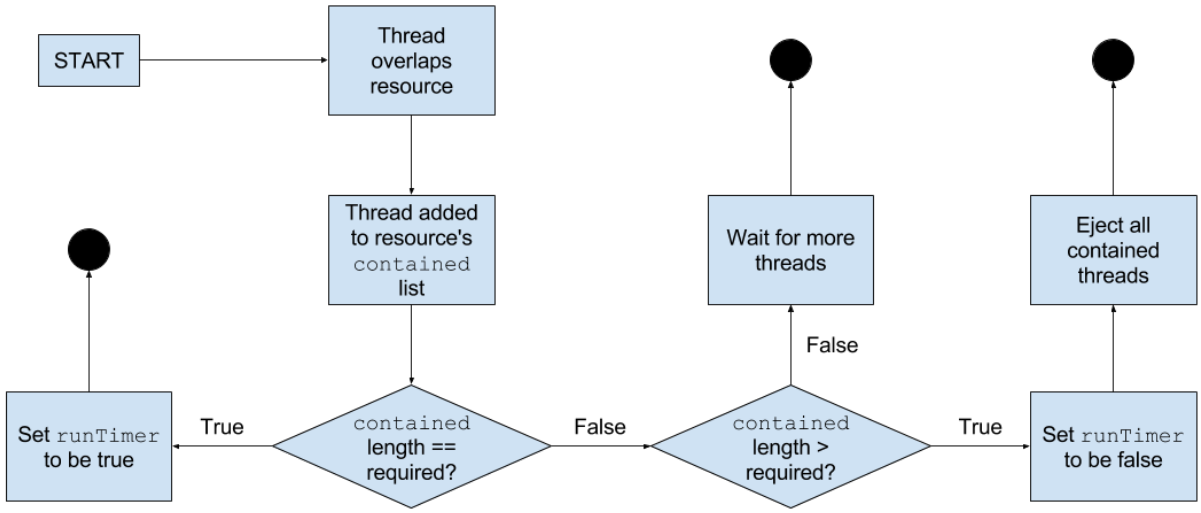

Figure 4.4: Flow chart representing the sequence of events when a thread overlaps a resource

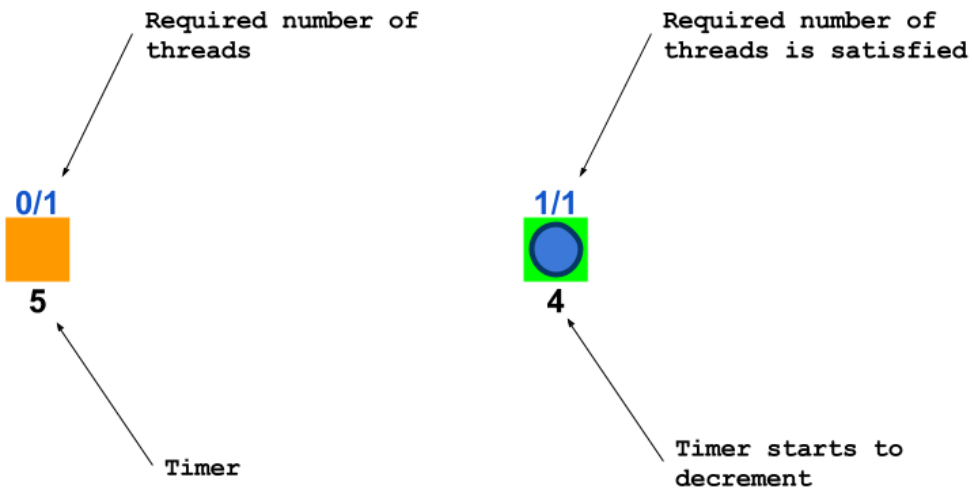

Figure 4.5: Description of the basic interaction of resources and threads

It was then discovered that this method posed a problem: as soon as a thread was ejected, it was still overlapping the resource, and the collision process would begin again. To stop this from happening, and the game being stuck in a permanent loop, collision detection was disabled for the resource sprite for half a second. This gave enough time for the threads to travel far enough from the resource that they were not picked up as a separate overlap before the resource reinstated collision detection and started receiving threads again.

Once a resource had the required number of threads, the timer would start, as described above. Once the timer reaches 0, the threads have completed running and no longer need the resource, so the `completed` boolean is set to true. In this state, the resource will not accept any threads, and ejects the contained threads out at random angles, as with the case of the game starting or a collision occurring. The Phaser `update` function constantly checks all the resources to determine when they have all completed - once they have, the victory state can be run.

4.3.4 Locks

Once basic thread spawning, random thread movement and resources had been implemented, the next step was to include locks. Using Phaser's overlap detection, a function was called when a lock overlapped a resource to change its state. To do this, an additional parameter, `locked`, was required on each resource when they were instantiated. The overlap function set this parameter to `true`.

Phaser does not, however, have inherent 'off overlap' detection, so it wasn't possible to tell immediately when a lock was moved off a resource. A workaround that was eventually found was to set all resources to be unlocked in Phaser's `update` function immediately before checking for the overlap. The `update` function runs quickly enough that the check is made and resources are locked again if they are overlapped with a lock, and the brief unlocking caused no problems.

The way locks worked in the first implementation of the game was very simple: if a resource was locked, it would accept only as many threads as were required and cause any other threads to bounce off - the lock itself did not dictate how many threads were allowed access. In a later iteration this was deemed as a poor representation of real lock structures, which is detailed in subsection 4.5.1.

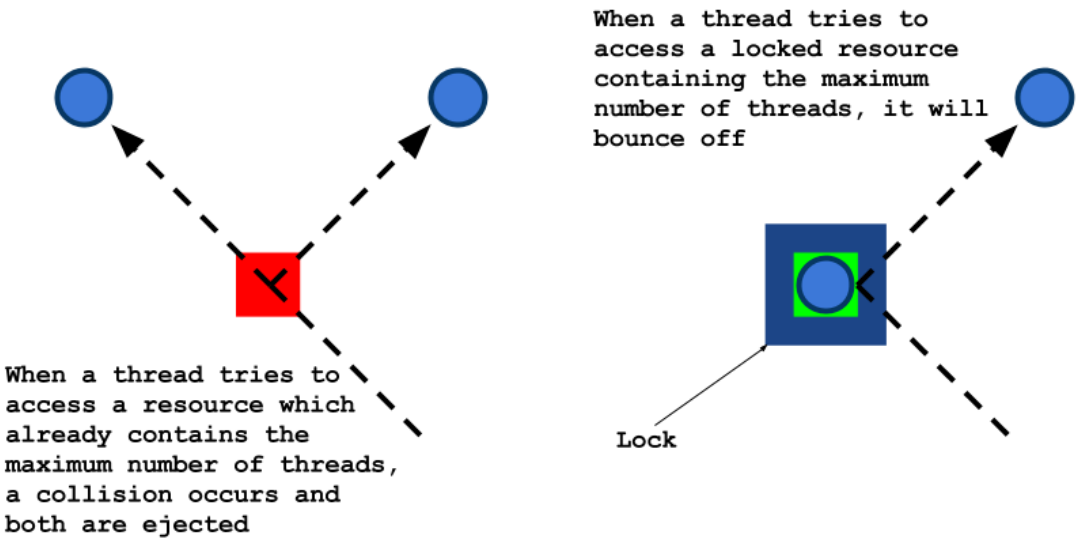

Figure 4.6: Visualisation of the operation of locks on resources

4.4 Additional Game Features

The following section describes the implementation of game features which were important for conveying elements of concurrency, but not strictly part of the core game framework.

4.4.1 Linked Resources

Another feature that was required in the game was linked resources: a pair of resources which the process would not use until both had all the required threads. It represented a situation in which a process requires concurrent access to two resources, so has to have threads accessing both before it can proceed. This required some reworking of the level data as at the time there was no way to distinguish between different resources, which was necessary to indicate which particular resources were linked. Each resource in the JSON object was given an `id` number and an optional `linked` number, which would indicate the `id` of the resource it was linked with.

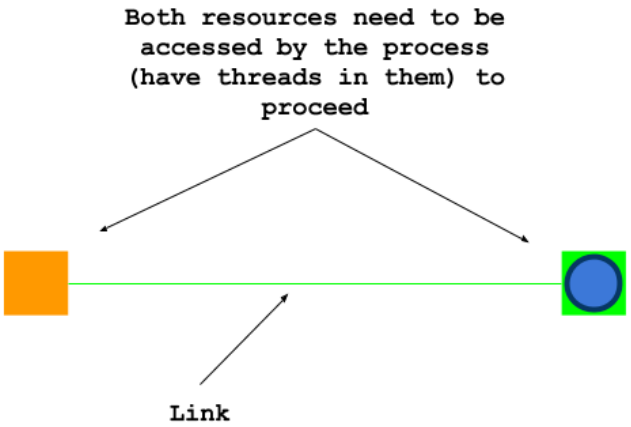

Figure 4.7: Visualisation of linked resources

With these details now loaded, the `create` function checks to see if a resource has a link: if so, it draws a connecting line between the two to indicate the link. It was also decided to give the resources an extra parameter, the `ready` boolean, to streamline the timer starting process: rather than setting the `runTimer` boolean to true when a resource contained the right number of threads, the `ready` boolean was set to true. If a resource had no linked resources and `ready` was true, it would run the timer, but if it did have links it would wait until both resources were marked as ready, and then run both timers.

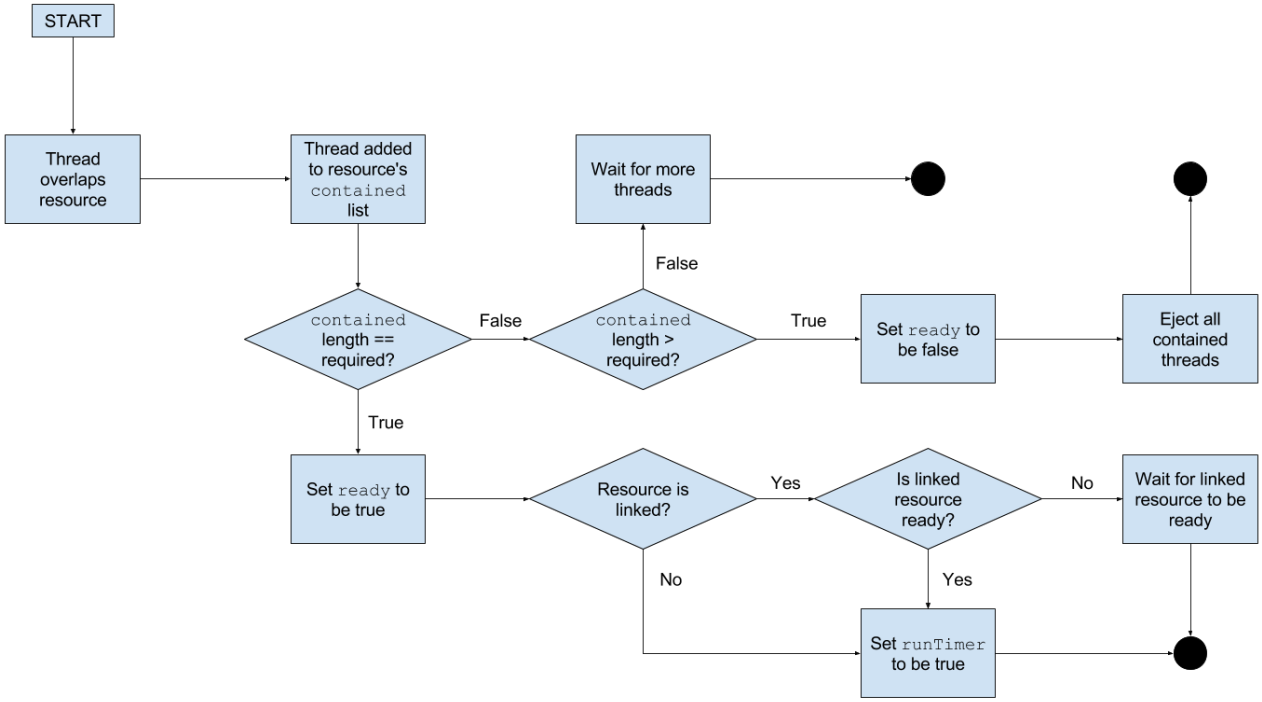

Figure 4.8: Flow chart representing the sequence of events when a thread overlaps a resource

4.4.2 Directional Locks

Directional locks were designed to 'fire' threads from one region to another, regions which otherwise threads would not be able to access. Firstly, this required another change to the level design JSON: some locks now required a `direction` parameter. The direction would be set to one of four possible values, corresponding to up, down, left and right.

When loading a lock in the `create` function, a check was made to see if the lock had a `direction` parameter associated with it. If so, a specific graphic would need to be loaded (a lock with an arrow in the corresponding direction) and a `direction` parameter was added to the lock's sprite with the appropriate angle assigned.

Now, when a resource was completed, a check was made to see if a lock was currently overlapping the resource. If so, then a further check was made to see if the lock had an assigned `direction` parameter. If so, then the threads would be ejected only at the angle specified; if not, they would be ejected as usual.

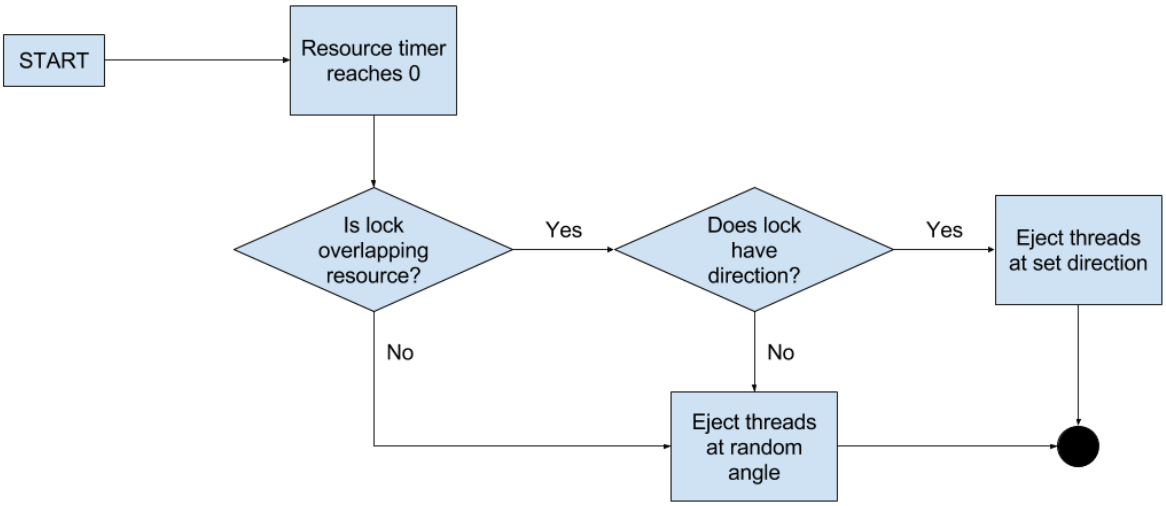

Figure 4.9: Flow chart representing the sequence of events when a thread overlaps a resource locked by a directional lock

Random Angles Causing Problems

A problem which occurred with this implementation was that no matter how small the 'gap' for threads to be fired through was made, even at its minimum threads which had not been directly fired were still managing to get through the gap by chance. It was decided that there was no direct way to stop this from happening, and initially the issue was left as it was and a note made that in concurrency, poorly managed threads can make it through gaps in security and gain access to regions to which they are unauthorised.

Upon revision, however, this was deemed as being unfair on the user. It's true that such issues do occur and have to be dealt with in concurrency, but usually they are the fault of the developer implementing them. As it was, the user had no control over this problem and the entire issue was down to chance, so it was felt that the problem should be eliminated.

Solution: Fixed Angle Paths

It was decided that to overcome these issues the framework needed more control over the threads. The random element of the threads' movement was very important, but a way to predict how the threads were going to behave at least to some degree was necessary to stop them from reaching areas where they weren't supposed to be.

A solution was to limit the threads' travelling angles to one of four directions (corresponding to North-East, South-East, South-West and North-West on a compass) to ensure they only moved in set diagonals - a constraint applied to the game's start and whenever a thread was ejected from a resource. With this knowledge, it was possible to track paths which **would** cause the threads to slip through the gaps, and force the threads never to be spawned along these paths. If the threads couldn't access these paths, then they could never get close enough to the gaps to break through.

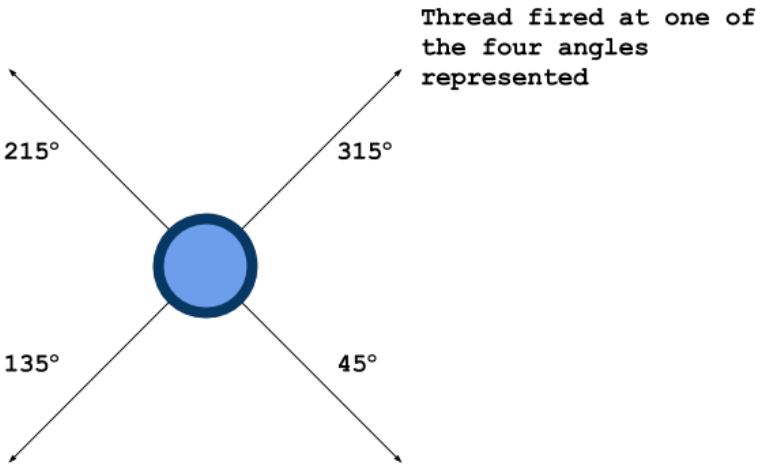

Figure 4.10: Visualisation of new thread firing angles

4.4.3 Multiple Processes

It was felt that a necessary aspect of concurrency to represent was not only the concept of multiple threads all trying to gain access to resources simultaneously, but multiple threads from different processes. This was represented in game by having threads of three different colours: green, red and the original blue.

This addition required extensive restructuring. The level details JSON now had to specify not just how many threads were required, but how many of each colour. Every resource had to keep track of how many threads from each processes were required for completion, and this had to be reflected in the level details JSON as well as including different collision rules for each coloured thread and a textual representation of how many of each coloured thread were contained.

Despite the fact that large sections of code had to be rewritten and restructured, the threads all followed the same basic set of rules which had already been put in place. This meant that this implementation stage, although introducing new complexity to the code, was not exceptionally difficult to implement but made a significant difference to how the game worked, providing several layers of new interactions for levels to focus on.

4.4.4 User Accounts and Scoring

The primary purpose of the user accounts and scoring system was to track the evaluated users to see how well they were doing, but also to provide them with session to session persistent states - both of these were functional requirements of the project. To control user account access, it was decided to use Flask's login management package - it is very easy to use and does not require any external libraries [6]. It has limited session control and security measures, but for the purposes of this project these features were considered secondary to convenience; while security was an issue, as users should not be accessing each others' accounts and manipulating evaluation data, it was felt that only basic measures were required to protect the data's integrity. Flask's login package does not set the world standard for security practices, but it does account for common security risks and ensures a standard level of security across the platform. No registration functionality was required either, as test users and their passwords were populated by an external script.

Once user accounts were set up, a scoring system was implemented. It was decided that this should be based on two aspects of gameplay: time taken and the number of collisions occurred. The reasoning behind this was

to encourage the users to avoid collisions, as is best practice in concurrent programming, and also introduced an extra element of challenge to keep the game interesting. A faster time would also indicate that the user has understood the system quickly, which is a sign that they have grasped the fundamental concepts well. The score was calculated by recording the time elapsed for a level plus the number of collisions multiplied by an arbitrary factor of five. The lower the score on a level, the better.

Next, a way to display the scores in a meaningful manner had to be decided on. Users would likely be familiar with the standard three star system prevalent in mobile games, and as the game followed a similar level-by-level structure as its mobile counterparts, it was felt that this would be an intuitive and easily implementable scoring system. Each level had to be checked and appropriate thresholds for each star rating decided upon - this was a complicated process, as some levels would simply take longer by nature, and some were designed in a way that completely limiting collisions was very difficult. Once appropriate thresholds were decided upon, they were added to the levels JSON structures to be loaded into the game framework.

The star-score for the level was displayed to the user after each attempt, and also saved to the database. The simplest way to save the scores was to assign each user with a JSON in the database, which contained their current high score for a level. Each time they completed a level, the score was updated and displayed to the user.

4.5 Late Stage Features and Alterations

The following section details additions made to the framework at later stages, usually after testing or driven by other significant changes. These additions are not necessarily essential, but aim to make the game interaction more streamlined and convey more accurate concurency concepts to the user.

4.5.1 Limiting Lock Access to a Set Number of Threads

Up until this point, locks had allowed threads to access a resource if it was not full, after which they restricted all access. Upon reflection, this was a poor representation of how locks would actually behave in concurrent programming, so it was decided that a change was necessary.

The new implementation of locks gave each lock an allowed number of threads (dictated by the `allowed` parameter, added to lock specification in the level details JSON) which it would use to accept or reject threads. Every time a thread overlapped a resource, a check was made to see if a lock was overlapping it and how many threads this lock allowed. If the resource already contained the maximum number of threads specified by the lock, the thread would be rejected; if not, it would be accepted into the resource.

Although a fairly simple change, this gave a much more realistic representation of how certain lock structures in concurrency work and the side-effects of using them incorrectly. For example, a lock which did not allow enough threads into a resource would constantly reject incoming threads even though the resource still needed them, and a lock which allowed too many would constantly be allowing collisions to occur within the resource.

The Zero Lock (or Block)

This change gave light to a previously unforeseen addition: the zero lock, or block. It was a lock structure with an `allowed` parameter vlaue of 0, which meant that it would restrict all access to a resource. This was seen as a very useful tool for users, as it could prevent resources from being used prematurely and reduce the number of collisions occurring. Also, when dragged onto a resource which already contained threads it would eject them

without causing a collision, which was useful to prevent deadlocks from occurring where threads were in short supply.

4.5.2 Reworking Restricted Access

As levels became more complicated and tried to express more involved aspects of concurrency, the established method of limiting access to restricted areas became less effective. Often, where there were multiple gaps, it was not possible for the entirety of the path to the gap to be restricted, so threads were eventually finding their way onto these paths and slipping through the gaps. Additionally, smaller environments could cause threads to be caught in bouncing patterns and never come into contact with a resource.

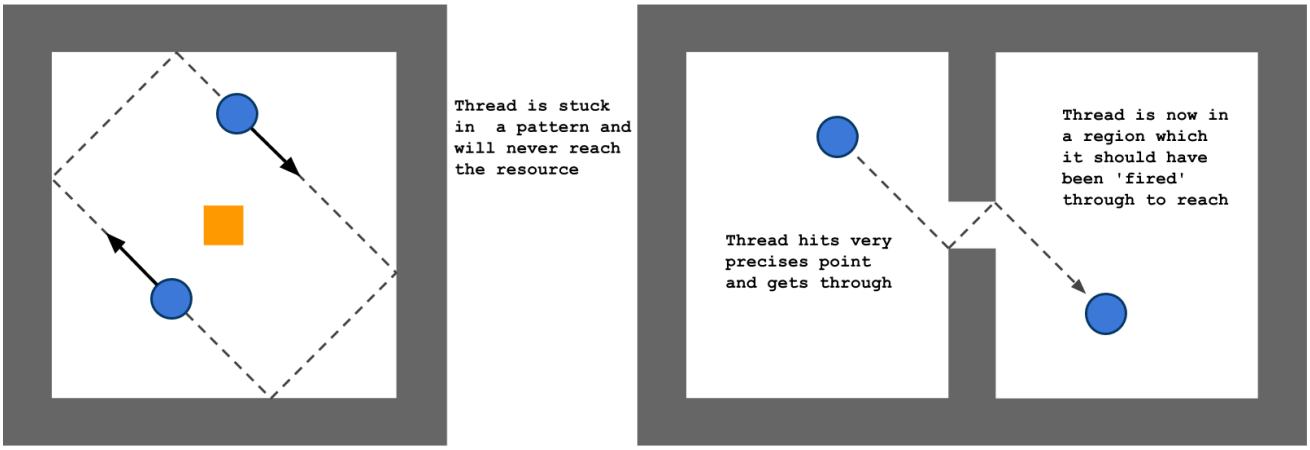

Figure 4.11: Demonstration of the two ways in which the fixed four-point direction firing of threads were causing issues

A solution to this problem was to introduce an invisible barrier at the gaps which could only be passed through if the thread had been specifically fired from a directional lock. The implementation involved adding an extra boolean parameter to the threads called `fired`, which would be set to true if the thread was ejected from a resource upon completion and the resource was overlapped with a directional lock. Next, invisible, immovable tiles were placed in the gaps which would make a check every time a thread collided with it to see if the boolean parameter `fired` was true. If so, they could pass. If not, they would bounce off.

With this new method of restricting access in place, restricted angles of travel were no longer necessary. However, rather than revert to the original method of sending threads at any angle, it was decided that some further refinement was necessary. The problem with sending threads at any angle was that there was a slim chance that the angle would be either a direct horizontal or vertical path. Depending upon the spawn point, this could cause a thread to never interact with a resource and cause a deadlock. Instead, the threads retained their current paths (North-East, South-East, etc) but with an offset of a random number up to 20° in either the clockwise or counter-clockwise direction. This gave a reasonably varied degree of movement without allowing the threads to get caught in loops.

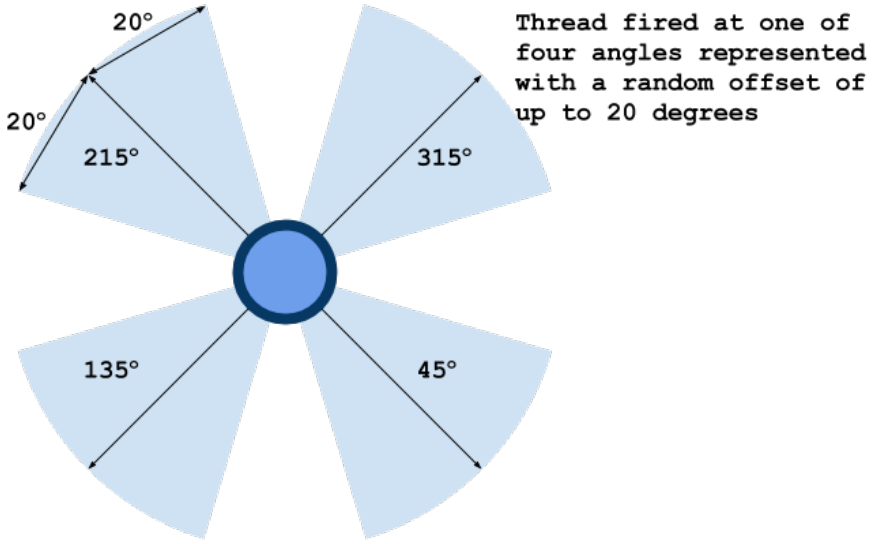

Figure 4.12: Visualisation of new thread firing angles

4.5.3 Snapping Unplaced Locks Back to Tray

Late in the development process, a bug was discovered which could allow users to cheat: users were capable of placing locks in such a way that they overlapped more than one resource, which was counterproductive to some of the lessons being taught about taking care and consideration with locks. The bug was fixed by recording each lock's starting location in the tool tray when they were loaded in, and whenever the lock was not placed on the centre of a resource they would snap back to their starting points in the tray. As a gird system was already in place, it was clearly visible when locks were placed on the centre of a resource. With this functionality in place, the overlap of multiple resources was no longer possible.

4.5.4 Help Modals

Each level had an associated tip to guide the user, but it was decided that this wasn't enough information to guide them in the finer points of gameplay and help them understand the connection of the concepts with real life concurrent programming. Therefore, several help modals were created which were associated with levels and loaded when the level was started. The modals consisted of text and images or gifs, which would display game elements in action with some description of what was taking place.

Chapter 5

Evaluation

Once the implementation of the game was complete, it required testing against users with an investigation of how effective it was at achieving the project's aims, the process of which is detailed by this chapter.

5.1 Evaluation Method

The evaluation involved testing a user's knowledge of concurrency, having them play the game and then test their understanding again to see if their knowledge had improved.

The method of testing users understanding was to have the user answer custom written quizzes before and after playing the game. The quizzes before and after had to be different, to avoid users simply learning answers. The quizzes were made to be six questions long, which was decided as being a good number of questions because it was enough to gauge the user's understanding but not enough for them to start experiencing fatigue.

Two quizzes were written: Quiz A and Quiz B (which can be found in the appendices.) Because it could not really be guessed at how difficult the quizzes were, it was decided that the best way to balance the results was to present Quiz A to half the users before the game and Quiz B to the other half, with them alternating for the post-game quiz. With this method, anomalous results relating to the difficulty of the quizzes could be identified and accounted for.

The project was tested on 12 users of varying backgrounds in concurrent programming, ranging from no prior experience to well practiced. It was decided that a wide range of experience should be tested to see which groups found the application the most useful - this could be tracked by the difference in quiz scores before playing the game and after playing the game. The experience level of the users was not explicitly recorded, but derived from their pre-game quiz scores: a high score indicates a high level of experience, and a low score indicates less experience.

5.2 Evaluation Results

Below can be found a box chart detailing the results of the quizzes. As can be seen, there is a considerable rise in the average score between pre-game quizzes and post-game quizzes. Although there is some deviation between the two groups who started with each quiz, the averages of both groups are very close. We can take this as an

indication that on average users gained a better understanding of concurrency concepts by playing the game, indicating a degree of success of the project.

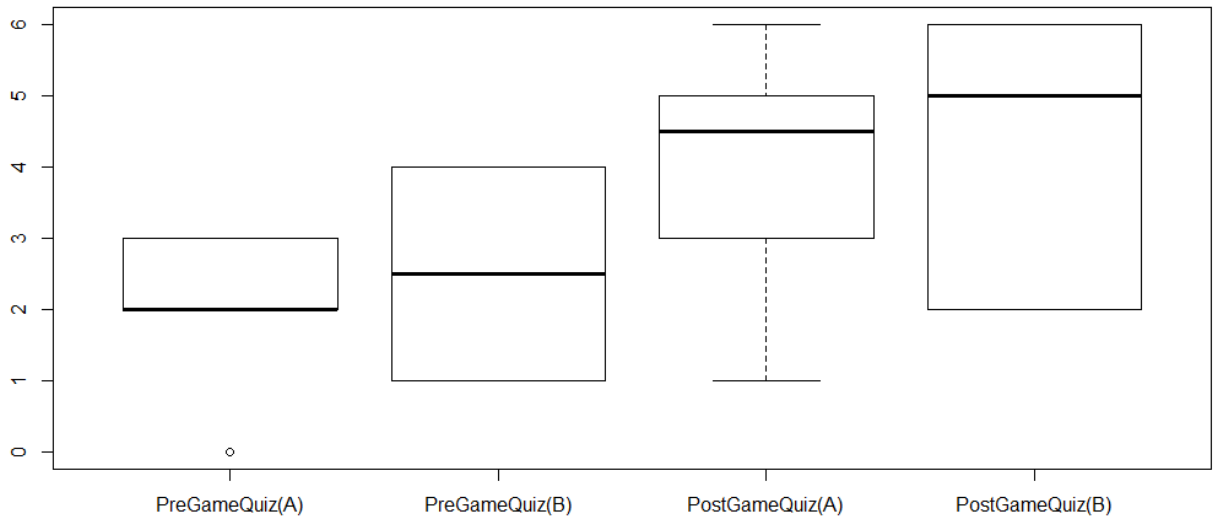

Figure 5.1: A chart displaying the spread and averages of scores across pre- and post-game quizzes, for both Quiz A and B

Below can be seen the difference of each user's correct answers in their pre-game quiz and their post-game quiz. Here, an increase in their score would indicate that they got more questions right in the second quiz, and as such have learned something while playing the game. As can be seen here, most users saw a significant increase in their scores, indicating that the game successfully instilled some knowledge of concurrency.

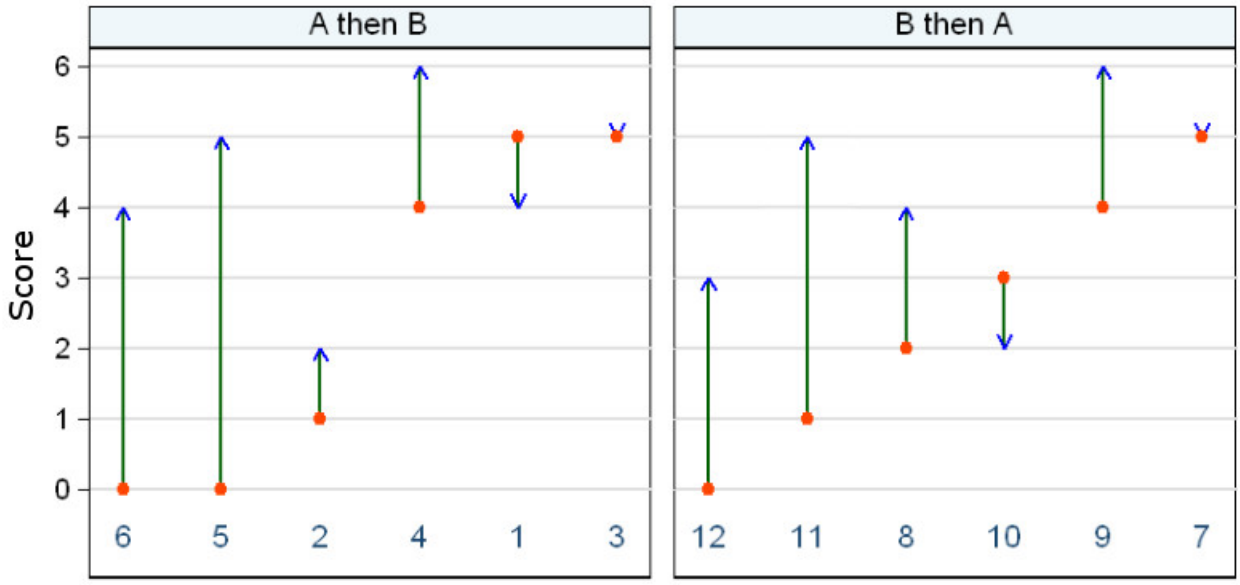

Figure 5.2: A chart displaying the difference between pre- and post-game quiz scores for each user. The chart has been ordered by each user's starting quiz score for each user group: the orange marker represents their pre-game score, and the blue marker represents their post-game score, with an arrow direction indicating an improvement or decline in scores.

It must be noted, that the users who showed the highest increases in scores began with a fairly low score on the first quiz. This would indicate that they were relatively unfamiliar with the concepts being explored upon starting their first quiz. We can then determine that the game has been most useful for players who have less experience with any kind of concurrency, and those who are already familiar with the concepts (i.e. scored highly in their pre-game quiz) did not show much improvement.

This is to be expected: it was established in the design stage of the project that the game would aim to teach fairly basic concurrency concepts. Players who are already familiar with these basics are not likely to see any improvements, but the game has still proven to be of significant use to those without any prior experience.

It was also interesting to examine the post-game quiz scores of users compared with their average star rating across all the levels they played. A higher star average indicates that the player has completed the game's levels to a higher standard - that is, have taken less time and allowed less collisions - and would imply that they are better at the game and have more thoroughly understood the concepts being taught. A comparison was made between these values and their final quiz score:

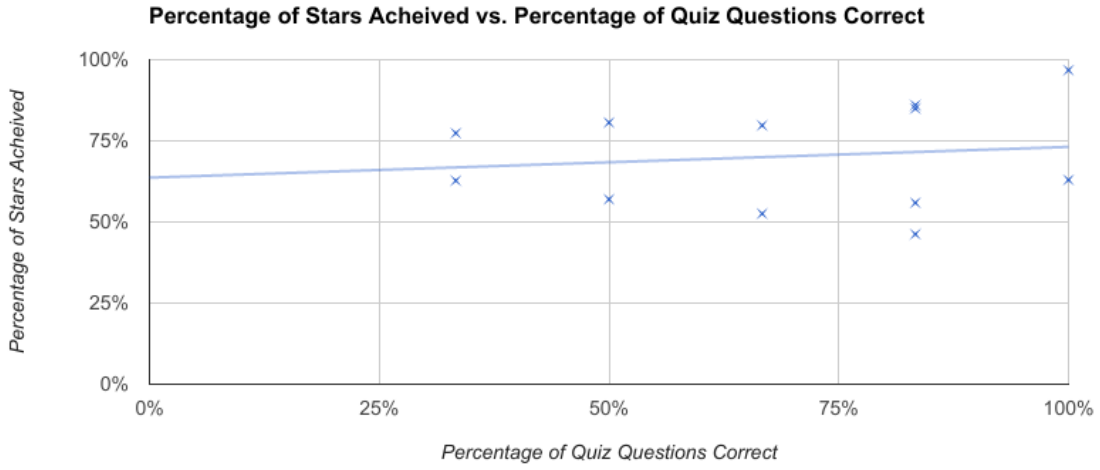

Figure 5.3: A comparison between the users' average star score per level and their post-game quiz score, with a linear trend line

It was expected that those who did well at the game (i.e. scored a high number of average stars) would have developed a good understanding of concurrency and would have scored highly in the post-game quiz, but this turned out not to be the case. Although the trend line demonstrated a positive slant (meaning that *on average* a high number of stars did correlate with a high post-game score) there is significant variation, indicating that the correlation is weak.

A possible conclusion to be drawn from this is that, while the game does in general benefit a user's understanding, the victory requirements imposed on the user do not necessarily promote a good understanding of concurrent theory. It must also be noted that a high score is dependant on the user making quick and precise mouse movements, which does not have any particular bearing on their understanding of concurrency - while a user coming up with a solution quickly and with minimal collisions *could* indicate that they have a good understanding of the theory, there is no way of telling if they would have gone on to score poorly in the game because their mouse movements were hesitant or imprecise.

These results indicate that being good at the game does not guarantee that the user has developed a good understanding of concurrency. I believe that further research into this would be useful: it is possible that

5.3 A ‘Fun’ Game

When the users were enlisted to play the game, no one was specifically encouraged to play every one of the 31 available levels - they were requested to spend as long as they felt comfortable, but not to force themselves to finish it. Despite this, many of them played all the way up to the end of level 31, with only one user completing less than half of the levels.

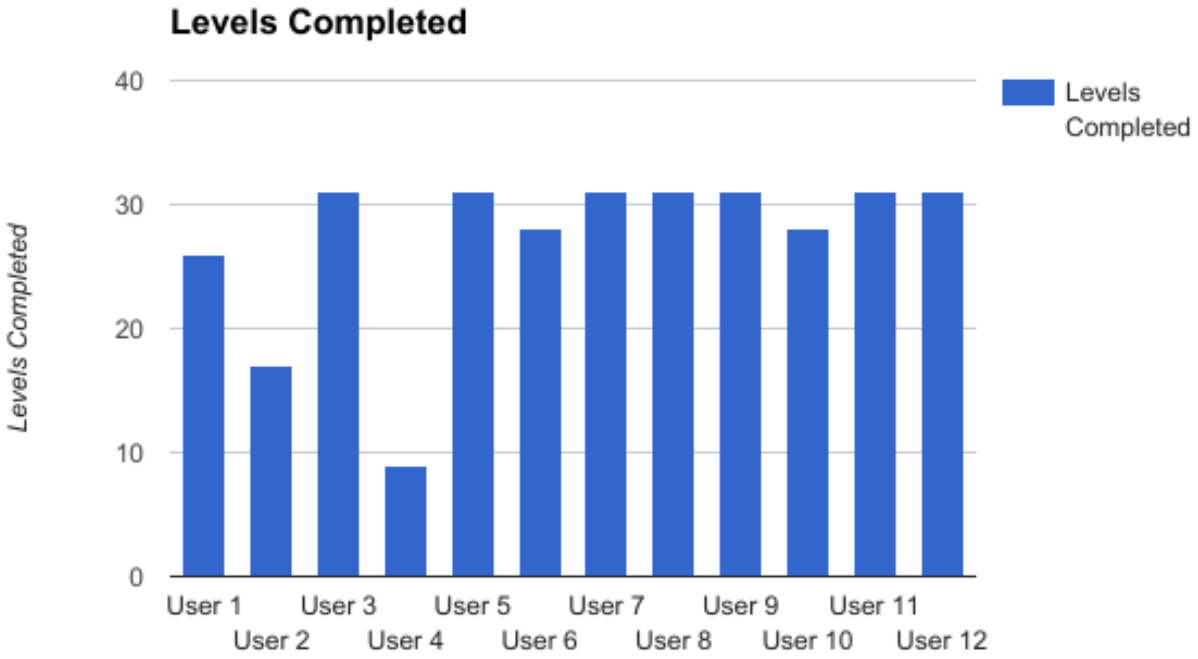

Figure 5.4: A chart displaying the number of levels completed for each player. A high number of levels completed indicates a significant time investment

Whether this is indicative of completionist human nature or an actual enjoyment of the game we cannot be certain, but it can be assumed that if the users went above and beyond to complete the game fully (which was expected to take on average over half an hour) that they must have been engaged enough to be enjoying it. This fulfils the first requirement: “must be an enjoyable and entertaining experience,” and makes the game a good classroom alternative to be enjoyed in free time.

5.4 Quiz Evaluation

After collecting the users results, it was possible to examine the quiz answers and look for patterns in the answering of the questions themselves. The two quizzes yielded a very similar number of correct answers: Quiz A had a total of 37 correct answers and Quiz B had a total of 41 correct answers. This indicates that there was very little difference between each quiz in terms of difficulty. As expected, the post-game results came out significantly higher than the pre-game results.

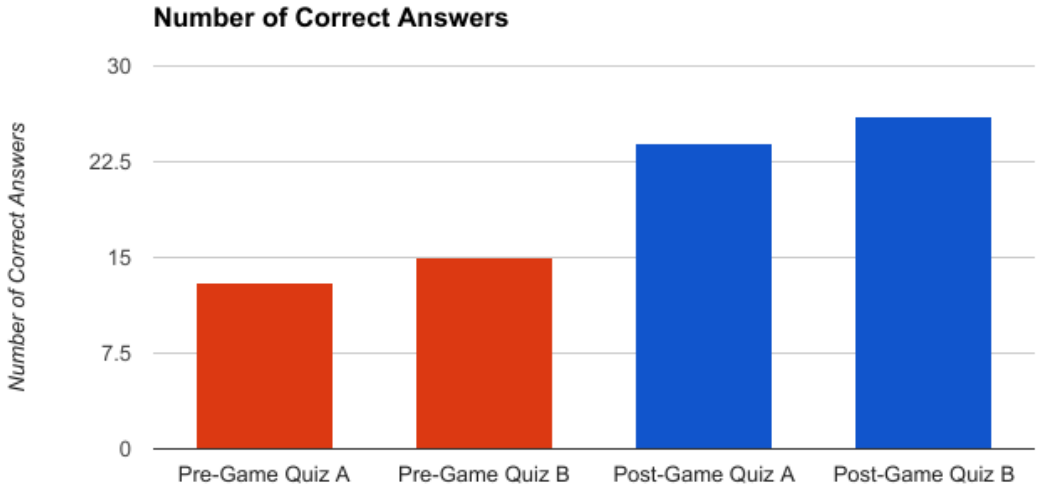

Figure 5.5: Breakdown of correct answers given for Quizzes A and B, between both pre- and post-game quizzes

However, in the examination of the results for individual questions, there were some which were identified as being significantly harder than others, as can be seen from the chart below.

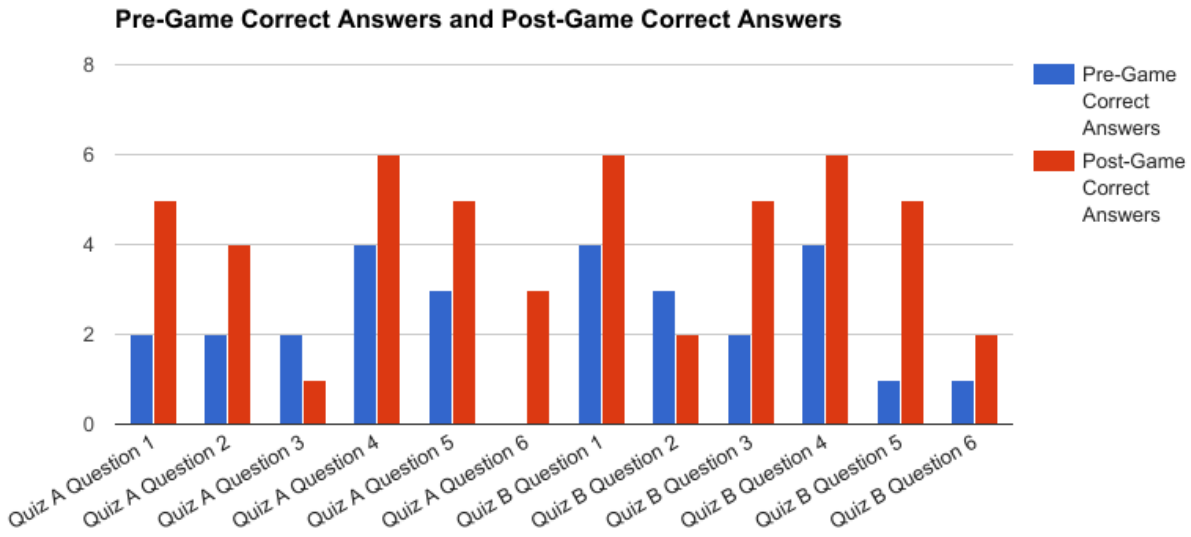

Figure 5.6: Breakdown of correct answers given for each question, between both pre- and post-game quizzes

Quiz A question 3 was a particularly interesting case because while only three correct answers were given in total, which is decidedly low, there were more correct answers given in the pre-game quizzes than the post-game quizzes. Quiz B question 2 also exhibited this behaviour, so it was deemed necessary to investigate these two questions.

Quiz A question 3 could be construed as being framed somewhat awkwardly, as it asks which of the following would NOT be in issue if mutual exclusion were not used in a scenario. This could have thrown users, because all the other questions are presented as requiring users to identify issues that WOULD be problems, and the difference could have been confusing.

Quiz B Question 2 was a little more surprising. The answer to this question was stated in one of the help

modals and underpins a fairly important concept in concurrent programming. However, while the user had to understand the relationship to complete the game, the explicit relationship was only described in real world terms once so it could have been missed.

A further limitation of the quizzes is that they do not specifically test the user's practical application of the relayed concurrency concepts, but rather their theoretical understanding of them. A challenge that could be taken forward in further development of the project could be to introduce a post-game quiz which requires the writing of code to solve a genuine problem. This would be a good test of the user's understanding of the practical application of learned concepts, but evaluation of the submitted code could be tricky, particularly if it was designed to be automated.

5.5 Evaluation Summary

Based on the results gathered from the evaluation, it is clear that users on average have shown a marked improvement in their understanding of concurrency concepts.

It was identified, however, that how good a user was at the game (i.e. completed levels in a short time and encountered minimal collisions) did not seem to directly affect how well they truly understood the concepts of concurrency in their post-game quiz. This may not necessarily be related to the user's understanding of concurrency, however, and could be simply due to poor fine motor control on the user's part. For example, it is very likely that a user using a mouse would have a distinct advantage over a user with a track-pad on a laptop due to the finer degree of control. A potential future development to address this issue could be to limit the amount of rapid, precise cursor control required to achieve a high score on a level, possibly by giving the user time to construct their solution in advance (in the style of Trainyard and The Incredible Machine) and not force them to rush with solutions.

While users were explicitly told to complete only as many levels as they felt comfortable with, many of them completed all available levels. This indicates that students are willing to practice with teaching platforms at a significant cost to their own free time (an estimate of close to 40 minutes) if they feel engaged and entertained while doing so - it appears that a game is very much a suitable platform for such teaching.

Overall, these results indicate that a game would be a very useful platform in teaching students concurrency in their own time. Students have primarily benefited from playing the game, particularly those with limited prior experience. Although a good understanding of the game does not directly translate to a good understanding of concurrency, the majority of users did learn from it and spent a significant amount of their own time engaging with it.

Chapter 6

Conclusion

The fundamental objective of this project was to investigate whether or not a game was a viable platform for teaching concurrency. After the evaluation of the preliminary results, it has been established that players showed a marked improvement in their understanding of concurrency after playing the game, so it appears that the game has indeed taught them concurrency concepts to a degree.

It is important to note that the users showing the most improvement were marked as having little prior knowledge of concurrency - this is to be expected, because the concepts included in this game were fairly basic. Anyone with prior knowledge of concurrency would likely be familiar with them, and are unlikely to gain significant experience from the game.

The platform is also very different from the typical classroom or lecture setting in which concurrency is usually taught. Given the level of success it has had in teaching students over a brief period of time, the game has indicated that alternative platforms are certainly viable for teaching concurrency, at least at a low level of complexity. Further research should be done to investigate whether this would scale up for more complicated concurrency concepts.

The users also engaged with the game for a significant amount of their own time. This indicates a willingness for users to interact with content outside of teaching hours in a game format, which can act as a good platform to supplement theory teaching in the classroom.

6.1 Preliminary Results

As has previously been mentioned, after playing the game the users showed a marked improvement in their understanding of concurrency. The majority of users also completed the levels to the end, seeming to indicate some enjoyment and engagement with the content.

The results did indicate, however, that being good at the game did not necessarily mean that the user would have learned any more than someone who did badly at the game. We can draw from this that while the game does teach concurrency concepts successfully, the method of evaluation of the user's ability in the game does not match up with the method of evaluating their knowledge after playing it.

Based on these preliminary results, I believe it is safe to say that it is possible to convey basic concurrency concepts in a game format and for users to learn from it, whilst also enjoying themselves.

6.2 Future Developments

While the project answered several important questions laid out at the start of the process, there are two particular areas that I feel could be developed to investigate the results. Firstly, the game could be made more complex to convey more complicated concurrent programming features, and an investigation should be undertaken to determine whether users with a greater understanding of concurrency will gain more from it, and whether the less experienced users will still be able to learn the basics. I believe this would be an interesting study, because it will indicate whether or not a game is only useful on a basic level of conveyance or whether it can be scaled up to exhibit more complex behaviour and still be understandable for those new to concurrency.

Secondly, while I feel that the quizzes are a useful gauge for determining how much the user has learned, it has not been specifically determined whether or not these skills are transferable to practical problem solving. As the problems posed in the game are very practical in their nature, one would expect to see a correlation, but it was not something that was explicitly tested. A future development could be to construct some environment in which the user can exhibit what they have learned in a practical coding exercise. The evaluation of such a practice would likely prove difficult, but it is still an unexplored idea and I believe further investigation would be useful.

6.3 Deployed Game

The game can, as of the submission date of this document, be found at:

`none`

This URL can be used to access the game without requiring any local setup. The game can be played without needing a user account.

Appendices

Appendix A

Source Code Structure

The following appendix details the submitted source code files, outlining the structure and briefly describing important files.

```
/
├── concurrency.py
├── db_conf.py
├── extract.py
├── help_data.py
├── populate.py
├── requirements.txt
└── static
    ├── css
    ├── fonts
    ├── img
    ├── js
    │   ├── bootstrap.min.js
    │   ├── game.js
    │   └── phaser.js
    └── templates
        ├── base.html
        ├── game.html
        ├── index.html
        └── login.html
```

- **`concurrency.py`:** this file contains the main Flask app. It handles rendering pages, establishing the database connection and retrieving/storing data as games are played.
- **`db_conf.py`:** a configuration file, containing important details for connecting to the database.
- **`extract.py`:** a file written for the evaluation stage which extracts user data from the database and displays it. It displays the username, number of levels completed and the total number of stars achieved.
- **`help_data.py`:** contains all the text and image paths for the help modals for each level, stored as a list of dictionaries.
- **`populate.py`:** contains all the level data, and when run populates the database with every level and a selection of predefined users.
- **`requirements.txt`:** a list of requirements for the project to run

- **`game.js`:** JavaScript file containing the definition of the main game
- **`phaser.js`:** Phaser source code, required to run the game's functions
- **`base.html`:** base HTML template, all other templates inherit core elements from this template
- **`game.html`:** template in which the game div is run.
- **`index.html`:** template for the main page, populated with every available level to the user

Appendix B

Evaluation Introduction

All the evaluated users were contacted via email once they had agreed to perform the evaluation. Below is the email which was sent out to each of them, which acts as the evaluation introduction.

Hello!

Thank you for agreeing to take part in my honours project evaluation. Read on for more information and instructions on how to proceed. The evaluation should take roughly 30 minutes, but feel free to stop playing whenever you feel like.

Gamifying Concurrency Teaching

Concurrency and parallelism are some of the most difficult fields of computing for CS students to grasp. As such, this project aims to investigate whether these concepts are grasped more effectively if presented in a format different from the usual lecture/lab setup - namely, a game. The game's intention is to present some of the most basic concurrency fundamentals in a way that's easily digestible and fun to engage with. If you have any further questions about the project as a whole, please feel free to email me at parkinsonj94@gmail.com.

Instructions

1. Complete the quiz at the following link to the best of your ability: **<LINK TO FIRST QUIZ>**. Don't worry if you are unfamiliar with the concepts and aren't sure of the answers - just skip questions if you have no idea. Your given username to put at the top of the form is: **<UNIQUE USERNAME>**
2. Go to the following link: **<LINK TO WEBAPP>**. Please log in with the following credentials:
    - username: **<UNIQUE USERNAME>**
    - password: **<UNIQUE PASSWORD>**
3. Complete as many levels as you can. Levels will get harder towards the end. If you get stuck, you can choose to stop playing or email me for a hint.
4. Once you're done (whether you finished all the levels or not) please complete the second quiz: **<LINK TO SECOND QUIZ>**
5. I would very much appreciate it if you could email me upon completion with any general feedback. Also, if you experience any issues, do not hesitate to email me and I will respond at the soonest possible convenience. Otherwise, you're done!

Notes

- Your first quiz might be marked "Set B" and your second one "Set A" - this is not a mistake. I've split the testing groups up so that half begin with one quiz and half begin with the other. This is to ensure that there's no bias based on the difficulty of the questions.
- The game does not work on mobile devices - you will have to use a laptop or desktop PC.
- If you're stuck on quiz questions, please don't look them up - that would defeat the purpose. Instead, leave them blank and move on. Don't worry, the quizzes are anonymous, so even I won't know who scored what!

Finally, thank you!

Thanks for your support in the evaluation of this project - it is extremely valuable to me.

All the best,

Jack Parkinson

Appendix C

Quizzes

For each question in the quizzes the correct answer is displayed in bold.

C.1 Quiz A

1. Which of the following best describes the behaviour of THREADS?
   (a) They follow a set of instructions predictably at a predictable time
   (b) They follow a set of instructions randomly at a predictable time
   (c) **They follow a set of instructions predictably, but at an unpredictable time**
   (d) They follow a set of instructions randomly at an unpredictable time
   (e) I don't know
2. Why is mutual exclusion necessary when dealing with concurrency?
   (a) Threads are likely to stop using a resource when they are not finished with it
   (b) Processes spawn too many threads for the operating system to effectively keep track of them
   (c) **Threads complete activities at different rates and so will interact with resources unpredictably**
   (d) Resources need to be excluded from processes so that they don't get used up before other processes can access them
   (e) I don't know
3. If mutual exclusion wasn't used in concurrent programming, which of the following would NOT be a problem?
   (a) Data used by multiple processes could be overwritten or corrupted
   (b) Threads could become deadlocked
   (c) **It would be impossible to know what a resource was used for**
   (d) Threads could cause collisions
   (e) I don't know
4. How would you describe a case where two threads access the same resource at once and data is corrupted?
   (a) Handshaking

(b) **Collision**

(c) Headbutt

(d) Corruption Field

(e) I don't know

5. Which of the following best describes a DEADLOCK?

   (a) When two threads both try to access the same resource

   (b) When a thread gets into cache memory and corrupts it

   (c) When a resource fails to load from the disk and threads are kept waiting

   (d) **When two threads are each waiting for the other to complete before proceeding**

   (e) I don't know

6. What best describes the difference between a RENDEZVOUS and a BARRIER?

   (a) **A rendezvous will allow two threads to access a resource and a barrier can allow more**

   (b) A rendezvous will allow one thread access to a resource and a barrier can allow two

   (c) A rendezvous will allow threads to access a resource and a barrier will block all access

   (d) A rendezvous structure can be reused, while a barrier cannot

   (e) I don't know

C.2 Quiz B

1. Which best describes the idea of CONCURRENCY in computing?

   (a) More than one computer running at once

   (b) A computer with more than one CPU core

   (c) **A computer allowing more than one task to run at once**

   (d) A program which does not require processor power to run

   (e) I don't know

2. Which best describes the relationship between a PROCESS and a THREAD?

   (a) **Processes create threads to perform multiple actions at the same time**

   (b) Processes are handled by the CPU cache, threads are handled by memory

   (c) Processes are instructions to the processor, threads are the hardware artifacts which run them

   (d) Threads are used to "tie" multiple processes together, so that they all run at the same time

   (e) I don't know

3. Which best describes the term mutual exclusion?

   (a) Locking a resource so that no threads can access it

   (b) Locking a resource so that only it can be used by a thread which is of the same data type as the resource

   (c) **Locking a resource so that only a specified number of threads can access it at once**

   (d) Locking a thread so that it cannot use a specified resource

   (e) I don't know

4. How would you describe a case where two threads are each waiting for the other to complete?
   (a) Timeout
   (b) Collision
   (c) **Deadlock**
   (d) Stagnation
   (e) I don't know

5. Which statement best describes what would happen if two threads attempted to access an unguarded resource at the same time?
   (a) No collisions would occur - the processes would access the resource sequentially by default
   (b) The processes would query the system admin, which would decide which process had higher priority
   (c) The resource would be overloaded and disconnect
   (d) **A collision would occur**
   (e) I don't know

6. How would you best describe a RENDEZVOUS in concurrency?
   (a) Two processes working together
   (b) A structure that denies any access to a resource
   (c) A structure that brings two resources together to allow a thread to access both of them concurrently
   (d) **A structure that denies access to a resource to more than two threads**
   (e) I don't know

C.3 Quiz Results

Below can be found the responses of every user. Users 1-6 performed Quiz A before playing the game, and users 7-12 performed Quiz B before playing the game. A cell filled with red indicates an incorrect answer, and a cell filled with green indicates a correct answer.

QUIZ A	USER 1	USER 2	USER 3	USER 4	USER 5	USER 6	USER 7	USER 8	USER 9	USER 10	USER 11	USER 12
Q1	3	5	3	1	5	5	3	2	3	3	3	3
Q2	3	5	4	3	5	5	3	3	3	2	3	5
Q3	3	5	2	3	5	5	4	1	3	1	2	5
Q4	2	2	2	2	3	5	2	2	2	2	2	2
Q5	4	3	4	4	5	5	4	4	4	3	4	4
Q6	3	3	3	3	5	5	1	6	1	2	1	3
QUIZ B												
Q1	3	3	3	3	3	3	3	3	3	3	2	5
Q2	3	3	2	1	1	4	1	4	1	1	5	5
Q3	3	4	3	3	3	3	3	1	3	2	4	5
Q4	3	3	3	3	3	3	3	3	3	3	4	1
Q5	4	2	4	4	4	4	1	2	2	2	4	5
Q6	3	3	4	4	3	3	4	1	3	5	5	5

Appendix D

Running the Project

Python 2.7[14] is required to run this project. It also has several external dependencies, including Flask and SQLAlchemy, which can be installed by installing from `requirements.txt`. This can be done by running the following command:

```
pip install -r requirements.txt
```

This command requires `pip`, which should come packaged with any Python release after Python 2.7.9[8]. The project can then be run by executing the following command:

```
python concurrency.py
```

This will start a local development server, which can be accessed by opening a web browser and navigating to `http://127.0.0.1:5000`. This should display the project's index and the application should be usable.

Appendix E

Ethics Checklist - Signed

School of Computing Science
University of Glasgow

Ethics checklist form for assessed exercises (at all levels)

This form is only applicable for assessed exercises that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, or getting information about how a system could be used, or evaluating a working system.

If no other people have been involved in the collection of information, then you do not need to complete this form.

If your evaluation does not comply with any one or more of the points below, please contact the Department Ethics Committee for advice.

If your evaluation does comply with all the points below, please sign this form and submit it with your assessed work.

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.

   Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback

2. The experimental materials were paper-based, or comprised software running on standard hardware.

   Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, mobile phones, and PDAs is considered non-standard.

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.

   If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

   Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

4. No incentives were offered to the participants.

   The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

6. No participant was under the age of 16.
   Parental consent is required for participants under the age of 16.

7. No participant has an impairment that may limit their understanding or communication.
   Additional consent is required for participants with impairments.

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. All participants were informed that they could withdraw at any time.
   All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

10. All participants have been informed of my contact details.
    All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation.

12. *All the data collected from the participants is stored in an anonymous form.*
    All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Module and Assessment Name L4 INDIVIDUAL PROJECT

Student's Name JACK PARKINSON

Student's Registration Number 2092657

Student's Signature 

Date 21/03/17

Bibliography

[1] Manuel Carro, Ángel Herranz, and Julio Mariño. A model-driven approach to teaching concurrency. *Trans. Comput. Educ.*, 13(1):5:1–5:19, February 2013.

[2] CreateJS, official website. http://www.createjs.com/.

[3] Phaser - HTML5 Game Devs Forum. http://www.html5gamedevs.com/forum/14-phaser/.

[4] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2016.

[5] Flask, official website. http://flask.pocoo.org/.

[6] Flask Login, official website. https://flask-login.readthedocs.io/en/latest/.

[7] Flockers, official website. http://www.flockersgame.com/.

[8] Installing Python Modules. https://docs.python.org/2/installing/.

[9] Lemmings Universe, website. http://www.lemmingsuniverse.net/games.html.

[10] Naoki Akimoto and Jingde Cheng. An Educational Game for Teaching and Learning Concurrency. http://www.aise.ics.saitama-u.ac.jp/KEST/KEST03W/papers/39-akimoto.pdf.

[11] Phaser, official website. http://phaser.io/.

[12] Rob Pike. Concurrency Is Not Parallelism.

[13] PlayCanvas, official website. https://playcanvas.com/.

[14] Python, official website. https://www.python.org/.

[15] Daniel Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. Daniel Shiffman, 2012.

[16] SQLAlchemy, official website. https://www.sqlalchemy.org/.

[17] A. Stevenson. *Oxford Dictionary of English*. Oxford Dictionary of English. OUP Oxford, 2010.

[18] The Deadlock Empire, website. https://deadlockempire.github.io/.

[19] The Incredible Machine, official website. http://www.sierragamers.com/aspx/blob2/blobpage.aspx/msgid/660032.

[20] Three.js, official website. https://threejs.org/.

[21] Trainyard, official website. http://trainyard.ca/.

[22] WebFaction, official website. https://www.webfaction.com/.