School of Computer Science and Informatics

Coursework Submission Cover Sheet

Please use Adobe Reader to complete this form. Other applications may cause incompatibility issues.

Student Number	C1421079
Module Code	CM3203
Submission date	05/05/2017
Hours spent on this exercise	>425
Special Provision	

(Please place an x is the box above if you have provided appropriate evidence of need to the Disability & Dyslexia Service and have requested this adjustment).

Group Submission

For group submissions, *each member of the group must submit a copy of the coversheet.* Please include the student number of the group member tasked with submitting the assignment.

Student number of submitting group member	

By submitting this cover sheet you are confirming that the submission has been checked, and that the submitted files are final and complete.

Declaration

By submitting this cover sheet you are accepting the terms of the following declaration.

I hereby declare that the attached submission (or my contribution to it in the case of group submissions) is all my own work, that it has not previously been submitted for assessment and that I have not knowingly allowed it to be copied by another student. I understand that deceiving or attempting to deceive examiners by passing off the work of another writer, as one's own is plagiarism. I also understand that plagiarising another's work or knowingly allowing another student to plagiarise from my work is against the University regulations and that doing so will result in loss of marks and possible disciplinary proceedings.

Cardiff University

Computer Science

CM3203: One Semester Individual Project, 40 credits

Final Report

Real-Time Networking Game

Author:
Braden Marshall

Supervisor:
Frank C. Langbein

5th May 2017

Abstract

A modern browser-based multiplayer implementation of the game Tron, featuring an artificial intelligence capable of competing against human users. With focus being on demonstrating the use of popular multiplayer video-game design techniques, and their implementation within a web-application. Existing solutions either rely on external browser-extensions, or lack support for multiplayer. We design and implement a web-application using common, and natively available, technologies. The result is a web-application capable of running multiple simultaneous game lobbies, enabling users to play Tron with their friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is somewhat well known that in the early 2000s the web began to transition from static pages into web-applications, i.e. those powered by user-generated content and dynamic HTML. This new form of the web was coined Web 2.0 (Graham, 2005). Over the past few years, the potential of these web-applications have evolved massively. In particular, many new technologies, such as those specified within HTML5, have been introduced into the average web-browser. These technologies have broadened the capabilities of a web-application; making them an incredibly viable medium for many software projects. In the past, browser-based video-games required the use of third-party browser-extensions, such as Adobe Flash. However, with the aforementioned improvements, there now exists curiosity regarding the feasibility of implementing a browser-based video-game using solely the technologies natively available within the web-browser.

Due to those reasons, for this project we are to develop a multiplayer implementation of the video-game Tron. It shall feature a computer opponent, powered by some artificial intelligence enabling it to compete against able-bodied computer users. The focus of the project is not to develop a fully-polished, consumer-ready product; but instead to create a fully functioning prototype, demonstrating the use of various video-game design techniques and the feasibility of their implementation within a web-application.

The web-application will support a server capable of running, and arbitrating, multiple simultaneously game lobbies, each holding their own game instance. A group of users are then able to connect to a particular game lobby, in order to become players of the game instance and thus able to compete against one-another over a network. A variety of networking techniques will be employed to improve the user-experience, by diminishing perceptible latency.

The intention of the artificial intelligence, controlling the computer opponent, is not to play the game perfectly, besting the majority of human combatants, but instead to act with human-like strategy and behaviour; remaining both challenging,

but possible for an average user to beat - keeping the game enjoyable. This will entail the use of simulation-based techniques and concurrency.

Chapter 2

Background

Before we begin to delve into the development of our project, it is important we first discuss the prior state of the problem. In particular, we introduce the reader to some key ideas that helped to compose the final presented solution. This shall also give us a chance to elaborate on the aims of the project.

2.1 The Game of Tron

The game of Tron first emerged in response to a scene from the 1982 film bearing the same name, Tron (Walt Disney Productions, 2014). These games often took their own spin on the concept, adjusting certain game mechanics. Hence, to help avoid ambiguity, we first describe, in detail, the rules of the game. These are the rules adhered to throughout the project's development.

Tron is a free-for-all game played by two or more players. Players are spawned, in a uniform distribution, around the four sides of a square, enclosing the game's arena. Each player is themselves a smaller square, 1 unit in size, that is initially *directed* facing inwards of the square. Once the game has commenced, all *alive* players are perpetually in motion, travelling around the arena at the same constant speed. However, at any time, the player can direct themselves 90° anti-clockwise (left) or 90° clockwise (right). As each player travels around the game arena, they leave a *wall* occupying the arena they previously covered. This continuous wall is know as a *trail*, and can be though of as the travelled path of a particular player. A player is considered to be *dead*, and out the current game, when they collide with borders of the arena, any other player, or the trail of any player (including their own). Hence, the goal of the game is for a player to travel around the game arena long enough to outlive all other competing players. This also introduces strategic play, as, for example, players are able to act aggressively in hopes to block off their opponents - defeating them sooner. A round of the game is considered finished

once only a single player remains *alive*. In this case, the single remaining player is considered the *winner*. If no players are left *alive*, the game is considered to be a draw.

2.2 Existing Games

As it stands, there are quite a few implementations of Tron in existence. Some are relatively small 'indie' titles, whilst others are in the form of an extraneous minigame within a AAA title. We shall begin to discuss some of these existing implementations, providing a brief overview of the functionality they offer and their incorporated techniques. It is worth noting, we focus primarily on the smaller titles as they better relate to the scope of our project.

First up is Fltron (Hsu, 2017). Fltron is an Adobe Flash based, two-player, grid-base implementation of Tron. It features a fairly skillful AI opponent, sufficiently capable of mimicking human behaviour. The game is also complete with a fair amount of polish, such as sound effects and aesthetic graphics. However, it does not support multiplayer over a network. It also happens to be quite outdated, incompatible with many modern browsers, due to its reliance on Adobe Flash which slowly being phased out in favour of HTML5. (Wikipedia, the Free Encyclopedia, 2017b)

Cycleblob (Shalom, 2017) is yet another Tron implementation. Although, it is unique in the regards that the game arena is a three-dimensional object; configurable to a variety of interesting shapes, such as a rounded cube or torus. It is quite a modern development, utilising many of the exciting HTML5 technologies, such as *WebGL* (Mozilla Developer Network, 2017). Similar to the aforementioned Fltron, the game features a capable AI opponent but still lacks multiplayer functionality.

Slither IO (team@slither.io, 2017) is an extremely popular browser-based game. Whilst it is not an implementation of Tron, it is very similar in concept. What makes this game particularly interesting, is that it demonstrates a high-level of technical capability. The game is multiplayer, enabling hundreds of players to compete against one-another in a single game instance. This clearly advocates some of what is possible within a modern web-browser. However, it is worth noting that the game is entirely player vs player; there are aspects of artificial intelligence.

Rounding off on what has been discussed, it is evident that these technologies, powering the modern web-browser, do hold a lot of promise. There is strong indication that they are indeed capable candidate platforms for running robust video-games of the arcade genre. However, there does not yet seem to exist a modern browser-based multiplayer implementation of Tron.

2.3 Browser Technologies

A modern web-browser is now equipped with a wide array of different technologies, many of which accomplish very similar goals. When it comes to settling on a particular technology, there is no 'one-size-fits-all' solution. Each are viable options under under certain circumstances. It is because of this, in the following section, we compare some of these competing technologies to enable us to better justify our future design decisions.

Firstly, we shall discuss some of the technologies whose purpose it will be to render the game's graphics. Some of the possible contenders include: animating DOM elements around using JS, or even CSS; SVG; a canvas element powered by Canvas 2D; a canvas element powered by WebGL. However, from those options, there are only two feasible candidates: WebGL and Canvas 2. WebGL is a low-level graphics API, and is based upon OpenGL ES. It provides the developer with a high amount of control over the graphics pipeline and is also capable of 3D graphics. On the other hand, canvas 2D is (as the name suggests) intended for relatively simple 2D graphics, and abstracts quite significantly away from the graphics pipeline - trading off control for ease-of-use.

For the task of network communication, there are many choices. However, the two main relevant technologies, for modern browsers, are WebSockets and WebRTC. There are a few key differences between the two. Firstly, WebSockets utilises the Transmission Control Protocol (TCP) protocol whilst, on the other hand, WebRTC can utilise either TCP or the User Datagram Protocol (UDP). Another key difference is that WebSockets only supports full-duplex communication between a web-browser and a web-server, whilst WebRTC's extends upon this by supporting full-duplex communication between two web-browsers. WebRTC requires that a web-server be set-up to handle the initial signalling required to establish a connection between two browsers. As one would expect, WebRTC is slightly more complicated to use, it also happens to be a newer standard and is not currently very well support amongst web-browsers.

2.4 Artificial Intelligence

As discussed previously (see section 2.2: *Existing Games* (Page: 4)), there are already several Tron implementations out there in existence. Some of these already feature an adequately sophisticated computer opponent that is controlled by an artificial intelligence. Sadly, some of these implementations do not have publicly accessible source-code, or developer documentation; making it very difficult to understand the techniques they're using. However, after further research it was discovered that conveniently, in 2010, Google sponsored an AI challenge (Google,

2010), which received over 1400 entries. This objective of this challeneg was to devise a artificial intelligence capable of playing the game Tron; albeit, with a slightly different set of rules and in slightly different circumstances. The winner of said competition, *a1k0n*, released a descriptive post-mortem (a1k0n, 2010), complete with source-code, detailing the core ideas behind what powered their submission. However, the produced artificial intelligence would be able to outmatch the majority of human controller players.

Chapter 3

Design

Throughout the following chapter, we shall analyse the intricate specifications of our application and accordingly propose sensible design solutions. This will consist of describing the application as a whole, then delving into each of the individual aspects. By the end, the reader should obtain a strong understanding of what exactly our application does along with the utilised core techniques.

3.1 Software Development Process

Firstly, we shall describe the overall methodology and workflow that have been employed throughout the duration of development. These are of important consideration as they promote both an effective and productive approach to the creation of software. Much of what is to be described is inspired by the ideas expressed in 'The Pragmatic Programmer' (Hunt, Thomas and Cunningham, 1999).

As the application is not a collaborative undertaking, there is little-to-no need in sticking strictly to any specific paradigm or framework for the software development process; as it would most probably lead to unnecessary overhead and distraction from any relevant progress. However, it is still of purpose to state that development most closely resembles the set of principles described under Agile (Beck et al., 2001), in that the application's design and implementation occurred concurrently, with continuous adaptation.

Over the course of the development process, the Git (Open source, 2009) version control system has been used to track file changes. Among other benefits, this allows us to maintain a full history of our application's source code, enabling us to revert back to previous versions - removing undesired changes. To further the backup capabilities, the Git repository is hosted online at GitHub(Marshall, 2017).

In order to manage the various short-term goals throughout development, a todo-list (in the form of a plaintext file) has been maintained. The purpose of which

was to help maintain an efficient workflow and provide a medium for documenting design thoughts between development sessions.

To aid future developers, or those studying the application, all source-code has been passed through a lint utility. This helps to identify potential bugs and preserve a consistent coding standard - by identifying syntactic discrepancies. Also, the source code itself contains extensive documentation; in the form of both relevant variable names and comment annotations.

3.1.1 Package Scripts

We have also assembled a repertoire of scripts that serve the purpose of automating some of the tedious tasks which are executed on a regular basis. In abstracting away from these laborious procedures, we have reduced the chance of unnecessary complications whilst also improving the efficiency of working with the application.

Below is a curated list of the available scripts, along with a short description of their purpose.[1] However, please mind that some reflect ideas we have yet to discuss. (for more detail, please see subsection 3.2.3: *Boilerplate* (Page: 12))

Listing 3.1 – Creates an `webpack-bundle-analyze` session against the production build of the client bundle.

```
1 yarn run analyze:client
```

Listing 3.2 – Creates an `webpack-bundle-analyze` session against the production build of the server bundle.

```
1 yarn run analyze:server
```

Listing 3.3 – Builds the client and server bundles, with the output being optimized.

```
1 yarn run build
```

Listing 3.4 – Builds the client and server bundles, with the output including development related code.

```
1 yarn run build:dev
```

[1] Also, when a developer attempts to push to the GitHub repository, our unit-test are automatically performed; aborting the push upon failure.

Listing 3.5 – Deletes any build output that would have originated from the other commands.

```
1 yarn run clean
```

Listing 3.6 – Deploys your application to `now`.

```
1 yarn run deploy
```

Listing 3.7 – Starts a development server for both the client and server bundles.

```
1 yarn run develop
```

Listing 3.8 – Executes `eslint` against the project.

```
1 yarn run lint
```

Listing 3.9 – Executes the server. It expects you to have already built the bundle-susing the `yarn run build` command.

```
1 yarn run start
```

Listing 3.10 – Runs the `jest` tests.

```
1 yarn run test
```

Listing 3.11 – Runs the `jest` tests and generates a coverage report.

```
1 yarn run test:coverage
```

3.2 Preliminary Design

The application itself is quite a large undertaking, as it comprises a multitude of different aspects. Most of which can be categorised into one of three groups: game mechanics, network communications, and artificial intelligence. Therefore, where possible, we shall try to discuss each one of these categories separately. But first of all, we shall introduce the reader with an overview of the application and describe the relation between some of its core components.

3.2.1 Architecture Overview

Much of what has been designed is derived from ideas that relate to the game as a whole, that being the overall encompassing architecture. To aid the reader in understanding said content, we shall introduce them to a brief summary of the foundational ideas in which the game is to be built upon.

At its heart, the game follows a client-server multiplayer game architecture. This is where a single device is designated to be responsible for processing user-input, updating the game state and communicating said game state to the connected players. The connected players are principally responsible for relaying input to the server and rendering graphics based upon the state communicated from the server.

3.2.2 Core Technologies

As the game is to be played in a web-browser, we will be making use of the three fundamental web languages; HTML, CSS, and JavaScript. These three languages, and the standards which govern them, allow us to confidently write portable code which will run predictably within the majority of web-browsers. Although, sadly this is not always the case as it is notorious for many web-browsers to not always fully support the most up-to-date standard - assuming it follows it in the first place.

The primary programming language for our client-side source-code is JavaScript the standard core web-browser technology for web-page DOM manipulation. To help tackle the compatibility issue mentioned above, all our JavaScript will conform according to the ECMAScript2015 standard (Ecma International, 2017). It shall then be transpiled to a more universally supported syntax using Babel (Babel (Open Source), 2016).

Transpiling to JavaScript has become somewhat the norm in the realm of modern web-development. In recent years, many entirely new programming languages have emerged for the sole purpose of being transpiled into JavaScript. For this application, the decision to use an updated standard of JavaScript, as opposed to one of these entirely new languages, is to reduce the amount in which we abstract from the underlying code that is to be executed; removing another layer where issues could arise.

As our game is designed to be played within a web-browser, we already have access to the wide range of graphical components and other features described within the HTML specification. Notably, this includes elements such as buttons, lists, hyper-links and CSS - which is used to describe the presentation of our web-document.

As the logic and state of our game exists within JavaScript, we will need

to frequently manipulate these HTML elements - such that a proper reflection of our game state is maintained. This type of structure closely resembles the Model–view–controller (Wikipedia, the Free Encyclopedia, 2017d) software architectural pattern for implementing user interfaces. However, handling the view layer can prove to be quite a cumbersome task as many naive solutions scale very poorly, such that future adjustments may have adverse side affects or may just be very awkward to implement; making correctness difficult to conserve.

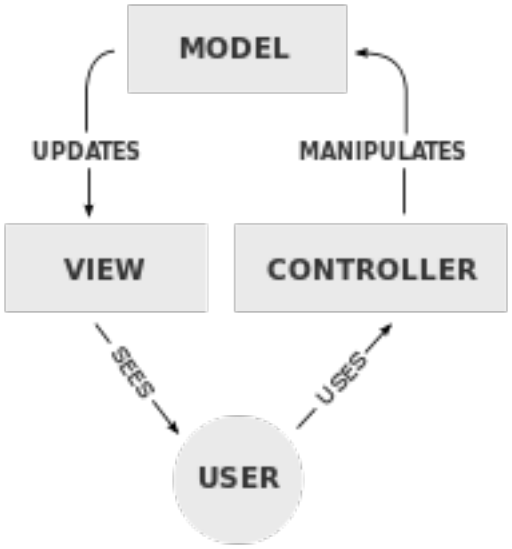

Figure 3.1 – *Diagram depicting the Model–view–controller software architectural pattern, from (Wikipedia, the Free Encyclopedia, 2017d).*

Thankfully, this is a very common problem faced during web-development, and there already exist many capable candidate solutions. With this in mind, along with the desire to learn a new framework, and the curiosity regarding its applicability in integration with this project; React(Facebook Inc., 2017a) and Redux(reactjs, 2017) have been employed to help manage the view layer.

React is a popular JavaScript framework that makes it relatively easy to create advanced user-interfaces for web-applications. It allows you to design your application as a set of simple views for each state in your application, and will efficiently update and render only the appropriate components when data changes. In essence, this is done by constructing almost your entire application's view layer within JavaScript by building encapsulated components - each charged with managing their own state.

Redux is a separate library, but plays very well when used in conjunction with

React. Simply put, it is a predictable state container for JavaScript applications. It aids in allowing the developer to write applications that behave in a consistent manner, whilst providing tools to improve the developer's experience in regards to processes such as debugging.

Redux, in principle, requires the application's state to be stored in an object tree inside a single data *store*. Each state is represented by a single immutable object. Changes to the state tree are made by emitting *actions*. An *action* is an object describing what happened during some event to the state. These *actions* are then used to transform the state tree through the use of *reducers*; a pure function, with `(state, action) => state` signature, describing how an action transforms the given state into the next state.

Given that we already have to use JavaScript for our front-end game code, it makes sense to not bring in another language for our back-end. This bears a myriad of benefits, such as not having to deal with the additional quirks of a separate programming language. Also, it eliminates the hassle of maintaining two separate implementations of code that are identical in purpose. This becomes especially relevant later on, when we introduce client-side prediction (see section 3.5: *Network Communication* (Page: 20)). Hence for the back-end, that is the code which will be executed on a dedicated server, we shall be utilising Node.js(Node.js Foundation, 2017b).

Node.js, is a technology which serves as a JavaScript run-time environment allowing JavaScript to run in standalone, outside of a web-browser. It has become a major proponent of the 'JavaScript everywhere' paradigm, allowing the meat of web-application development to unify around a single programming language. A concise description is provided on the front-page of its website:

> 'Node.js® is a JavaScript run-time built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.' - Node.js Foundation, 2017b

3.2.3 Boilerplate

Somewhere early on during development, having already settled on the aforementioned technologies, it began to be quite difficult to maintain an efficient development workflow. That inspired the decision to transition the project to the use of the React Universally (ctrlplusb, 2017) starter-kit; a boilerplate project with a bare-bones base structure along with various scripts to help automate the tedious build/watch transpile process required for our JavaScript code.

React Universally also sets up Server-side rendering with React. This is helpful as it enables the server to perform an initial render of our application's components,

then serve the result to the client. This results in the web-page appearing to load faster, as the main layout would've been pre-rendered; so the client user does not experience the initial flickering of DOM elements as the view is mounted.

Regarding the Redux store, the state is simply injected directly into the web-document prior to being served to a client. Upon initialisation, the client immediately rehydrates the application using the transmitted state and mounts the React components.

The source code directly composing our application is split into three main directories: *shared*, *client*, and *server*. The *shared* directory contains the bulk of our application, including source which is rendered server-side to be served to a client. Whilst the *client* directory is for browser specific source that is not to be used server-side, not even for the purpose of server-side rendering. In particular, it includes functionality such as user-input and establishing a live communication session with the server. In a similar respect, the *server* direction contains all the source specific to our Node server.

3.3 User Interface

It should be declared, once again, that the primary focus of this project is not to create a polished video-game, but to simply explore the related techniques and their integration within the realm of modern web-development. However, in reality, if a video-game's user-interface is sub-par, the likelihood of the user having a positive experience will be highly diminished. It is for that reason, we shall discuss some core aspects related to user-interface design and it's correlation with this project.

The user-interface is not solely related to the layout and style of graphical components, but also about how their behaviour can impact the user's experience. In particular, the way in which they're expected to interact an application can be a detrimental factor.

There is also no single solution to user-interface design. Every scenario is different, with its own unique specifications, and must be treated as such. However, there are general guidelines and conventions which are commonly found to be incorporated into designs. Therefore, we must develop a design that accommodates the unique characteristics of Tron.

With this taken into account, along with inspiration drawn from analysing the existing browser-games (see section 2.2: *Existing Games* (Page: 4)), the following set of interface design specifications have been devised:

1. **Single-page web-application**: the interface should be a single HTML page, which dynamically updates its content. This is to avoid irritating the user with page reloads differing by only small amounts of content.

2. **Responsive layout**: the interface should scale nicely on a variety of devices - accommodating different resolutions. This is done through the use of techniques such as CSS media queries and relative units.
3. **Conventional user input**: as the game is quite simple, the average user should almost instinctively know how to operate it. Primarily, this is done by sticking to convention; such as *WASD* keys in order to move.
4. **Lobby invitation**: the interface should provide users with a very straightforward way in which they're able to play a game of Tron with their friends.
5. **Simplistic URL structure**: the interface must make intuitive use of the site's URL, as the game will be a web-application; so should conform to convention.

After careful assessment of the above criteria, it became possible to develop some concepts and realise them prototype designs. Throughout the remainder of this section, we shall elaborate on these prototypes.

We conceived the idea to solve both the 'Lobby invitation' and 'Simplistic URL structure' criteria with a single solution. This solution is to allocate the entire URL path to represent a *key* to a single particular Tron game lobby. That is, if a player wants to invite his friends to a game; they would simply append some arbitrary sequence of characters to their URL then forward it to their friends. This solution is very easy for the user to understand. Although, the *privacy* of their game is dependent on the predictability of the chosen lobby key.

In Tron, it is pivotal for there to be minimal obstruction towards the user when they wish to apply their ability to redirect their player within the game arena. This rules out the use of any form of button, due to the unacceptable latency involved in positioning the cursor. Instead, the user shall control their player using the *WASD* keys (that is W: north, A: west, S: south, D: east). Or, as an alternative for mobile touch devices, the user is required to simply tap the portion of the screen reflecting the direction they wish to now travel along.

3.4 Game Mechanics

Compared to other arcade games, Tron has relatively few fundamental game mechanics. However, with that being said, there are still many challenges that arise due to the game's fast-paced nature and critical requirement for accuracy. Throughout this section, we shall discuss some of the more integral problems regarding the game's core mechanics along with the designed solutions, and the techniques in which they use.

Before we delve into the specifics, it is vital we first provide an overview of the game's state object. In video-game design, the game state refers to an object - or other data store - which contains all the data representing a game instance at some particular point in time.

Below is a breakdown our game state object's structure:

tick : the number of times this state has ticked; inclusive of the current tick update.

progress : the amount of time which has passed since the last tick update.

started : a boolean indicating if a round of Tron is in progress.

finished : a boolean indicating if the current round has finished. *undefined* if started is *false*.

arenaSize : the number of cells within our game arena.

playerSize : the number of cells a player occupies within our game arena.

speed : the number of cells each player travels over the course of a millisecond.

players : an array containing an object for each player part of the current game. The following describes the structure of a single player object:

- **id** : a unique string used to identify the player.
- **name** : an arbitrary string for other players to recognise this player.
- **alive** : a boolean indicating if the player is alive, or otherwise dead.
- **direction** : the direction (north, south, east, or west) in which the player is travelling.
- **position** : a *point* (an array, in the form of $[x,\ y]$), representing the player's current position in the grid.
- **trail** : an array of *points* at which the player has changed direction. Used to construct a path of the area in which the player has travelled. However, the array has two special cases: the first element is the player's spawn point, and the last element is the point for the player's previous position.

cache : a nested object containing various cache structures required by our state. The following describes the structure of the cache object:

- **collisionStruct** : a data-structure in which we can check for player/trail collisions within our arena. See for more detail subsection 3.4.1: *Collision Detection* (Page: 17).

Our game state itself must be implemented as a mutable object. This is due to

the large overhead that immutability generates; primarily from computationally expensive processes, such as copying an abstract data-type. As immutability is optional, it therefore is simply not worth introducing it within the game loop - as it would degrade performance.

Although having the game state be a mutable object does not directly interfere with our server, it does raise concern in regards to our client and their utilisation of Redux. This is because one of Redux's key principles, is that all data held within the *store* must be immutable, such that all modifications to the state are performed solely by *reducer* functions.

To avoid creating an unnecessary reliance and coupling of the project onto optional technologies, we do not want to meld our internal game code with Redux. Hence, for our client, when we receive a game state update from the server we copy it and commit both the original and copy into the *store*. One of these stored game states will remain immutable, whilst the other will be fully mutable - hence unaware of updates.

This solution may currently appear rather strange, as at this stage it is too early to introduce the entire reasoning. However, simply put, it allows us to keep an authoritative state (what is known) and a predictive state (what should be). Our DOM will reflect the authoritative state, whilst the drawn graphics of the game will reflect the predictive state (for more information, see section 3.5: *Network Communication* (Page: 21)).

The next major aspect of our game is the game loop. In principle, it is a section of code that runs continuously during game-play, at some interval. User-input is processed, without blocking, during each *tick* of the loop. The remainder of the *tick* is then used to update the game's state and render graphics. It of critical importance, and is arguably the most employed pattern in game-design as it provides a very convenient and deterministic way in which the developer can structure their game.

Our game shall be utilising the game loop, and is heavily based around said pattern. In principle, the server needs to spin the game loop at a rate fast enough to process user-input and calculate updates without introducing noticeable latency. Whilst the client is only required to spin fast enough to create the illusion of animated graphics.

By default, both client and server will have their game loop configured to run at a tick-rate of once every 15 milliseconds - approximately 66 times a second. [2]

[2] This is considered the standard for video-games. In most cases, a faster rate does not provide any distinguishable benefit.

3.4.1 Collision Detection

Many existing Tron implementations internally use a grid-based arena, that is a player occupies an entire single cell of the arena. This enables collision look-ups to be performed in constant time, by simply indexing an array. However, our implementation allows players to move with an incredibly high-degree of precision (Refsnes Data, 2017). Unfortunately, this does complicate collision lookups.

Instead of checking for collisions against every object in the arena, we have utilised the uniform grid spatial data-structure. This allows us to divide the arena into a grid of arrays, where each array holds references to the objects that reside within its bounded space. Thus, when checking for a collision, we only need to check against objects that are held within the array(s) that our target object intersects with; reducing the search space quite substantially.

To populate our collision data-structure, we generate a series of rectangles representing each individual line-segment that forms the player's trail - from their current position, to the position they were spawned at. This is the stage at which we take into account the players' size.

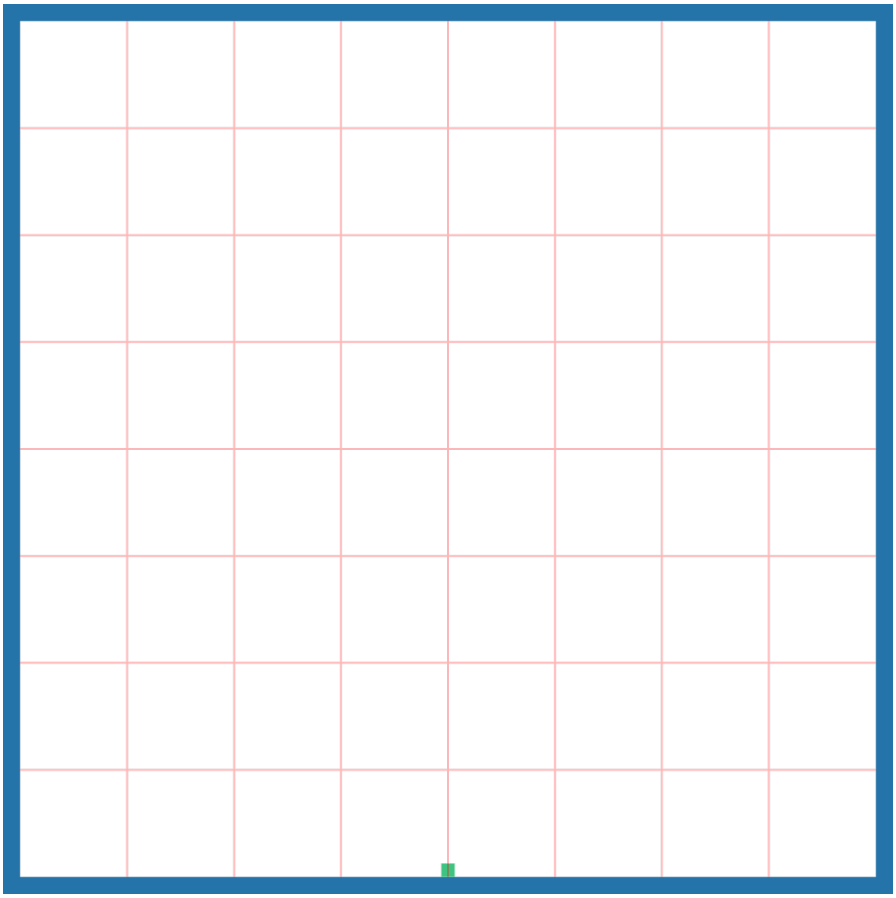

Figure 3.2 – *Visualisation of uniform-grid data-structure, drawn by our game's draw debug mode.*

Once the search space has been reduced, we now need to check to see if any of the obtained objects collide with our target object. Conceptually, this is just checking if there exists an overlap between two rectangles; a very simple calculation. However, as we want players to stop at the precise moment at which they crashed, we must calculate the intersection point and then offset it appropriately using the size of the two players.

This boils down to two distinct cases: [3]

1. **Case**: player collides with another player - who is not heading in an opposing direction.
   Solution: reposition the crashed player by appropriately calculating an offset distance from the player they hit - based upon both their sizes. Using the crashed player's travelling direction for the offset's sign.

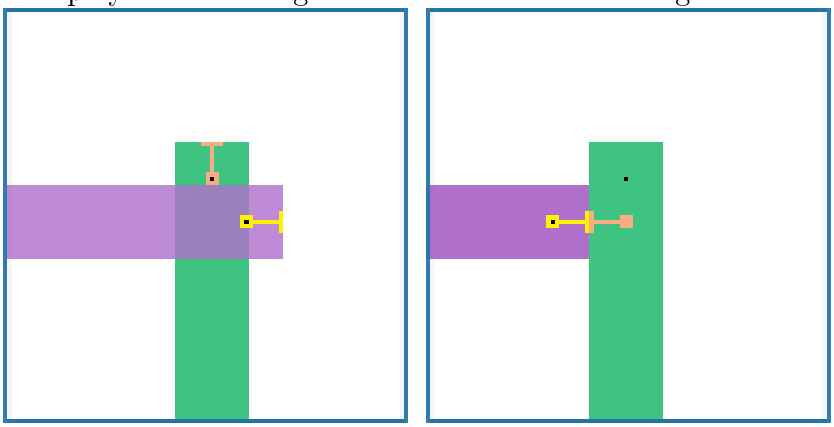

Figure 3.3 – *An example diagram demonstrating a sideways collision between two players. On the left, they're depicted in their unfixed positions. Whilst on the right, they're now shown in their fixed positions.*

2. **Case**: head-on collision between two players.
   Solution: move each player backwards by half the sum of their *overlap* and *overshoot*.

[3] Please note that, in our collision case diagrams, the black dot within a player represents their position point.

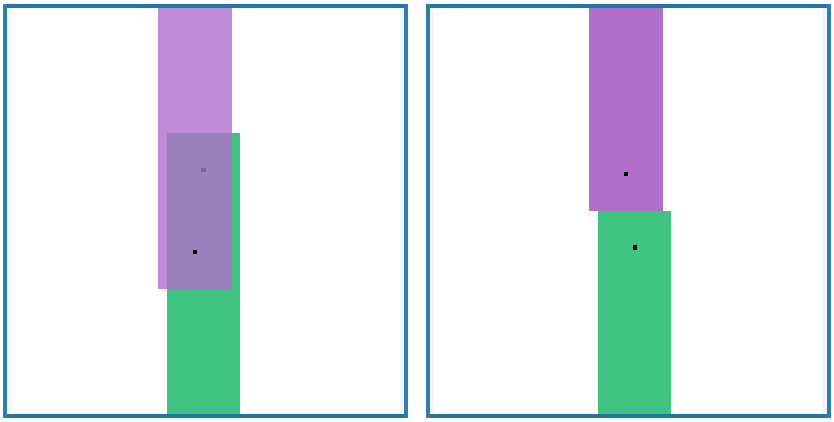

Figure 3.4 – *An example diagram demonstrating a head-on collision between two players. On the left, they're depicted in their unfixed positions. Whilst on the right, they're now shown in their fixed positions.*

3.5 Network Communication

For a game to be multiplayer, all players need to share the same consistent experience across a network, i.e. they need to all be playing on the same game state. So this raises the question on how to synchronise the game state between all connected players. There are two main methods to achieve a synchronised game state and, to some degree, we will be using ideas from both.

The first method is known as *peer-to-peer lockstep*. It centres around the idea of the game being modelled as turn-based and, before each turn, all non-deterministic events (such as user-input) are broadcasted to all players. Once all players have sent their input for this turn, each player then individually updates their game state; which will of course all end up being identical. The primary flaw in this technique is that all players are forced to play at the latency of the player with the weakest connection. This would be extremely frustrating for Tron, as each time you try to move you have to wait for the player with the weakest connection.

The second method is known as *client/server*. It entails having a single authoritative game-state kept on a server, that is then communicated between all players. When a player wants to perform an action, they must communicate said action to the server. The server will then apply the action and communicate the updated state to all connected players. However, this solution makes no compromises for latency. For example, the server's state is forever being updated and it will often receive actions from connected players that were made under the assumption the game is still at some previous state.

Our developed solution entails the use of several techniques. At its core, it is a

client-server model. But we make use of two mechanisms known as *lag compensation*(Valve, 2011a) and *client-side prediction*(Valve, 2011b) which help to reduce the negative effects of latency.

Simply put, lag compensation enables the server to 'rewind' time when applying the input of a user; compensating for any latency that may have occurred. Whilst lag prediction allows clients to mimic the server, including the immediate processing of their input. See (Bernier, 2011) for a more thorough description.

Communicating the entire game state after each tick is an incredibly infeasible task; each player would be required to communicate with the server at a rate faster than the game's tick-rate.

Our solution to this problem is quite a simple one. When the client receives a message containing the updated state, they immediately reply to the server with an acknowledgement. The server will interpret this acknowledgement as a request to prepare the next state transmission.

To further reduce the latency of communication, we can also reduce the payload size of each state transmission. This is made possible by identifying that over the course of several ticks, the game state remains largely unchanged. Knowing this, we are able to keep in memory the last state communicated to each player and when we need to transmit, just calculate a snapshot of the differences [4] between the current state and the last one which was sent. We also don't bother sending the cache, and instead simply regenerate it on the client - which isn't too expensive of an operation, and doesn't hinder our server's authoritative game state.

As introduced within the background section (see section 2.3: *Browser Technologies* (Page: 5)) there exist several technologies capable of handling the actual communication with a web-server; most notably WebSockets and Web-RTC. However, we have chosen to use WebSockets as our communication technology. This is because it is well supported, well documented and is does not provide us with extra hassle as we already have a dedicated game-server.

3.5.1 Game Lobbies and Concurrency

Our web-application is intended to support the simultaneous play of numerous Tron game lobbies. This is to allow separate groups of users to each be allocated their own game instance, allowing them to play with one-another - away from undesired guests.

When a user visits our web-application's site, the server will receive a *HTTP GET* request and serve them the necessary files. Once the client receives said files, they begin to render and mount the React view.

As a reminder, the URL path is interpreted as the unique game lobby key (see

[4] We calculate and apply the differences using *fast-json-patch*(Starcounter-Jack, 2017).

section 3.3: *User Interface* (Page: 13)). Hence, once the view has been mounted, if the client's URL path is not blank a WebSocket's connection is then established with the server. The client will then request to the server that they wish to join said lobby.

Once the server receives a request from a client, who wishes to join a game lobby, it will check if the lobby corresponding to the submitted key currently exists. If not, it is created. The server will then add the requesting client as a player in said lobby.

Once registered with the game lobby, the client is added as a player within the current game state and then is sent a full copy of said state. This update to the game state is distributed to the other existing lobby clients identically to any other update (i.e. via our snapshot system, see section 3.5: *Network Communication* (Page: 20)).

The game lobby contains a special object called the state controller. This object keeps a history of all the states currently held within our lag compensation's history. It is also responsible for spinning our game lobby's game loop and handling all updates to the game state.

However, Node.js is a single-threaded environment, which means that only one request can be processed at any given time. This requires all forms of I/O, and other computationally expensive tasks, to be performed asynchronously; to avoid creating a lock on Node's event loop.

Knowing this does create concern, as our game will include tasks that are very computationally expensive - such as the aforementioned state updates, along with others including artificial intelligence. However, Node.js is equipped with a module, called `child_process` (Node.js Foundation, 2017a). This module is of vital importance, as it enables us to delegate the computationally expensive tasks onto a separate process that shall be executed in parallel - without blocking our main Node.js event loop.

Therefore, our state controller will delegate all state updates to a dedicated Node.js child process; keeping the event loop of our main Node.js process spinning fast.

3.6 Artificial Intelligence

In order for our computer opponent to play the game, they need to be controlled by some artificial intelligence; some program which has an understanding of the game and can make sensible moves, as if it were a human player. This is quite a complicated, especially given that our game is played in real-time.

Our solution to this problem centres around the idea of the modelling the game to be both played on a grid and turn-based, then running simulations which play

out the possible scenarios in which the game could develop. However, as we require our AI to make decisions within a very short amount of time, we are unable to check the entire search-space and instead must make use of a variety of techniques to help optimise the process.

Probably the most radical technique is to only run each simulation up until some fixed depth, at which point we then evaluate the current game state using an heuristic evaluation function. The effectiveness of this technique is heavily reliant on both the chosen fixed depth as well as the evaluation function.

The heuristic evaluation function we've designed uses an optimised version of the flood-fill algorithm, featured in the very high-performing implementation that was the winner of a Google AI competition - introduced in section 2.4: *Artificial Intelligence* (Page: 5). The core idea is to count player *ownership* of grid-cells, in our model of the arena, based upon a heuristic distance measurement.

First, all players have their heuristic score initialised to 0. Then we calculate the minimum Manhattan distance between the AI player and all other alive players. We then apply flood-fill to calculate the distance from the current player to each empty grid-cell. Flood-fill is optimised by halting its process once the distance surpasses the previously calculated minimum distance. We are now left with a heuristic distance measurement.

For each grid-cell, we then apply the following process to score based upon cell ownership: for each player, increment 1 to their score for every other player that has a distance that is greater-than, or equal to, the current cell. If the other player does not have a recorded distance to the current cell, instead increment by 2.

Thus, a greater score indicates a stronger position; although not relative to other players.

Even with the reduction in search-space and optimised evaluation function, time is still very limited. In order to avoid wasting such time, we disregard branches which are not promising. This is done through the use of the Monte Carlo tree search algorithm (Wikipedia, the Free Encyclopedia, 2017e) with UCB1.

However, UCB1 takes into account *wins* and *loses*. Clearly, this conflicts with the idea of running simulations to some fixed depth. To combat this, when we reach the end of a simulation, we decide whether the said simulation was a *win* or a *loss* based upon some naive rules:

- If our player is dead, increment the losses by the total number of alive players.
- Otherwise, increment wins by 1 (to reward being alive) plus 1 for each player that is either dead or have a lower heuristic score.

On top of all of this, the simulations involved in calculating the artificial intelligence's move run in parallel to the rest of the game-server. This is done by

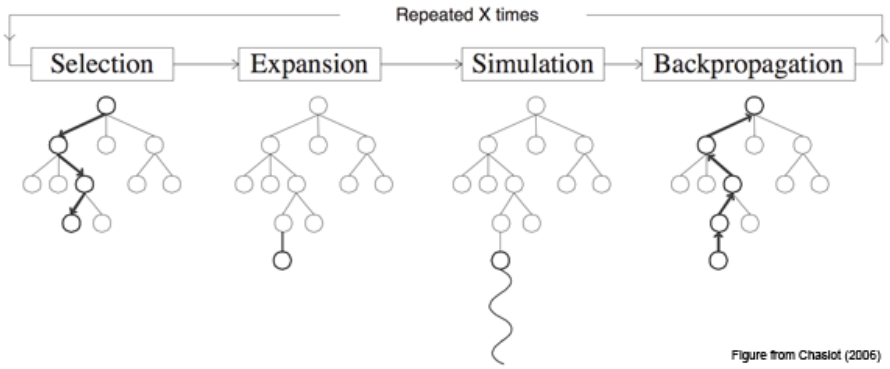

Figure 3.5 – *Diagram depicting the Monte Carlo tree search algorithm, from (ziggystar, 2012).*

Let t be the total number of simulations that have involved node s, let n_i be the total number of times the move to s_i was chosen as the next move (i.e. $t = n_1 + \ldots + n_k$), and let w_i be the number of times this led to a win. If the simulation ends up in node s, and s is not a final state of the game, then as the next move, we choose the one that ends up in node s_i such that the following score is maximised:

$$UCB1 = \frac{w_i}{n_i} + \sqrt{\frac{2 \ln t}{n_i}}$$

Figure 3.6 – *A description of the UCB1 formula and its application with regards to a turn-based game-tree.*

spinning off a child process, which will run on a separate core. Once a move has been calculated, it will communicate the result back to the server and then die.

The server will then apply lag compensation, once it receives the communicated move. This effectively allows simulations to take longer than the duration of a single tick; leading to moves that are generally better.

Chapter 4

Implementation

Both the design and implementation processes occurred simultaneously, that is the implementation of our application evolved alongside the development of its final design. During the design chapter (see chapter 3: *Design* (Page: 7)), we only discussed the final state of the system. This was done to avoid creating any confusion with regards to what was actually created. However, throughout this chapter we will be discussing some of the mistakes made, difficulties faced, and the solutions allowing us to overcome them.

4.1 Source Overview

Great effort has been invested into ensuring that our implemented source-code is of a high-quality. Among other aspects, this entails conscious attention to certain factors, such as maintainability and reusability. This is especially important due to the multifaceted nature of the project.

In particular, certain software development principles have been utilised, such as high-cohesion and low-coupling. This results with an implementation that is both easier to understand, and whose components can be easily integrated into future projects.

Throughout the remainder of this section, we shall provide the reader with an overview of our application's source-code. This is to help establish an understanding of what exactly has been developed, and how the implementation is structured. However, for an in-depth description regarding specific detail of our code, we advise the reader to further their understanding by consulting said source-code directly. In order to ease this, high amounts of documentation have been embedded throughout the codebase.[1] [2]

[1] We've omitted all unit-tests to try and keep this list concise.

[2] In order to retain focus on our project's primary aspects, we avoid delving too far into React

./ : the root directory of our project.

- **./.babelrc** : configuration file for Babel (see subsection 3.2.2: *Core Technologies* (Page: 10)).
- **./.editorconfig** : configuration file for EditorConfig, providing a means for developers' to define and maintain consistent coding styles between different editors and IDEs.
- **./.env_example** : example configuration file describing environment variables intended for use during deployment. This is provided by React Universally.
- **./.eslintignore** : configuration file containing patterns identifying specific files that we intentionally do not wish to lint. This includes third-party and post-build source-code.
- **./.eslintrc** : configuration file for our JavaScript linter (see section 3.1: *Software Development Process* (Page: 7)).
- **./.modernizrrc** : configuration file for Modernizr. This allows us to detect the supported technologies within a client's web-browser. This is provided by React Universally.
- **./.nvmrc** : configuration file stating this application's intended version of Node.js.
- **./LICENSE** : copy of the application's licence, that being *GNU GENERAL PUBLIC LICENSE Version 3*.
- **./package.json** : configuration file for Node.js containing much metadata regarding our application, such as dependencies and scripts. Extended from what is provided by React Universally.
- **./README.md** : short piece of text introducing developers to the application.
- **./TODO.todo** : todo-list containing a log of tasks completed throughout development (see section 3.1: *Software Development Process* (Page: 7)). This became somewhat disregarded towards the later stages.
- **./yarn.lock** : log of the exact dependency versions, for predictable portability. This is provided by React Universally.

build/ : output directory containing our source-code once it has been transpiled and passed through a module bundler. We refrain from discussing its contents in further detail.

Universally (see subsection 3.2.3: *Boilerplate* (Page: 12)).

client/ : directory containing browser specific source that serves no purpose server-side, not even for server-side rendering.

- **index.js** : kickstarts our application within the client's browser. Extended from what is provided by React Universally.
- **registerServiceWorker.js** : installs the offline plugin, which instantiates our service worker and app cache to support pre-caching of assets and offline support. This is provided by React Universally.
- **components/** : directory containing the React components not intended for server-side rendering.
  - **ReactHotLoader.js** : enables React hot-loading for development builds of our application. This is provided by React Universally.
- **game/** : directory containing all the Tron game functionality that is strictly only required client-side.
  - **draw.js** : draw functionality that attaches to the canvas element of our document, enabling it to draw the graphics for our game of Tron.
  - **drawdebug.js** : optional draw functionality that overlays the standard Tron graphics to provide the user with a visualisation that is more useful for debugging. In particular, highlighting the bounds for each node within our collision data-structure.
  - **gameloop.js** : a version of the game loop optimised for the web-browser (for more information, see subsection 4.2.1: *Game Loop* (Page: 34)).
  - **polyfills/** : directory containing polyfills that offer workarounds for clients whose browsers lack certain functionality. This is provided by React Universally.
    - **polyfills/index.js** : example polyfill for clients that lack the picture element.
  - **state/** : directory containing Redux functionality that powers the state of our application (see subsection 3.2.2: *Core Technologies* (Page: 10)).
    - **index.js** : entry point file which sets up our root Redux reducer and root Redux Saga (see section 4.2.8: *Redux* (Page: 40)).
    - **input/** : directory containing Redux functionality for processing user-input.
      - **host/** : directory containing the Redux functionality usable by the host user to configure the game lobby.

sagas.js : Redux sagas used to execute actions on behalf of the game host for the current game lobby.

keyboard/ : directory containing the functionality required for Redux to emit actions on keyboard input.

actions.js : Redux actions for the key-down and key-up keyboard user-input events.

sagas.js : Redux Saga which forks out to constantly listen for keyboard input from the user; distributing actions for the appropriate events.

player/ : directory containing the Redux functionality usable by the user in order to control their player.

sagas.js : Redux sagas used to execute actions on behalf of the user, so that they're able to control their game player.

lobby/ : directory containing Redux functionality enabling the user to communicate with their game lobby.

sagas : Redux sagas used to send/receive messages between client, and the server holding the game lobby.

sockets/ : directory containing the means allowing Redux to communicate, in a natural manner, with a server via WebSockets (see section 4.2.8: *Redux* (Page: 40)).

actions : Redux actions used to establish and communicate using a WebSocket connection.

reducers : Redux Reducer which attaches the WebSocket connection status to our Redux store's state.

sagas : Redux sagas facilitating integration between WebSockets and Redux.

config/ : directory containing general configuration for our web-application and the build/deploy process. Extended from what is provided by React Universally.

internal/ : directory containing most of the internals powering React Universally.

node_modules/ : directory containing many of the third-party dependencies, as specified within our `package.json` file.

public/ : directory containing miscellaneous static files useful for hosting a web-application. This is provided by React Universally.

server/ : directory containing functionality specific to our Node.js server.

index.js : kickstarts the server component of our application. Extended from what is provided by React Universally.

sockets.js : initiates the WebSocket functionality and hooks up our Tron game server, supplying the necessary Node.js dependencies for dependency injection.

game/ : directory containing all the Tron game functionality that must be specialised for a particular platform, in this case Node.js. This will interface with the remaining game files using dependency injection.

gameloop.js : a version of the game loop optimised for the Node.js run-time environment (for more information, see subsection 4.2.1: *Game Loop* (Page: 34)).

processes/ : directory containing the entry-points for our Node.js child processes (see subsection 3.5.1: *Game Lobbies and Concurrency* (Page: 21)).

aimove.js : child process entry point used by a computer player in order to determine a move calculated using some artificial intelligence.

update.js : child process entry point used by our game's state controller to update the game state.

middleware/ : directory containing middleware for our Node.js server. Extended from what is provided by React Universally, in order for compatibility with Redux.

shared/ : directory containing the bulk of functionality developed for our application. Its contents are intended for use both on the client and server.

components/ : directory containing our React components which compose the view layer.

App/ : directory containing the components specific to our application.

index.js : base component of our application.

globals.css : CSS style rules which apply to our application as a whole.

Error404 : directory containing components for a simple 404 page. This is provided by React Universally.

GameCanvas/ : directory containing the components for the canvas element which we draw Tron's graphics onto.

index.js : the component representing our game canvas, including the functionality enabling it appropriately adjust in

size.

Lobby/ : directory containing the components used to create a panel of information and options for when the user is connected to a lobby.

index.jsx : base component for our game lobby panel. This component will initiate the WebSocket connection to the appropriate game lobby.

host/ : directory containing the components of our game lobby panel's host section, providing additional configurations to users who are the host of their current game lobby.

index.js : base component containing the graphical controls for host operations.

Welcome/ : directory containing the components used to create a welcome screen for when the user is not connected to a lobby.

index.jsx : base component containing a brief introduction to our Application along with the ability to set the player name and join a random lobby.

HTML/ : directory containing a generic component used as the foundation of a HTML document. This is provided by React Universally.

game/ : directory containing the majority of Tron's game code.

gameloop.js : an abstract class of our game loop, intended to be extended upon by adding a platform specific timer.

ai/ : directory containing the functionality powering our computer players through the use of artificial intelligence (for more information, see section 3.6: *Artificial Intelligence* (Page: 22)).

index.js : perform some checks before initiating the simulations and returning the calculated move.

minimax.js : deprecated simulation technique (for more information, see subsection 4.2.5: *AI Simulations* (Page: 36)).

montecarlo.js : implementation of Monte Carlo tree search used to identify a strong move.

wintree.js : data-structure used by our Monte Carlo tree search to keep a history of the executed simulations and their outcomes.

heuristics/ : directory containing the implemented heuristic evaluation functions which provide insight of the game state.

index.js : calculate the heuristic scores for each player by using the distance map provided by the flood-fill algorithm.

floodfill.js : perform our optimised flood-fill algorithm for a single player within the game arena.

network/ : directory containing the core functionality powering the architecture enabling our game to be multiplayer (for more information, see section 3.5: *Network Communication* (Page: 20)).

lobby.js : class charged with managing an individual game lobby.

server.js : class that processes client connections and arbitrates players within game lobbies.

snapshot.js : utility functions to obtain and apply snapshot comparisons that are taken between two states.

statecontroller.js : class which handles the authoritative game state (asynchronously) and can apply changes utilising lag compensation.

input/ : directory containing the various attachments to the player-server connection. These are primarily used to read messages sent from user players in order to perform some operation on the game lobby.

index.js : attach a player to all other attachments contained within this directory.

host.js : attach events to a player allowing them to perform host operations.

player.js : attach events to a player allowing them to direct their player.

operations/ : directory containing a collection of handy operations that are applied to the game state.

collision.js : operations used for the purpose of collision detection, including a function used to create the rectangle objects representing a line-segment composing player trails.

general.js : general-purpose operations, including those to initialise the game state and rebuild the state cache.

player.js : operations relating to the game state players, including their creation, removal, and safe repositioning.

update/ : directory containing the various tasks performed during each tick of our game loop.

- **index.js** : entry point to our game loop tick updates. At the end, this will also check if the current game round has reached termination.
- **collision.js** : apply collision detection against our game state (see subsection 3.4.1: *Collision Detection* (Page: 17)).
- **move.js** : move all alive players by the appropriate distance.

utils/ : directory containing utility functionality that our game requires.

- **geometry.js** : contains some geometry functions used throughout our game, particularly by our collision detection.
- **spawn.js** : calculates the spawn position for a particular player within the game instance.
- **collision/** : directory containing the data-structures utilised by our collision detection.
  - **grid.js** : implementation of a uniform grid.
  - **object.js** : standard object interface used by collision data-structures.
  - **quadtree.js** : implementation of a quadtree.

state/ : directory containing the shared-scope Redux functionality that powers the state of our application (see subsection 3.2.2: *Core Technologies* (Page: 10)).

- **index.js** : sets up the shared root Redux reducer and root Redux Saga (see section 4.2.8: *Redux* (Page: 40)).
- **configureStore.js** : set-up or Redux store to integrate with React Universally and its development workflow.
- **input/** : directory containing the Redux functionality enabling a by the user in order to control their player.
  - **host/** : directory containing the Redux functionality usable by the host user to configure the game lobby.
    - **actions.js** : actions which are emitted to the store in order for the user to perform host operations.
  - **player/** : directory containing the Redux functionality usable by the user in order to control their player.
    - **actions.js** : actions which are emitted to the store in order for the user to perform operations regarding their player.
- **lobby/** : directory containing Redux functionality enabling the user to communicate with their game lobby.

actions.js : actions relating to the user and their relationship with the game lobby.

reducers.js : reducers relating to the user and their relationship with the game lobby.

utils/ : utility functions provided by React Universally.

4.2 Notable Challenges

4.2.1 Game Loop

Constructing the game loop is a seemingly simple task. In essence, it is a callback that some timer would make a call to at some fixed interval equal to our tick-rate. However, there were a number of quirks related to JavaScript's standard API which made this task not as straightforward as initially expected.

On the client, we are able to use JavaScript's *window.requestAnimationFrame()* (Mozilla Developer Network and individual contributors, 2017b) function. This is called whenever the canvas is preparing to draw the next frame, giving us the opportunity to update the game state - basically hijacking the internal loop used within the browser. This technique worked during the early stages of development, but when we sought to run the game on the browser; it soon became apparent that obviously this function wouldn't exist in a Node.js environment. So, a separate solution needed to be realised.

Research turned in the direction of JavaScript's standard built-in timer, *setTimeout()* (Mozilla Developer Network and individual contributors, 2017a). However, after some short tests, said timer was concluded to be improper for the task as it did not tick with sufficiently reliable accuracy - often too late by many milliseconds. However, there existed an alternative function, *setImmediate()* (Mozilla Developer Network and individual contributors, 2017c).

Sadly, this function occupied a high amount of system resources due to it ticking at an incredibly high, and uncontrollable frequency. This resulted with us having to internally disregard calls if a tick was not scheduled.

Thankfully, after extensive research online, it was discovered that this problem has been tackled by others. A proposed solution (timetocode, 2017) was suggested that recommends to use *setTimeout()* as the primary timer function then, when we enter the window of potential error, switch to *setImmediate()* providing us with a far greater accuracy. This seemed to work very well!

4.2.2 Head-on Player Collisions

Within the game loop, during the update process of single tick, we first move each and every alive player. Afterwards, we perform a series of checks to identify which players have crashed. Once all players have been checked, we process said newly crashed players by officially *killing* them - which mercifully only entails switching their alive flag to `false` and reposition them to their theoretical position at the exact moment of impact.

However, once we began testing, a bug was discovered that players would be repositioned to seemingly incorrect locations after a collision. After further investigation, it was clear that this unexpected behaviour only occurred in the case where two players would collide in a head-on collision, that is they were travelling in opposing directions and met one-another.

After further thinking with regards to the problem, a solution was devised that modified the way in which players were repositioned. Instead, we would backtrack a player by an amount calculated by half the sum of the overshoot and overlap of the two colliding players. However, this was only necessary for the case in which a head-on collision had occurred. For more information, please see item 2: *Collision Detection* (Page: 19)

4.2.3 Mutability

Initially, the game's state was developed to be an immutable object; where all functions made no changes to the given game state, but instead returned an entirely new object with the desired changes applied. This helps to reduce obscure bugs and makes the code far easier to test - especially for unit tests.

However, it also required performing a deep copy of the game state prior to any single change to be made. This is an extremely costly operation to be performing within a game loop, specially when it would have to be performed several times per tick. Realising this wasn't viable, the application was modified to internally use a mutable game state and to manually copy it where need be (e.g. for lag compensation, see section 3.5: *Network Communication* (Page: 21)).

Irritatingly, this caused some difficult to trace bugs. One of the most notable of these was related to copying the game state and collision data-structure within our game state's cache. In short, each object within the collision data-structure keeps a reference to the player which created it. When we then copied the game state, the stored objects did not have their player references updated to match those of the newly created player objects themselves. This resulted in faulty collision detection, related to the workings of our lag compensation. However, the bugs have since been fixed.

4.2.4 Collision Data Structures

In the early stages of the game's development, we would check every object within the arena when searching for collisions. Although this method worked well, and didn't result in any noticeable lag, further research into game development revealed how commonplace spatial data-structures were. This highlighted that our current collision detection mechanic was an urgent area for plausible improvement.

Progress on the game soon advanced and the use of the hierarchical structure of a quad-tree was implemented and integrated into our game's collision detection, to help reduce the total search space when checking for collisions. This did show some considerable performance gain. However, after playing the game, it became apparent that trail line-segments were almost always intercepting multiple nodes - making the need for a hierarchical data-structure somewhat meaningless.

Hence, a uniform grid was instead implemented and integrated to help optimised our game's collision detection. A uniform grid also has the perk of being far easier to deal-with and works well for the case of Tron.

Also, to aid in both the creation and debugging of these two spatial data-structures, the game features a special *debug* mode which would visualise the bounds for each node onto the game arena in the form of a semi-transparent overlay.

4.2.5 AI Simulations

As per our design, the game's computer players were powered by an artificial intelligence that ran simulations using a variant of the minimax algorithm (Wikipedia, the Free Encyclopedia, 2017c) that featured alpha-beta pruning (Wikipedia, the Free Encyclopedia, 2017a) to avoid looking down unnecessary branches of the game tree; improving performance.

However, it was taking far too much time in order for the AI to calculate a suitable move. By the time the AI had calculated a move, the game state would've significantly progress; making their move now redundant - often leading to their undesired death.

There existed two clear ways in which this performance issue could be tackled. We could either restrict the breadth of our game-tree search, or restrict the depth. Restricting the depth of our AI's game-tree search exponentially reduced its ability to make educated moves - it performed extremely poorly. On the other hand, restricting the breadth of our game-tree search was highly dependent on the situation the AI player is current in; as many of the possible early moves are never simulated.

This inspired the idea to migrate to using the Monte Carlo tree search algorithm, as discussed in section 3.6: *Artificial Intelligence* (Page: 22). This al-

lowed our AI to only explore branches of the game-tree that seemed promising; ignoring those branches that almost always led to poor outcomes.

4.2.6 AI Concurrency

In the early iterations of the game's artificial intelligence, the move for a computer controller player would prepare to calculate their next move of a tick; limiting the duration of the entire simulation process to at the most <16 milliseconds. However, it soon became very apparent that this was an inadequate amount of time.

This provoked research into the possibility of running our artificial intelligence concurrently with the rest of the game, allowing simulation to take any pre-defined amount of time. Irritatingly, as developing the artificial intelligence was the final stage of the game's development, I then discovered that Node.js is strictly a single-threaded environment. This was an absolutely devastating discovery...

Thankfully, after an extortionate amount of research, Node's `child_process` module (Node.js Foundation, 2017a) was discovered and successfully integrated into our game sever. This was the cause for a lot of hassle, especially in regards to integration the build workflow provided by React Universally.

In short, this Node module allows us to create a fork of an entirely new, and separate Node process capable of communicating with our main process. Not only did this allow us to execute our artificial intelligence's simulations without blocking our game loop, but it also reduced the time in which we kept our Node event loop occupied. It is critical for our Node event loop to not be used for computationally demanding tasks, as they prevent our server from processing time-sensitive requests such as processing actual user-input or even serving up the application's static files.

Despite this, it still seemed as though our artificial intelligence was preventing our event loop from spinning fast enough. After extensive debugging, it was determined that the cause for this delay was due to the fact lag compensation was being applied after we had received the suggested move from our artificial intelligence. This greatly hindered the game's experience, as it made game-play sluggish and inaccurate; obviously this type of behaviour was unacceptable.

This spawned a large-scale redesign regarding how the game lobby class dealt with the game state. Previously, the lobby, and therefore the main thread, would not only be responsible for running our tick updates to the state, but also for applying the sequence of updates required during lag compensation. However, the greater amount of time we have to compensate for equals a greater amount of time it takes to perform the compensation. This caused our Node event loop to get severely, and frequently held-up.

The problem was solved thanks to the development of our `StateController` class. Each game lobby instance would, within its constructor, instantiate a new

instance of this `StateController` class. Within this `StateController` instance would live a separate process that was solely dedicated to applying individual updates to the game state. For a more in-depth description, see subsection 3.5.1: *Game Lobbies and Concurrency* (Page: 21).

4.2.7 AI Timing

Many aspects of the application rely upon the use of concurrency. Concurrency holds many benefits, particularly when it comes to performance. Irritatingly, it massively complicates aspects reliant on determinism. This became particularly evident when implementing the game's artificial intelligence. The artificial intelligence is to operate alongside concurrent tasks, such as updating the game state (see subsection 3.5.1: *Game Lobbies and Concurrency* (Page: 21)) and for even running the AI simulations (see subsection 4.2.6: *AI Concurrency* (Page: 37)).

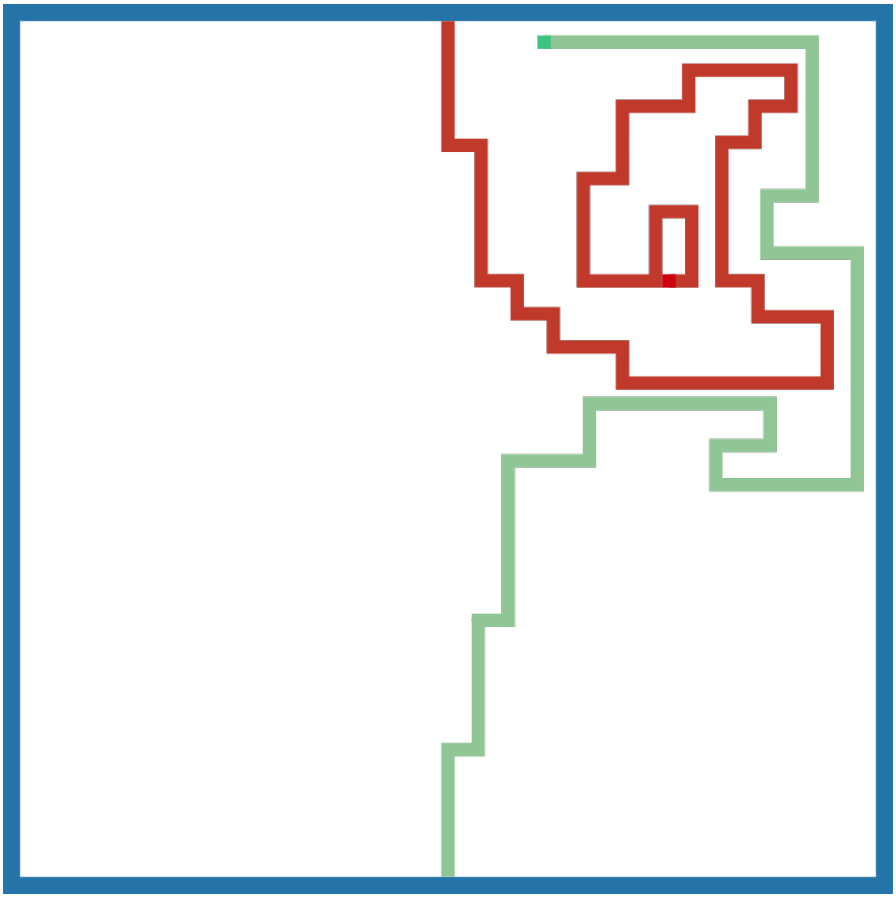

Figure 4.1 – *Diagram depicting a 'close-call' situation gone wrong for an artificial intelligence controlled player (red) against a human controlled player (green).*

However, our artificial intelligence was previously designed to run simulations under the assumption it had full control over the exact time in which moves could be played. The AI would spend a fixed amount of time running simulations centred around a turn-based model of our implemented game. Once all players had taken their respective turns, the game state would be updated, using a progress time equal to the minimum time required before changing move, leading to our simulations converging on strategies which could never be carried out. This is because the strategies depended on future moves being played at specific times, these times did not coincide with the fixed amount of time allocated to running the simulations. The solution to this issue was quite simple, the simulation's state update would use a time progression equal to the duration of the previous AI move calculation. This would give some indication as to when our AI would play their next move. Although not a perfect solution, manipulating the time-progression between state updates, did alleviate much of the negative impact.

4.2.8 Technology Integration

Some of the most prevalent issues faced were those indirectly related to the project. Specifically the issues that came about due to the discretional use of third-party libraries, and other technologies.

Redux

Redux did a pretty superb job in handling our view's state. It kept our client's view layer reflecting the current game state (see subsection 3.2.2: *Core Technologies* (Page: 10)). It even played a critical role in many of the tasks occuring behind the scenes, that is those not directly related to our game's state. Despite its successful integration into our application, it was the spawn of many challenges.

For example, Redux is not natively equipped to deal with asynchronous events; such as communicating with the server, or detecting user keyboard input. Fortunately, there exist many additional libraries which tackle this problem. One of the asynchronous libraries is Redux Saga (redux-saga, 2017).

Redux Saga is a middleware that can be attached to Redux. It is used in conjunction with the new JavaScript generators, to make dealing with asynchronous code in a React/Redux application easier and more natural to work with. The use of generators play as an alternative to callbacks, which can quickly grow to become out-of-hand. In principle, it builds upon the saga pattern (Hector Garcia-Molina, 2017).

For most jobs, Redux Saga is quite simple to use. However, integrating Redux Saga with WebSockets proved to be quite complicated - especially with regards to the use of JavaScript generators.

Our implemented solution begins by first immediately requesting then establishing a WebSocket connection with the server, and proceeds to maintain said connection. A saga is then set-up and tasked to constantly listen out for messages from the server, and upon receiving a message will spit out the appropriate Redux action containing the payload. A write saga is also set-up which listens for actions emitted to the store from the client. Once an action is received, it is appropriately packaged into a payload and shot off to the server using the maintained WebSocket connection. This WebSocket/Redux Saga adapter works very well, and is written in a generic manner, decoupled from the rest of the application, in order for it be feasibly integrated into other projects.

Node.js Child Process

Both the networking and artificial intelligence aspects of our application rely heavily on the use of Node.js's child process module. This allows us to delegate computationally expensive tasks to be computed concurrently within a separate process. However, as threading is not supported, we are unable to share memory between any two processes. As an alternative, we were able to serialise our game state and communicate that to our separate process. This was the cause of a serious issue. During development our application was experiencing some odd unexpected behaviour - input seemed sluggish at times. After some rigorous benchmarking, it was discovered that the time required to serialise, and communicate our game state to a separate process was dangerously greater than the time spent on the task inside the process. In fact, it would often take longer than several ticks to apply a single update to the game state.

After extensive debugging, a pattern emerged in that as the elapsed time of a single game of Tron grew, the communication delay when messaging our separate process also grew. After some more analysis, communicating the game state's cache was identified as being the primary cause of this extra delay. This was due to the serialised representation of our cache occupied too much memory. Although not a perfect solution, our application now disregards game state cache when communicating with a separate process. On the receiving side of communication, the cache is efficiently rebuilt using data from the remainder of the game state. This solution produced wonderful results.

Chapter 5

Results and Evaluation

It is now time to evaluate our implemented application by performing a critical analysis to help determine the extent of our application's fulfilment to the initial project goals. This shall be done by performing a series of both objective and subjective tests aiming to expose its strengths and weaknesses. All experiments are performed on machine with an Intel Core i7 3770K CPU (3.5 GHz) and 16GB RAM (1600 Mhz).

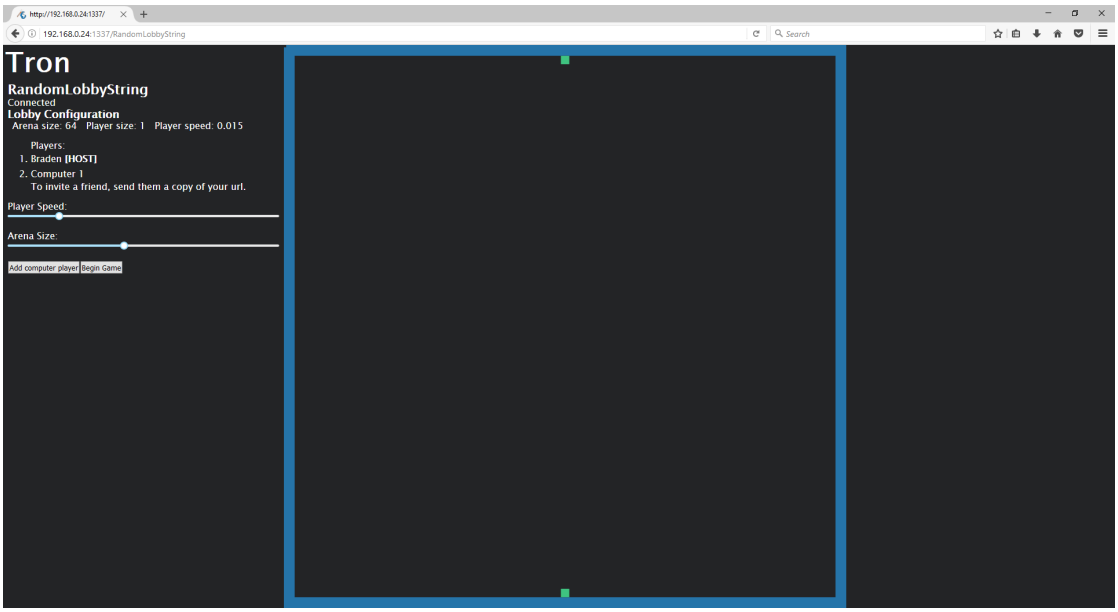

Figure 5.1 – *Screen capture of the implemented application, demonstrating the game within a web-browser.*

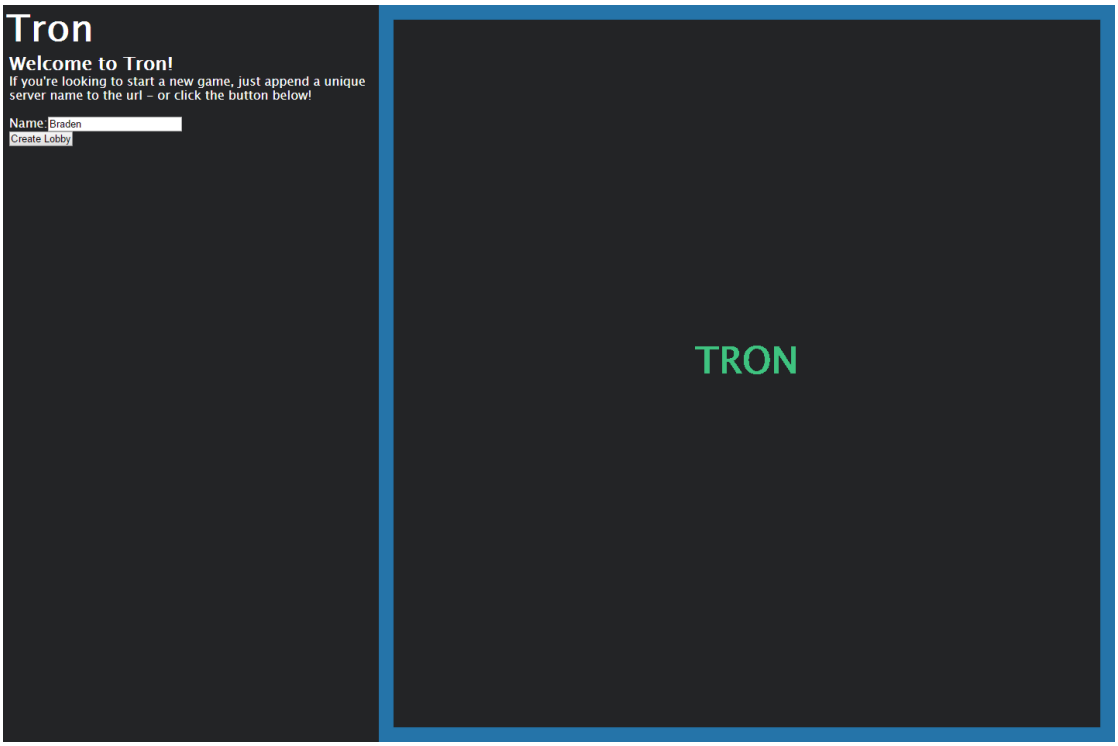

Figure 5.2 – *Screen capture of the implemented application, demonstrating the game's welcome screen.*

Figure 5.3 – *Screen capture of the implemented application, demonstrating the game in progress.*

5.1 Unit Tests

Unit testing is arguably the most reliable software development technique for the purposes of scrutinizing an application in order to prove its operational correctness. A unit is the smallest testable part of an application, such as an individual function. A unit test is a short code fragment intended to test a single unit of an application.

During the development process, an application will evolve as modifications are introduced. This is normal behaviour, although the process has a habit of leaving behind bugs. Having a complete suite of unit tests enable the developer to automatically identify bugs and ensure that their code retains its previous validity. Unit tests can be implemented in a number of ways. As the meat of our application is programmed in JavaScript, we will be using Jest (Facebook Inc., 2017b) - one of the many available JavaScript testing solutions. Jest also offers specialised support for React via snapshot testing. Various modules within our application contain unit tests in a subdirectory named '`__tests__`'. These tests have been developed according to both the black-box and white-box testing methods, to best guarantee

the completeness of our application.[1]

```
PASS  shared\game\utils\collision\__tests__\quadtree.test.js
PASS  shared\game\utils\__tests__\spawn.test.js
PASS  shared\game\operations\__tests__\general.test.js
PASS  shared\game\operations\__tests__\player.test.js
PASS  shared\game\utils\collision\__tests__\grid.test.js
PASS  shared\components\App\Error404\__tests__\Error404.test.js

Test Suites: 6 passed, 6 total
Tests:       24 passed, 24 total
Snapshots:   1 passed, 1 total
Time:        8.126s, estimated 13s
Ran all test suites.
Done in 11.14s.
```

Figure 5.4 – *Capture of the console output produced from executing our Jest unit tests.*

As seen from Figure 5.4, we have unit tests covering several modules of our application. Should a new bug be discovered within one of these modules, a new test would be created capable of detecting said bug. For information regarding what is actually being tested, we encourage the reader to refer to the JavaScript test file's source-code; they are well documented.

5.2 Game Performance

The game has been developed up to a playable state, yet there is still interest in seeing just how much is capable before it begins to breakdown. To gain a stronger understanding of just what our game is capable of, we conduct experiments measuring the performance impact caused by various game mechanics. For each of these experiments, the goal is to derive a sensible evaluation based upon an accumulation of quantifiable data.[2]

[1] Due to the time-sensitive nature of the project, the currently implemented unit tests do not yet have 100% coverage of our application.

[2] A separate Git branch was created for the purposes of recording benchmark data. It can be found on our GitHub repository (see section 3.1: *Software Development Process* (Page: 7)).

5.2.1 Tick Update Performance

The game loop is responsible for drawing the graphics and, more importantly, updating the game state. It is absolutely critical that the update function's call duration does not exceed the duration of its associated tick. If this were the case, our game's tick-rate would begin to fall - leading to undesired game behaviour. Hence we perform a series of experiments measuring the implemented update function's execution duration under various scenarios. These scenarios will vary by two controllable factors: the total number of connected players[3], and the elapsed time since the beginning of the game round. Each experiment scenario is repeated five times to help mitigate erroneous results.[4]

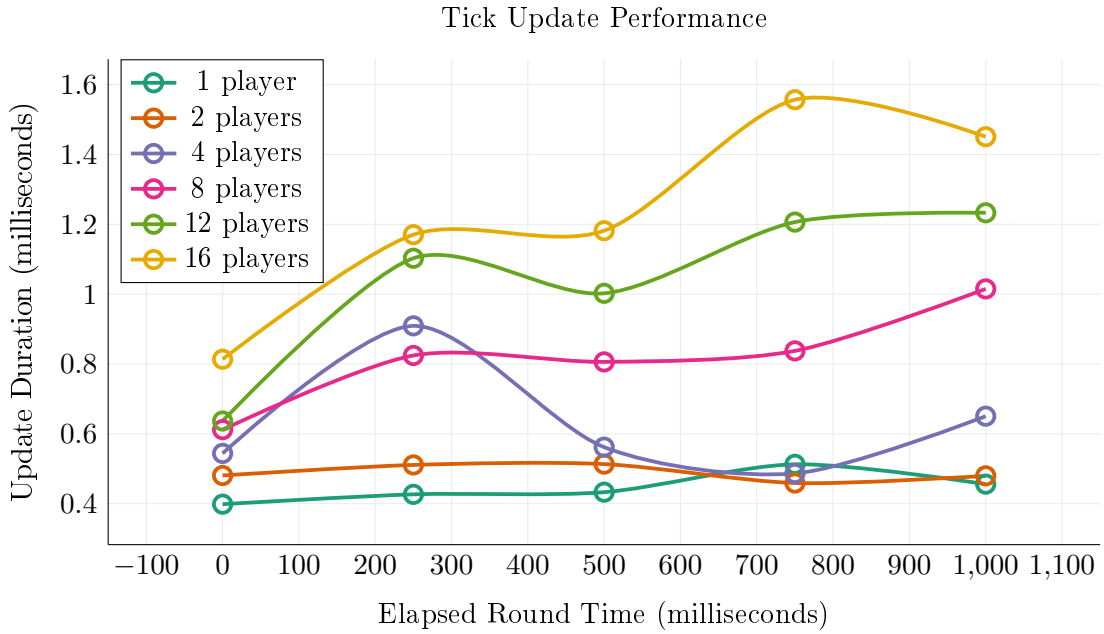

Figure 5.5 – *A graph depicting the impact to the performance of our game's tick updates with relation to the number of total connected players and the elapsed round time.*

As can be seen in Figure 5.5, the results indicate there is definitely a significant correlation between our game's performance and the two examined factors. It can be seen that an increase to either factor will result in our game state update duration being prolonged. Not only this, but performance is subject to the com-

[3]All connected players, besides the host player, are computer players controlled by artificial intelligence. This means our results better resemble those of worst-case performance.

[4]For a full log detailing the exact results gathered from the experiment, refer to Table 8.1 (Page: 59).

bination of either factor. The reasoning for this is quite simple, and our collision detection is the primary cause: in general, our collision detection is required to perform more extensive searches if our game arena consists of a greater number of objects. Furthermore, as the round naturally progresses in time, each player will be travelling around the game arena; contributing additional objects. Hence, round progression produces objects proportionally to the number of alive players; further burdening the task of collision detection. However with this being said, even in the worst case, our game state updates occur at an extremely fast rate - well within the required bounds of once every 15 milliseconds (see section 3.4: *Game Mechanics* (Page: 14)).

5.3 Capability of Artificial Intelligence

Each computer player determines their move using an artificial intelligence, powered by the culmination of various techniques. The goal of which is to mimic the behaviour of a human controlled player, giving the competing human user(s) a realistic game experience. This happens to be quite a difficult requirement to test, due to the subjectiveness of what can be considered 'human behaviour' (for a relevant subjective experiment, see section 5.4). Despite this, we can obtain empirically quantifiable data by running a series of experiments, recording the win-percentage gathered from pitching our implemented AI against a human controlled player. Ideally, we would like the AI player to perform equally to the human user, i.e. a 50/50 win-ratio. However, the results are subject to the skill-level of the competing human user. This is not something that we can easily regulate, so we assume our chosen human user is of an average skill-level. To get a broad understanding, we shall be performing the experiment under several different scenarios, varying the game by certain factors. These factors include the game arena size and player movement speed. We shall capture the total simulation count at the start of each round, then select the median result from five repeats to help mitigate erroneous results.

Table 5.1 – *Table of results stating the outcome of the many games in which an artificial intelligence controlled player would compete against a human controlled player.*

Player Speed	Arena Size	AI Wins
0.01	16	4/5
0.025	16	4/5
0.05	16	5/5

Continued on next page...

...Table 5.1 continued from previous page

Player Speed	Arena Size	AI Wins
0.075	16	4/5
0.01	32	3/5
0.025	32	5/5
0.05	32	5/5
0.075	32	3/5
0.01	64	2/5
0.025	64	1/5
0.05	64	3/5
0.075	64	3/5
0.01	128	0/5
0.025	128	1/5
0.05	128	1/5
0.075	128	3/5

Concluded

As can be seen in Table 5.1, the computer player performs somewhat respectably in comparison to a human user. From observing the experiments, the artificial intelligence would often begin with reasonable strategy, but then occasionally decide to play a move that was not so favourable. This was especially a problem in the larger arena sizes, once the two competing players had become disconnected from one-another. Once the two players would separate, the human controlled player was able to simply wait for the computer player to make a mistake. Hence, the AI performed particularly well when the arena size was smaller, due to the generally shorter game duration. The computer player thrived at faster speeds, as the human player was unable to react expeditiously. However, a combination of the tested fastest speed and smallest arena size, made the game feel somewhat random - as both players were unable to make strategic moves. To better understand where improvements could be made, analysis was performed regarding the performance of our simulations. This prompted investigation into how the total simulation count was affected by the size of the game arena.

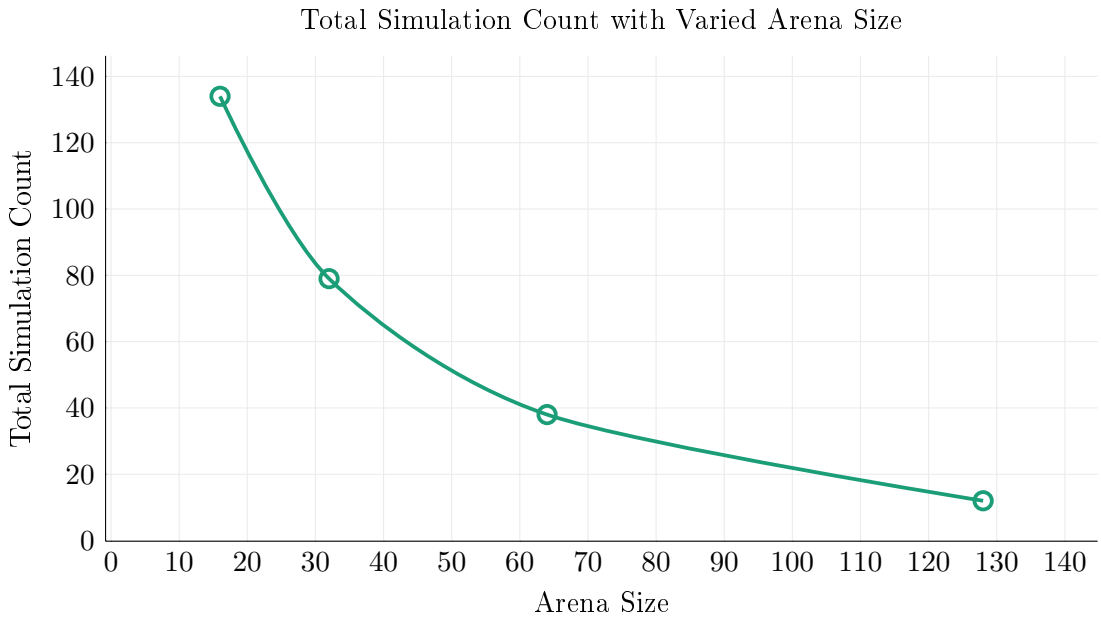

Figure 5.6 – *A graph depicting how the game arena size impacts the total number of simulations performed by our artificial intelligence when calculating a single move.*

As expected (see Figure 5.6)[5], the total simulation count is considerably greater for arenas of a relatively smaller size. This indicates something within our simulation task was hindering performance. However, expanding from what was shown in subsection 5.2.1: *Tick Update Performance* (Page: 46), it was suspected that game update function was not to blame, but instead the heuristic evaluation function. This suspicion was proven correct by dramatically increasing the simulation depth and measuring how it affected the total number of simulations (see Figure 5.7)[6]. This experiment will be performed with two players, one controlled by a human and the other by our artificial intelligence. We shall record the median value of five initial computer moves, made on an arena size of 32. If the total number of simulations sparsely changed, we would know that our heuristic evaluation function was to blame; as it ran only at the end of a simulation - unaffected by depth. This loss of performance would prevent the proper construction of belief regarding the game tree, leading to the move being decided upon despite not having fully explored the consequences.

[5]For a full log detailing the exact results gathered from the simulation count experiment, refer to Table 8.2 (Page: 63).

[6]For a full log detailing the exact results gathered from the simulation depth experiment, refer to Table 8.3 (Page: 64).

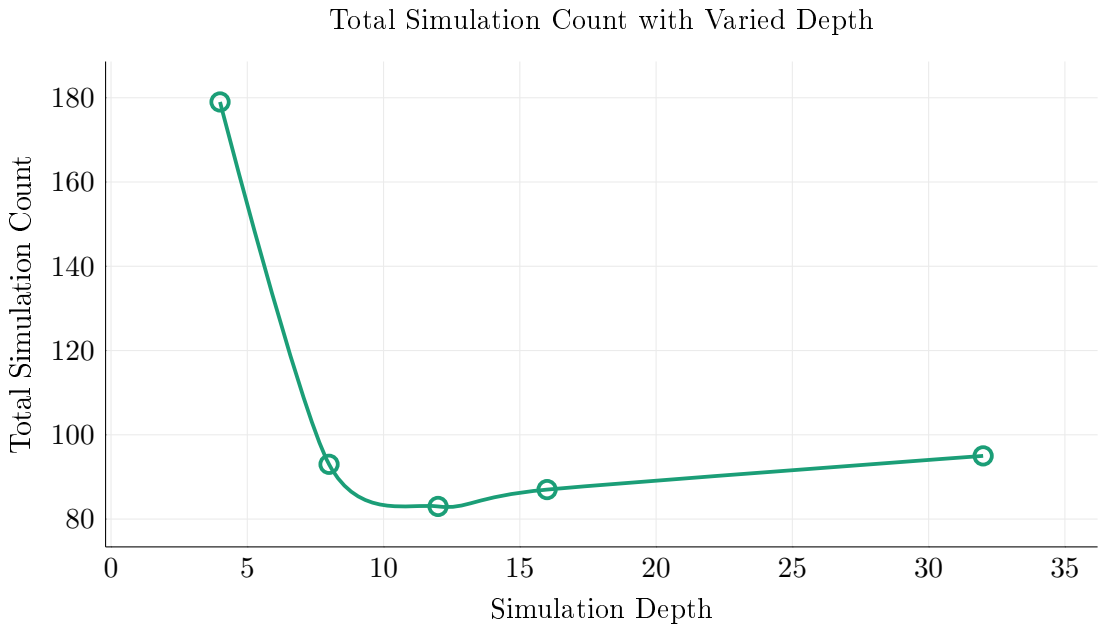

Figure 5.7 – *A graph depicting how the individual simulation depths impacts the total number of simulations performed by our artificial intelligence when calculating a single move.*

Another fault of our artificial intelligence is regarding timing. The simulation tree would often try to play optimally, for example by redirecting at the very last instant. This is caused by the simulation believing it has control over the next move. Sadly, this is not the case due to the non-deterministic nature involved for the concurrent aspects of our application. Despite efforts made to remedy this (see subsection 4.2.7: *AI Timing* (Page: 38)), the issue still persists - although not to its previous extent.

The artificial intelligence also has no explicit means to prevent itself from blocking off 'owned' areas of the game arena. This results in the computer player often making moves that will worsen their total potential lifespan. The issue has been tackled by others (see section 2.4: *Artificial Intelligence* (Page: 5)). but was never included in our implementation.

As discussed in subsection 4.2.6: *AI Concurrency* (Page: 37), our artificial intelligence calculates moves in a process separate to that running the main application. This functionality works quite well for a game consisting of only several computer players, but begins to fall down when a larger quantity of computer players are introduced. In part, this is due to each computer player being assigned their own process to calculate moves; an architecture which does not scale well, due to the executing computer only have a limited number of cores.

5.4 User Feedback

The implemented game could be objectively perfect, however if users believe that the project fails to hold promise, then our project's investigated ambition would conclude as a failure - indicating the designed techniques fail to perform sufficiently. Therefore, it is worthwhile to collect a sample of user opinions, enabling us to form a concrete understanding of where our application stands. Hence, we introduce five users to play a few games, then document their opinions by having them participate in a short survey, as described in Figure 5.8.

> Please respond to the below questions on a scale from 1-10 (where 1 is strongly disagree, and 10 is strongly agree):
>
> 1. The graphics of the game appear to run smoothly.
> 2. The game's user interface is functional, with near unnoticeable loading times.
> 3. Movement around the game arena seemed sufficiently accurate, in response to my input.
> 4. The multiplayer functionality felt responsive, with little indication of latency.
> 5. The computer controller player works well; challenging, but not impossible to beat.
> 6. The foundational mechanics of the game hold promise, and could produce a good game with a little more polish.

Figure 5.8 – *A series of questions to be surveyed to test participants.*

For the most part, user feedback (see Table 8.4 (Page: 64)) indicates that the game is somewhat a success. Although this is pleasing, it fails to highlight the weak-points of our application.

Chapter 6

Future Work

The project has demonstrated that native web-browsers are a suitable platform for video-game development. This was done by designing and implementing Tron, using a variety of popular game-design techniques. For the most part, the project has been success, although, as with anything, there are definitely areas for improvement. Throughout the remainder of this chapter, we shall elaborate on some potential future work that could be undertaken. This will also give the reader a more concise understanding of what the project lacks.

6.1 Deployment

There are big differences between developing an application and developing an application as a usable product. In particular, deploying a web-application, as a product intended for general users, can require huge amounts of work in designing deployment infrastructure and implementing the necessary software tweaks to accommodate this 'general' user-base. As it stands, the focus of the project has been on constructing the foundations of Tron, so factors related to wide-scale deployment have not received much attention. Hence, if stability is of any concern, it is not currently suitable for our implementation to be publicly deployed on the internet.

6.1.1 Security

Breaches in security are an extremely grave threat to any application that is publicly accessible. It is of the utmost importance that serious consideration be invested into deterring users whom bare malicious intent. With that said, we make critical note identifying that, as of current, the implemented application lacks adequate security measures. In particular, user input has not been extensively

sanitised and there is likely to exists potential bugs, which could be triggered by malicious users.

6.1.2 Large-scale performance

Although the implemented application is capable of an arbitrary number of game lobbies, and connected players, it will most inevitably result in a hindrance to performance. Great attention has gone into designing a concurrent architecture. The ambition of which is to enable the possibility of deployment utilising a cluster of machines; to distribute the load of our computationally expensive tasks.

6.2 Artificial Intelligence

The implemented artificial intelligence can occasionally outperform human players, but unfortunately it is subject to some rather unusual behaviour (see section 5.3: *Capability of Artificial Intelligence* (Page: 47)). There do exist more advance artificial intelligences for the game of Tron, but their integration would likely result in a computer player that is neigh impossible to beat. Furthermore, one key area for future work is regarding the question as to how we can remedy the timing inaccuracies related to our artificial intelligence.

6.3 Gameplay

As we have repeatedly stated, the focus of this project has been on designing and implementing an application which demonstrates the foundational functionality required for multiplayer video-game development. This leaves a lot of room for the addition of feature related purely to entertaining gameplay. To accommodate this, much effort was devoted in constructing an implementation that is both flexible and extendible, so future developers can more feasibly implement modifications. It is once again worth noting that the front-end, being built with React, is decoupled into components that can be extended upon without too much hassle. Also, the game mechanics maintain an individual high-cohesion; reducing reliance on a particular architecture or technique.

Chapter 7

Conclusions

The project has now come to an end, giving us the chance to round off and conclude our findings. This chapter will provide the reader with a summary, and brief elaboration, on what has been accomplished, including both the successes and failures. We shall begin with a general overview of the project, before delving into some of the more specific aspects.

To begin with, the most prevalent blunder was deciding to base the game on free movement, as opposed to cell-based movement. This proved to be quite an inconvenience, complicating a range of tasks, without offering all that much benefit to the user experience; due to the visual difference being near indistinguishable. However, many of these inconveniences were overcome and the project now demonstrates a higher level of capability.

7.1 Networking

Since the conception of the project, the game has been designed according to the client/server network architecture model (see subsection 3.2.1: *Architecture Overview* (Page: 10)). This has suited the project quite nicely, especially with the additional techniques used to reduce latency (see section 3.5: *Network Communication* (Page: 20)). Players are able to interact with each-other in real-time, with minimal perceptible latency. This is primarily due to lag-compensation and client-side prediction. However, occasionally client-side prediction will result in *flashes*, indicating an inconsistency between the client's predicted state and that communicated by the authoritative server. In particular, this occurs for a client when another player redirects themselves moments before collision - causing their trail to flash red, as their death was incorrectly predicted. This could either be manipulated to be a feature (possibly change to orange during prediction, then red once death is confirmed by the server) or make the client prediction only predict player move-

ment, not collisions.

A large amount of time was also invested in enabling the game to support an arbitrary number of game lobbies. The functionality does exist and works as one would expect. However, it simply does not scale very well. Despite many efforts, such as those related to concurrency (see subsection 3.5.1: *Game Lobbies and Concurrency* (Page: 21)), a single dedicated game server is only capable of feasibly running just a few game instances, and is dramatically hindered by high-numbers of computer players - due to their artificial intelligence. Quite early on during development, this issue started to get recognised as a potential problem that may later arise. In fact, significant efforts were made to prepare for an adjustment regarding the network architecture. In particular, it was conceptualised that the dedicated server would pick the most-capable connected player in a lobby to play the role of the game server, perhaps cycling through the players as an anti-cheat measure. This alternative strategy would dramatically reduce load on our dedicated server, as it would then only be responsible for establishing the connection between players in the same lobby. To reduce the amount of work required in transitioning to the aforementioned strategy, specialised software-development techniques, such as dependency injection, have been utilised throughout the game's networking functionality, to ensure minimal coupling to Node.js.

Despite this, a complete transition was not viable due to numerous setbacks. The main setback faced, was that regarding the fact Web-Sockets is not capable of browser-to-browser communication. This leaves open two potential solutions: proxy messages through the dedicated server, or, more favourably, convert the communication medium to WebRTC. However, time did not permit the ability to undertake either solutions, without impeding other aspects of the project.

7.2 Artificial Intelligence

The game features a computer controller player that determines their move based upon some artificial intelligence techniques. These techniques centre around the idea of running simulations, exploring the various scenarios derived from the current game state.

Artificial intelligence based upon simulation is scarcely utilised when performance is of a critical concern. This is primarily due to the large computational expense involved with running simulations, as they are of a 'trial-and-error' nature. Alternatively, a more ideal solution would analyse only the current game state to determine a move. Despite this, many of existing Tron AI solutions saw great success from utilising game tree simulations; indicating that it was indeed a viable direction. In practice, our implemented artificial intelligence performs quite well, and can even be considered a strong contender against human users. Regardless of

this, the goal was to create an AI which mimicked human strategy and behaviour; as opposed to being of an incredibly high skill-level. Hence, it is in this regard we must state that the implemented artificial intelligence does not perform entirely as desired. In particular, it would occasionally play bizarre moves in which the strong move is easily identifiable by even a novice human player. Conflictingly, the artificial intelligence also suffered issues regarding the timing of moves. However, this is not strictly a bad thing as it also happens to be a trait of human users.

In conclusion, the artificial intelligence that has been implemented does suit our particular use-case quite well, that is the realistic playing of Tron. But questions are raised regarding a simulation based AI being suitable for more complex video-games, where performance and accuracy are of a critical concern.

Chapter 8

Reflections on Learning

When working on a new project, it is expected there will be times of confusion and challenge. These moments are often demoralizing, and can impede in one's motivation to work. Ironically, they are also the times in which *learning* occurs. For something to be worth doing, it must hold some reward. One of the most valuable rewards, especially to a beginner, is the attainment of new skills. Hence, for a project to be worthwhile, it must bear many challenges. This idea is especially prevalent in software development. Almost every project will expose the developer to new ideas. Often these ideas are expressed through the use of a new library or design pattern. However, the developer will also acquire skills that are inexplicit in nature. These skills are often called 'meta-skills', referring to the cognitive strategies that an individual applies to the processing of new information. They are often equated to experience, and impact the way we approach future projects. Throughout this chapter, we reflect upon our approach to this project, scrutinizing the assumptions that we made. We do this to help better develop our ability as an individual to learn, and tackle challenges that we may later face.

The realisation that the project exists purely as a learning exercise did, undeniably, encourage the selection of certain decisions. Although some of these decisions may not have been entirely suitable for a commercially motivated project, they did provide a wealth of new skills. One decision, in particular, was in regards to the somewhat excessive use of third-party libraries and techniques. Many of such were employed with little purpose other-than to understand their promoted ideas and intricacies; that are frequently appreciated for being well-designed. However, the decision did introduce the concept of 'technical debt'. This resulted from the plethora of issues solely related to the arguably unnecessary intricacies we had subjected ourselves to. Unfortunately, this snatched time away from the core aspects of the project. Admitting that this was an entirely bad decision, would not be entirely true.

Another mistake made was the lazy assumption that techniques utilised by

similar projects would transfer well onto our own. Not only did this turn out not to be the case, but resulted in greater amounts of work due to the inevitable transition to a more suitable technique. If more time was spent researching and testing a particular technique, we would have most probably saved time overall.

Despite their hindrance to the final state of our project, it is not entirely unfortunate that these mistakes occurred. They unlocked the opportunity for us to recognise them, understand them, and to build on them going forward. The experiences they taught are an invaluable asset, transferable to many aspects in life, and will evolve our mindset going forward with future endeavours.

Glossary

Table of Abbreviations

Appendices

Below we list various materials that have been referenced throughout, but impractical to include within the text:

Table 8.1 – *Table of results following the experiment described in subsection 5.2.1:* Tick Update Performance *(Page: 46).*

Players	Elapsed Round Time	Update Duration
1	0	0.421
1	0	0.489
1	0	0.378
1	0	0.332
1	0	0.373
2	0	0.464
2	0	0.456
2	0	0.465
2	0	0.609
2	0	0.41
4	0	0.439
4	0	0.584
4	0	0.484
4	0	0.461
4	0	0.752
8	0	0.552
8	0	0.69
8	0	0.698
8	0	0.608
8	0	0.512
12	0	0.607
12	0	0.623
12	0	0.729
12	0	0.604
12	0	0.621
16	0	0.934

Continued on next page...

... Table 8.1 continued from previous page

Players	Elapsed Round Time	Update Duration
16	0	0.697
16	0	0.948
16	0	0.77
16	0	0.719
1	250	0.422
1	250	0.507
1	250	0.373
1	250	0.442
1	250	0.39
2	250	0.497
2	250	0.485
2	250	0.575
2	250	0.501
2	250	0.497
4	250	0.926
4	250	1.066
4	250	1.002
4	250	0.643
4	250	0.909
8	250	0.787
8	250	0.742
8	250	0.765
8	250	0.983
8	250	0.845
12	250	1.449
12	250	0.82
12	250	0.91
12	250	1.219
12	250	1.116
16	250	0.889
16	250	1.019
16	250	1.409
16	250	1.474
16	250	1.061
1	500	0.424
1	500	0.478

Continued on next page...

... Table 8.1 continued from previous page

Players	Elapsed Round Time	Update Duration
1	500	0.405
1	500	0.434
1	500	0.424
2	500	0.542
2	500	0.429
2	500	0.473
2	500	0.535
2	500	0.588
4	500	0.593
4	500	0.609
4	500	0.627
4	500	0.464
4	500	0.517
8	500	0.595
8	500	0.728
8	500	0.785
8	500	0.819
8	500	1.103
12	500	1.06
12	500	0.884
12	500	0.856
12	500	1.147
12	500	1.065
16	500	1.269
16	500	1.178
16	500	1.265
16	500	1.014
16	500	1.183
1	750	0.422
1	750	0.428
1	750	0.665
1	750	0.564
1	750	0.485
2	750	0.44
2	750	0.472
2	750	0.503

Continued on next page...

... Table 8.1 continued from previous page

Players	Elapsed Round Time	Update Duration
2	750	0.417
2	750	0.464
4	750	0.547
4	750	0.397
4	750	0.565
4	750	0.504
4	750	0.42
8	750	0.775
8	750	0.748
8	750	0.765
8	750	1.108
8	750	0.792
12	750	1.435
12	750	1.344
12	750	1.133
12	750	1.11
12	750	1.01
16	750	1.396
16	750	2.763
16	750	1.171
16	750	1.266
16	750	1.188
1	1000	0.444
1	1000	0.467
1	1000	0.425
1	1000	0.437
1	1000	0.507
2	1000	0.519
2	1000	0.425
2	1000	0.583
2	1000	0.376
2	1000	0.496
4	1000	0.696
4	1000	0.702
4	1000	0.703
4	1000	0.611

Continued on next page...

... Table 8.1 continued from previous page

Players	Elapsed Round Time	Update Duration
4	1000	0.541
8	1000	1.124
8	1000	1.453
8	1000	0.846
8	1000	0.788
8	1000	0.864
12	1000	1.162
12	1000	1.282
12	1000	1.249
12	1000	1.133
12	1000	1.339
16	1000	1.653
16	1000	1.21
16	1000	1.374
16	1000	1.6
16	1000	1.417

Concluded

Table 8.2 – *Table of results, following the experiment described in section 5.3:* Capability of Artificial Intelligence *(Page: 47), stating how the game arena size impacts the total number of simulations performed by our artificial intelligence when calculating a single move.*

Arena Size	Total Simulations
16	134
32	79
64	38
128	12

Concluded

Table 8.3 – *Table of results, following the experiment described in section 5.3:* Capability of Artificial Intelligence *(Page: 47), stating how the total number of artificial intelligence performed simulations are affected by their respective depth.*

Simulation Depth	Total Simulations
16	134
32	79
64	38
128	12
	Concluded

Table 8.4 – *The collective results gathered from the user survey described in section 5.4:* User Feedback *(Page: 51).*

Question 1	Question 2	Question 3	Question 4	Question 5	Question 6
9	8	7	9	8	10
10	10	10	10	10	10
8	9	9	8	7	9
6	7	9	9	8	10
7	8	10	9	8	10
		Concluded			

Bibliography

[a1k10] a1k0n. *Google AI Challenge post-mortem*. 2010. URL: https://www.a1k0n.net/2010/03/04/google-ai-postmortem.html (visited on 17/04/2017).

[Bab16] Babel (Open Source). *Babel · The compiler for writing next generation JavaScript*. 2016. URL: https://babeljs.io/ (visited on 26/01/2017).

[Bec+01] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: http://www.agilemanifesto.org/.

[Ber11] Yahn W. Bernier. *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization*. 2011. URL: https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization (visited on 16/04/2017).

[ctr17] ctrlplusb. *React Universally*. 2017. URL: https://github.com/ctrlplusb/react-universally (visited on 12/04/2017).

[Ecm17] Ecma International. *ECMAScript® 2015 Language Specification*. 2017. URL: http://www.ecma-international.org/ecma-262/6.0/ (visited on 12/04/2017).

[Fac17a] Facebook Inc. *A JavaScript library for building user interfaces - React*. 2017. URL: https://facebook.github.io/react/ (visited on 12/04/2017).

[Fac17b] Facebook Inc. *Jest Painless JavaScript Testing*. 2017. URL: https://facebook.github.io/jest/ (visited on 30/04/2017).

[Goo10] Google. *Google AI Challenge*. 2010. URL: https://csclub.uwaterloo.ca/contest/ (visited on 17/04/2017).

[Gra05] Paul Graham. *Web 2.0*. Nov. 2005. URL: http://www.paulgraham.com/web20.html (visited on 26/01/2017).

[Hec17] Kenneth Salem Hector Garcia-Molina. *Alpha–beta pruning*. Tech. rep. Department of Computer Science Princeton University, Jan. 2017. (Visited on 19/04/2017).

[Hsu17] Dan Hsu. *Fltron - Light-Cycle and Tron Games*. 2017. URL: `http://www.fltron.com/` (visited on 09/04/2017).

[HTC99] Andrew Hunt, David Thomas and Ward Cunningham. *The Pragmatic Programmer. From Journeyman to Master*. Addison-Wesley Longman, Amsterdam, 1999. ISBN: 020161622X.

[Mar17] Braden Marshall. *Braden1996/tron.io*. 2017. URL: `https://github.com/Braden1996/tron.io` (visited on 12/04/2017).

[Moz17a] Mozilla Developer Network. *WebGL - Web APIs | MDN*. 2017. URL: `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API` (visited on 09/04/2017).

[Moz17b] Mozilla Developer Network and individual contributors. *WindowOrWorkerGlobalScope.setTimeout()*. 2017. URL: `https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout` (visited on 17/04/2017).

[Moz17c] Mozilla Developer Network and individual contributors. *window.requestAnimationFrame(* 2017. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Window/requestAnimationFrame` (visited on 17/04/2017).

[Moz17d] Mozilla Developer Network and individual contributors. *Window.setImmediate()*. 2017. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Window/setImmediate` (visited on 17/04/2017).

[Nod17a] Node.js Foundation. *Child Process | Node.js v7.9.0 Documentation*. 2017. URL: `https://nodejs.org/api/child_process.html` (visited on 27/04/2017).

[Nod17b] Node.js Foundation. *Node.js*. 2017. URL: `https://nodejs.org/en/` (visited on 12/04/2017).

[Ope09] Open source. *Git - Fast Version Control System*. 2009. URL: `http://git-scm.com/` (visited on 25/04/2017).

[rea17] reactjs. *Predictable state container for JavaScript apps*. 2017. URL: `https://github.com/reactjs/redux` (visited on 12/04/2017).

[red17] redux-saga. *redux-saga/redux-saga*. 2017. URL: `https://github.com/redux-saga/redux-saga` (visited on 19/04/2017).

[Ref17] Refsnes Data. *JavaScript Numbers*. 2017. URL: `https://www.w3schools.com/js/js_numbers.asp` (visited on 15/04/2017).

[Sha17] Shy Shalom. *Cycleblob - A WebGL lightcycle game*. 2017. URL: http://cycleblob.com/ (visited on 09/04/2017).

[Sta17] Starcounter-Jack. *Starcounter-Jack/JSON-Patch*. 2017. URL: https://github.com/Starcounter-Jack/JSON-Patch (visited on 17/04/2017).

[tea17] team@slither.io. *slither.io*. 2017. URL: http://slither.io/ (visited on 09/04/2017).

[tim17] timetocode. *An accurate node.js game loop inbetween setTimeout and setImmediate*. 2017. URL: http://timetocode.tumblr.com/post/71512510386/an-accurate-nodejs-game-loop-inbetween-settimeout (visited on 17/04/2017).

[Val11a] Valve. *Lag compensation*. 2011. URL: https://developer.valvesoftware.com/wiki/Lag_compensation (visited on 16/04/2017).

[Val11b] Valve. *Prediction*. 2011. URL: https://developer.valvesoftware.com/wiki/Prediction (visited on 16/04/2017).

[Wal14] Walt Disney Productions. *Tron (1982) - "Light Cycle Battle"*. 2014. URL: https://www.youtube.com/watch?v=7DgL_w5qwIw (visited on 26/01/2017).

[Wik17a] Wikipedia, the Free Encyclopedia. *Alpha–beta pruning*. 2017. URL: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning (visited on 19/04/2017).

[Wik17b] Wikipedia, the Free Encyclopedia. *Comparison of HTML5 and Flash*. 2017. URL: https://en.wikipedia.org/wiki/Comparison_of_HTML5_and_Flash (visited on 09/04/2017).

[Wik17c] Wikipedia, the Free Encyclopedia. *Minimax*. 2017. URL: https://en.wikipedia.org/wiki/Minimax (visited on 19/04/2017).

[Wik17d] Wikipedia, the Free Encyclopedia. *Model–view–controller*. 2017. URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller (visited on 27/04/2017).

[Wik17e] Wikipedia, the Free Encyclopedia. *Monte Carlo tree search*. 2017. URL: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search (visited on 17/04/2017).

[zig12] ziggystar. *Monte Carlo with UCB applied to complex card game*. 2012. URL: http://stackoverflow.com/questions/12523221/monte-carlo-with-ucb-applied-to-complex-card-game (visited on 17/04/2017).