VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Simulating and Visualising a Model Railway

Nicky van Hulst

Supervisor: Dr David J. Pearce

October 15, 2016

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

This project is concerned with creating a software simulation and visualisation of a model railway to improve the efficiency of testing software controllers. Testing a train controller on a physical railway is time consuming and limited by what is physically available to test on. A simulation and visualisation of the model track will reduce these issues by providing a quicker testing platform for the controllers. The simulation, visualisation and a track builder were successfully created in Java. A controller was developed on the simulation to control the trains and transferred over to the physical system. The controller was able to successfully control the trains.

Acknowledgements

I would like to thank my supervisor David Pearce for the continued feedback and advice throughout the project. This support has improved the quality of the project and report.

Contents

Figures

Chapter 1

Introduction

Simulations are used to model real-world systems. Simulations are able to emulate different scenarios and configurations without the significant cost of failure. Testing control software for trains is made easier with a simulation. Gaining access to trains and a track would be extremely costly, if something were to go wrong it could also be dangerous. A simulation of the hardware reduces this problem. Therefore a model railway has been created in the School of Engineering and Computer Science department (ECS). The railway has the purpose of emulating the important safety concerns and features that a railway system would have. For example, trains can collide or derail if not managed correctly.

One of the reasons this is important is that a software controlled railway system falls under the category of a safety critical system. Safety critical systems are concerned with systems that, when things go wrong, the consequences could cause serious harm to people, the environment and or equipment. These types of systems are difficult to test safely and cost-efficiently, which is why simulations are a commonly used tool to help with the problem.

An important safety concern of a model railway is the potential for collisions. The model railway was specifically designed to model software interlocking to prevent collisions [1]. Interlocking considers a number of tracks and signals that work together to work out if a route is safe [2]. The system is designed to make it impossible to allow a train down a route that is unsafe. In the model railway, this is achievable in software by using the sectioning system.

The model railway can be controlled by a software program running on a machine connected to the railway via the Java Model Railway Interface (JMRI). The software can individually control trains using unique identification numbers. The trains can be instructed to move forward or in reverse. The Junctions of the track can be toggled. The sensors on the tracks can generate events based on the entering or exiting of trains in sections.

1.1 The Problem

The physical model railway has many benefits compared to testing the controllers on real trains, however, there are still issues that can be addressed. Setting up different tests is still time-consuming to run, partly due to having to manually reset the track and trains. Other limitations include having a limited number of tracks and trains to test with unless more are purchased. Therefore there needs to be a way to make testing and extending tracks easy and cost-efficient. The solution to this is to create a full software simulation of the model track that will emulate the features of the physical track.

1.2 The Solution

A simulation has been created using Java and the user interface implemented using JavaFX. In order to get the simulation running a controller can be selected to control the trains or the user can manually send events to the simulation. The layout of the track should be specified, along with the trains and rolling stock that will be part of the simulation. The track and train configurations can be loaded with a structured JSON file into the simulation or controller. The simulation can be executed without a user interface but one is created for convenience. In order to simulate the model track more accurately, a simple physics engine was implemented. The physics engine models important aspects of a real system: mass, acceleration, friction, etc.

The positioning of trains is done using an absolute positioning system on a two-dimensional plane. The simulation enables you to create different types of controllers that can be tested on the simulation first before testing on the model track.

The project also has a track builder used to edit the track layout and create testing configurations. It allows custom tracks and custom trains to be added. To make this efficient, loading and saving of these configurations is also implemented. This allows the creation of a new track with a number of trains in different locations, which can be saved and loaded later for simulation. It also supports different software controllers. Two examples have been created to show the potential a controller can have. One controller is a locking controller that makes sure that two trains are never in the same section at once. Adding another controller is very simple as it only needs to communicate with a single interface.

In order to make the simulation more user-friendly, a user interface and visualisation was created. The interface was created using JavaFX which is one of the standard user interface libraries that comes with Java. It is what is used for the menus buttons and canvas. The visualisation displays the train's location on the track from a top-down perspective. Vector graphics are used to draw the track, showing the trains moving around the track in real time. A log of events that are generated by the track and controller software is also displayed.

The simulation was evaluated by comparing it against the physical hardware. The first requirement is accuracy. For example, the accuracy of how the trains move along tracks including stopping time and acceleration. This can be tested as covered in the evaluation section. The simulation also has to be accurate in how and when it send events to the controller. The event format needs to be identical in order to substitute the simulation for the hardware and the timing of events ensures the trains exhibit the same behaviour. This can be checked by using a controller on the hardware that was created on the simulation and observing the train behaviour is identical.

1.3 Contributions

- A Simulation and visualisation of the Model Railway have been developed. This allows user-defined software controllers to be developed and tested prior to use with the physical hardware.
- A Track builder and loading/saving functionality were created to simplify creating a variety of configurations of track layout, trains, and rolling stock.
- Two example controllers were developed to evaluate the simulation and compare against the hardware. A Routing and locking controller to control the track and trains.

Chapter 2

Background

This chapter briefly covers existing train simulator research and important components that impact the project.

2.1 Existing Train Simulators

Real world train systems do use software to automate certain parts of the process of controlling trains and tracks. However, there are still a lot of tasks humans have to conduct. There is research aimed at making routeing more efficient and there are a number of different simulators that try to simulate different aspects of the system. The main goals of these simulators are safety and efficiency [3].

Japan's Railway Technical Research Institute (RTRI) has created a number of these simulators to work towards these goals. One example of this is a simulation that estimated track damage after an earthquake. The earthquake simulator makes use of three modules, an earthquake motion simulator that calculates seismic motion in the deep subsurface. Another simulator simulates the movement on of the surface and the third simulator calculates the dynamic behaviour of the railway structure. Analysis models use the output of these three simulations to calculate the total damage. To validate the simulation, actual data from real large earthquakes that occurred were compared to the simulation output. The simulation output, that 26% of railway structures would need to be repaired, the actual repairs required was 14%.

Another simulation studies the effect of the wind and aeroacoustic effects on trains moving at very high speeds. The wind simulation uses the Cartesian grid method. The aeroacoustic calculates the sound effects using Howe's theory of vortex sound.

Finally, a simulator has been developed that model the contact between the train wheels and the track [3].

The other set of train simulators which exist and are not aimed at research are simulations for hobbyists [4]. Open rails are one of these simulators. Open Rails is based on Microsoft's train simulator which is open source and runs on the Windows platform. The goal of Open Rails is to create an open and extensible architecture for simulation, to allow for community contributions and interactions. The simulator Supports rolling stock, trains and activities. The intent for these simulators is different than the research simulations. The goal of most of the hobby simulations is to provide an immersive visual simulation experience.

The mentioned simulators are vastly different to the simulator in this project. They are either aimed at very specific parts of the train and track or for visual appeal. In contrast, this project is about enabling software control on a physical model of a railway. The project is very specific to the track that is modelled so other previous research on simulation of a railway will not be of much benefit.

2.2 Model Railway Hardware

The simulator is directly modelling the physical track and therefore it is important to know how the railway hardware functions in order to accurately simulate the system. The model railway uses the Digital Command Control protocol (DCC) to control the trains. DCC uses modulation of voltage on the track to encode packets of information. Also, junctions are controlled in roughly the same manner. Each train on the track can be controlled independently using unique identifiers anywhere along the tracks. The track can receive events to change the train speed and direction.

Train location is vital in controlling the trains in a safe manner. To achieve this the concept of sectioning is used. The track is split up into sections, where each of these sections has a unique identification number and can be made up of a number of tracks.

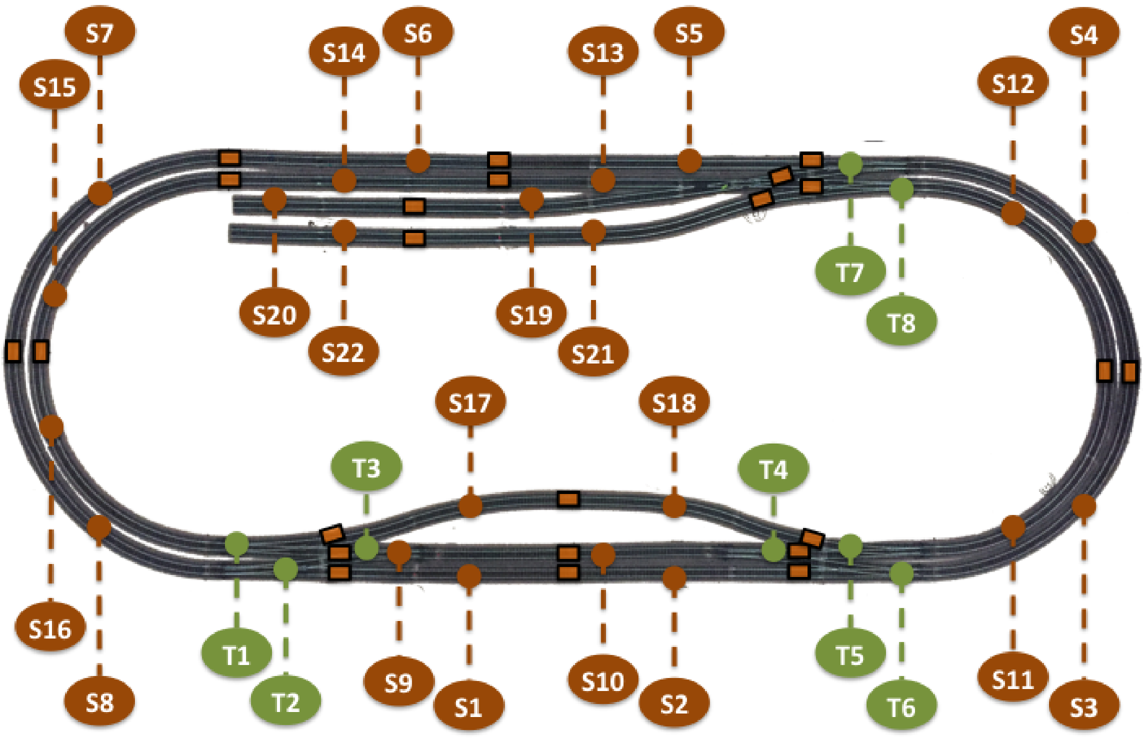

Figure 2.1: Track Layout - The image shows the sections and junction of the track. The sections are indicated with 'S' and the section ID. Every odd number in the sections are detection sections and the even number are non-detection sections. Junctions are labelled with 'T' and the junction ID. The junctions are used by the train to diverge into other lanes. [1]

A section is either a detection or non-detection section. Figure 2.1 shows the layout of the hardware. A detection section contains a sensor that can detect if a train is on it or not but not which train or how many. Rolling stock is not detectable and has to be managed by software to avoid collisions. A non-detection section is a standard piece of track that has no real interaction with the controller. To make efficient use of the sensor hardware the detection and non-detection sections alternate. Efficiency with sensors is important as the hardware support a maximum of 16 sensors. The detection alternation systems mean that when a train leaves a detection section we know it is now in the non-detection section next to it. The track can detect a change in section by looking at the state of the sensors. When a train moves from a non-detection section to a detection section the state of the sensor changes from a low to a high state. When a train moves from a detection section to a non-detection section it changes from a high to low state. Using this sectioning system we are able to keep track of the trains. This does, however, require the controller to receive starting section information of all the trains on the sections.

These sections are the only way the controller software can keep track of the trains. There are however more aspects to consider. Within these sections there are tracks. Most of these are simple and do not provide extra functionality but there are junction tracks which are used to switch the trains between tracks. Therefore in order for the controller to know where the train went it must also know the starting state of these junctions.

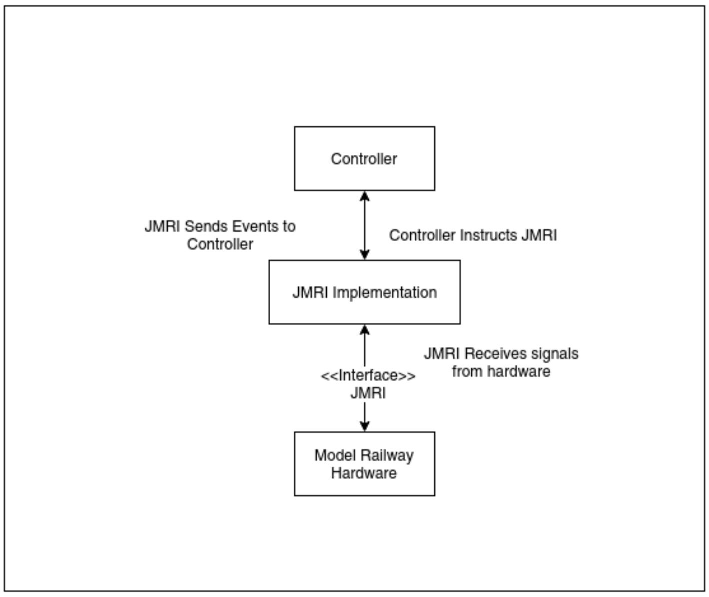
Controller
JMRI Sends Events to Controller
Controller Instructs JMRI
JMRI Implementation
JMRI Receives signals from hardware
<<Interface>> JMRI
Model Railway Hardware

Figure 2.2: The Hardware Architecture diagram shows the communication between components in Model Railway

Figure 2.2 shows the architecture of the model railway. The arrows in the architecture indicating communication between the components. The Model Railway Hardware generates events and sends it onto the Java Model Railway Interface Implementation which interprets the event and passes it onto the controller. The controller then updates its state based on this data and can send an event back to instruct a train or toggle a junction. The events supported are:

- **Section state changed event** - This is what the controller uses to determine the location of the trains. It passes through the ID of the section

- **Set Junction** - The controller can manage junctions on the track by specifying the ID of the junction to toggle the state to either thrown or not thrown.

- **Set train speed** - Sets the target speed of the train, 0-100 percent power, trains vary in top speeds.

- **Set train Direction** - Sets the direction of the train to forward or reverse.

2.3 Java Model Railway Interface

The purpose of the Java Model Railway Interface project (JMRI) is to create a range of tools to control model railways from the computer [5]. JMRI is an open source project widely used by hobbyists to control model railways. As previously mentioned the model railway uses Digital Command Control, this can be quite complicated to program for, therefore JMRI has created interfaces to smooth out this process. There is also a standalone program that can be customised to your needs. For the purposes of this project, the tool was used as a library. JMRI provides an interface to connect to the hardware through a USB port, it establishes a connection and handles the transfer of messages. An important method in the interface is the message method, there is where all the messages from the hardware go through [5]. There are a couple more interfaces used in the project from JMRI, `LocoNetListener` and `ThrottleListener`, which are used to listen for events from the hardware.

2.3.1 JMRI Implementation

A small standalone client was developed at VUW on top of JMRI for use with the model railway. The client uses the JMRI interfaces to control the trains on the hardware. The implementation provides a command line interface for controlling trains on the physical track. It simplifies the JMRI software into a simple event based system. There are a number of commands to control the track and trains. For example `Start id speed` to start the train at a given speed. `Turnout id` to set a turnout. `Loop id array` sections to make the train loop. These commands can easily be extended by creating a new keyword and defining a method with the behaviour you desire.

The software has a direct connection to the model railway which it uses to communicate events and instructions. The messages from the hardware then get translated into simple events in the Event class. For example, a Section changed event. The class also implements a listener interface with a `notify(Event e)` method. This is used by the controller to send Events back to the hardware. The notify method translates the event back into a message the JMRI interface can understand. The events the client supports are:

- **Power Changed** - Railway is off or on
- **Section Changed** - This provides an integer section identifier, and a boolean indicating whether the event represents a train entering or leaving the section.
- Speed Changed: This provides an integer for the train identifier and a float for the speed.
- **Direction Changed** - This provides an identifier for the train and a boolean for direction, forward or backwards.
- **Emergency Stop** - The identifier of the train to stop.
- **Junction Changed** - The identifier of the junction and a boolean for thrown or not.

The controller also implements the listener interface and registers with the `ModelTrack`. The `ModelTrack` also registers with the controller. That way they can send events to each other. The software also has a controller package which is where the controllers that are created from the simulation will go into. The simulation has been created so that the interfaces are the same as the client, therefore, transferring controllers into it is trivial.

2.4 JavaFX

JavaFX is used for obtaining user input and drawing the user interface within the simulator. JavaFX is created by Oracle and is “a set of graphics and media packages that enable developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms” [6].

JavaFX is one of the two standard GUI libraries that come with Java. It is used throughout the program to render the window, menus, buttons, text, etc. JavaFX also provides a canvas, which has a range of methods used to draw shapes, this is what is used to draw the trains and tracks in the middle of the screen and everything is updated at 60fps.

To decide on which User interface framework to use the pros and cons needed to be considered. The default style of the windows, buttons and menus of the frameworks are different I found applications made by both and found the modern look of JavaFX a better fit. Looking into some tutorials on JavaFX the code looked more concise and easy to understand for my purpose, therefore, I decided to go with it. Contrary to Swing which I have used in the past where the lines of code required can escalate quickly. After finishing the User Interface I concluded that the right decision was made. Creating layouts was easier and required less code.

Chapter 3

Design

This chapter covers the design of the program and the important design decisions made throughout the project. Before explaining these design decisions, it is important to describe all of the components that make up the project. There are four main components: the Java Model Railway Interface (JMRI), the JMRI Client, Train Controllers, and the Simulation.

3.1 Model Railway Components

- **Model Railway Hardware** - The main focus of the simulation is emulating the physical hardware. The track needs to be modelled as software; to do this, the physical attributes of the track and trains needs to be considered, along with how events are generated and received to control the trains and junctions.

- **Java Model Railway Interface** - The Java Model Railway Interface handles the connection to the Model Track Hardware through a USB connection and provides an interface to receive the events generated by the track, as well as a way to send events back to the track.

- **JMRI Client** - A standalone Java program created at VUW that implements the JMRI interface to receive and send events. The program converts the events received from the hardware into a simple format that the controllers can use to manage the trains. The controller can then process received events, as well as send new events back to the hardware, which is done by sending an event to the JMRI implementation. The JMRI converts it back into a format that the hardware can process, and finally, the event is executed on the hardware.

- **Train Controller** - The Train Controller is the software that decides what the trains should do based on the events received from the hardware. Also, it can control parts of the hardware, such as junctions. Separate controllers can be implemented to show this, two controllers have been implemented for testing: a locking controller and a routing controller. The locking controller ensures that no two trains are in the same section at once, while the routing controller uses Dijkstra's shortest path algorithm to find the shortest distance to a trains destination. The controllers are placed inside the controller package inside the JMRI implementation. The controllers should work interchangeably in the simulation and the JMRI client.

- **The Simulation of the Hardware** - The Simulation is the main aspect of the project, which substitutes the hardware and the JMRI client for the controllers. The hardware is simulated and outputs events that are converted into a format the controller can understand. The controller then sends events back to the hardware to control the trains, the output of which is shown in the visualisation.

3.2 Model Railway Hardware

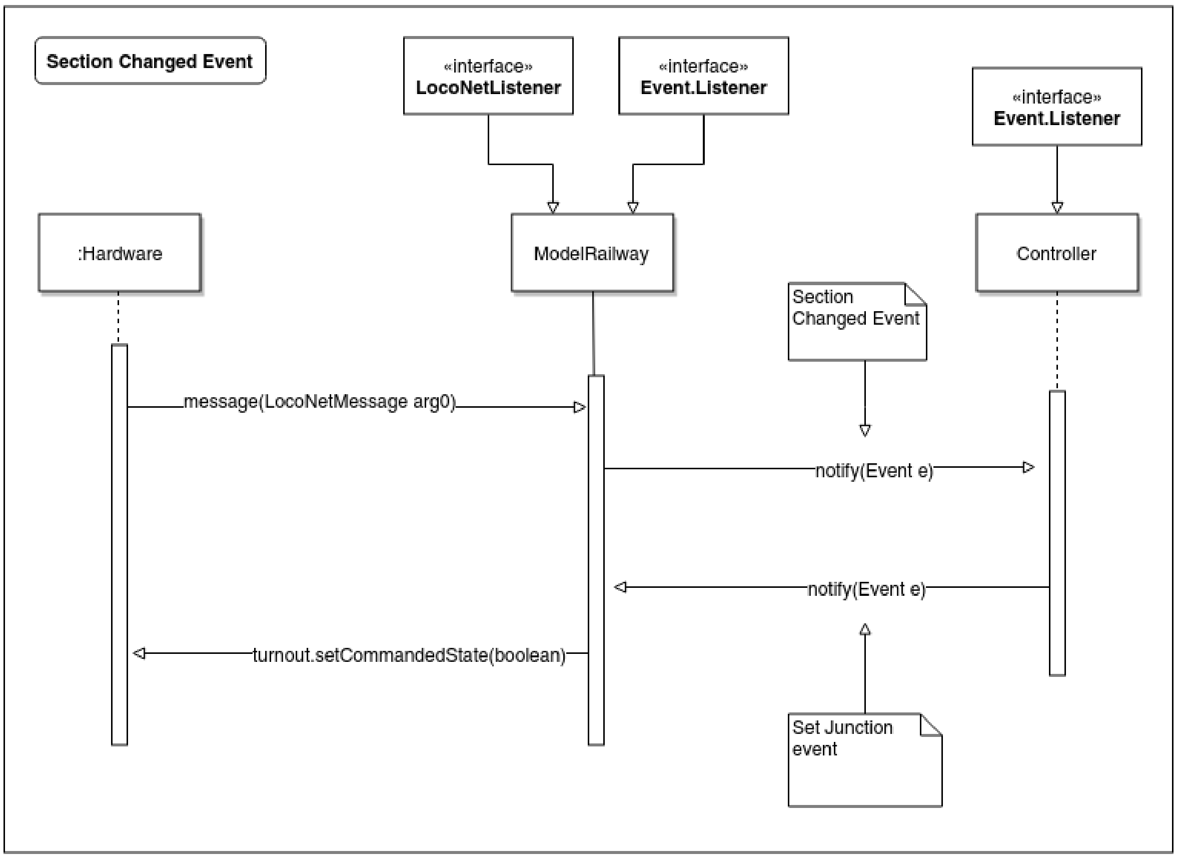
Section Changed Event
«interface» LocoNetListener
«interface» Event.Listener
«interface» Event.Listener
:Hardware
ModelRailway
Controller
Section Changed Event
message(LocoNetMessage arg0)
notify(Event e)
notify(Event e)
turnout.setCommandedState(boolean)
Set Junction event

Figure 3.1: Event Generation Sequence

The `SectctionChanged` events are fired when trains move in and out of sections and communicate with the JMRI client impacts the design of the simulation. A detailed example of how this occurs in the hardware can now be discussed.

The diagram 3.1 shows the approximate sequence of events that occur when a train changes section on the hardware and events fire, and how the controller responds. The hardware sends a message through the JMRI `LocnetListener` Interface, then the `ModelRailway` class converts this into an event to send to the controller (`SectionChanged` event). The controller then decides whether a junction should be toggled to route the train where it needs to go and sends an event back to the `ModelRailway` class. The model railway interprets the event and calls the appropriate method on the JMRI interface to toggle the junction on the hardware.

The `Event.Listener` interface is a key interface. The interface is used to pass the events from the hardware to the controllers. This is where the hardware is abstracted away and replaced with the simulation. The consequence is that the simulation must produce identical events to that of the hardware to ensure controllers can be used for both platforms. Following the event format is not too difficult, for example for a section changed event the

ID of the section is all that is required. However the exact time the hardware sends these events is not known. The simulation assumes the event fires when half of the train is in the next section, however, if needed this can be updated as the length of the train and the exact location is known to the simulation.

3.3 The Simulation

The simulation is based on the hardware, therefore, the design of the simulation has to model the constraints of the hardware. The simulation must also model real world factors that impact the train and rolling stock behaviour. The hardware supports different track types. The types modelled in the simulation are:

- **Regular track** - Contains a source and destination.
- **Junction** - A junction contains one source and two destinations, the destination for a train depends on whether the junction is thrown or not.
- **Buffer track** - A buffer track is a track with a source but no destination.

These tracks are then combined into sections and given an ID identical to the hardware. These tracks can be connected in many different configurations for different layouts.

The simulation models train with multiple attributes these include, acceleration, max power, brake force, length, and weight. These attributes are important to the accuracy of the simulation. Rolling stock requires the weight and the length to be known. Weight to impact the speed of the train pulling it and length to model collisions. The train is connected to a rolling stock by driving into it at a low speed. Alternatively, the stock can be connected to the train on initial placement. The simulation also includes a basic physics engine that uses the train attributes to calculate the speed acceleration and stopping time which allows for more accurate location updates.

The simulation works in real time, an update method updates all the elements on railway based on how much time has passed. This is separate from the update to the visualisation, this is so the simulation can run without a user interface or visualisation. The purpose of separation is that it enables automatic testing to be set up by using the update method.

The hardware sensors handle positions on a section by section basis. In contrast, the simulation uses an absolute positioning system in order to detect collisions and produce smooth movement along tracks. However from the perspective of the controller the location is on a section by section basis identical to the hardware.

Supporting multiple controllers inside the simulation is key and considered in the design. A controller needs a `register(Event.Listener listener)` method which allows the controller to send events back to the simulation using `notify(Event e)`. The controller also needs to implement `Event.Listener` to enable the simulation to send events to the controller. To make this process even easier when using the user interface a drop down menu lists the possible controllers.

3.4 Design Features

There are a number of key software design principles that were decided on at the beginning and were followed throughout the project, separation of concerns being one of the top priorities. The design of the project can be separated into four main parts: the simulation, the user interface, the track builder, and controllers. Abiding by the principles of separation of concerns in this way has a number of benefits. For example, when adding a new feature, the part affected should have limited effect on other parts of the project. It will also assist when testing, as separate units can be tested in isolation, making debugging much easier.

The control of communication of information between the controllers and the simulation is important. The railway hardware passes limited information to the controller, therefore, the simulation should send identical information. A controller that works on the simulation must also be able to work on the hardware, this will not be the case if it received extra information. e.g. the simulation knows of the speed of the trains and their X and Y coordinates. The controller could, in theory, use this information for extra control by using the speed data in calculations to make sure that the trains never collide. However, this information is not available from the hardware. This means that if the simulation is substituted with the hardware, the controller would not work. This is accomplished by using two interfaces: The `Event` interface that specifies the format of events and the `Event.Listener` interface that specifies a method to listen to events. The interfaces are the only way the simulation communicates with the controller which is identical to the hardware. However, there is one more concern, the input of starting state to the controller. The controller cannot use state from the simulation as it would not have access to it on the hardware instead the controller loads the configuration from a file.

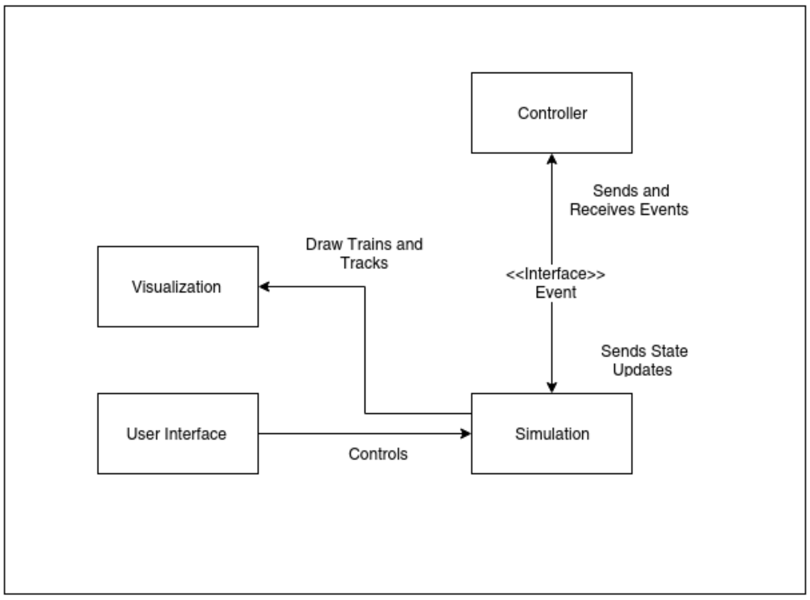

Figure 3.2: Simulation Architecture, separated out into MVC components the model is the Simulation the view is the visualisation and the controller the User Interface.

Extensibility was also a focus. With simulations, there is always the potential desire to add additional features, so it is important for the project to be extensible. An example of how extensibility was ensured with this project is exemplified by the ability to add new track types. To add a new type of track, the `DefaultTrack` abstract class needs to be extended and the relevant methods implemented, but no changes need to be made to the way the rest of the system works. Another example of this is the user interface; it is separated from the logic of the software, therefore an entirely different user interface could be added at will, or the existing one could be extended with ease.

The architecture of the project 3.2 follows the Model/View/Controller design pattern. The Model holds all the data about the trains and tracks and controls their movement and logic (Simulation). The Controller is anything that the user interacts with, e.g. the toolbars and menus, as well as the mouse and key listeners (User Interface). The View is the canvas that the trains and tracks are drawn on, and is updated by the model (Visualisation). By following this architectural model, tests can be conducted on the simulator without the user interface or visualisation needing to be run. Furthermore, a tick method can be called to mimic the simulation running in real time. With this, tests that would otherwise take a long time can be conducted quickly, easily, and automatically by setting the number of updates, and certain conditions can be checked, such as making sure that the trains in the system never collide. One more aspect to consider is the control of information between the simulation and the controller.

3.4.1 User Interface Design

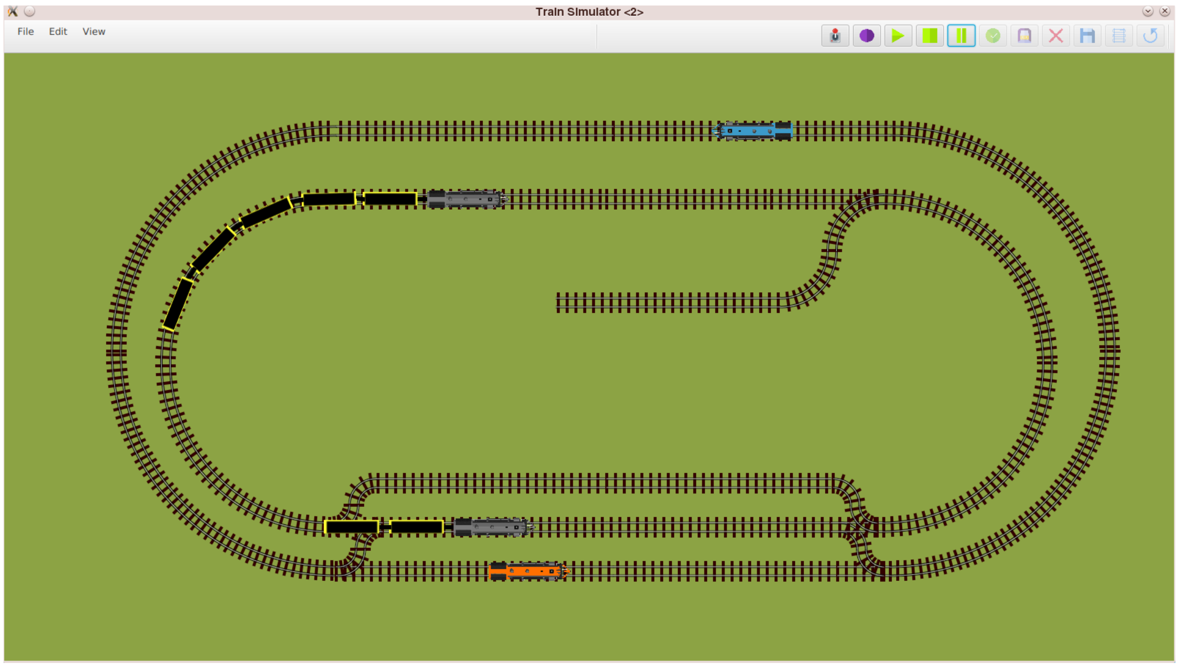

Figure 3.3: The Main Screen of the program that shows the created track layout and trains in the current configuration.

The most important part of the project is the railway layout and the trains. This is reflected in the user interface design. The main control of the simulation is handled through the buttons on the toolbar. See figure 3.4. This toolbar is aimed to be easy to use, so hovering over a button provides the user with a tip indicating its function. The buttons not applicable to the current mode are greyed out. Also, the buttons and interface of the program take up very little space to allow the track to be as large as possible. See figure 3.3.

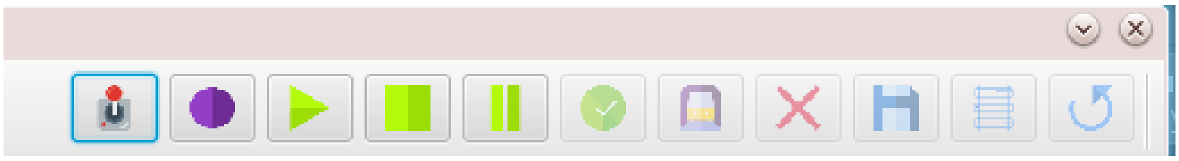

Figure 3.4: Toolbar

The controller button allows selection of one of the available controllers. The circle button is used to manually send events to trains on the track. The play, pause buttons are used to suspend and resume updates and the stop button resets to the starting configuration.

There is an extra element that reduces the space the track has, which is the event log. However, the user is able to minimise the log when its not needed. Furthermore, most of the menus that the user interacts with are pop-up menus, which means that they are only there when needed and hide themselves as soon as they are not. These pop-up menus are used when the user wants to add a train to the track or send a speed event; for simpler actions like toggling junctions, the mouse is used directly.

3.4.2 Track Builder

The track builder user interface is similar to the main screen; See figure 3.5. This is so that the user encounters an easy learning curve. The main difference in this mode is the extra available buttons. The buttons from the simulation mode are still shown but greyed out to indicate that they cannot be used. The user can easily select a track from the example tracks listed on the right side of the screen and dragged onto the canvas to place them. Tracks can only be placed in valid locations - which is in the proximity of another track, or anywhere for the starting track. To help the user with placement, when placing a track, it is highlighted green or red to indicate valid or invalid placement respectively. There are also keyboard shortcuts to make track selection more efficient: pressing R cycles through the available tracks, and E changes the direction of the tracks.

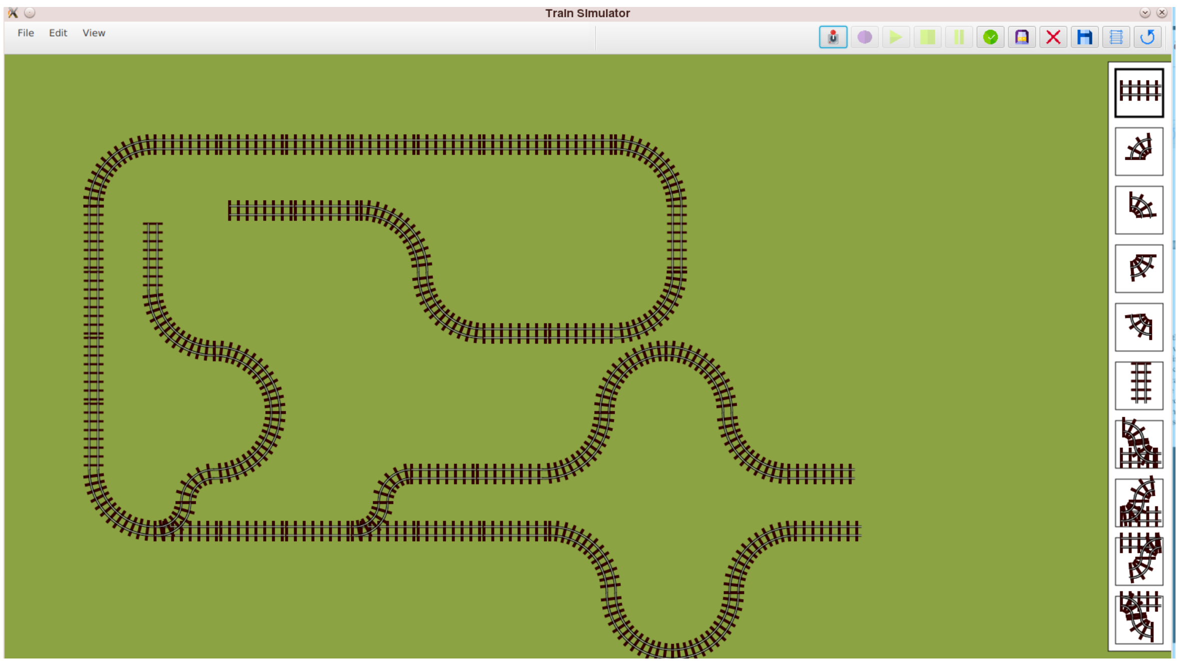

Figure 3.5: Track Builder

Chapter 4

Implementation

This chapter covers implementation details of the Simulation, Track Builder, and controllers.

4.1 Modelling Physical Objects

There are three main objects from the physical railway that need to be simulated. These are the tracks, trains and rolling stock. Objects that are simulated require fields such as their location, weight, and shape.

4.1.1 Tracks and Sections

The hardware splits the railway into multiple sections. Each of these sections can contain multiple tracks. The simulation implementation is similar, the `Section` class is used to store an array of tracks and IDs for each section.

The tracks are split up into three different types: straight, curved, and junctions. The simulation uses a two-dimensional plane for object locations, therefore the starting location of a track piece can be represented by an x and y coordinate. All track types inherit from the abstract class `DefaultTrack`. The common functionality across all track types is placed in the abstraction to reduce code duplication see figure 4.1.

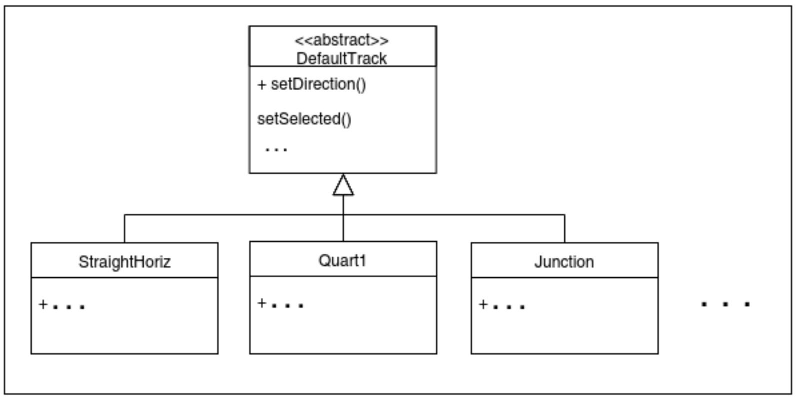

Figure 4.1: The Track hierarchy shows is a simplified version

To set the location of a track, it requires a track to connect to (unless it is a starting track). If it is not a starting track, it will be placed based on where the track it comes from ends using the `setStart(DefaultTrack from)` method. This ensures tracks are lined up accurately and trains can move from one to another. It also simplifies specifying layouts, instead of storing the x,y value of each track the ID of the track it originates from is stored.

Another implementation considered was allowing tracks to be placed anywhere and having a system that checks if all the placements are valid afterwards, or even allowing an incomplete track. However this would not be convenient for the user as they would have to line up tracks perfectly on a user interface or specify exact (x,y) coordinates in a file.

Tracks know their own shape which means they can work out the trains next location based on the train's current location and speed. Tracks have two main methods used by the simulation to move trains `getNextPoint(..)` to update the train's location and `checkOnAfterUpdate(..)` to check if the train will still be on the track after it moves. The track needs to know the direction the train is moving (forwards or backwards) and the orientation of the train relative to the natural direction of the track, this information is stored as boolean values in the train object.

The train receives its next position by asking the current track where it would move to next at a given speed. The track does not know inherently which direction the train is coming from. Therefore the track also needs to know if the orientation of the train goes along the direction of the track and if the train is moving forward or in reverse. Consider the following example in 4.2:

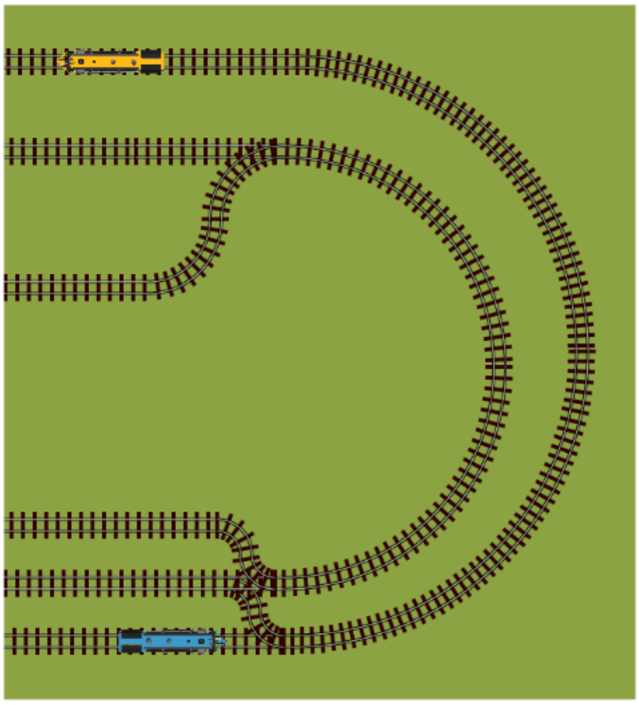

Figure 4.2: Direction indicates track direction. The dots separate the individual tracks and the triangles indicate the trains orientation.

The natural direction of the track the bottom train is on is right, while the natural direction of the track at the top is left. When the track object checks where the bottom train should go it knows to increase the x coordinate because the train is going forward along the natural direction (right). The top tracks direction is left which means it knows to decrease the x co-ordinate.

There are more cases to consider. Trains can be placed down in a different starting orientation against the natural direction of the track. See figure 4.3

1. The first case shows the simple case where the train is moving forward along the track

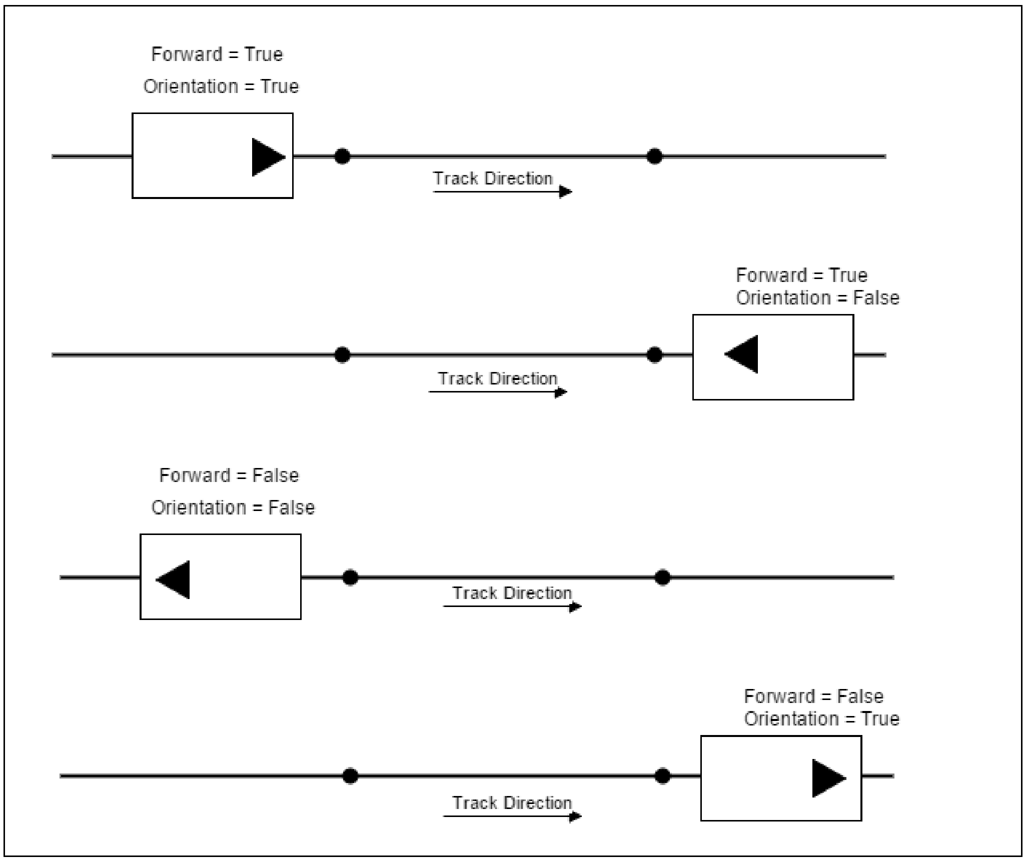

Figure 4.3: Different track directions example

direction (train moves right).

2. In the second case the train was placed down against the orientation of the track, therefore the orientation inside the train is set to false. Note that both trains are moving forward, however, for one the track has to increase the x co-ordinate the other decrease it. (train moves left).

3. In the third example the train is going backwards and is placed against the direction of the track (train moves right).

4. In the last example the train is going backwards but is placed with the direction of the track (train moves left).

Junctions are a special track, they are used to diverge trains into different lanes. Unlike a regular track, there are two exit points. This increases the use cases to consider. When the train enters from one of the tracks exit points the above solution no longer works. Instead, the track the train came from must be stored. Other use cases include, when the junction is thrown, there are certain directions that trains cannot cross the junction on, as doing so would lead to derailment. This was fixed by checking where the train originated from, to work out which track of the junction it is on.

The implementation reused existing track classes and composed three of them into a single junction piece. This meant that the junction class held three track objects - two curved and one straight. In some ways, this did simplify things, as methods already existed for moving the train along these simple tracks, along with methods for bounds checks, etc. The downside was that somehow the junction needed to know which internal track the train is on. Another side effect of this decision was that when the simulation would assign the next track if it was a junction it had to also store the internal track of the junction.

4.1.2 Simulating Trains

There are two types of train object in the program: `Train` and `DrawableTrain`. A drawable train has the information required for it to be displayed in the visualisation and the simulation, as well as attributes like its exact coordinates, current speed, and acceleration. The standard train object only holds variables like its length, orientation, and direction. Orientation describes whether or not the train goes along with the orientation of the track, and direction describes whether it is moving forward or backwards.

The trains physical attributes can easily be adjusted while the simulation is running with sliders. See Figure 4.4.

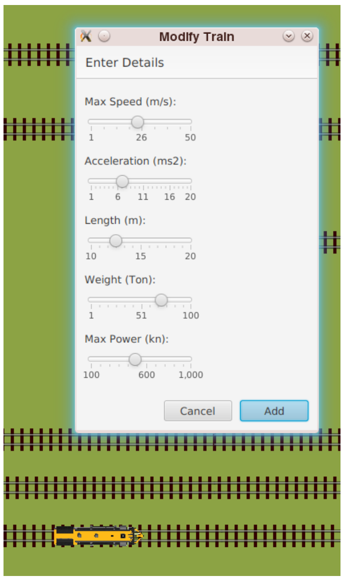

Figure 4.4: These sliders can modify train attributes that effect the physics calculations

Rolling stock is anything connected to a train. They are invisible to the hardware recall section 2.2. A standard stock object has an ID, length, and weight. The length is important for the controller to work out what a safe distance is behind a train with one or more rolling stock connected. The weight is used in the physics calculations. If there are a large amount of heavy rolling stock connected to a train, it would have an impact on its acceleration and stopping time, so this has to be modelled for accuracy [7].

The drawable rolling stock object has some additional information. To the simulation, it is perceived similar to a train. The way it updates its location and is drawn is the same as a drawable train. The main difference is that, when connected to a train, it does not calculate the distance it has to move - this is instead done by the train pulling or pushing it and the information is passed to it.

In order to make the simulation more accurate, these objects need to have a sense of scale compared to the real world. This is the ratio of distance in the real world and distance on the screen. For example, if we decide one pixel on the screen should be equal to five metres in the real word, then if the user creates a train 15 metres long in the new train menu, this will be converted to 75 pixels. Using this conversion factor, speed in metres per second can also be converted to distance in pixels over time changed, e.g. if 4 seconds have passed and the train is moving at 10 metres per second, then we then know that it has moved 40 metres. Converting that to pixels: $40 \times 5 = 200 pixels$.

The actual size an object takes up on the screen may vary across devices. The scaling is based on the number of pixels an object should be and does not consider the physical size of these pixels. This could be changed in the future by considering the pixel density of the screen, but it is not an issue since the objects on the screen are still to scale relative to each other.

4.2 Collision Detection

Collision detection accuracy is important in the simulation, as a situation where the train crashes in the real world but not in the simulation should not be able to occur. There are multiple ways to do collision detection. The implementation that was chosen was a bounding box around the trains and rolling stock. Points on the front and back of the train are then checked if they are inside the bounding box of another train or rolling stock.

Creating a bounding box for a rectangle may seem simple at first, since you simply need to take one of the corners and use the width and length to calculate if a point is within the rectangle, but this only works if the rectangle never changes orientation (see figure 4.5). The conservative approach would be to create a naive bounding box that uses the length of the train to create a square bounding box around the train, but this would lead to false positives in collision detection.

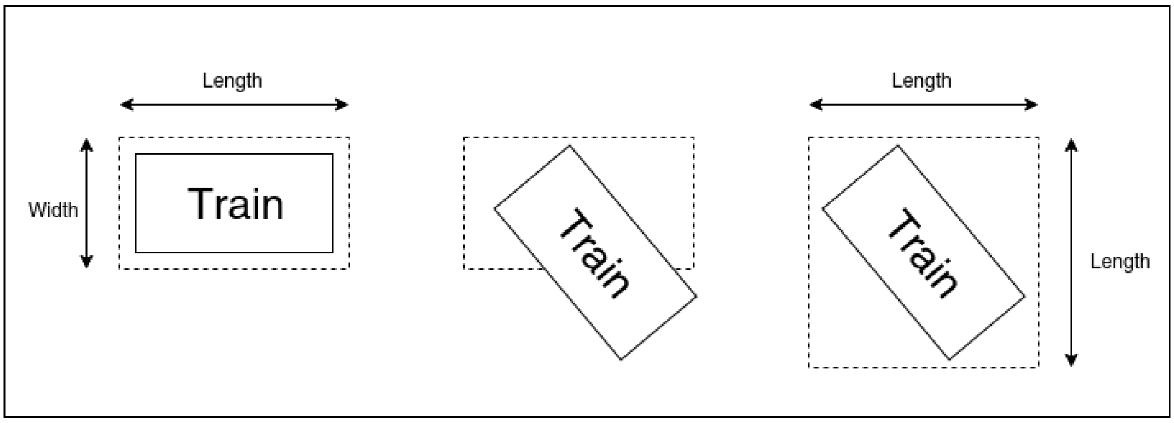

Figure 4.5: Naive Bounding Box

The naive bounding box was too inaccurate; it made trains going around curves next to each other crash, despite them not touching. Therefore an alternative approach was tried and successfully implemented. The simulation knows the centre point of the rectangle bounding box and the rotation of the train, so this means that the cross product and magnitude can be used to work out the exact coordinates of the four corners of the rectangle. After these four points have been calculated, the point that needs to be checked whether its inside the rectangle is used along with the four points to create four triangles, as shown in fig 4.6. The area of these triangles are calculated and added together then compared against the area of the rectangle; if the resulting total area is greater than the rectangle, then it means that the point being checked is not within the trains bounding box.

However, this calculation is relatively expensive and has to be computed every tick. An optimisation that was made was to use do naive bounding box calculation first; then, if the point is inside this naive bounding box, the more expensive calculation described above can be done to make sure that the point is inside the actual bounding box.

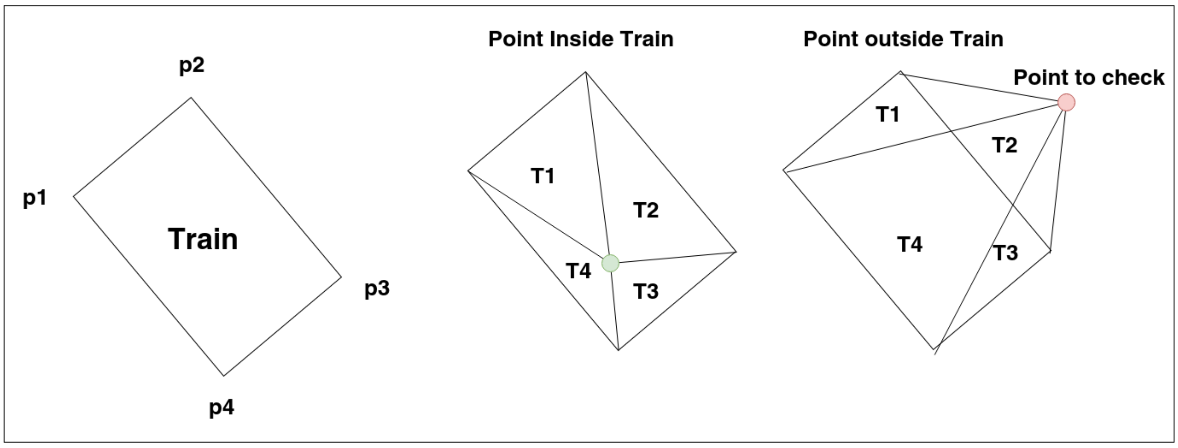

Figure 4.6: Accurate Bounding Box

4.3 Physics Modelling

To more accurately simulate a railway track, physical modelling is performed. This is done by a simple physics engine. This engine models a trains acceleration, weight, forces, and friction on the track. The methods used are similar to that used in force directed layouts [8]. The formula used is:

$$force = mass \times acceleration \tag{4.1}$$

This is rearranged to find the acceleration:

$$acceleration = force/mass \tag{4.2}$$

The force represents all of the forces acting on the train. The forces considered for this simulation are the engine force applied in a direction (forward or reverse), the braking force (applied to the opposite direction it is travelling), and the friction forces which essentially makes it more difficult to move. The mass is based on the weight of the train and any connected rolling stock. Acceleration is the change in speed over time measured in metres per second per second. The formula used for force is:

$$Force = trainpower(frictioncoefficient \times (mass \times gravity) \tag{4.3}$$

A simplification was made with the weight. A train and its rolling stock are considered to be one unit by the simulation, so the weight value is the total weight of the train and the rolling stock together if it has any. This could mean something that is able to be pulled in real life may not be able to in the simulation [7]. This is because, in reality, trains have a slack between the rolling stock and the train, so only one is pulled at a time. However, this is a minor detail that can be fixed by reducing the static friction to the kinetic friction coefficient. The simulation uses the total weight of trains and rolling stock and applies static friction when the train is not moving.

An interesting problem arose when trains were sent an event to change direction. Without physics modelling this procedure is trivial; the direction of the train is changed instantaneously, and the next update the train moves at the same speed it was going, but in the opposite direction. However, with the simple physics engine engaged, the train is required to brake to a stop with braking power and friction, then speed up again in the opposite direction. To make this realistic, instead of instantly changing direction, a boolean for changing direction is set to true which causes braking to be applied until the trains speed reaches 0. At this point, braking is turned off and the train begins acceleration in the new direction.

Another consideration was the power of the trains. The formula used to calculate the acceleration of a train requires all the forces on the train to be calculated. The only impacting force pushing the train forward is the engine force, so the amount of power the train engine can create needs to be known. In the first implementation, trains had unlimited max power, unlike the hardware which is capped. The simulation slowly increased the amount of power output at every tick if the target acceleration had not been reached - similar to slowly pushing down the gas pedal in a car. This implementation had many issues: it required a certain point to be reached to know that the amount of acceleration was enough, so the train could stop increasing power. Also, when the target speed was reached it would start decreasing power, but there was too much of a delay and this led to the train going over the target speed. It would then re-adjust to below the target, and this cycle would repeat until it finally narrowed in on the target speed.

The second implementation saw the creation of a helper function that calculated the exact power required for a train to reach a certain acceleration by rearranging the acceleration formula. The acceleration of one of the hardware trains was measured and it was approximately $7m/s^2$. This value was given to the function which uses the formula

$$frictionforce = frictioncoefficient \times ((trainweight + rollingstockweight) * gravity) \tag{4.4}$$

$$power = targetacceleration \times (trainweight + frictionforce) \tag{4.5}$$

The friction coefficient varies depending on if the train has already started moving or not.

The power formula deliberately does not take into account the weight of the rolling stock, as adding rolling stock should inherently decrease the acceleration of the train. This assumes a train would apply the same power independent of whether or not a rolling stock is connected.

4.4 Clock Tick

All updates in the software are based on a tick which, when called, records the time difference between the last time it was called. This difference can then be used to update everything accurately according to how much real time has elapsed. For example, this is required to update the location of a train accurately. A train has a speed in meters per second, and depending on how much time has elapsed since the last tick, we can convert that difference to the actual distance the train should move that tick. Once everything has updated, locations and other details on the screen can be redrawn, showing the new positions. This ensures that everything moves at expected rates regardless of the host CPU speed or frame rate.

Running the simulation with the visualisation uses the real world time as it is running. However, when running the simulation on its own the time between updates needs to be set. To enable this the `setTestMode(true,tickTime)` method needs to be called. Changing the simulation to this mode applies the tick time to each update instead of using real time.

4.5 Generating track events

In the physical world, generating events is achieved partly with the hardware. When a train arrives at a detection section, the hardware sends a signal stating that it has been triggered, and in turn, an event is generated saying that the state of the section has changed. It does not care what train triggers the event. It has no idea about any of the tracks around it, nor its length or curve or slope, etc. Therefore, since this is part of modelling the physical track, it would make sense to generate the events in the part of the program responsible for modelling the hardware. The problem is knowing when the event should be sent. A train that is in the simulation only knows of the track that it is currently on. In the implementation of the train, there is no way of knowing where the train is relative to any one section of the railway.

The way the program knows where a train is is by employing a two-dimensional absolute positioning system. All objects with a position are given (x,y) coordinates specifying their location on the plane.

These are used to draw all the elements on the screen and also to determine where the train should move to next. The idea for this comes from the fact that, in the physical world, the train is bound by the physical layout of the track. To model this, the track object works out where the train should be. The track is able to do this given the current location of the train and the speed at which the train is moving. This makes adding more types of track easy as the new track is in charge of determining where the train goes next. No changes need to be made to the train or any of the other tracks.

This implementation also allows for a simple way to check when a train has entered a new section. Each piece of track knows its starting location, length, and shape, and can, therefore, calculate its bounding box. Because the bounds of a track are known, the track can work out if an update (the next tick) of a train at a certain speed will be outside these bounds, and consequently on the next track. This means that we have a point where the program knows an event should be generated. This is handled by checking if the train will be in the same section each time it moves: if it is not in the same section, then update the track and send an event to the modelled track with the ID of the section that has changed. It can then send this on to the control software which only sees the section ID, but it can use this to interpret which train has moved where based on previous state information.

Train movement accuracy is important. A point where this is important is when moving the train from one track to another when a check is made to determine whether the train will still be on the track if it moves. If this is not the case then it will be placed directly at the start of the next track. This is not accurate because with one update it may need to be moved a small amount further than the start of the next track. This problem is fixed by checking how much distance is left to move after it moves to the end of the track. For example, if a train moves 10 pixels in one update but there are only 5 pixels left on the track, the train will be placed 5 pixels after the start of the next track.

This chosen design means that there is no extraneous information in the software controller class, and there is still a separation between the visualisation and the model track.

Another benefit as the time could be input as a parameter per update to run the simulation outside of real time constraints.

4.6 Rendering

The curved tracks support 90-degree turns. The implementation was based on the assumption of only needing sections with 90-degree turns. With this, the only information required is the previous piece of track and the direction of the track, and from that, it can figure out where to place the next piece.

From the beginning, two different ways of implementing the rendering of tracks were considered: either have only 90-degree curves - which is limiting but would still achieve the initial goals - or allow tracks of any curvature. The decision to only allow 90-degree curves was made, 90-degree tracks meet the specifications and also because the other option would take up a lot of time that could be spent elsewhere. At first, implementation to allow tracks with any curvature to be rendered was attempted, but it was taking too much time.

Drawing the train in the correct location is simple on straight tracks, as working out the next location simply requires the train's speed and orientation information and a conversion of those values into pixels and a change in the X coordinate on horizontal tracks and Y on vertical ones. Figuring out the next location on a curve requires a bit more work, but because of the simplification of them always being 90 degrees, a couple of calculations are used to work out the next location.

The train has a field which holds the number of degrees it has already moved through a curve. Another field stores the amount to move in one update (speed). The train has to rotate 90 degrees through a curved track, to calculate how much it should move each time the degrees left to turn are used along with the potential number of updates to do it based on its speed. For example, if a train has moved 45 degrees out of the 90 and every update moves 5 pixels along the curve with a curve length of 100, this means that there are 10 more updates before the end of the track is reached. 45 degrees divided by 10 updates = 4.5 degrees per update. This formula has to be used at every update in case the train increases in speed the next update, which is common due to acceleration. Another consideration is changing direction while on the curved track, in which case the degrees completed will have to be inverted.

4.7 Track Builder

Creating new railway configurations was slow and tedious. Each track had to be manually specified along with the track it comes from and goes to. This process was also very error prone. The solution for this was creating a track builder. The track builder provides an easy way to create and save new tracks and configurations. Other than creating tracks, it also allows the additions of trains and rolling stock to sections.

A helpful feature of the track builder is the ability to drag a track piece to the end of the track you want to connect it to. In the initial implementation, clicking on one of the available pieces would add it to the end of the previously placed track. This was fine when only one circuit was being built, but the existence of junction tracks makes it ambiguous where you want to add the next track. The ability to drag track pieces onto the design solves this issue and allows for more complex tracks to be created.

To smooth out the process more for the user, the track builder also checks the location you drag the piece to, to see if it is a valid placement. If it is valid, it highlights the track green; if not, red. When the user releases the track in a valid location, it does not place it where they released it exactly. Instead, it searches for the nearest track that it could fit against and uses that piece to find the exact location to place the new track. This way, the user does not have to spend time placing the track in a pixel-perfect spot; instead, it detects if it is within a certain distance from a valid spot, and moves it to the perfect location when the user releases the mouse so that the tracks line up properly.

There is another consideration when connecting the tracks together: see 4.7. Here, when the final piece between the junctions is inserted, the junction on the left would be set as the straight piece origin, however, the destination would not be set. The solution is to have a final pass through of all the track pieces, and any track without a destination are then checked to see if any other track can connect to it, and if so, it is set to its destination. But it is also valid to have a track without a destination; these are buffer tracks.

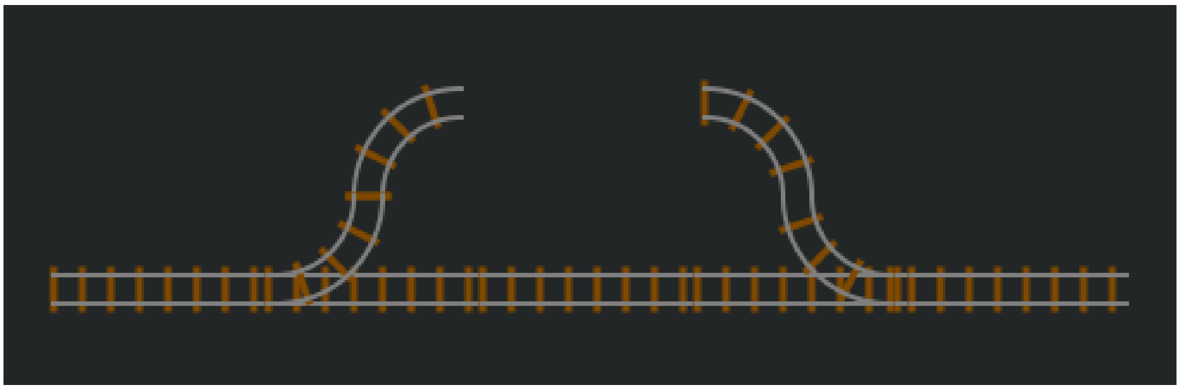

Figure 4.7:

4.8 Saving and Loading

To help with the convenience of testing different train configurations, loading and saving functionality was added. This way, if time is spent in the track editor creating a large track, it can be saved and used it again at later. The same goes for custom trains with specific weights and engine power values. Save files are in the form of JSON files, which have an object oriented structure and support arrays. The main elements of a save file for this software include arrays of sections and arrays of tracks for each of the sections. There are also arrays for trains and rolling stock.

Saving with this structure makes it easy to load back in. Section objects have attributes like the ID of the section and the section it comes from and goes to, as well as whether it contains a junction section and whether it is the starting section of the track.

The tracks are saved with what type they are: a straight track or a curved one, as well as what degree the curve is, if applicable. Also saved for each track is which track it comes from and where it connects to. This could be multiple other tracks in the case of a junction track. The saved data for a train includes whether it is a custom train or one of the default trains; if it is custom, it will have several attributes, such as engine power, length, etc. If it is one of the default trains, it will just have its name and the starting section of the train. The save data for a rolling stock has a starting section and if it is connected to a train, and if so, the ID of that train.

4.9 Controllers

The controller has to be completely separated out from the simulation. Although the simulation and the controller are essentially modelling the same domain, the data they have access to is different. The controller only has data that comes from events and the initial configuration, which includes track layout and IDs and the starting locations of trains. This meant that the same classes had to be created for the controller: `ControllerTrain`, `ControllerTrack`, etc. The controllers implemented currently respond to the events generated by the physical track or simulation by implementing the controller interface which had the method `receiveSectionEvent` with the ID of the section.

An abstraction was made for controllers by creating an abstract controller class. This is because all controllers receive an event and have the same behaviour for determining which train relates to the event and if it is a section entry or exit event. The two current controllers that subclass the abstraction `DefaultController` are the locking controller and the routing controller. The locking controller ensures that no two trains are in the same section at the same time. The routing controller uses Dijkstras search algorithm to find the shortest path to the destination; it creates a graph out of the sections and junctions and uses the length of the tracks to determine the shortest route, automatically toggling the junctions in the track so that trains will take this path.

4.9.1 Loading Configurations

The controller requires the track layout and the trains and their starting locations on start up. There are two ways to load the information into the controller: the first is to pass in an array of `ControllerSections` and a list of `ControllerTrains`. This is the method used by the simulation. Converting the format of the simulation sections and trains into controller sections and train is trivial and does not require saving and loading of files.

The second way is to create a configuration file with the layout and trains, and then specify the file path inside the controller. The file can also be created by the simulation by creating the layout in the track builder and saving the file. By creating a configuration file, the simulation is not required at all, any many different configurations can be created and saved without modifying the source code of the controller.

Chapter 5

Evaluation

This chapter is covered in two parts. The first part covers the process used to test the simulator and the second part covers the experiments carried out to check that the requirements of the projects are met.

5.1 Testing the Simulation

There are three main aspects to test. The first testing the simulation this was achieved with JUnit unit tests and manual testing. The second testing the controllers created behave the way they are supposed to this was done manually with the simulator. Finally testing the simulation vs the hardware system.

5.1.1 Simulation testing Framework

Part of the design of the program was to make sure the GUI is separate from the model and simulation. The reason for this is so you can run the simulation without the GUI. This was done by separating out the updating of the program logic and the refreshing of the graphics. Therefore the update method can be called from the testing framework without needing a GUI following the Model View Controller (MVC) design pattern.

Initially, testing was done manually by creating the tracks in code and verifying correct behaviour on the GUI. This was tedious and slow and did not provide consistency between tests or a record of what was tested. Instead, it was decided to use JUnit, a unit testing framework for Java. This provides a better way of testing. Unit tests are now used to create easily runnable tests repeatedly without the need for a GUI or user input.

An example of a simple test that is run is creating an instance of a train and two simple track pieces connected to each other. The train is then placed on the first track and the update method is called while checking if the current track of the train updates to the next one. Another test is concerned with attaching some rolling stock to a train. At slow speeds when the back of a train collides with a rolling stock, it should connect. This is testable by creating a train placing it on a section in front of a rolling stock and reversing it while checking is the rolling stock becomes connected after a certain number of updates. Other tests include having many trains on the track at once. Checking collisions detection tracks entering junctions in different orientations and directions.

5.1.2 Controller Testing in Simulation

Before the controllers could be tested on the hardware they needed to function correctly inside the simulation.

The first one created was a simple locking controller. The track is divided up into sections and each controller can keep track of where the trains are by receiving a starting location of the trains and updating this location by responding to events sent by the simulated track. A locking controller ensures no two trains are in the same section at once. Before a train moves onto the next section it tries to acquire a lock for that section. If it cannot, it stops, if it can, it enters the sections and releases the lock from the previous section. This was tested by creating two trains one faster than the other. The faster train was placed a few sections behind the slow train. What should be observed is as the fast train approaches the section the slow train is in, it should stop. Once the slow train is out of the section, the fast train should then enter. This was the case. Unlike the simulation, the controller is not continually updated. It only responds to events sent out by the model and bases it decisions on what the event means to the state.

5.2 Controller Evaluation Experiment

Testing controllers on the simulator is relatively straightforward since many different scenarios can be created and automatically checked to meet certain conditions. In contrast, testing on the hardware is time-consuming, tedious, and limited to the available hardware. In order to run a test on the hardware, the trains need to be manually placed in the correct sections, after which the software that interfaces with the hardware need to be started. For every trial, this has to be repeated. The trains also tend to get stuck which means the trains need to be reset. Another problem is tests have to be visual, a person must wait to observe the result. In the simulator, a test can be set up and the simulation run without visual output. The simulation does have known differences to the hardware that cannot be avoided. Therefore, we can never be sure that a controller which worked on the simulator will definitely work on the hardware. But, still, the simulator is useful for catching problems as early as possible.

Therefore the first use case for the simulator is to create a controller on the simulation and test that it functions correctly and then transfer it over to the hardware. To test this functionality, a case study was conducted to check if this can be done in practice. To achieve this, different controllers were developed and tested in the simulation and then moved to the physical system. The results of this case study will be discussed.

5.2.1 Setup

The controller created in the simulation has to be moved to the JMRI client. The client already has interfaces for events and controllers. The interfaces in the simulation had to be modified to match the client in order for a smooth substitution of the controller.

The way the events interface was implemented in the simulator was a bit different to the physical system, therefore it took approximately a day of work to make it work identically to the Model Railway. An adaptor was created between the two. The adaptor received the events from the simulator interface and transforms the data into events that the physical system can understand.

The controller requires the layout of the track and the train starting locations to be able to manage them. When the controller is connected to the simulator it can receive this state when the controller is constructed. However, this is not possible when the controller is separated from the hardware. Therefore a class was created for the controller to load the layout data from a JSON file. This data in the file can be added manually or the layout created inside the track builder and saved. See figure 5.1.

After fixing these two issues, the controller test could begin.

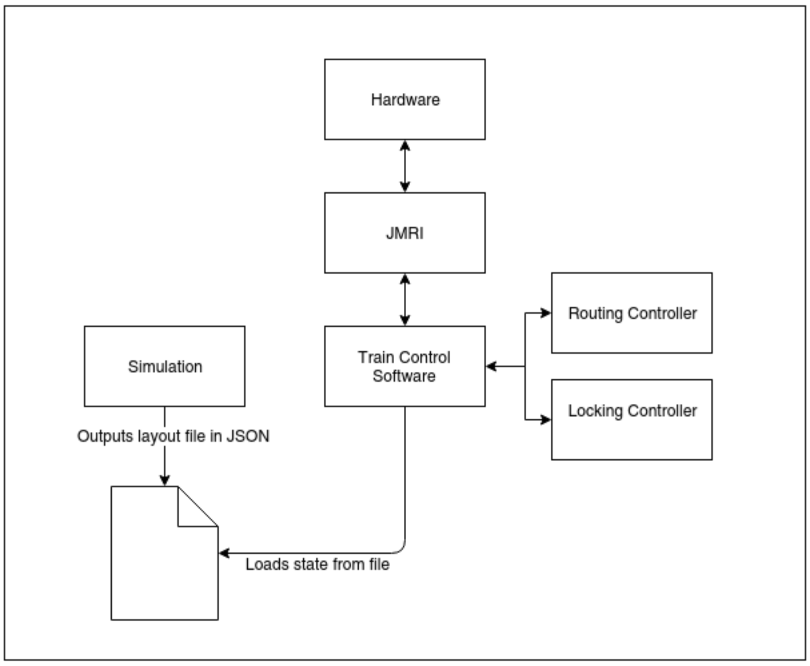

Figure 5.1: The simulation creates a configuration file for the controller to load the railway state.

5.2.2 Methodology

This use case consisted of transferring the locking controller that was created and tested on the simulation onto the physical system and checking for identical behaviour.

The locking controllers stop two trains entering the same section, therefore the test aimed at creating a condition where without a controller the trains do enter the same sections and collide see figure 5.2.

The test setup started with placing two trains on the track, one section in front of the other, followed by sending speed events to the trains. The train at the back needs to move faster in order to try and cause a collision (i.e which the locking controller should prevent) therefore, the speed is set to 100 percent, the one at the front set to 50 percent. See figure 5.3; as it is, without intervention they would collide. The locking controller should lock the section the front train is on and refuse entry to the train at the back. Once the front train leaves the section it will release the lock to the previous section and the controller should send an update speed event to the back train to start it now that the section is clear.

Setting up the test takes some time. First, the trains need to be placed on the track, but the wheels are small and they are difficult to line up. They have to be placed in the correct sections in order for the software to have the same starting state. The software that connects to the hardware then needs to be started. The software that interfaces with the hardware (Model Railway) already existed at the start of the project (recall section 2.3.1). The controller should control the train, therefore, a method has been added to send all the events to the controller. After sending the `control` command the locking controller will start responding to events from the hardware.

5.2.3 Results

A problem was uncovered. An assumption that the IDs of the tracks would be the same as outlined in the paper [1], i.e. that every detection section would be oddly numbered;

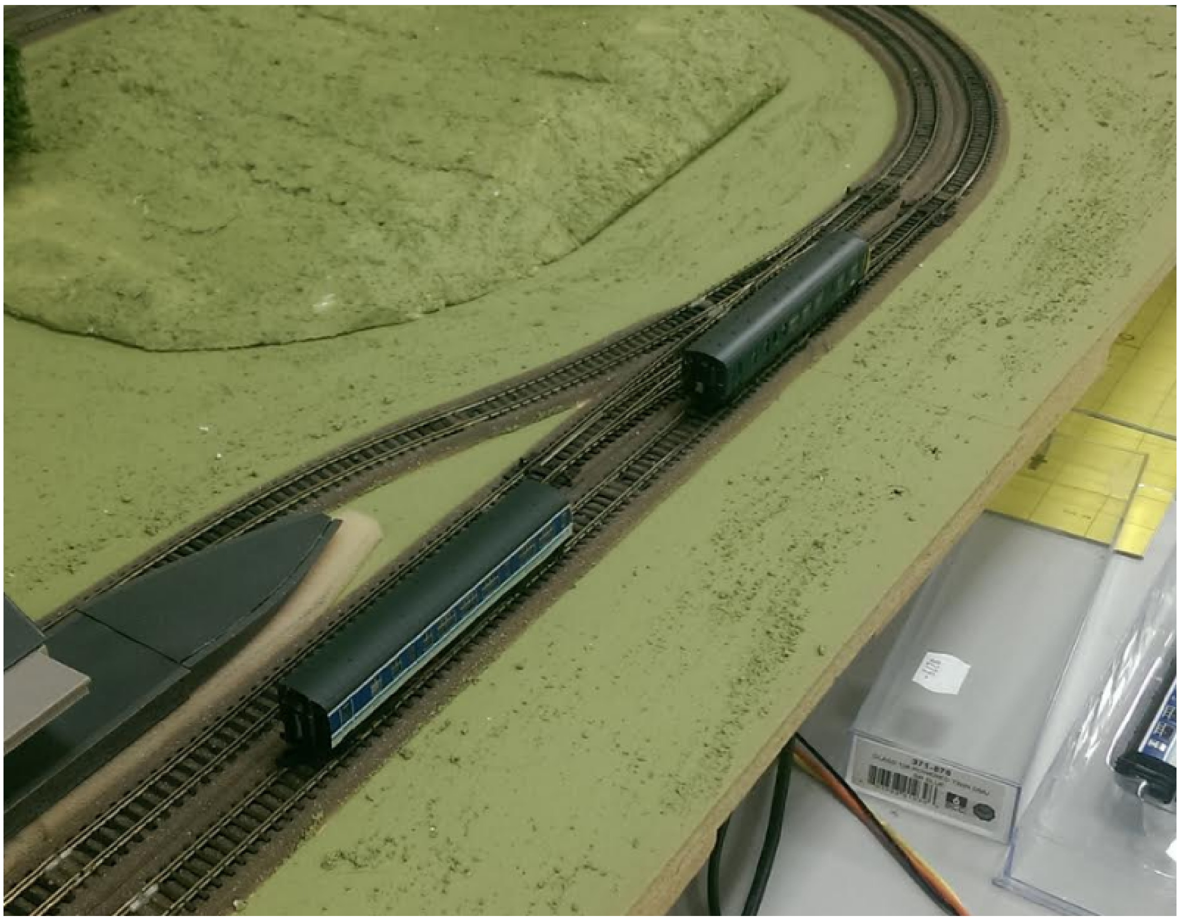

Figure 5.2: The trains on the model track one section behind the other. The locking controller should prevent a collision.

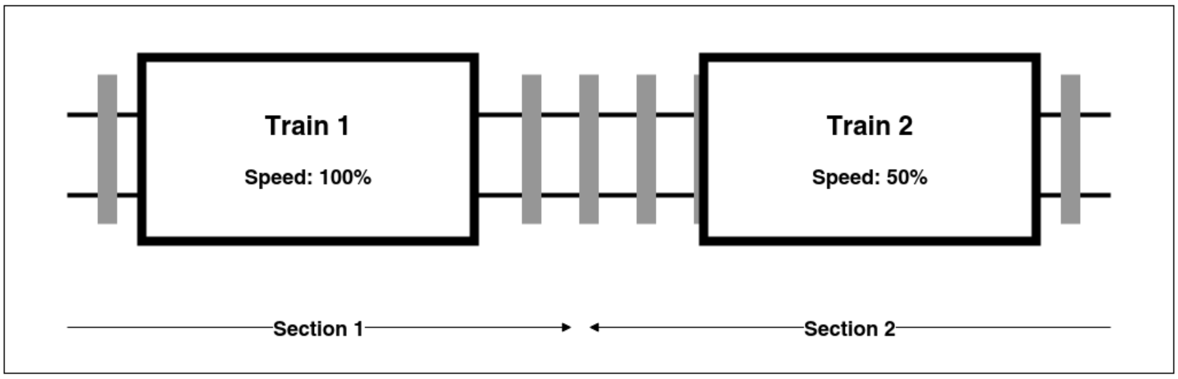

Figure 5.3: Locking Test: the train at the back is set to a faster speed to cause a collision.

whereas in my implementation, the IDs are simply incremented. This meant the controller representation of the track was incorrect causing the controller to become confused and lose track of the trains position. This is easy to adjust for once noticed, and the simulation and controllers now handle the IDs the same way as the hardware.

The controller now functioned as intentioned on the hardware. This was confirmed visually by observing the train stopping as it approached the section the front train was in and resumed when the train holding the lock left the section.

5.2.4 Discussion

Overall, after fixing minor issues the experiment was a success. It was indeed possible to move a controller developed entirely on the simulator to the hardware.

5.3 Simulation Accuracy Experiment

An important consideration for any simulator is to understand how accurate it is compared with the physical system being modelled. Accurately means that when the hardware and the simulation have the same starting state and receive the same inputs, the outcome should be the same. However, there will be differences in real vs simulated results which can be measured. Observing the visualisation of the simulation and the hardware in real time, the trains should be in the roughly the same location on the track and events sent to both should have the similar effects. This means that the speed, acceleration, and stopping time of the trains needs to be considered in the simulation. It also means that scale should be accurate in the visualisation.

The first step to making the simulation match up with the hardware was running benchmarks on the hardware. On the train side, the information that needed to be recorded was acceleration, stopping time, and max speed. For the tracks, only the length was important. The track length was also used to work out the max speed of a train.

The outcome of the experiment should be that the movement of trains and the scale of the track is closer to that of the hardware by inputting the results into the simulation.

5.3.1 Methodology and Results

Average Speed To work out the average speed of the train the train on the track and set to reach full speed using the hardware controller. Once it had done a couple of loops around the track and was at full speed, a timer was started and recorded the time of each lap the train did around the track for 5 laps. Next, the power of the train was set to 50 percent and times recorded for 5 laps. This benchmark was repeated on the controller software on the computer. This was then repeated for the second train.

At full speed there was a 0.2-second variance in lap time at full speed this increased to 1-second see A.1 and A.2.The total length of the tracks came to approximately four metres and this number was then used to work out the max speed of the train: $368cm/14.3s = 25.7cm/s$ (approximately 920m an hour). This value could then be set as a target speed in the simulation to match the hardware.

Stopping Time To work out the stopping time, the train was set to full speed and once it had reached full speed and reached a marked point on the track, the controller was set to start braking. The distance between the marked point where braking started and the end point where the train came to a stop was measured. On average this distance was 45.8cm over 5 trials, for full results see A.3.

Acceleration To calculate acceleration, the train was placed at a fixed point and the train set to full speed, the time was measured when it got to full speed with the formula

$$acceleration = velocity - velocityInitial/time \tag{5.1}$$

The average time over the five trials was 3.2 seconds. Substituting the results into the equation gives $25cms - 0/3.2s = 7.8cms^2$ The full results can be found in the appendix see A.4.

5.3.2 Discussion

After gathering the data required, it was then entered into the simulation. The target speed was set to 25.7ms and the tracks set to the measures length. The experiment was then repeated on the simulation. At first, it was not accurate; there were some things to fix in the simulation. The way the power was updated in the train was not fine enough, so the way that worked had to be changed. That was the main issue. After fixing it, the benchmark for checking the time it takes a train to complete a lap at full speed was repeated on the simulation and was now within a second of the hardware.

The accuracy of the stopping time is not clear when the controller changed from the forward setting to braking setting there seemed to be a delay before the brakes would be applied. Also stopping the train by reducing the speed stop the train almost instantly. However as long as the simulation is accurate to the hardware the requirements are met, also the train and track attributes can easily be changed in the simulation if needed, using the sliders in the user interface.

The physics engine in the simulation models a lot of variables that would make it more accurate when it comes to real trains, the model trains are at a smaller scale so these variables do not have as much effect. However modelling all these attributes means if different weighted trains or different tracks were to be modelled the physics engine would be suitable, this would not be possible if only the acceleration and stopping time was hard-coded into the simulation.

Chapter 6

Future Work and Conclusion

This chapter covers further projects made possible by the completion of this project, recommended extensions to the simulation, and concluding remarks.

6.1 Future Work

The main opportunity the project has opened up is that of creating and testing more complex controllers for routing trains. Before the project was completed, testing controllers was difficult and slow, but now they can easily be added to the simulation and selected from a drop down menu to use. The two controllers developed with this project are fairly simple, but there are endless possibilities for controllers. For example, a controller could take stopping time into account to make sure trains never collide, or could use A* heuristics for route searching, or could even manage congestion by taking trains all over the railway system into account. The length of the rolling stock could be considered along with the weight of the rolling stock, which impacts the stopping time. Currently, the available controllers wait for events to respond to however an interesting project would be to create real-time controllers.

There are a number of features I would add to the simulation. Improving the UI for creating trains and allowing new trains with different parameters would be beneficial. Currently, it is easy to do this in code, but there should be improvements in the user interface to make this process easier.

Another feature would be a better system for exporting or importing controllers. Currently, the controller needs to be developed inside the simulation program. Having a way to specify where the controller is located and having the simulation check if it is valid before using it would be very useful.

Changing the tracks system so that it supports curves of any angle would make more track layouts possible and would make it easier to represent realistic tracks. Furthermore, support for different types of tracks, such as a diamond track, and railway signals would be great.

Random events to derail trains and see how other trains respond or allowing the user to crash trains in incorrectly set routes.

6.2 Conclusion

Simulations are often used to model real world systems to reduce costs and mitigate the safety issues that come with doing tests on the real system. Simulations are also useful when the system has a broad range of variables, meaning that all inputs cannot be tested by brute force. A railway is one of these kinds of complex systems, therefore a simulation was created to simulate a model railway in order to assist with testing software controllers.

The benefit of the simulation comes from improved testing capability compared to the physical system, as there are a number of factors that make testing controllers difficult.

Placing trains on the track takes time as they are difficult to line up. Instructions are sent through the command line and anytime a different configuration needs to be tested, the source code has to be modified and the program restarted - this includes any changes to the controllers. Changing controllers requires the program to be closed down, a few lines of code changed, and restarted. The trains also tend to get stuck on the track, which means they have to be reset, and therefore the software has to be restarted. Doing this once only takes a few minutes but the time spent resetting quickly adds up to significant amounts. This is what occurred when trying to get the locking controller working for the first time, which made it take a significantly longer time than it otherwise would have.

Testing different configuration is even more difficult. The hardware has to be physically changed for different track layouts. This layout has to then be manually hard-coded into the JMRI implementation. There is also the limitation the number of physical tracks and trains and rolling stock that are owned.

The simulation has reduced these problems. In the simulation, placing trains can be done quickly by double-clicking on a track and selecting one of the default trains, or creating a custom train if desired. An even faster way to place a train is by loading a configuration file which does it automatically. Resetting is also very easy; the stop button simply needs to be pressed, and it resets all of the trains and junctions by storing which configuration is being run. Starting the simulation with a controller means clicking the controller button and selecting one. This can be changed later without having to restart the program. New layout configurations can also be created using the track builder, and they can be saved to file. The simulation also has a physics engine, therefore trains with different attributes can be tested without having to purchase the physical trains. The simulation also provides a log of all the events that occur when running, which is useful to see the sequence of events that have happened, or work out where things went wrong.

The simulation is able to emulate important aspects of the hardware; it outputs the same events and sends the same state to the controllers, which allows controllers to be tested on the simulation and transferred over to the hardware system with the same result given the same inputs on the simulation and the hardware. The simulation has a simple physics engine for accuracy and supports trains and rolling stock. The simulation also contains two controllers as examples. Along with a track builder to create and test new configurations, the design is modular to allow future extensions.

Appendix A

Experiment results

Table A.1: Train lap time (s) at full speed

Light Blue Train				
	Controller		PC	
Lap	Full Speed	Half Speed	Full Speed	Half Speed
1	14.5	26.0	12.9	17.6
2	14.4	25.7	12.8	17.8
3	14.2	25.8	12.8	18.0
4	14.2	25.6	12.7	17.8
5	14.0	25.5	12.9	17.9

Table A.2: Dark Blue Train lap time (s) at full speed

Dark Blue Train				
	Controller		PC	
Lap	Full Speed	Half Speed	Full Speed	Half Speed
1	14.0	25.8	13.2	19.1
2	14.2	25.1	13.1	19.2
3	14.1	25.0	13.1	19.2
4	14.0	25.2	13.2	19.0
5	14.2	25.0	13.2	19.1

Table A.3: Distance it takes a train to stop travelling at full speed

Stopping Distance at full speed	
Lap	Distance (cm)
1	44
2	45
3	47
4	46
5	47
Average	45.8

Table A.4: Time it takes a train to accelerate at full speed

Acceleration	
Trial	Time (s)
1	2.6
2	3.2
3	3.2
4	3.2
5	3.2
Average	3.08

Bibliography

[1] D. J. Pearce, *A Model Railway for Investigating Safety Critical Software*. 2016.

[2] Alstom, *Rail interlocking: how does it work?* 2013.

[3] I. Mitsuru, "Railway simulators: tools for novel railway research." `http://www.railjournal.com/index.php/rolling-stock/railway-simulators-tools-for-novel-railway-research.html`, 2016.

[4] O. Rails, "Open rails - discover - open rails." `http://openrails.org/discover/open-rails/`, 2016.

[5] J. M. R. Interface. `http://jmri.org/`, 2016.

[6] Oracle. `http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm#JFXST784`.

[7] R. Allain, "How do you get a train moving?," 2014.

[8] K. Roman, *An Empirical Evaluation of Force-Directed Graph Layout*. PhD thesis, Victoria University of Wellington, 2014.