UNIVERSITY OF
LIVERPOOL

2018/19

Student Name: Jack Gloyens

Student ID: 201164129

Project Title: Ticket to Ride

Supervisor: Sven Schewe

DEPARTMENT OF COMPUTER SCIENCE

The University of Liverpool, Liverpool L69 3BX

Abstract

Ever since the home computing craze of the 1980's, there have been companies developing games and entertainment software for commercial systems. With the advent of technologies like the internet online multiplayer has become a huge part of the games industry. Fighting against players on the other side of the world in arena-style, first-person shooters or working alongside them in more cooperative games has been an attractive concept to gamers for decades, and improvements in network technology has allowed for larger, more complex worlds that can handle millions of players simultaneously.

This project aims to explore the various different network based technologies that games employ to create immersive multiplayer experiences that appear seamless to users, but are often very complex in actuality. The final outcome of the project being an online multiplayer version of the board game *Ticket to Ride* that uses these same techniques.

Massively Multiplayer Online Role Playing Games (MMORPGs) often use a database to store the current state of the game world and the players that inhabit it; players can only access the data that is necessary to them such as their immediate surroundings. Whereas smaller arena-style games often require players to set up their own servers that other players can connect to, here all the game data is stored in system memory rather than a database.

To reduce latency and network traffic whilst maintaining consistency and synchronicity, many games take inspiration from distributed systems design. The current state of the game is stored as a log of update commands that describe how the game has changed, almost like a timeline. The server and clients then send concise update commands rather then sending the entire game state.

This project also explores Artificial Intelligence (AI) in video games over the years and the way games provide realistic and challenging, yet also entertaining opponents for players. Although the final system uses algorithms that have been understood for many years, such as Dijkstra's shortest path, the design and evaluation stages explore more complicated solutions that could've been employed. This includes systems that would give weights to different choices based on their importance and the other players apparent strategies, the choices could then be ordered by their priority.

The final product has been tested in a controlled environment as well as a real world environment on non-development computers. Human participants were asked to use the system and provide feedback which was analysed to better evaluate the final system and even alter or entirely remove features that users didn't like.

Introduction

The aim of the project was to produce an online multiplayer version of the popular board game *Ticket to Ride* where multiple users on different computers would be able to connect to the same game and play together over a network. The server admin would also be able to add AI controlled players to the game which would play automatically when it's their turn. The system should also have a number of failsafes in place in case of server crashes or unexpected disconnects to maintain consistency and some of the other possible features included a graphical user interface, save games and basic security.

The main challenge of this project was creating a system that allowed the server and clients to seamlessly send and receive messages and remain synchronised but without using too much bandwidth. The system also had to be able to recover from unexpected behaviour on the network including dropouts and desyncs without crashing or affecting other users. Creating challenging yet fun AI players was difficult; they had to be able to plan ahead and optimise their play, but also have an element of randomness in order to appear more human, as a perfect AI opponent isn't much fun to play against.

There were a number of possible solutions, such as using a web-based application with a database on the backend, but for this project I decided to use Java to create a socket- based system whereby a server program listens for incoming connections from clients on a network, and to then send and receive messages to and from clients when the game state changes.

This solution works well on local area networks but for wide area networks, it does require the server admin to set up port forwarding on their router they obviously need some technical knowledge for this, but most routers make the process fairly straightforward. The use of Java meant the program could easily be made cross-platform and Java also has a number of built in libraries for graphics and networking which made the development process somewhat easier.

The final product adhered to almost all of the requirements set out in the specification. It plays very much like the original board game (except for a few minor features) and the netcode is stable and reliable. If users suddenly disconnect, then the server carries on with the game and simply skips the players turn until they rejoin. If the server suddenly stops, then the client program will safely close the connection and drop the user back at the home screen. The game features a full 2D graphical user interface for both the client and server programs, however, it is missing animations and the design does leave a little to be desired. The AI players provide an interesting challenge on smaller games, but lack foresight and cunning which means they rarely win on larger games.

Background research

Ticket to Ride

Ticket to Ride is a board game for between 2 and 5 players. Players are given a large map with lots of cities and train routes between those cities which can be claimed. At the start of the game players are given a set of destination objectives which they must fulfill by the end of the game. On a players turn they can draw train cards from the table or deck or use the cards in their hand to buy unclaimed routes on the board. Players earn points by purchasing routes (longer routes are worth more points) and by fulfilling any of the route objectives which they were given at the start of the game. The game ends when a player runs out of trains or there are no more routes available to claim.

The game has gone through numerous iterations since its initial release in 2004. Different versions feature different maps including America, Europe and India as well as adding different mechanics such as stations, tunnels and ferries. However, the basic rules have remained the same.

Networked games

The biggest problem involved in realising a solution was developing stable and reliable netcode that keeps the client and server synchronised. Online multiplayer games have been present since the 1970s in the form of Multi-User Dungeons (MUDs) and from then, networking technology has become significantly faster and more reliable, allowing for complex forms of online play. During the 80s, people would often use MIDI cables to connect computers and play multiplayer games, a famous example is 1987's *MIDI Maze* which allowed up to sixteen users to play simultaneously.

With the introduction of broadband in the 90s, which replaced slow dial-up connections, players could connect from all over the world through the internet. Fast paced first-person shooters like *Doom* and *Quake* were among the first to allow players to connect over wide area networks and play in arena style matches. Users would have to set up and manage their own servers which required relatively high-end hardware and some technical know-how but the entertainment value made it worthwhile. Multiplayer tournaments were even held which kick started the E-sports craze which continues to this day with games like *Dota* and *Counter-Strike*.

1997 also saw the release of *Ultima Online*, the first ever Massively Multiplayer Online Role-Playing Game (MMORPG) where thousands of players could meet up, fight and quest together in a giant simulated world. Players would connect to a central server managed by the developer, Origin Systems, which meant less computer-literate players could easily connect and play the game. This system obviously means that MMORPGs are very expensive to maintain, as the servers have to be kept up and running constantly. However they give the developer much more control over the game, for example if a bug is found after the game is shipped, then the developer can upload a fix to the server which is then

downloaded onto the client's computer the next time they connect. Before *Ultima Online*, this kind of patch wasn't possible. This also meant the developers could keep adding new quests and features to the game as well as hosting seasonal events, which were only available to players for a limited time.

AI game opponents

Since video games started to become popular in the 70s, there has always been some form of artificially intelligent opponent or assistant. For example some early Pong consoles featured modes where the other paddle would automatically move to follow the ball. This is obviously a very simplistic form of intelligence, but it servers the same purpose as modern video game AI, providing a fun and interesting challenge to the player.

Arcade games during this time also featured computer controlled enemies for the player to face, but often times they wouldn't react to the player and would just follow pre-programmed paths, for example *Space Invaders* or *Galaxian*.

The commercial availability of more powerful hardware in the 80s allowed video game AI to become more complex. *Pac-Man* (released by Namco in 1980) featured four ghosts which would chase the player around the screen. All the ghosts responded to the player and based their movements on how the player moved. Initially they would simply surround and trap the player but in testing this wasn't very entertaining, so the designers programmed the ghosts to behave differently to each other. One ghost might head straight for the player while another might head for the point a few blocks in front of the them to try and outwit them whilst another might try to circle behind the player. The programming was actually fairly simple but the ghost's AI appeared complex and interesting because they behaved in different ways and felt reactive.

In the following decades video game AI became more and more complex. *Halo* (released in 2001) introduced the concept of behaviour trees. Non-player characters (NPCs) had a number of possible states they could be in and different ways of transitioning between states, triggered by things that happened in the world and the actions that the player took. Enemy tactics would even change based on which state they were in and this made it appear as if they were strategizing rather than just chasing the player around the game world.

Requirements analysis

Essential functional requirements

The game obviously needs to follow the same basic logic as the original board game, with players taking turns, and on their turn, they can either draw a card or claim a route, earning points for completing these routes. The final product does adhere to this requirement but is missing a few minor features such as giving bonus points at the end of the game to the player with the longest continuous route.

The system also needs to be able to host games for up to 5 players on the server simultaneously, including AI controlled players. It also needs to handle sudden random connection errors without crashing or interrupting the game for players who are still connected. As well as this the server and clients have to remain synchronised at all times and maintain the same game state between updates, however, they should limit their use of the network and send messages only when necessary.

I decided to focus less on the user interface and overall user experience in order to focus my efforts on creating a stable backend for the system. That said it does feature a basic graphical user interface (GUI) that allows users to interact with the system without needing to type commands into a console window.

Desirable functional requirements

The server program should be able to save unfinished games to a file and load them at a later point without any errors. The final program also has an auto-save feature which automatically saves the current state of the game once every 2 minutes.

The server admin should be able to add password protection to their server and users won't be able to connect without the correct password. Admins can also manually kick players from the server. I initially also wanted the server program to have the ability to whitelist and blacklist IP addresses but this feature wasn't implemented due to time constraints.

Some desirable requirements that I laid out in the specification were fulfilled but later removed due to feedback from testers. This included a turn timer that limited the amount of time players could spend on their turns, and this was done so that even if a player was away from their computer it wouldn't hold up the game for everyone else. However, users complained that they felt hurried and didn't have enough time to plan their strategies. Another feature was that if a user disconnected then an AI player would take over until that user reconnected, however, the AI would often play very differently from the player which would be frustrating. In the final version, if a user disconnects then the server simply skips their go until they reconnect.

I had also planned for the game to feature a home screen, game over screen and an options menu, but didn't have time to fully implement these screens. They aren't necessary to the core functionality of the system but would have helped to improve the user interface.

Non-functional requirements

Since the program is written in Java, it can easily be made to run on different operating systems (OSs) but I also wanted the ability for players with different OSs to be able to play together in the same game. Fortunately, this didn't require any extra networking code and so this requirement was easily fulfilled.

Comparison to existing solutions

There already exists an online version of the game developed by Days of Wonder which is available on Windows, Mac, Android and iOS. This follows the exact same layout and logic as the board game but also features extra maps and AI opponents. Due to the similarities it shares with this project, I will be using it as a comparison.

The PC and Mac versions of the official *Ticket to Ride* video game are published through Steam, an online platform for distributing and managing digital copies of games. Steam uses the SteamWorks application programming interface (API) which has a number of features that developers can make use of including in-game achievements, digital rights management (DRM) and online multiplayer by connecting different accounts. Since online connectivity is handled by the Steam servers it's fairly simple for developers to set up and manage multiplayer systems in their games.

Conclusion of research

As discussed earlier there are a number of ways games in the past have handled online connectivity, MMORPGs tend to favour large databases that store information about the world and player characters, a dedicated server that clients connect to interprets this data in order to build the game world for the player. Smaller head-to-head or arena style games tend to favour having clients directly connect to each other or to a small server computer, the smaller amount of game data means it can be stored directly in program memory instead of in a database. Since the game I produced is only meant to handle up to 5 players, I decided to go with the latter option.

Since Valve, the company that manages Steam, obviously have exclusive servers, I was not able to follow the system employed by the official *Ticket to Ride* video game. Therefore, I had to write the netcode myself and decided to use the Java socket programming library which I'd had experience with before.

For the AI opponents I began with a fairly simple system that had little strategy or foresight, I would have expanded to a more complex system if time allowed.

Data required

The project as a whole wasn't very data driven during development but sets of test data were created to test edge cases on individual units of the program, for example, providing a username that contains invalid characters.

After the game had been finished, it was tested in a real world environment on human participants who had never seen the game before. Three full games were played and then users were asked to provide verbal feedback as well as fill in a questionnaire where they rated various aspects of the program out of 10. All the testers signed consent forms before playing the game and their questionnaires were confidential.

However, I found that even with feedback and questionnaires it was difficult to gauge people's reaction to the system and whether they found it intuitive to use or not. I think often people don't know what they want out of a system until they actually see it. I would have liked to have given users different interface options that they could try out and rate but I unfortunately couldn't do this due to time constraints.

Design

The client-server connection

As discussed there were a number of possible systems that could have been used to achieve the online multiplayer aspect of the project including web-based front ends, that access game and player information from a database, or a smaller client based network connection. Databases are more suitable for games with very large player bases and vast quantities of world data that needs to be stored. Since the game I designed is only meant to support up to 5 players, I decided to go with a socket-based solution whereby clients directly connect to a server computer which actively maintains the game state and sends out updates to all the clients when they occur.

With a socket-based system it is more likely that the server and clients will stay synchronised as there is a direct link between them. However, since games aren't hosted on a central server that's in my control, it will require users to set up servers on their own computers. For local area networks this should be straightforward, but on large area networks, users will need to port forward their routers so that incoming data is routed to the correct device. This obviously requires more technical knowledge from the user than if games were hosted on a central server.

For this project, I used Java's built in-socket programming library which I'd had experience with before.

Storing the game state

The server and clients must maintain the same game state and remain synchronised at all times. If a client has an incorrect game state then they may end up sending invalid update commands to the server and receive erroneous data which would disrupt the game for the user.

The naive approach would be for the server to periodically send out the entire game state to all the clients. The problem with this is that most of the game data won't change between updates so you're essentially sending out large chunks of redundant information across the network that the clients already have. This is very wasteful of bandwidth and can even cause latency issues if a user is constantly flooding the network.

Many online games instead keep a log of changes to the game state, so when an update occurs, it gets added to the log and the servers send out a short command to all the clients indicating what pieces of data have changed and how. The clients then use this command to update their own internal game state and effectively emulate how the game will run themselves. When the user inputs data, then the client program sends an update request to the server which fulfills the request and sends an update command to all the other clients. This reduces the amount of redundant data being sent over a network as you're only sending specific data when a variable changes.

This approach did require a more complex implementation to get working but has many advantages in the long run.

In the final version of the game, update commands are stored as string objects that begin with the command name followed by a list of data relevant to the given command.

Maintaining consistency and random number generation

As stated the client computers basically simulate the outcome of the game in between receiving updates from the server. However, the clients also have to know the initial state of the game including player profiles and how the decks are shuffled.

Player information is sent as a concise list to all clients at the start of the game. Each client then initialises their games accordingly.

Rather than sending the entire deck of cards as a long list of data I opted for a system that uses seeded random number generation (RNG) on both the server and client computers. At the start of the game the server generates a random integer and sends it out to all the clients. Whenever the clients need to do something randomly (such as shuffling a deck of cards) they use this number as a seed to generate more random numbers to be used in the shuffling algorithms. Since the RNG algorithms and the seed are the same on all computers they all produce the exact same results without having to directly share data across the network.

This system significantly reduces traffic over the network and in testing hasn't produced any errors or erroneous results, all the clients have come up with the same values.

Graphics and the container system

The game features a 2D graphical user interface (GUI) which obviously means that certain objects will have to be rendered to the screen. I decided to create a basic container class that has a set of fields and methods that made it simple to render, any object that had to be drawn could extend this container class which made would automatically make it renderable.

Each container object also has a list of child objects which inherit values from the parent object. So, for example, if the parent is moved then all of the child objects will automatically move relative to it. This is very useful for interfaces that may contain hundreds of individual elements (such as the map) as they can all be managed from a single parent object. A big use of this in the final product is to switch between two interfaces, which may each contain hundreds of individual renderable objects, by simply making one container invisible and the other visible. This effect will trickle down and switch the child objects either on or off.

AI opponent design

For my initial design of the AI opponents I went with quite a basic strategy whereby the computer used Dijkstra's algorithm to find the shortest path between cities that it had to connect. It then checked if it could purchase any of the routes using the coloured train cards

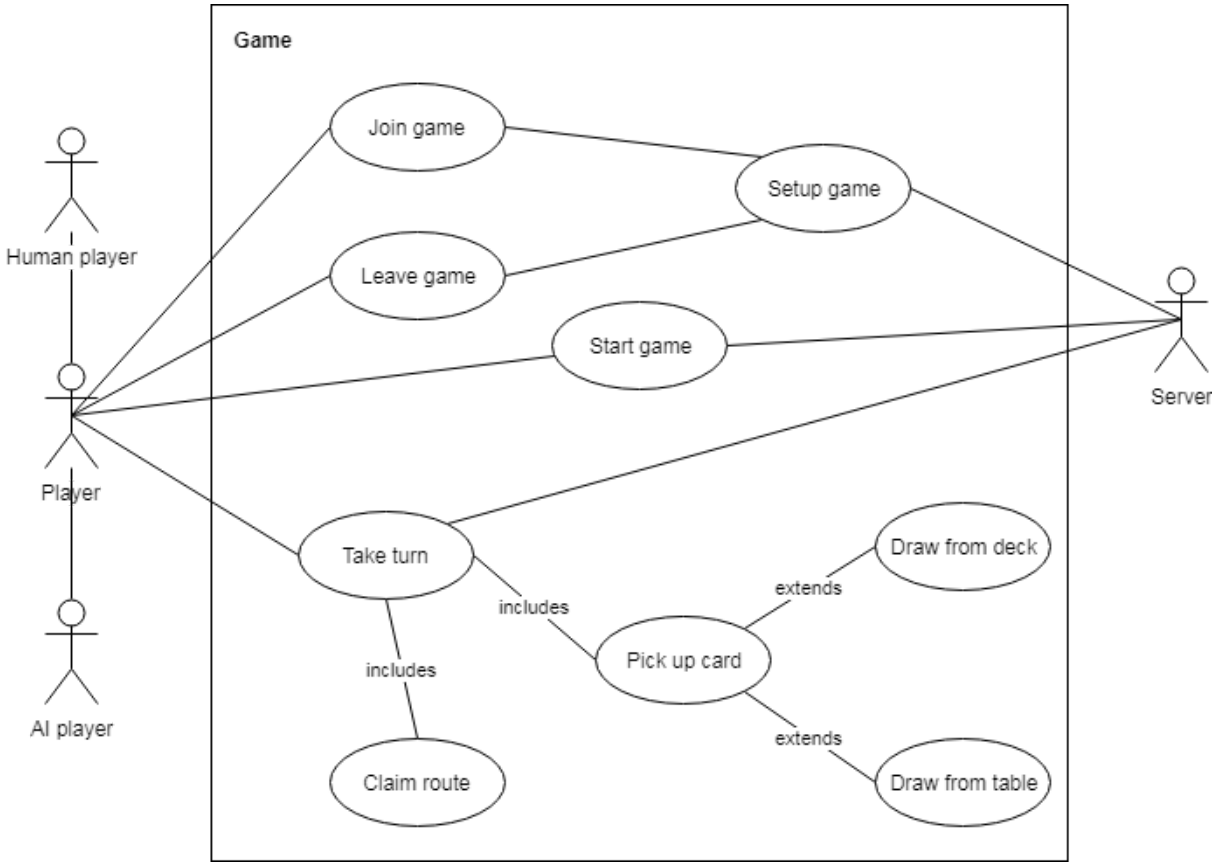
that it had in its hand which it would then buy. It would also keep track of the frequency with which certain colours appear while tracing the routes. If it couldn't buy a route then it would use this frequency distribution to pick up cards from the table that it needed.

I had planned to implement a more complex system if there was time left at the end of the project. This would have included a system that could weigh different routes based on how important they were and how likely they were to be claimed by other players. For example if a particular route had lots of routes next to it that had already been claimed by other players then it's likely that route will be claimed in the next few terms so it should be of high priority. Routes could then be ordered by how important they were to the player and purchased as necessary.

The AI's turns are simulated on the server and client computers individually and no update commands are sent over the network during an AI's turn. Provided the AI algorithms, RNG algorithms and random seed are the same on the server and all the client computers they should simulate the turn in exactly the same way.

Use-case

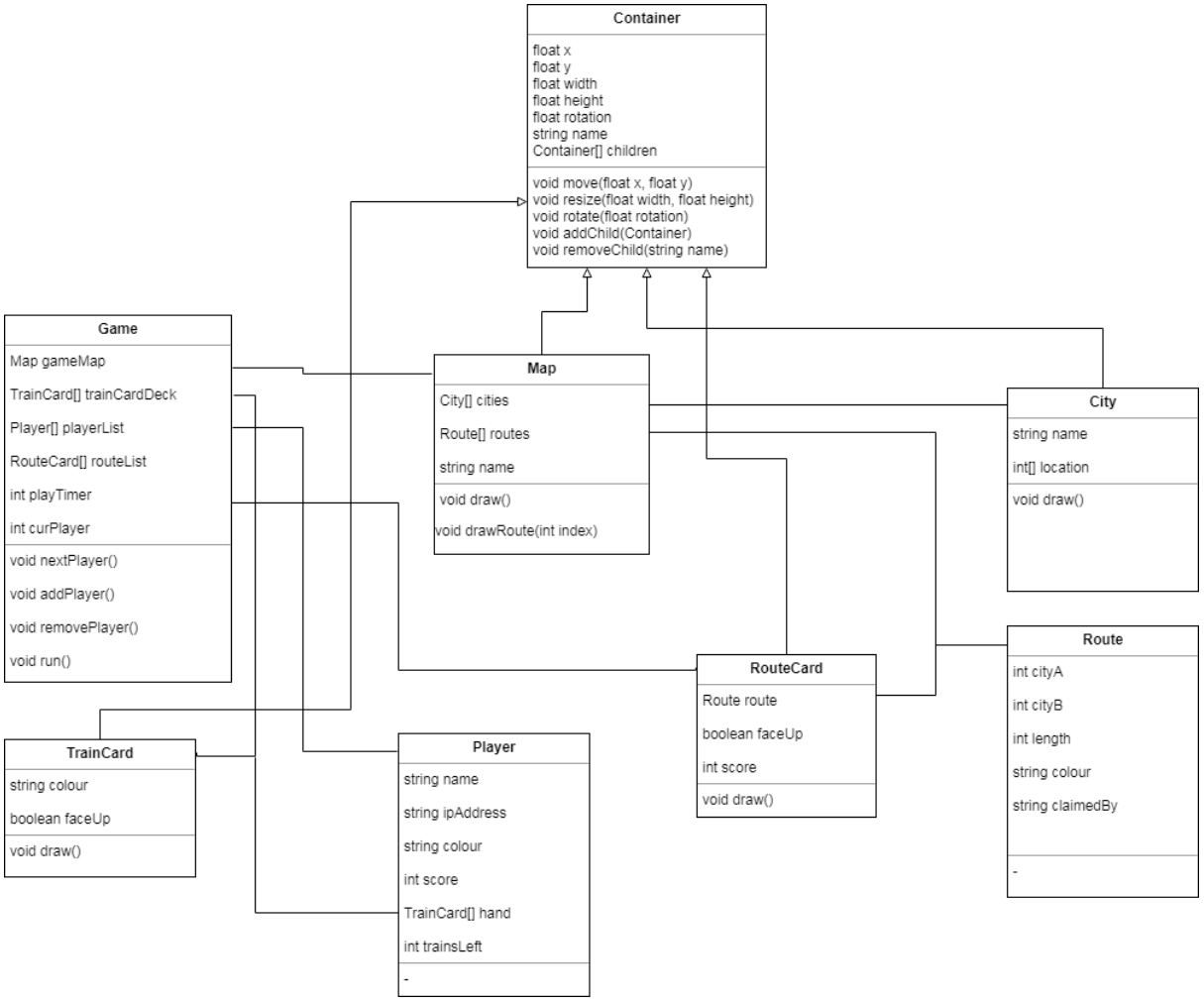
Below is a use-case diagram that shows the communication between the client and the server as well some of the basic functionality of the system



Classes

As stated, all renderable objects extend the container class which drastically simplifies the rendering process. The diagram below shows the basic class system for the main game including the container class.

Class diagram



Interface design

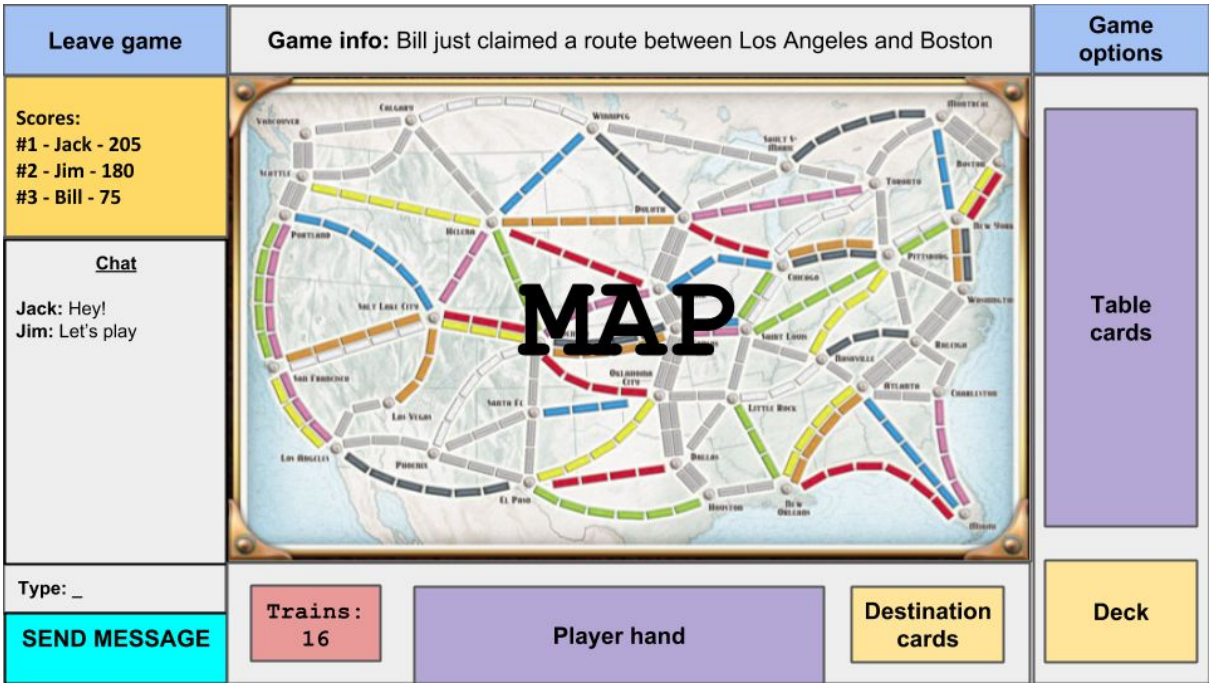
Below is the initial design for the user interface of the main game screen and in the centre is the map which displays all the cities and the routes between them. Originally, I wanted the user to be able to zoom in and out of the map and move it horizontally and vertically but this wasn't implemented in the final product. When it's a players' turn they are able to interact with the map in order to buy routes.

On the left is the list of connected players and their scores. There is also a chat log which shows what players are saying and an option to disconnect from the game. Neither of these last two options are in the final game due to time constraints.

At the bottom is the number of trains the player has left as well as the players hand and their destination cards. If the user mouses over the destination cards then a pop-up appears showing more details about their destination objectives. This popup disappears when they mouse off the destination cards.

On the right there is a panel for game options which would have allowed the user to mute the music and sound effects however in the final game there is no sound so this option isn't necessary. Below that is the set of train cards that are currently on the table, when it's the players go they are able to interact with these cards and pick them up and underneath that is the deck of train cards. Again, these become interactive when it's the players turn.

At the top is the game information section. Whenever an update is received and a player makes a move it is displayed here. This gives the user a better idea of what's going on in the game.



Realisation

Client-server communication

The first aspect of the program I implemented was the connection between the client and server. I've developed a number of small programs that use Java's socket library and much of the code was based on these programs. For this reason I managed to get a working system up and running fairly quickly, the difficulty came in making it crash-proof by adding in various failsafes.

The initial system allowed clients to type commands into a console window, this command would be sent to the server which would check it was valid and then send it out to all the other clients. Commands could also be typed into the server window which again would be verified and sent out to all the clients.

The server maintains socket connections for each client on individual threads. This allows the server to accept incoming client messages simultaneously and reduces latency between the clients and server.

However if a client disconnected from the server without warning or the server suddenly stopped then the whole system would crash with a network error exception. This was fixed by using a try-catch clause that could catch these errors and safely close the socket connection while keeping the system running for everyone who was still connected.

Some simple game logic commands were also implemented to test the system including adding/removing players and changing player names.

Graphics renderer

The network aspect of the program was working but there was no graphical interface to speak of, thus commands had to be manually typed into a console window. So my next task was to implement the renderer and the aforementioned container object system.

I had initially planned to use a Java based implementation of OpenGL but it proved tricky to set up and since the graphics are fairly simple anyway I opted to instead use the Java 2D API which is native to the language and relatively easy to use. I created the container class and created some test objects, every object that extends the container class has a render method that's called recursively at the end of each loop i.e. a parent object is rendered, then all its children, then all the children's children etc.

I also created a function to check if the mouse pointer was inside the bounds of a container for use in the interface manager. This worked well for containers that were aligned to the vertical or horizontal axes, but any objects that were rotated had to have a number of trigonometric functions applied to correctly determine if the pointer was inside or not.

The container class also has an empty update method that is called during the update loop. Any object extending the container class can override this method to add functionality to the object such as reacting if the object is clicked.

Interface and map screen

After the renderer had been fully implemented I started creating some basic interfaces for the server screen and main game screen. I created a set of interface classes for things like buttons and textboxes (which extended the container class) which could be created and added to any screen. These objects automatically manage themselves once they're added to a screen and they include functions to check if they're clicked and get input from the user as well as enable and disable them.

The map is made up of two renderable elements; cities and route segments. Cities are straightforward to render, but routes require a lot more logic and took some time to fully implement.

A route object contains pointers to the two cities it's connecting, the number of individual segments that make up the route, the colour of the route and the player who owns it, if any. Since the distance between the connecting cities and the length of the route (the number of segments multiplied by the segment size) are rarely the same a set of cases for each possible outcome have to be accounted for:

1. The distance between the cities and the length of the route are the same. In this case render each route segment with the same orientation and with no spacing between them. The orientation can be calculated using the cosine rule and the position of each segment can be interpolated fairly simply.
2. The distance between the cities is greater than the length of the route. This is similar to the first case in that all the segments will have the same orientation. However, extra spacing is added in between each segment so that overall the route covers the correct distance.
3. The distance between the cities is less than the length of the route. This is the most complex case, since keeping all the segments the same orientation would require you to have negative spacing between them causing them to overlap. The solution I came up with is to basically have the segments form an arc between the two cities. The position and orientation of each segment is automatically calculated (rather than having to be placed manually) using a set of circle theorems and trigonometric formula.

For the last case, I actually ended up getting help from a third year maths student at the university who provided some trigonometric functions that could be used to correctly draw the arc.

Since for the last case the arc could be on either side of the line between the cities the route class also has a boolean variable to flip which side the arc is on. Some cities also have two routes between them which have to be rendered side by side. For this reason the route class

also has a variable that affects their position perpendicular to the line that connects the cities, so one route could have an offset of 16 pixels and the other -16 pixels meaning they would be rendered either side of the line. The offset variable can also be used to move any route that overlaps another route or city.

In order to declutter the interface I designed it so that information on routes and cities would only be displayed if the user moused over them. There are two objects, the city name object and route information object, that change position according to the cursor and will become visible when the cursor is above a city or route.

Game logic

The game manager and game objects are fairly simple. Whenever a client or the server submit an update command the gamemanager will call the appropriate method which will change the game state accordingly. After the game state is changed various checks are performed including:

- If the deck needs to be reshuffled
- If a player has fulfilled one or more of their route objectives
- If the game has finished

As stated in the design segment these updates and checks are simulated on both the server and client machines rather than having the server simulate the updates and then send out the game state. For this reason there are no update commands to reshuffle the deck or finish the game since the server and client side game managers will automatically update if those checks come up positive.

While it's not a players turn, many of the interface objects are disabled so that a player can't draw cards or purchase routes out of turn.

The game follows essentially the same logic as the original board game except for a few minor features which weren't fully implemented due to time constraints. These include:

- Getting extra points for having the longest continuous route at the end of the game
- Being able to pick which cards are to be discarded from the players hand when a grey route is claimed, instead cards are chosen automatically by the program
- Being able to discard destination objective cards

AI opponents

The initial system that I designed and implemented for the AI was fairly simple and when testing, I found that the AI players weren't particularly challenging and didn't appear human. I slightly altered the path finding algorithm to try and use routes that the AI already had some of the cards for even if it would still have to pick up more cards to actually claim those routes. I also added in random wait times that made the AI appear as though it was 'thinking' and made it so that they would occasionally make random, unpredictable decisions.

These changes went someway to making the AI opponents more challenging, but in the final implementation of the system, they still lack foresight and in games with more players they

are constantly having to recalculate routes and change their strategy which often leads to poor decisions.

Again, due to time constraints, I wasn't able to implement the ideal system I had envisioned for the AI opponents, but the framework is definitely in place to add that functionality in the future. The path finding and frequency distribution algorithms are a good base for more challenging opponents.

Testing

Some basic unit testing was performed as the system was being developed to check that each individual part of the system worked correctly. The final version of the game was then tested in a controlled environment on multiple computers in the computer science labs. Once I was satisfied the system was working correctly, I tested it in a real-world environment using human players who hadn't seen the game before. They used their own non-development computers for the test, a mix of PCs and Macs.

The individual units that were tested were the:

- Network code and game manager
- Graphics renderer
- Interface manager
- AI opponents

Below are a series of tests performed on each unit with the expected outcome and any problems associated with those tests.

Netcode and game manager

Test description	**Expected result**	**Implementation problems**
Can a client can connect to a server and send a message?	The server correctly receives and displays the message.	
Can the server send a message to any connected clients?	The clients correctly receive and display the message.	There was a small bug here that was easily fixed, basically the server would send updates to everyone including the client who sent the update so the sender would effectively simulate the update twice. In the final system the sender waits for the server to validate the command before processing it.
Can the game manager correctly processes valid commands?	Invalid commands are rejected and valid ones update the game state.	

Do the server and clients shuffle the decks the same way?	The server sends out the RNG seed at the start of the game and the server and clients produce the same results.	
Do the clients and server maintain the same game state between updates?	The clients and server are all synchronised and will update the game state in exactly the same way.	
Do the clients correctly close the socket if they lose connection to the server?	The client programs close the socket safely and return the user to the join game screen.	
Does the server correctly close sockets when a user disconnects?	The server safely closes the socket and carries on managing the game for all the players still connected.	
When a user draws a table card, is the new table card correctly drawn for all clients?	The same card is drawn in the same place on the table for all players.	
When a user claims a route, is the route correctly claimed for all clients?	The route that was claimed is displayed in the colour of the user who claimed it for all the clients.	
Can a user only interact with the system when it's their turn?	When it's not a user's turn they can't claim routes or draw cards but can still mouse over cities and routes.	
When a player runs out of trains or there are no routes left, does the game end correctly?	The interface is locked out for all clients and a message is displayed saying who's won the game.	
When a player isn't connected, does the system skip their go?	If a player isn't connected, then the system simply moves onto the next player but will start letting them play again once they reconnect.	
When a client reconnects do they re-sync with the	The server sends a log of update commands to the	One problem with this was that when the AI opponents were

server and other clients?	client who simulates the game up to that point, the client should end up with the same game state.	added they had arbitrary wait times to make them appear more human. When users were simulating a game after reconnecting to it these wait times were also simulated which slowed down their initial connection speed. To fix this I added a flag that said whether the game was being simulated in real time or not, in which case it ignored any sleep commands.

Graphics renderer

Test description	**Expected result**	**Implementation problems**
Are textures displayed correctly?	Image files are loaded and displayed at the correct coordinates and with the correct size and rotation.	
Does changing a container's position affect its children?	When a container is moved the transformation is recursively carried down to all children of the container.	
Does changing a container's size scale the it's children?	When a container is scaled up or down this effect is recursively carried down to all children of the container.	
Does changing a container's rotation also rotate and transform it's children?	When a container is rotated this effect is recursively carried down to all children of the container.	Child containers aren't actually transformed correctly around the central point of their parent but will alter their rotation if the parent rotation changes. This feature would have been nice to get working properly but it's actually never used for any of the interfaces so it wasn't a priority to fix.
Does making a container visible/invisible change the visibility of its	When a container is made visible or invisible this effect is recursively carried	

children?	down to all children of the container.	
Does the system handle missing textures correctly?	If a texture can't be loaded the system should carry on without the texture but report an error in the console.	
Is text displayed correctly?	Text is displayed with the given font and font size and at the correct position.	
Are routes displayed correctly?	As discussed in the realisation section there are three possible outcomes for how the routes should be displayed, the most complex being where the segments form an arc between the cities. In each case the routes should be rendered correctly and connect the two cities.	The algorithm to calculate the positions and rotations of segments that form arcs proved to be very complicated. I actually ended up getting help from a third year maths student who provided some trigonometric functions that could be used.

Interface manager

Test description	**Expected result**	**Implementation problems**
Does the system detect if the mouse cursors is over a container that is axially aligned?	The mouse over function of the container will return true if the mouse is currently inside the bounds of the container and false if not.	
Does the system detect if the mouse cursor is over a container that has been rotated?	Same as above, the mouse over function will return true if the mouse is inside the bounds of the container or false if not.	This is significantly harder than the above case as the cursor and container have to first be rotated in the opposite direction so that they are axially aligned, for this reason it took longer to get working.
Does the system ignore mouse detection if the cursor is outside the window boundary?	If a user clicks outside the window or when the window isn't focused then the clicks are simply ignored by the	

	system.	
If a container object has an update function, is it processed correctly when the container object is clicked on?	When the cursor is inside the bounds of the container and the user clicks the update function should be called.	
When a textbox is clicked on does it gain focus?	A cursor appears inside the textbox when the user clicks on it.	
When a textbox is clicked off does it lose focus?	The cursor disappears from any textbox that was focused when a user clicks outside of it.	
When the user types does the focused textbox update with the keyboard?	If a textbox is currently focused then it's contents will update according to keyboard presses.	
If the user presses backspace does the textbox input get deleted correctly?	If a textbox is focused and the user clicks backspace then the last character of the input will be deleted and the cursor will move backwards. If there is no input then pressing backspace does nothing.	
Can the data from a textbox be read correctly?	If the program calls the textboxes get input function then it should receive the current contents of the textbox as a string.	

AI opponents

Test description	**Expected result**	**Implementation problems**
Does the AI's pathfinding work correctly?	The algorithm should find the shortest valid path between the pairs of cities on its destination objectives.	
Does the system draw cards optimally?	The algorithm should take cards from the table of the colours it needs most to	

	fulfill its destination objectives.	
Is the AI simulated the same way on every device?	The AI should draw cards and claim routes in exactly the same way for every client and the server. They should also wait the same amount of time.	The only problem I had with this was that when a user reconnects to a game the AI's turn wouldn't always be simulated in the way it should be. This was due to the fact that it would skip the sleep commands (to reduce connection time) but also skip the RNG commands which meant future random numbers were incorrect. To fix this, the final system still generates random numbers where necessary but doesn't always do anything with them.

For user testing I converted the programs to .jar files so that they didn't have to be run from a console window. However, I realised that this caused the client programs to constantly send blank messages to the server. This bug took some time to fix but I eventually realised that it was because I had included a feature where clients could type update commands into the console window rather than using the GUI, the .jar files ran without a console window so the program was trying to get input from an invalid source and then send the missing input to the server. To fix this, I simply removed the feature in the final version.

Evaluation

Method

After the project was finished, I got a few real life participants to play some games together . They signed consent forms and after the test was finished they filled out a questionnaire and also gave some verbal feedback on the system's usability. The answers to the questionnaires were given anonymously and any save data of the games played was deleted from the system after testing and users were allowed to completely uninstall the game from their computers.

The questionnaire I created was based on a set of questions from the system usability scale. This can be used to get a numerical rating of the usability of different aspects of the system. The questions are as follows:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The final system also has to be tested against the list of essential functional requirements, set out in the specification stage, to make sure it meets the minimum requirements for the system and includes all the core functionality. It also needs to be checked against the list of desirable requirements although these obviously aren't necessary to the basic functioning of the system.

User feedback

Three games were run in a real-world environment with four human players, and one AI player, each using their own computers to play. I had initially planned to only play two games but the testers found the experience very enjoyable and actually asked to play an extra game.

Most users found the system difficult to set up and get to grips with, they didn't understand the requirement for the Java runtime environment (JRE) to run the game correctly and needed assistance to get the system up and running. However,,, once they were away they found the system fairly intuitive and enjoyed playing the game.

Some users commented that the interface was lacking in some areas. For example, it wasn't completely obvious when it was someone's turn as this was only indicated by an asterisk next to their name. There also isn't a proper victory / game over screen, the system just pops up with a short message saying who won and locks out the interface to stop the game. They also commented on the lack of animations which made some actions unclear, such as drawing cards.

Users who'd played the board game also commented on the lack of some minor features present in the original, however these features aren't integral to the game's core functionality.

Unfortunately the numerical feedback I gathered from the questionnaires wasn't as useful as I had initially envisioned as players tended to rate either 1s or 10s rather than giving completely honest evaluations of the system. I think if I did a project like this again, I would probably try and test it on people I didn't know as they would likely be more inclined to give an accurate assessment of the system. For the sake of thoroughness, I have however listed the average user rating for each question below:

Question	**Average rating (out of 10)**
I think that I would like to use this software frequently	10
I found the software unnecessarily complex	1
I thought the software was easy to use	10
I think that I would need the support of a technical person to be able to use this software	2.7
I found the various functions in the software were well integrated	10
I thought there was too much inconsistency in the software	1.3
I would imagine that most people would learn to use this system very quickly	10
I found the software very cumbersome to use	1
I felt very confident using the software	10
I needed to learn a lot of things before I could get going with the software	1

Strengths

The final system adheres to all the essential requirements laid out in the initial specification document, except for the fact that a few of the minor mechanics that are in the original board game aren't present. However, the system follows all the core logic and rules of the board game.

A server can be started on any computer with the JRE installed and can host a game for up to five players, including AI players. Users with the client program can connect to the server by providing the appropriate IP address and the host can add a password to the server, which users must provide to be able to connect. Providing an invalid IP address or password will result in the player not connecting correctly and an error message being displayed, but the user has the chance to try and connect again. Admins can also remove players from the game.

Overall, the netcode is fairly stable and can easily recover from network dropouts or desyncs. When a user disconnects in the middle of a game, the server carries on and allows the other players to have their turns and simply skips that users turn until they reconnect. If the server unexpectedly shuts down, then the client programs will safely close the connection and drop the user back to the join game screen with an error message. In real----world tests, I only encountered one error where a player randomly stopped receiving messages from the server, however this didn't affect the other players and all the user had to do was reconnect and the problem was fixed.

The game features a graphical user interface which, although lacking in some aspects, servers it's purpose and most users found it intuitive and easy to use.

Since it's written in Java, the final product is cross-platform and users playing on different operating systems can connect to the same server and play together.

Finally the game features a save system which basically keeps a log of the initial game state and a list of update commands since the start of the game up until the point it was saved. There is also an autosave feature that will save the current state of the game every two minutes.

Weaknesses

Most of the desirable requirements were met, however, a few unfortunately weren't fully implemented due to time constraints. Towards the end of the project I decided it would be better to focus on improving and tightening up the features that were already present in order to make them as good as they could be rather than cluttering the system with half implemented features.

There is no in-game chat system for the players to communicate with each other. However, the game isn't co-operative so there's little need for players to communicate outside of

wanting to taunt or congratulate each other. The framework is in place to allow for this feature so it could be added fairly easily in the future.

Although the system has a graphical user interface, it doesn't have any animations which makes some actions unclear, such as drawing cards. The final system also doesn't have a home screen or options menu and there is no way for players to quit out to the join game screen, short of closing and re-opening the program.

There is no music or sound effects in the final game. The code doesn't even have a sound manager so this feature would require some effort to fully implement as there is no framework in the current code base to support it.

Although the netcode is stable for the most part, it's not perfect, and I did have problems during testing. The system doesn't check that messages are being received and interpreted correctly and even though most of the time this isn't a problem it can occasionally cause errors. One way to solve this would be to add checksums to any messages being sent, which would then be checked by the receiver to ensure the message is correct and hasn't been corrupted in transit. When sending an update or request, the senders don't check that the message was actually received, they just continue under the assumption it was. Again most of the time this isn't an issue but can on occasion cause devices to desync, this can be fixed by reconnecting to the game which will simulate all the updates up to that point.

There is also a problem with multiple users disconnecting and reconnecting to the game. Since the server doesn't keep track of the IP addresses of users who've disconnected from the game, it's possible for two or more users to quit the game and then reconnect as a different player. I was unable to find a good way to solve this issue since it's possible for the IP address of the client computer to change before they reconnect to the server. Open TTD (a multiplayer tycoon game) manages this problem by having user profiles that are password protected. When a user reconnects to the game they select the profile they were playing as before and to prove it's actually them they have to provide the password associated with that profile. This system works well, but obviously requires extra interface design and back end management so I didn't end up implementing it.

In my interim report, I mentioned that I wanted to write a JSON (JavaScript Object Notation) or XML (eXtended Markup Language) parser that could be used to load different maps stored in either of those file formats, but since this wasn't in my initial design, I decided not to focus on this feature. After researching it, I realised that the effort required to implement this feature wasn't really worth the benefit gained from it. In the final version of the game there is only one map, the United States, that is generated in code through commands that add cities and routes to a map object.

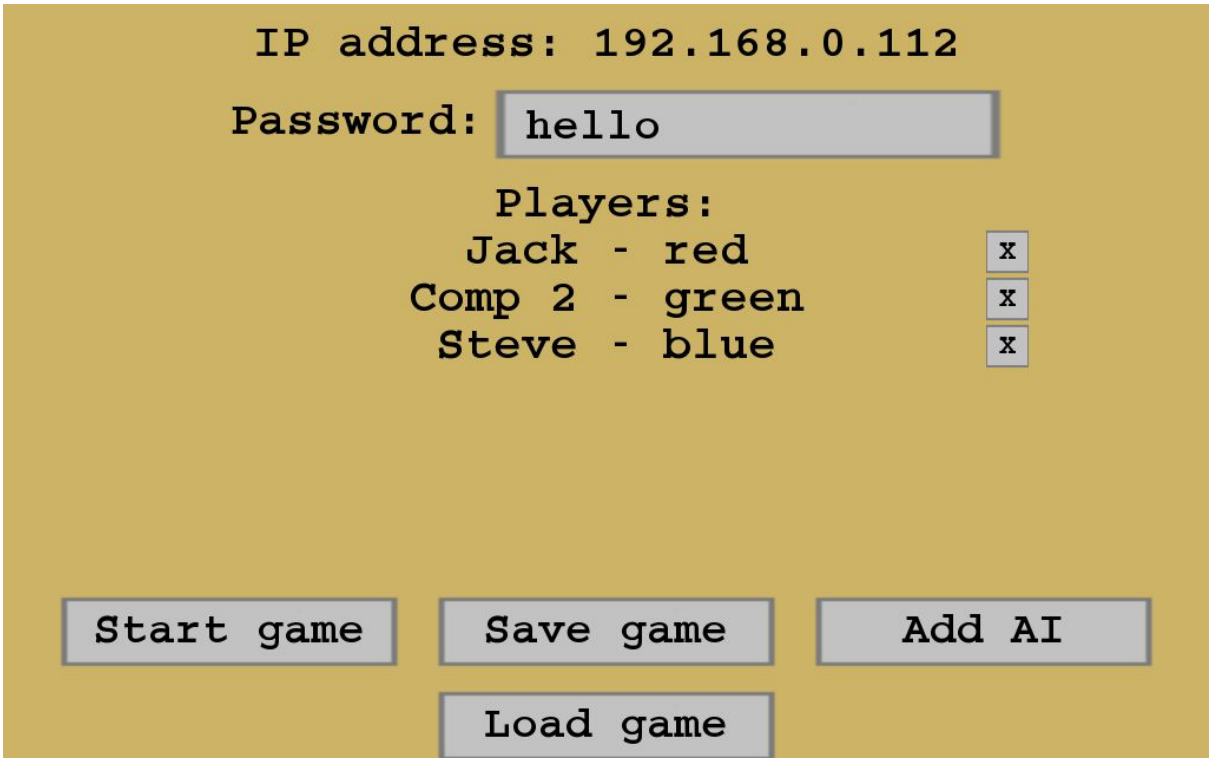
Finally some users commented on the system being difficult to set up, in part due to the requirement for having the JRE also installed to run the game correctly. If this was a professional product that I could devote more time to then I'd probably program it in something like C++ instead as it can then be released as a standalone piece of software that doesn't require any third-party software to run.

Changes to initial design

Some features that I had outlined in the initial design and specification were fully implemented, but after getting negative feedback from users, I decided to remove them from the final game. The first was a turn timer, this was added to keep the game flowing and so that any players who were away from their computer while it was their turn didn't hold up the game for everyone else. However many users said they didn't have enough time to plan out their strategies and felt rushed. The second feature was that when a user disconnected from a game that was in progress an AI player would take over for them until they reconnected. However, players said that the AI would often play cards they wanted to keep or claim routes they didn't actually want, so in the final version of the game any players who aren't connected simply miss their go until they reconnect to the server.

Screenshots

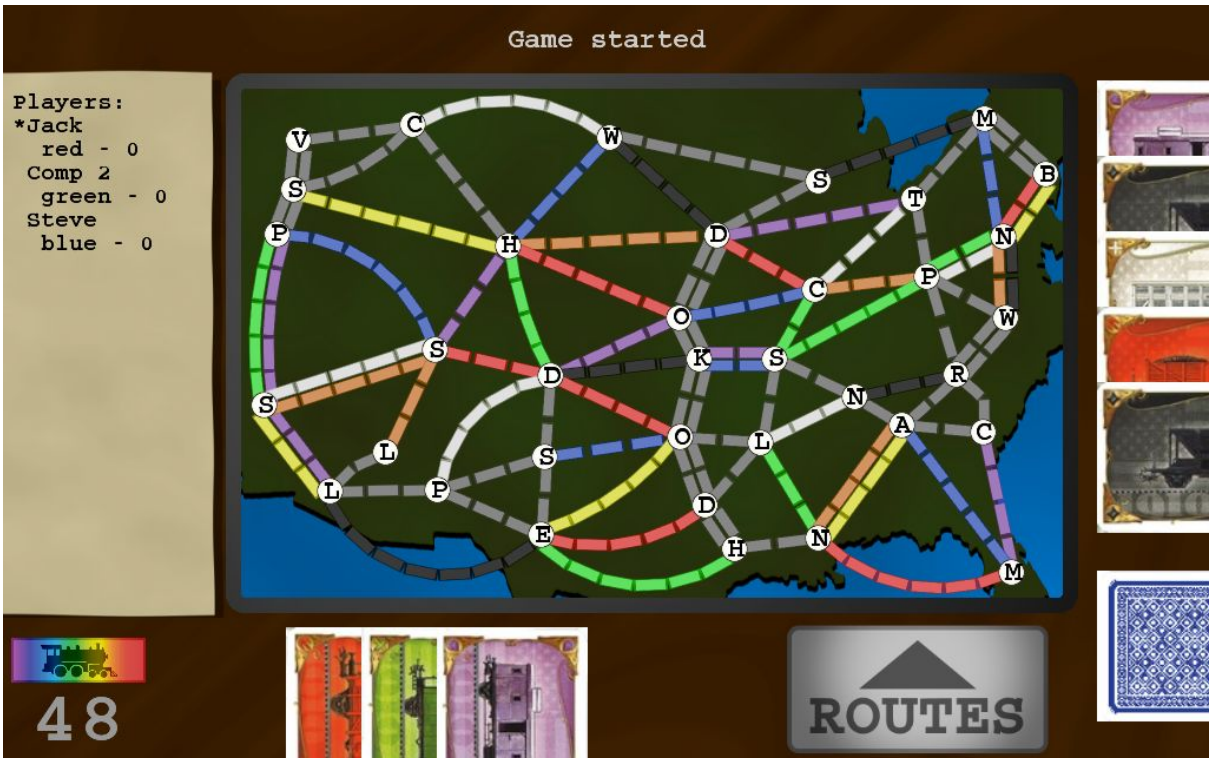
Included here are a series of screenshots that display the various interfaces of the system as well as how those interfaces operate.



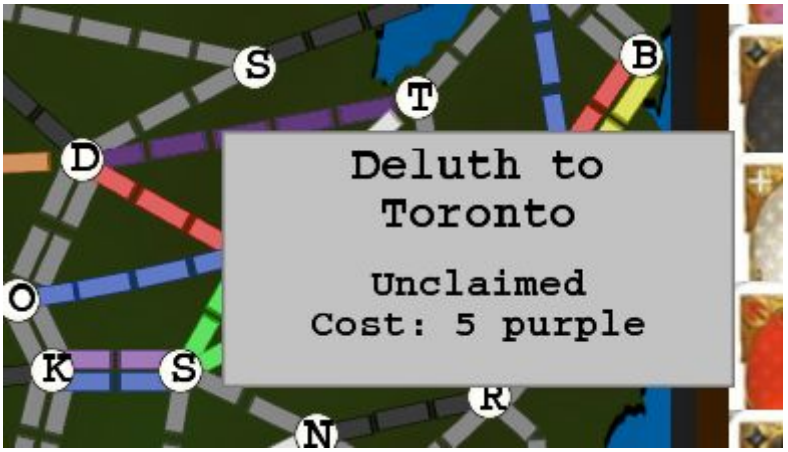
The server screen with the IP address of the server at the top, a text box for the password and a list of players. Admins can kick players by clicking on the Xs at the side as well as save and load games and add AI players.

The join server screen, users can type in the IP address of the server as well as the server password.

The pre-game screen where users can enter a username to be identified by before the game starts. They can also see a list of other players in the game.



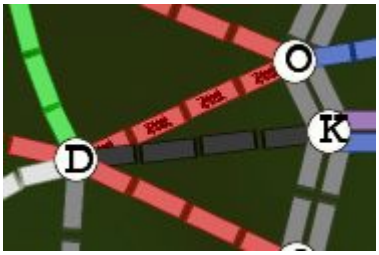
The main game screen when a game starts for the first time.



Users can mouse over routes to get information on them.

Users can also mouse over cities to see their name.

They can also mouse over the routes button to view their route objectives.



When a user claims a route it turns their colour and other users can’t claim it.

Learning points

At the start of this project, I already had a limited understanding of many of the aspects that went into the design of the system. So a lot of what I've learned isn't completely new but is simply expanding on previous knowledge.

I'd had some experience programming basic networked systems either through PHP and SQL or more involved systems that used sockets. However, this project required a stable and reliable backend that could recover from network errors and keep all the clients and server synchronised whilst reducing unnecessary traffic and latency. I took inspiration from the distributed system design of large databases, such as those used in MMORPGs, where the game state is stored as a log of update commands that are then processed client side. Through this, I've learnt a lot about distributed systems and the methods they employ to remain synchronised.

I also researched the various ways that different games handle network dropouts in order to implement a series of fail safes that would allow the system to recover and carry on operating normally.

Although I didn't implement the feature in the final product, I also learnt about the use of checksums to ensure data is consistent and doesn't get corrupted whilst being sent over the network.

Before the project, I had a fair amount of experience using different rendering engines, especially OpenGL, however I learnt a lot about how Java's 2D library handles graphics. The container system I developed was inspired by PixiJS, a Javascript based graphics library which I've used before, but I had to do some extra research to understand exactly how PixiJS manages graphics objects.

The artificially intelligent opponents use techniques and algorithms that I was already mostly familiar with. Dijkstra's algorithm and optimisation through frequency analysis are both fairly straightforward systems and the concepts behind them certainly weren't new to me. However I did do some research into how other games create interesting and challenging opponents that behave somewhat realistically using techniques such as behaviour trees and linear regression.

Overall, although many of the different systems that I researched (related to networking, graphics rendering and AI) weren't fully realised in the final product, they proved very helpful in developing a system that was stable, reliable and intuitive. The research process also gave me a far greater understanding of how these systems actually operate as opposed to just having a surface level knowledge on how they are implemented.

Professional issues

Throughout this process I have tried my best to follow the British Computing Society code of conduct and code of good practice to ensure smooth development.

When I was specifying the requirements and designing the system, I was sure to be realistic about what I could achieve and what was outside the scope of the project. I based this on the experience I'd had programming similar systems as well as research into how existing solutions were designed.

When creating the netcode, I was conscious of the various pitfalls involved in designing and maintaining a distributed system and created a series of safety nets to ensure the final system was stable and reliable.

The system is programmed in a well-established language and using libraries and algorithms that are very well documented. The code is also clearly commented in a standardised manner and Java documents were generated from these comments including detailed descriptions of the various classes and methods. These JavaDocs could be used to easily maintain the system in the future.

When testing the system and gathering user feedback I made sure that participants all signed consent forms and agreed to take part in the test. Their answers to the questionnaires were all anonymous and any data they entered into the system was deleted after testing was complete. Their feedback was then used to improve the system and tailor it to the needs of the target audience.

Since the system is largely based on an existing product, I'm conscious of the fact that it cannot be released or sold to the general public without permission from the current licence holder, so as not to infringe on their intellectual property.

Bibliography

1. Gabriel Gambetta: Client-Server game architecture, http://www.gabrielgambetta.com/client-server-game-architecture.html, last modified 2017
2. David A. Watt and Deryck F. Brown: Java Collections - An introduction to abstract data types, data structures and algorithms. Published by John Wiley & Sons Ltd. Chichester, 2001
3. Tutorials Point: Java networking, https://www.tutorialspoint.com/java/java_networking.htm, last modified 2018
4. StackExchange: Javascript and PHP for real time multiplayer, https://gamedev.stackexchange.com/questions/33337/javascript-and-php-for-real-time-multiplayer, last modified July 2012
5. JogAmp: JOGL - Java binding for OpenGL, http://jogamp.org/jogl/www/, last modified October 2015
6. Binary Tides: PHP socket programming, https://www.binarytides.com/php-socket-programming-tutorial/, last modified July 2012
7. Glenn Fiedler: State Synchronization, https://gafferongames.com/post/state_synchronization/, last modified January 2015
8. Usability: System Usability Scale (SUS), https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html, last modified 2018
9. Days of Wonder: Ticket to Ride online, https://www.daysofwonder.com/online/en/t2r/, last modified 2018
10. Days of Wonder: Ticket to Ride rules (US), https://ncdn0.daysofwonder.com/tickettoride/en/img/tt_rules_2015_en.pdf, last modified 2015
11. The British Computing Society Code of Good Practice, https://www.bcs.org/upload/pdf/cop.pdf, last modified 2011
12. The British Computing Society Code of Conduct, https://www.bcs.org/upload/pdf/conduct.pdf, last modified 2015
13. David Kushner: Masters of Doom. Published by Random House, Inc. 2003
14. Keith Burgun: Game Design Theory - A New Philosophy for Understanding Games. Published by CRC press, 2013

Appendices

The full game, with both the client and server programs and all the assets, is available to download at:
http://gloyens.com/Ticket%20to%20Ride/Game/Ticket_to_ride.zip

The Java source code and manifest files are available to download at:
http://gloyens.com/Ticket%20to%20Ride/Source/Source.zip

The automatically generated JavaDocs are available to view at:
http://gloyens.com/Ticket%20to%20Ride/JavaDocs/

These files are also included in the .zip file as part of the final submission.

3rd year project testing consent form

This research session will involve testing a piece of software based on the board game ‘Ticket to Ride’ and providing feedback on your experience.

You will be required to install the software on your own computer as well as the Java Runtime Environment (JRE) in order to participate.

You will be playing a game as a group and then once the game is finished you will be required to give feedback on your experience.

No personal data will be stored and you are free to remove the program once testing has been completed.

By signing this document you confirm that:

- You have read and understood the information on the project
- You have had the opportunity to ask questions about the project and received answers to your satisfaction
- You understand your participation is voluntary and you are free to withdraw at anytime without providing a reason
- You understand you can withdraw your data from the study at anytime
- You understand that any information you provide may be used in the final project presentation
- You consent to photos / videos being taken as part of the project

Name of participant:

_____

Date:

_____

Signature:

_____