Extension Report

by

Group GVA

Introduction

This report is intended to explain and justify the changes that team GVA have made to the game Locomotion Commotion which was inherited from different development team. Changes have been implemented to try and bring the final game closer in line with the software requirements. Challenges that were encountered when expanding the game will also be highlighted.

Existing Bugs

The software we inherited came with a number of bugs. Below is a table showing the bugs that we managed to identify and the details of how they were removed, or the justification of our reasons for not removing them, if that was the case.

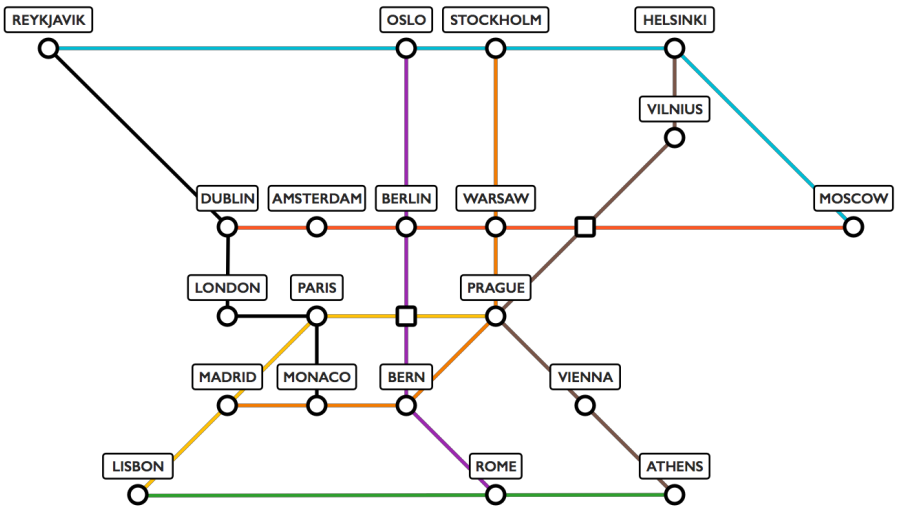
Bug Description	Removed (Yes/No)	Details of fix / Justification for leaving
Labels for cities all display that a player will receive 100 units of fuel per turn. This is a bug as the cities give amounts of fuel different to this value.	Yes	Fixed by calling the station.getTotalResourceOut() method instead of just displaying a constant (100).
Game resolution stuck at 1680x1050 and cannot be resized without rendering issues.	No	An attempt to fix this bug was made before realising that the task would be too time consuming. All of the art assets would have had to be redrawn and there were complications with how things like the top menu were drawn on the scene. The remaining issue is not a game breaking one, it just means that there has to be restrictions put on the hardware requirements for the game.
When starting a game both players select cities which they then own. If you then	No	This was not seen as a game breaking bug as you can just as easily close the

exit to the main menu and start a new game, city ownership information from the previous game is carried over meaning players cannot choose a new starting city.		game window and relaunch the game to create a new game. We did not fix this bug as its severity was not seen as costly enough to warrant spending the time to fix it over implementing extra functionality

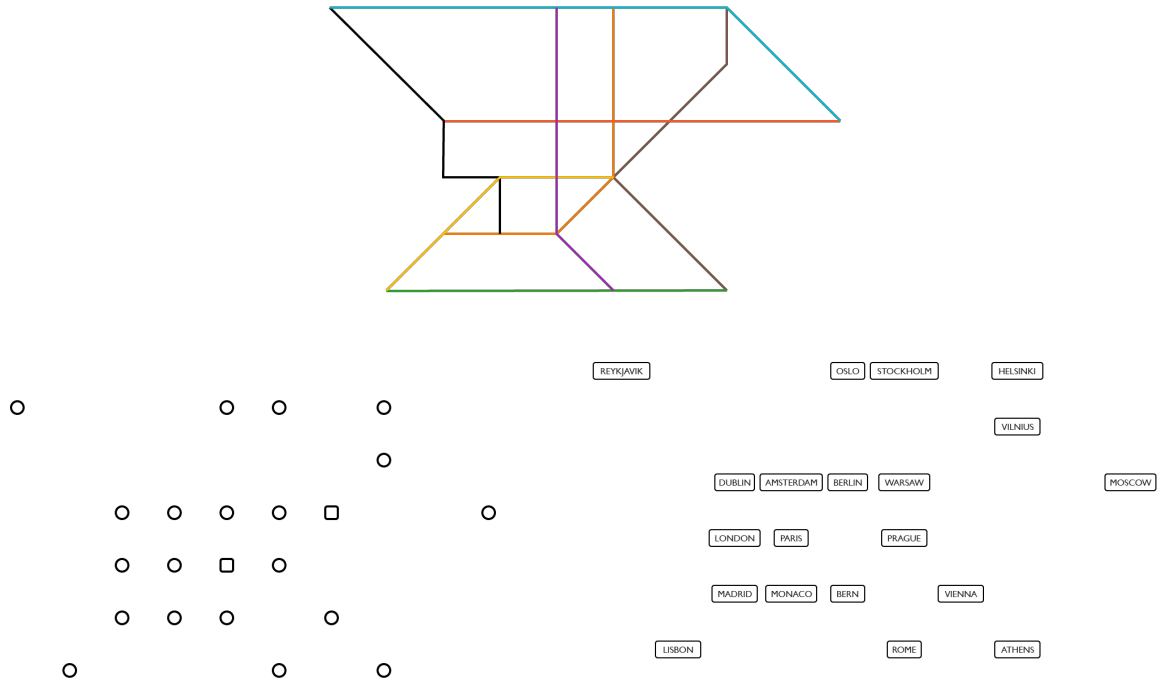
Implementing Obstacles

The requirement to implement obstacles in our game meant that we had to modify some of the existing art assets and even create new ones of our own. The game map was originally a single .png image with cities, rail lines and name tags all being drawn in the same layer. To implement obstacles (broken rail lines) between cities we needed to draw separate components of the game map on different layers so that when drawing-in obstacles we only have them overlapping rail lines and not things like name tags.

Original Full Map

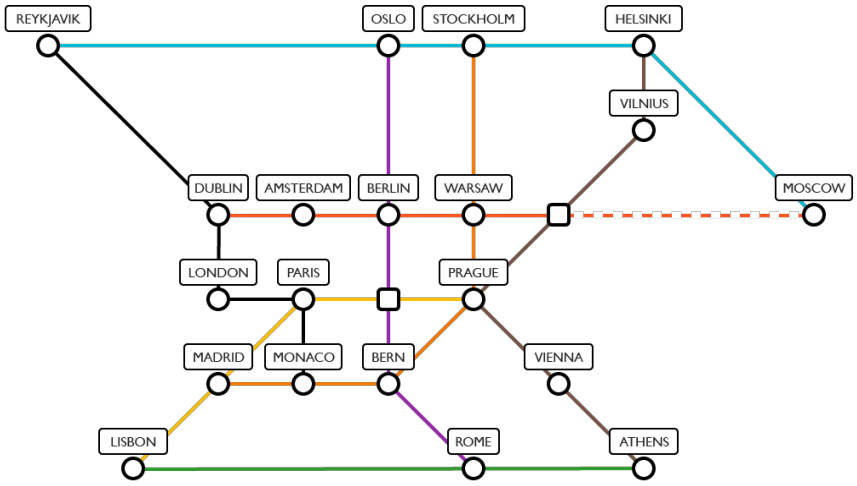


Separated Map Assets



The three separated assets are drawn on top of each other to create the full game map. Lines which are not traversable (blocked by an obstacle) are represented by dashed lines. Dashed lines are drawn on top of the lines layer of the map but below the names and cities layers. Separate obstacle images had to be created for the paths between every city or junction.

Example of the map with an obstructed line between Moscow and a junction



There are two methods for implementing the breaking and fixing of rail, aptly named breakConnection() and repairConnection(). We also had to create a separate class connectionSprite to help with deciding which broken rail asset to draw, or remove.

The method for breaking rail takes two Map Objects as parameters, it then goes through every city in the map and finds both objects, and then iterates through the connections from both of these objects finding the two connections which connect the two Map Objects passed in as parameters. It then checks through every train currently on the map to see whether any trains are currently on either of these connections, because breaking a connection with a train on is against the Geneva Convention, and so if there is the connection will not be broken. If the connections are both clear of trains however, they will both be set to untraversable which will mean trains currently on route will stop when they get to that connection and routes cannot be made across that connection. This is where the connectionSprites come in, a connection sprite has two Map Objects as attributes, therefore we then iterate over all the connectionSprites and draw (or make visible) the one that has the two Map Objects correlating to the two connections. This method will also fire a warning if there is no connection between the parameterised Map Objects or the connections are already broken.

The method for fixing the rail is very similar in the way it iterates to find the two connections, however it does not check whether there are any trains on the connections as there should never be trains on a broken rail. It then finds the correct connectionSprite in the same way and then sets it to invisible. It will also fire a warning if there is no connection between the parameterised Map Objects or the connections are already fixed.

We have also changed trains so that they are now obstacles to each other. This meant implementing a method, **checkForCollision()**, to detect weather trains have collided by testing whether any train actors overlap.

The method for implementing train collisions uses the actors bounds, which are found in a **getBounds()** method within the Game_Map_Train class. This returns a rectangle with the x and y coordinates and the width and height of the actor. We can then use the built in **overlaps()** function within rectangle to check whether any trains on the map collide. We do this by iteration through the list of all trains, and then iteration through all other trains creating every possible pair of trains apart from the same train with itself. With each of these pairs we then use the **getBounds()** and **overlaps()** methods to check whether they collide with each other. If they do, we first check if they are both on the same route, if they are we delete both trains from the game as they are destroyed on collision and break the rail that they are both on as it is damaged by the incident. If they aren't both on route and one of them is in a city however we do not need to worry about the connection as we do not break any connection if one is in a city. Therefore we just remove both the trains from the game. If the trains collide but are both on separate bits of rail (connections that aren’t the reciprocal of each other) then nothing happens.

Additional tests were written to check the newly added obstacle features. The tests are discussed in the the test plan document on page p7-11.

Implementing Quantifiable Goals

The game we inherited had a basic framework for goals in place, where random routes could be generated and displayed on goal cards. These goals came with no scoring method and we have had to implement a system for doing this. When implementing the goals, Dijkstra's algorithm is used to calculate the gold and points gained from each goal. Points are calculated by using a logarithmic function.

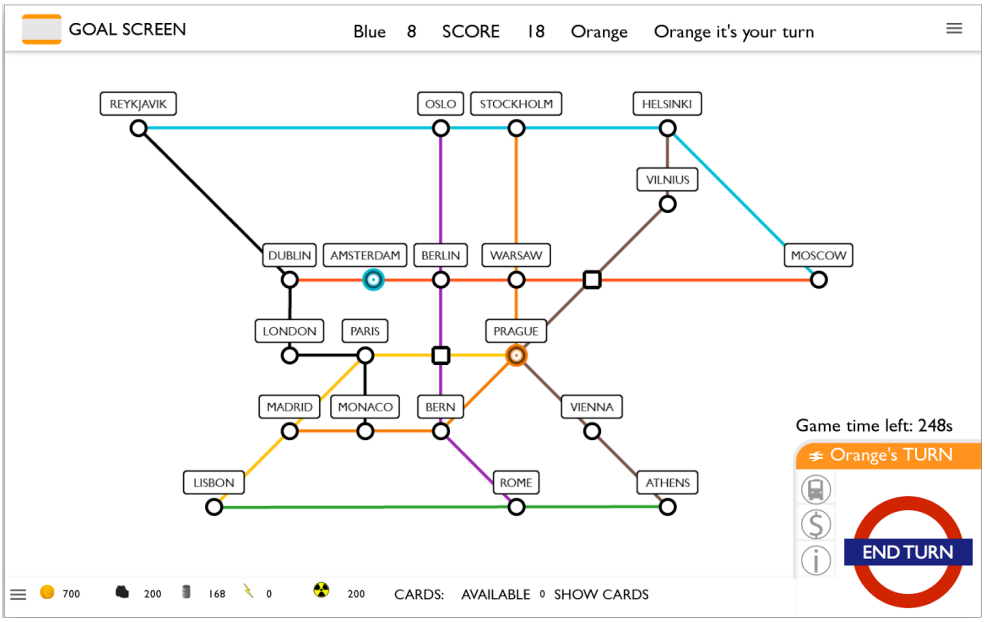
The tests for this feature can be found on page 11 of the test document.

Implementing Scoring

We decided that a player's score should be calculated as logarithmic function on the total gold they have accumulated over the course of the game. This is because the gold earned from completing goals is calculated in a balanced way using Dijkstra's algorithm and also because it adds another layer of strategy to the game, where players can risk selling resources to gain score. This dynamic scoring mechanism is to support the game's playability and entertainment value. This is achieved by giving the player a wide range of strategy options where they can risk selling resources, which they could potentially need for a later objective, to gain more points. Managing this mechanism wisely is crucial if a player hopes to win the game. Scoring was implemented by adding functionality to the addGold() method, where a players points are increased by a logarithmic function whenever a player gains gold.

The score is updated at the end of every turn and re-drawn on the game screen above the game map. The game timer is shown just above the bottom right HUD element.

The Game Screen with Scores Implemented



A game timer was also implemented so that the end of the game could be enforced, thus allowing for a winner to be determined. It was a requirement for the game to last for approximately fifteen minutes so this time option was included in the start new game menu. We decided to give players the option to play shorter games of five and ten minutes too as we felt that some players may be short on time and that this did not subtract from the fifteen minutes of play requirement, but instead added more flexibility for users.

The timer implementation uses the system clock to calculate the number of milliseconds elapsed since the game start (in particular, from the instant when both players have selected their start stations) and subtract this elapsed time from the selected game duration. Of course, the game duration is represented to the user in a number of minutes, the game scales this value up to milliseconds in order to calculate a precise value for the number of seconds left in the game. There were also considerations to include a timer for individual turns, so that it would be impossible for one player to take considerably longer than the other player. The amount of time we had discussed for a single turn was 60 seconds, an appropriately short duration to encourage a fast-paced game without rushing the players too much. However, we decided that this turn timer was unnecessary, as fairness in a hotseat style multiplayer game can be enforced by the actual players, providing infinitely more flexibility.

Tests for these added features are detailed on page 11 of the test plan document.

Game Extensions

We implemented a number of extra features into the game to improve playability and deepen the level of strategy that players can use. These extensions are not explicitly required for assessment 3, however they do not violate the requirements of the game.

Buying and Selling Trains

The ability to buy and sell trains was added because we feel it provides a good mechanism for game progression. The game came with a store where players could buy and sell fuel. It also had a button in-place to buy and sell trains, however this had no functionality and so that is what we implemented. In the game cities can be owned by a certain player and each city gives the owner a certain type of resources every turn. We took this into account when implementing the buying of trains. When you buy a train you select the city it is created in; if the city is not owned, then the player buying the train will be the owner of the city and the train created there will be powered by the fuel that the city provides the player. Selling trains simply removes the train actor from the game and gives the owner gold equal to 66% of train cost.

Teleport Card

The game features cards which each player can use to get an advantage over their opponent. Resource cards already existed in the game and along with the break and repair rail cards (see Implementing Obstacles). We decided to implement a teleport card which grants the player the ability to move one of their trains to any city on the map. This card addition was to add further progression and strategy mechanics.

The method behind the implementation of this card just sets a boolean *teleportTrain* to true, which then changes what happens when a train is pressed, because this variable is true when a train is clicked on it is set to the teleport cards train attribute. *teleportTrain* is then set to false which reverts back to the normal interaction with trains, *teleportCity* is then set to true which changes what happens when a city is pressed. Now when a city is pressed, the current teleport cards train, which is available through a *currentTeleportCard* variable is moved to that city. *teleportCity* is then set to false and *currentTeleportCard* is set to null, reverting back to the normal interaction with the game. The tests for these methods can be found on page 3-6.

Go Faster Stripes Card

A ‘Go Faster Stripes’ card was also added to the game so that players can increase the speed of their trains, allowing them to travel further per turn. This was to add another dimension of strategy to the game.

The method behind the implementation of the go Faster Stripes card is very similar to the implementation of the teleportation card, it sets a variable *goFaster* to true which changes the way we interact with trains. The next train that is clicked on is then given a speed upgrade and the variable *goFaster* is set back to false to resume normal interaction with trains.

Test methods on similar cards can be found on pages 3-6 of the test document.

Challenges Encountered

Working on another team's software has brought about extra challenges in the software development process, from dealing with existing bugs to editing completely unfamiliar code.

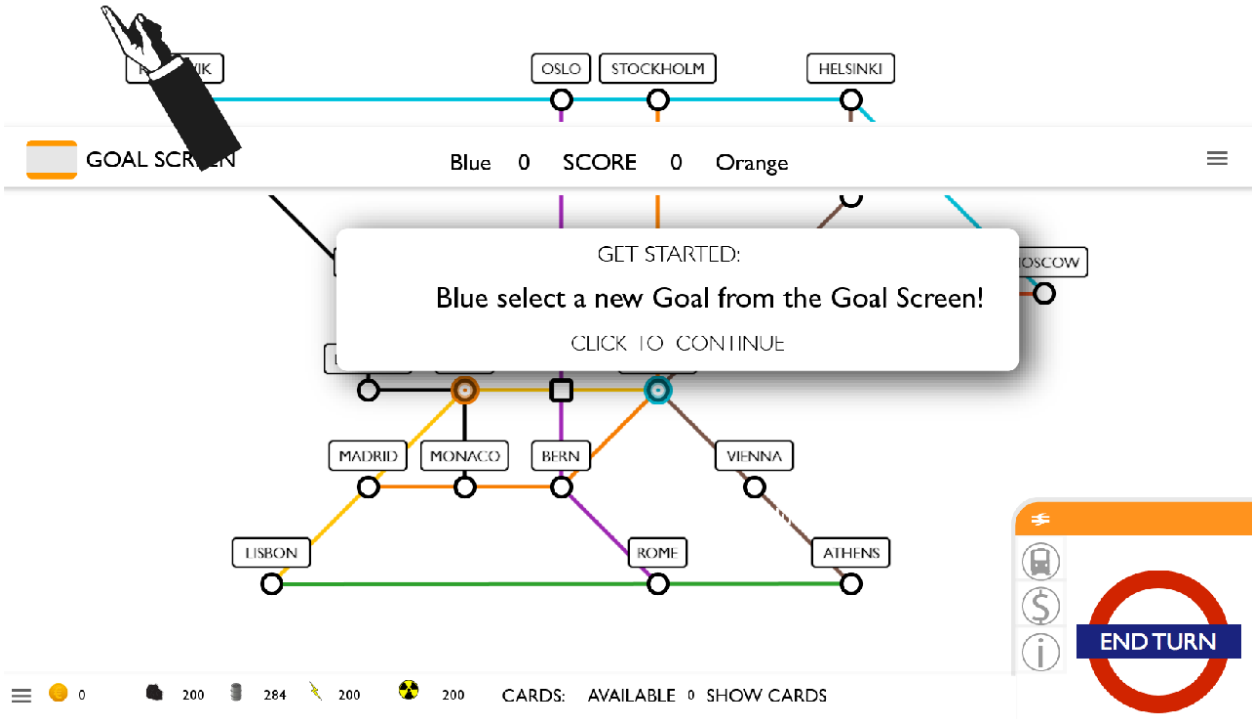
Code Structure

The first major issue was the way the code was structured. In no way was the structure of the code bad, however it was laid out in a completely different way than what we were used to. Our previous project was very simple in its layout, with a GameScreen class that included all the methods needed to make changes to the map. Team HEC however had a GameScreen and then the map could be changed from almost any class using the Game_Map_Manager which, although hard to get used to, was much more intuitive than our previous method. Another challenge was in the way that they had used Actors. We had used Actors as the map objects themselves, however in this game's objects are created and then each object has its own Actor attached to it. This made it a little bit more difficult to make changes to objects based on interaction, such as clicks and collisions, which all depend on the Actor rather than the object. However in most instances Objects had their relative Actor as an attribute, allowing for easy access. In the instances where this wasn't the case, and was needed, we added this in.

Scaling

When we first started working on Team HEC's project, we discovered that their strict single resolution support (that is, support for 1680x1050 monitors) presented large issues for several members of our team. Specifically, Jack, Marco, and Jordan found that their laptops did not have high enough resolution support for the game to render correctly. In an attempt to rectify this, Michael researched libGDX's Viewports functionality as a way of implementing virtual resolutions. We found that it would have been possible to scale the game's user interface such that it would fit on lower resolution screens, however as the graphics for the game are not vectorised, this led to severe graphical distortion, so we decided instead to use external monitors for testing the game.

On top of this, Team HEC's decision to render elements of the user interface on different stages made it very difficult to scale without causing the misalignment of some elements (generally the top_bar and Goal_Screen). Both of these described effects can be seen in the screenshot below (the distortion effect is more noticeable on a full-size image):



Previous Testing

Some main issues were the result of a lack of black-box testing by the previous team as there were specific instances in the game where null pointers were unhandled. These occurred in places such as aborting a route that currently had no connections.

Updating Documentation

The user manual was a slight challenge to update as it was not formatted in a good way and was rather verbose. The document mediator of our group had to get their head around the formatting decisions of the former group as rewriting the entire manual would have taken too long. In the end the document was cleaned up and relevant structuring was implemented, such as giving the game an introduction.

GitHub

The team encountered an error with the GitHub desktop client as it did not sync repository and local versions of the code. The team members who encountered this problem could not find a fix for this and so began using the git command line tools. Using the command line tools solved this problem.

Software Engineering Approaches

For this assessment we adopted an agile approach to development. The team met on a weekly basis for scrums, using the application Wunderlist to record the progress and delegation of tasks. Our sprints were scheduled as weekly, however we found that we often did sprints of a few days at a time. The roles of group members were more defined in this assessment as we believed it would help us work more efficiently as a software development team. We had one person in charge of testing, one person mediating the creation and extension of documentation, and three programmers. We found that this worked effectively however the programmers needed to help create the new documents as they had a greater working knowledge of their implementations.

The application Wunderlist should be highlighted as being a key tool that we used to monitor and influence the progression of the assessment. It is a shared to-do list, with which you can assign tasks to individuals. Notifications are also pushed to individuals devices so that everyone knew when a task was added or completed. A sample screenshot of wunderlist is show below: