

Software Testing Report

Testing methodology

Software testing is a process or an activity that is done on a software application that helps find software bugs. Throughout our software engineering project, a variety of testing methods were used to help discover as many software bugs as possible and to ensure our code was bug-free. Some of the testing methods include JUnit Testing, a unit testing framework for Java and white box testing. All our testing was done as the software was being developed as we are using Agile development method.

Unit testing was implemented using JUnit as Java was the programming language we use and because JUnit is very simple and efficient to use with Java applications, thus a lot of testing could be done. JUnit tests have a simple framework for self-verifying tests in Java, this allows tests to be run quickly, allowing the programme to be developed correctly. JUnit testing is very useful when new features are added as it isolates parts of the software that you want specifically to test, so new added features in different parts of the software cannot affect existing JUnit tests. JUnit testing produces formal testing results, this allows a team working on a project to provide proof of their products capabilities.

Whitebox testing is based on the inner workings of software and revolves around internal testing. In this time of testing, the code is visible to the tester. It focusses on verifying the input and output of specific parts of the software to make sure they are doing the right thing. This allows us to improve the design, usability and reliability of the programme. This method of testing allowed us to isolate and identify problems with specific parts of the programme early on which meant at any given stage of our project we had little to no bugs. Testing using Whitebox testing and using JUnit tests can help minimise the bugs in the development of our programme.

We considered usability testing for our testing report. Usability testing looks at the usability aspect of our programme. The user will be asked to complete tasks while being observed, the observer will note down any problems they encounter. More common problems will be prioritised to be fixed first with less common problems later. Feedback will be given by the user on what they think of the programme. As our user demographic involves students who come on open days, students who don't do computer science will be suitable for our testing. However, we decided not to use this testing methodology because it would require special permission along with ethics forms to carry out this testing. It also doesn't fit well with our Agile development approach.

Accepting testing involves testing the software that is tested for acceptability. The purpose of this is to test whether the software has met the core requirements and fulfilled the fit criteria. The software is constantly compared to the fit criteria to ensure all project requirements are met.

The main point of black box testing is to test the validity of the requirements of a project. It can be used to test functional requirements which is what we used it for in our testing report. It is the testing technique in which the functionality is tested without looking at the code.

Test report

White Box Testing

All our white box testing was done during the development of Kroy as we use the Agile development method.

White box testing was used when making small changes to the code to make sure the code worked with the changes and to make sure no problems or bugs were present. If there were any problems when the code was changed then the programmers made sure to debug the code, find the problem and then change the code causing problems and then rerun the test.

White box testing was considered in our development method as we make small changes in our programme and then run small tests during the development of our programme and not right at the end. Therefore, Whitebox testing complements our Agile approach.

Unit testing:

The Unit testing report can be found in the testing document. The link to the document is at the bottom of this software testing report. For our Unit testing report, we used a table to display all our results. For our unit testing report, each test was assigned a "Test ID", "Test function name", "function tested", "function use", "result of test" and "Test description". A "test ID" was given so any given test could be referred to relatively easily. The "test function name" was given to the specific method that tests some code within the Junit file. "Function tested" was given so we know what specific function was being tested by the test function. "Function use" was given so that more information can be provided about what was being tested if someone chooses not to go through the code or if someone does not understand the code. A "result of test" is given which can either be a pass or a fail depending on the outcome of the JUnit test. Finally, we have a "test description" which goes into detail about how each specific unit test works and what it's looking for so that it passes. The reason for choosing these columns for each test was so throughout our project, tests could easily be referred to and to make sure that these results were as reproducible as possible.

All our Unit testing was done using JUnit as our code is in java. In JUnit testing, the code is visible to the person testing the programme. All the JUnit tests were created to make sure that each specific part of the code was producing a specific result, so we know that the code is working correctly. The classes that we tested were Entity, Firetruck, Fortress and Projectile. The reason for only choosing these 4 classes to test is because we deemed that unit testing would be the most effective way to test in only these classes. Within the other classes, unit testing was not appropriate as they were more graphical and only changed screens. They also had very basic code, so we decided it wasn't necessary to have unit tests for them. In total there were 13 unit tests for the code and all of them passed.

Black box testing

The black box testing report can be found in the testing document. The link to the document is at the bottom of this software testing report. For our Black box testing, we used a table once again to display all our results. Each black box test was assigned a unique "Test id", "test information", "Relevant requirement(s)", "Expected result", "actual result", "result of test" and finally "test description and evidence". A test ID was given so any given black box test could be referred to relatively easily in our project. "Test information" was used because we need a description of what was being tested. A relevant requirement was given as each test relates to a functional requirement. We had an "expected result" which is what we expect to see to pass the test. We had an "actual

result” to describe what happened. We also have the “result of test” which indicates if the test was a pass or a failure. Finally, we had a “test description and evidence” to show how we conducted our test and to provide the necessary evidence of our test passing or if it didn’t pass, we say why it failed. We decided to use this structure for our testing table because it gives our results as much information as possible and it shows how we meet each functional requirement.

In total we had 11 black-box tests, only 1 test failed. The only test that failed was Fortress_Difficulty, this was a test to see if the difficulty increases as the game goes on. The reason for this test failing was because all fortresses have the same health as the function to increase the health of the fortress, to make the difficulty harder as the game goes on, has not been implemented in the game yet and will be at a later stage.

Acceptance testing

The acceptance testing report can be found in the testing document. The link to the document is at the bottom of this software testing report. For our acceptance testing, we used multiple tables which were split into functional requirements, non-functional requirements and constraint requirements. Each test was assigned a “Test id”, “requirement id”, “fit criterion”, “result” and evidence. A test ID was given so any given acceptance test could be referred to relatively easily in our project. A requirement id was given so we know what requirement the test relates to. A description is given to describe what requirement the test was looking at. A result was given which either was a pass or a fail. “Evidence” will describe the reason for failing or the evidence for passing the test.

For functional requirements we had 11 tests and 1 failed. F_FHealth failed because that feature of increasing fortress strength does not need to be implemented in the game at this stage. All the non-functional requirements and constraint requirements have passed.

Testing Material

The results of the testing can be found here:

Test Report

JUnit tests

Entity test (Run with JUnit EntityTest)

Test ID	Test function name	Function tested	Function Use	Result of test	Test description
Entity_gHealth	getHealth()	getHealth()	Gets the current value of health for an entity	Pass	Tests if the health returned is the correct amount of health which should be 100.
Entity_sHealth	setHealth()	getHealth() setHealth()	Gets the current value of health for an entity.	Pass	The tests sets the health of an entity to 50 and then checks using the getHealth() function that the setHealth() is working
Entity_tDamage	takeDamage()	takeDamage()	Removes a specified amount of health from an entity.	Pass	This test damages an entity by a specific amount of damage then checks the health of the entity to see if the damage reduced the health by the amount of damage.
Entity_iDestroyed	isDestroyed()	isDestroyed()	Checks if the health of an entity is less than or equalled to 0. That entity would be considered destroyed.	Pass	This test checks 3 scenarios. If the health is 0, if the health is less than 0 and if the health is greater 0. In our first scenario health of an entity is set to 0 then the function isDestroyed() is checked for it to be true. Damage is added to an entity so the health is less than 0, the isDestroyed() function is then checked again for it to be true. Finally the health of an entity is set to 0 and the isDestroyed()

					function is checked for it to be false . If it is false it will be accepted, and the test will be passed.
--	--	--	--	--	--

FireTrucks test (test with JUnit FireTrucksTest)

Test ID	Test function name	Function tested	Function Use	Result of test	Test description
FT_tInitialisation	testInitialisation()	getHealth() getWater()	Returns the current amount of Health in a firetruck. Returns the current amount of water in a firetruck.	Pass	The firetruck health and maxWater are initialised to 100 and 500 respectively. This test checks if they have been initialised correctly.
FT_tRefill	testRefill()	getWater()	Returns the amount of Water in the firetruck.	Pass	50 units of water is taken from the firetrucks water supply. Amount of water is checked if it's correct. Then water supply is refilled by 1 unit and it is checked again to see if it's equal to the amount refilled by.
FT_tTurning	testTurning()	getDirection()	Returns the firetrucks current direction	Pass	First, the truck direction is checked so it's in the same direction of where it has been initialised. The firetruck is turned right and the direction of the

					truck is checked.
FT_tAttack	testAttack()	attack()	The attack class deals out damage and reduces the amount of water in the firetruck.	Pass	This test checks if the attack class works. First the drops are checked if they are empty as there isn't any attacking going on. Then a truck attacks and the test checks if any drops are present to check as this indicates the attack is working.

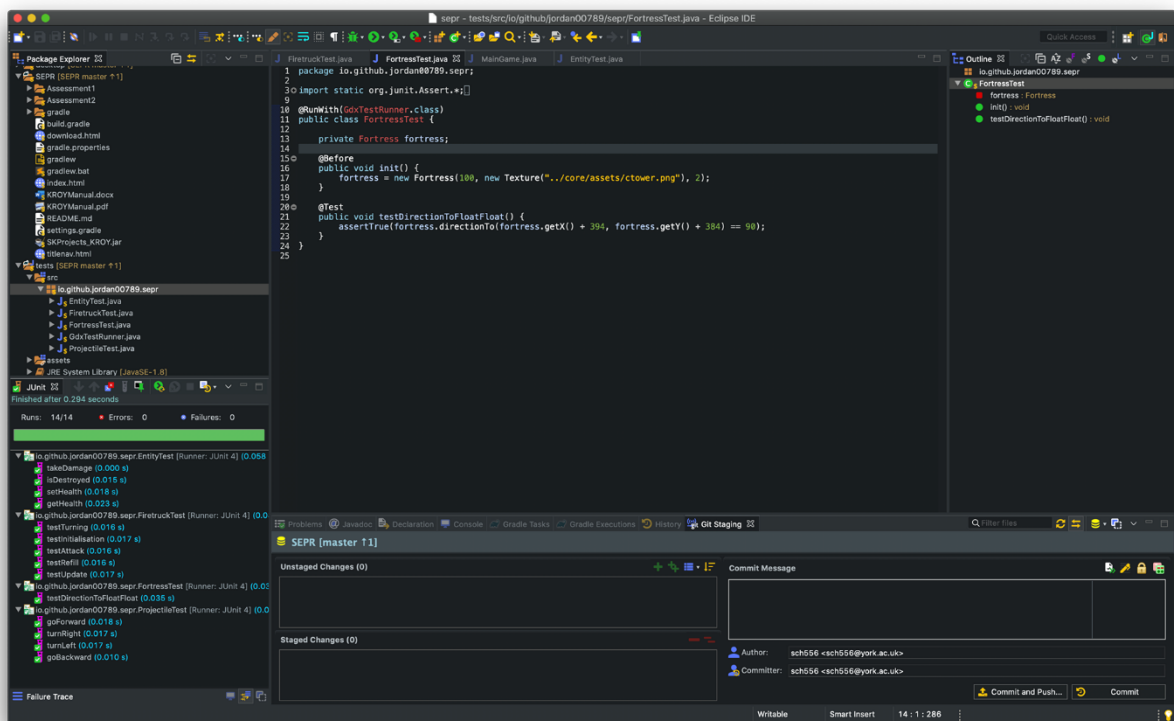
Projectile test (Run with JUnit ProjectileTest)

Test ID	Test function name	Function tested	Function Use	Result of test	Test description
Projectile_TL	turnLeft()	getDirection()	Decreases the direction by 1 unit	Pass	The test initialises the direction variable to 0. Then decreases direction by 1 and then checks if it gets decreased by 1.
Projectile_TR	turnRight()	getDirection()	Increases the direction by 1 unit	Pass	The test initialises the direction variable to 0. Then increases direction by 1 and then checks if it gets increased by 1.
Projectile_GF	goForward()	getVelocity()	Increases velocity by 1 unit	Pass	The test initialises the velocity variable to 0. Then increases direction by 1 and then checks if it gets increased by 1.

Projectile_GB	goBackward()	getVelocity()	Decreases velocity by 1 unit	Pass	The test initialises the velocity variable to 0. Then decreases direction by 1 and then checks if it gets decreased by 1.
---------------	--------------	---------------	------------------------------	------	---

Fortress test (Run with JUnit FortressTest)

Test ID	Test function name	Function tested	Function Use	Result of test	Test description
Fortress_DFF	testDirectionToFloatFloat()	directionTo(float x, float y)	Calculates direction from a fortress to a point.	Pass	Checks if the given point that is given that calculates a direction of a fortress is the correct direction.



The image above shows all tests passing.

BlackBox Testing

Test id	Test information	Relevant requirement(s)	Expected Result	Actual result	Result of test	Test description and EVIDENCE
Map_Load	Map loads up when you press Space-Bar on main menu.	FR_MAP	Map loads up correctly when no issues	As expected.	Pass	The game was launched. The space bar button was pressed and the game loaded. See figure 1 for evidence.
Truck_Shoot	When a player shoots water the water supply must decrease.	FR_SHOOT	Hitting the shoot button (space bar) decreases water supply.	As expected.	Pass	We controlled a fire truck, hit the space bar and observed the tank of water decrease. See figure 2 for evidence.
ET_Shoot	The ET fortresses shoot projectiles at the player.	FR_FORTSHOOT	Fortress shoot projectiles in the player's direction.	As expected.	Pass	We controlled a fire truck to go towards a fortress and observed that the fortress was attempting to attack the fire truck. See figure 3 for evidence
Arrow_Move	Player must be able to move the fire truck using the arrow keys.	FR_CONROLS	Pressing up on arrow keys makes truck go up. Pressing left makes the truck turn left. Pressing right makes the truck turn right. Pressing back makes the truck reverse.	As expected.	Pass	To test the fire trucks with movement, we pressed each individual button and measured what that did. See figure 4A, 4B, 4C, 4D for evidence, these screenshots show the movement works.
Truck_Repair	Fire truck repairs its	FR_REGEN	When a fire truck is in the	As expected.	Pass	We made the fire truck take damage

	health when the player moves the fire truck to the fire station.		red zone which is the fire station the health should go up.			and then took it to the red zone which is the fire station and saw that the health does go up. See figure 5A 5B for evidence.
Truck_R Water	Fire truck refills water when the player moves the fire truck to the fire station.	FR_REFILL	When a fire truck is in the red zone which is the fire station the water level should go up.	As expected.	Pass	We depleted the fire truck water supply then took the truck to the red zone and observed if the water went up. See figure 6A 6B for evidence.
Screen_WIN	Player can see the win screen when you destroy all fortresses without dying.	FR_WIN	Win screen should pop up after fortress is destroyed.	As expected.	Pass	We destroyed all the fortresses with the fire trucks and the win screen popped up showing the win screen works. See figure 7 for evidence.
Surface_Speed	Player experiences different speeds for the fire trucks depending on the surface they are on.	FR_SPEED	The grass should slow down the fire truck. The buildings and road should not slow down the fire trucks.	As expected.	Pass	We accelerated the trucks on different surfaces to see the difference in speed. No relevant evidence needed.
Screen_Lose	Player must see the lose screen if all fire trucks lose all their health.	FR_LOSE	After both fire trucks are destroyed the lose screen must pop up.	As expected.	Pass	We let a fortress destroy both fire trucks and saw the lose screen pop up. See figure 8 for evidence.
Truck_Barrier	The barriers	FR_COLLISION	The fire truck must not go	As expected.	Pass	We tried ramming the fire truck

	must stop the fire truck when the player attempts to go through them.		through brown barriers.			through the barrier. See figure 9 for evidence.
Fortress_Difficulty	Player fortresses must become harder to destroy the more fortresses you destroy.	FR_FORTRESSHEALTH	Fortresses have more health the more you destroy. So, they should take more water to destroy.	Fortresses all have the same health.	Fail	We measured how much water was used to destroy each fortress as we completed the game, this did not change therefore the difficulty remained the same. No relevant evidence needed.

Figure 1



Figure 2

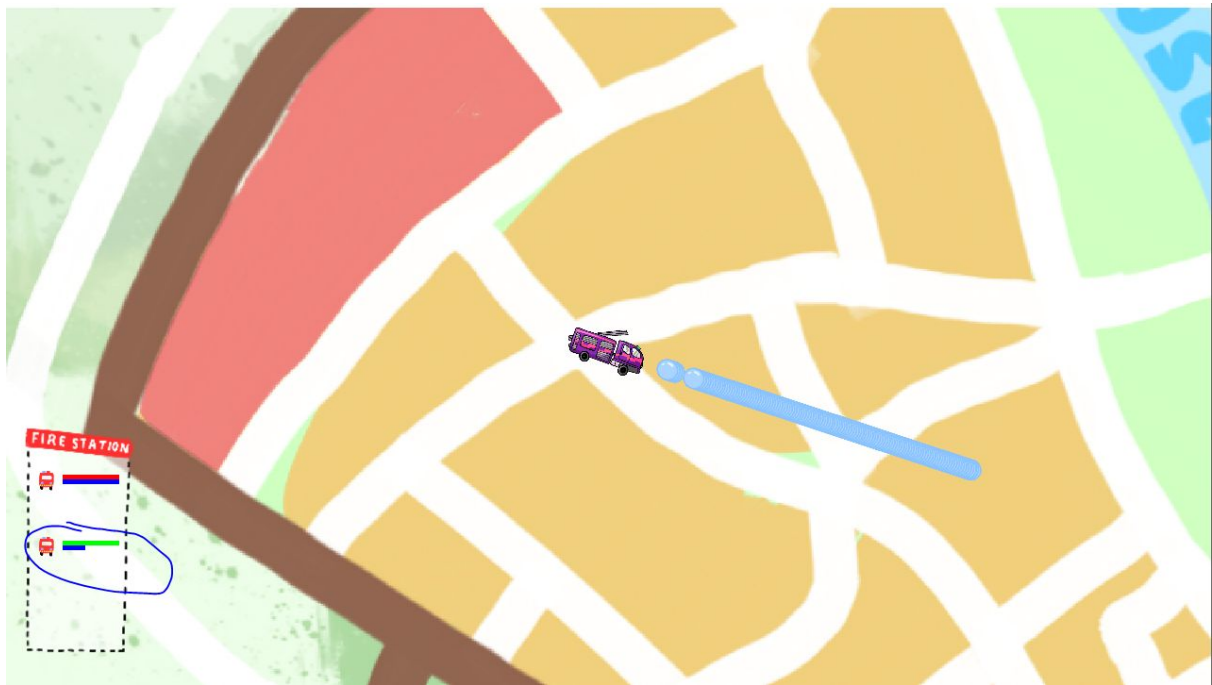


Figure 3

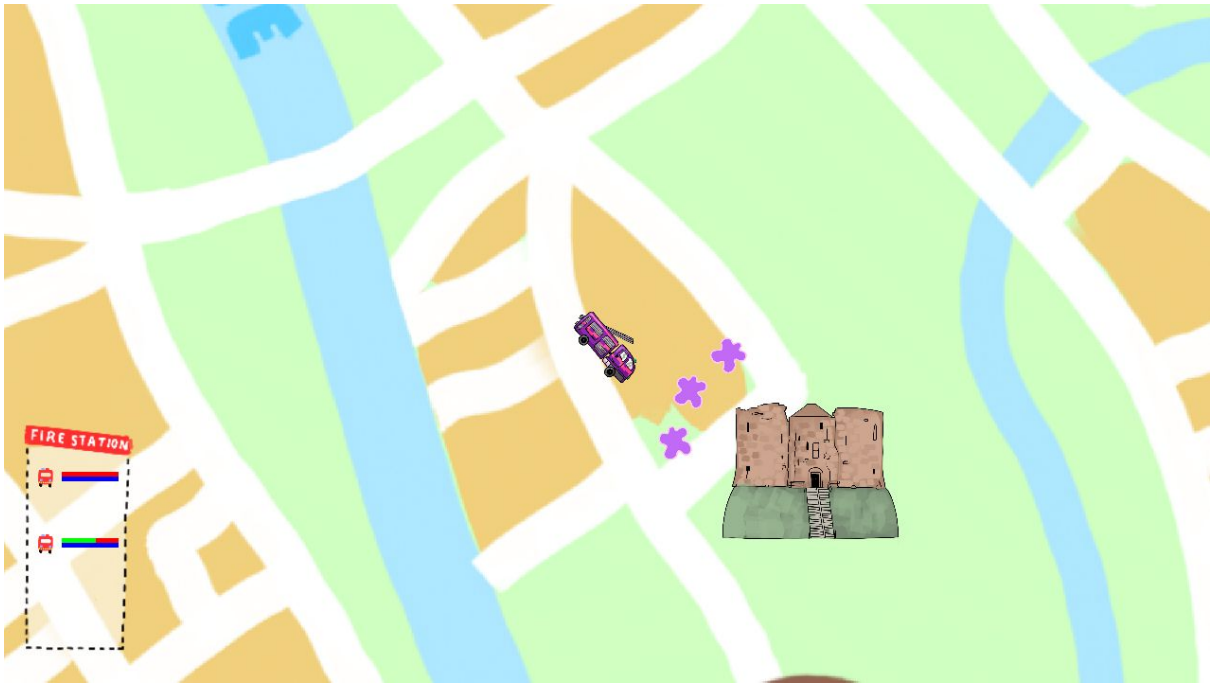


Figure 4A

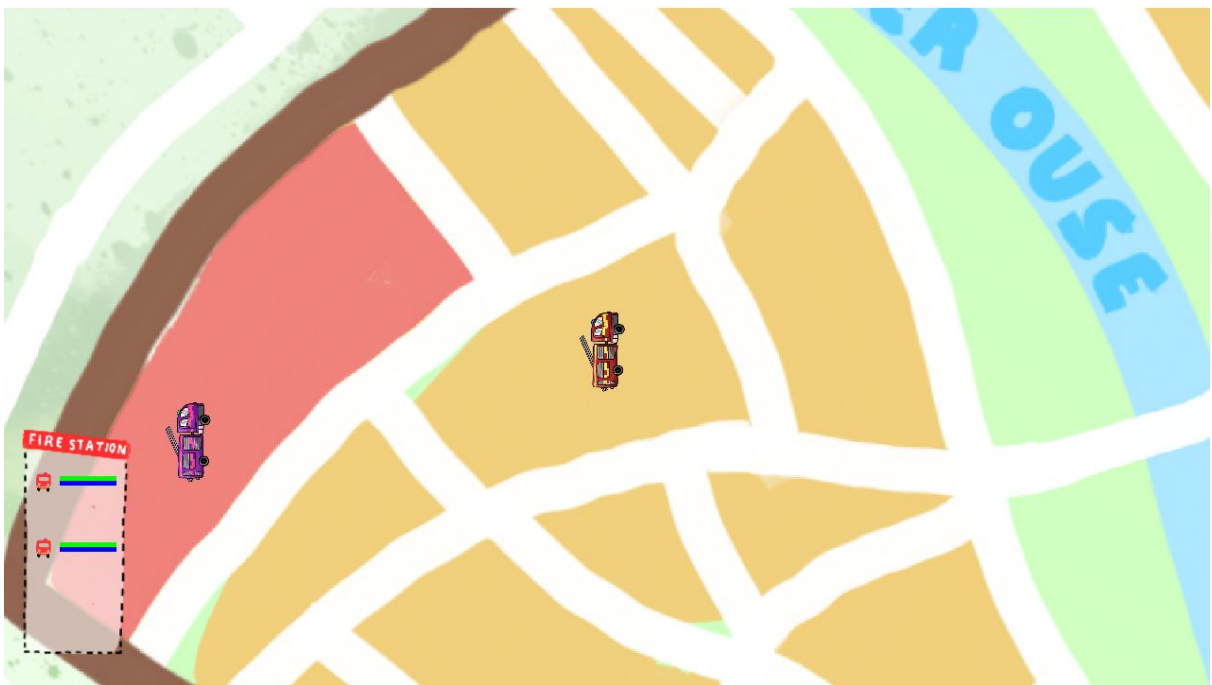


Figure 4B

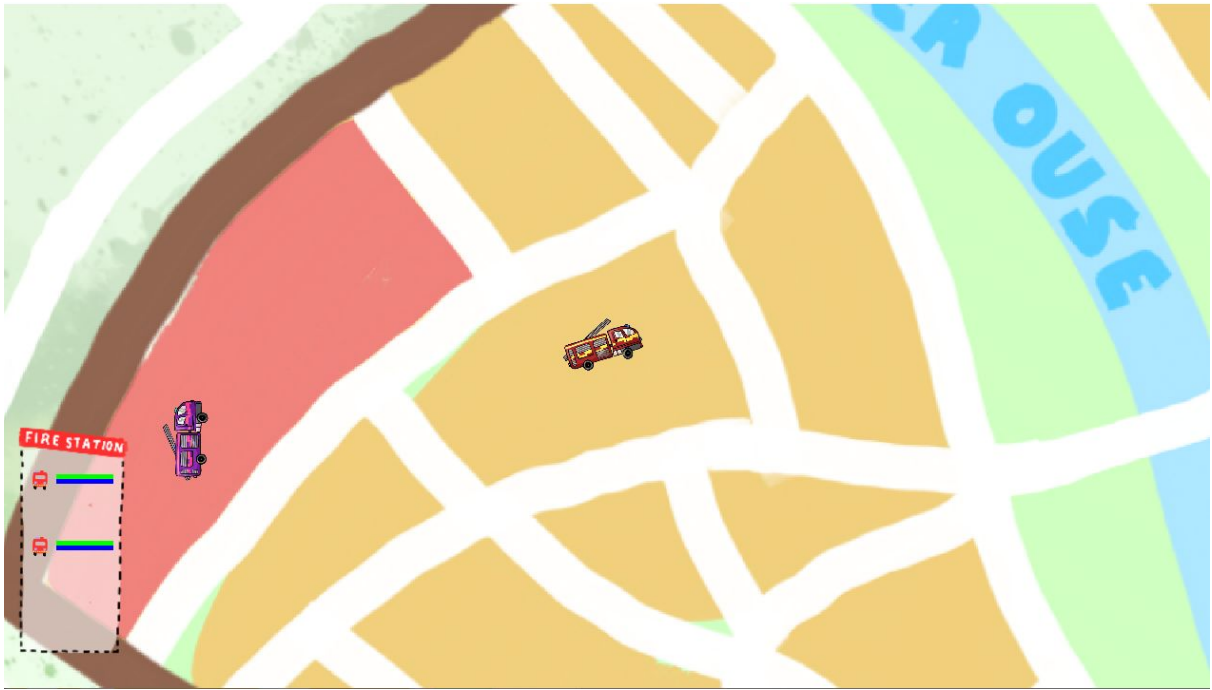


Figure 4C

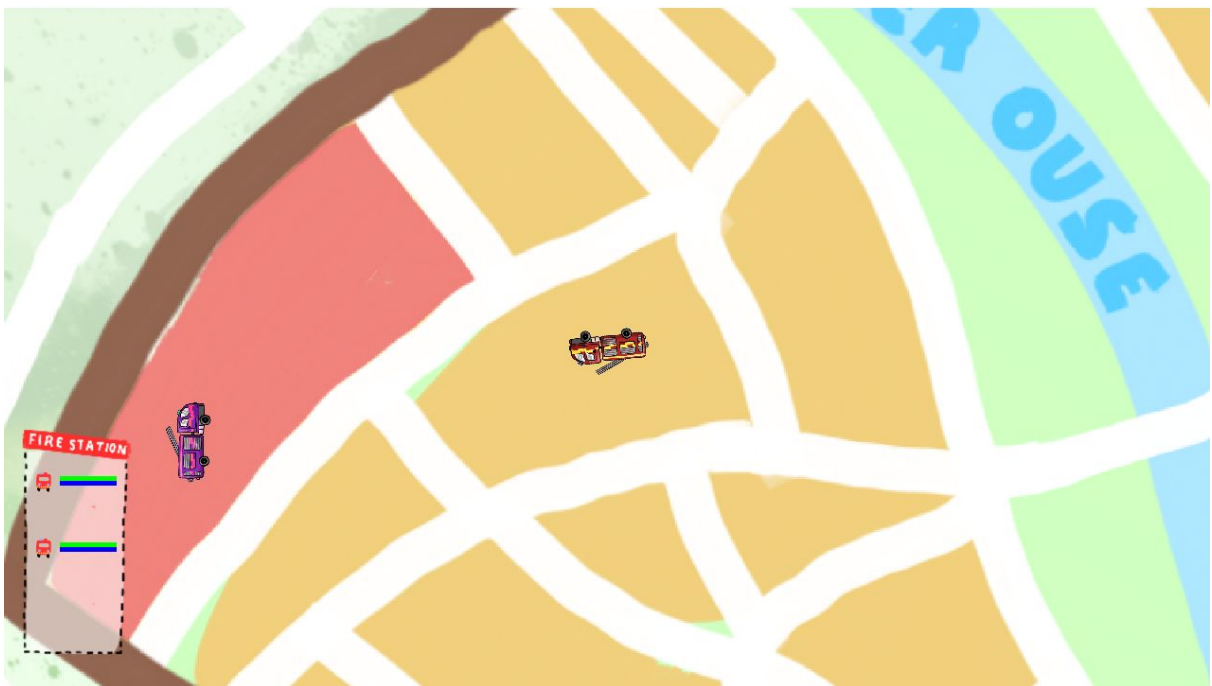


Figure 4D

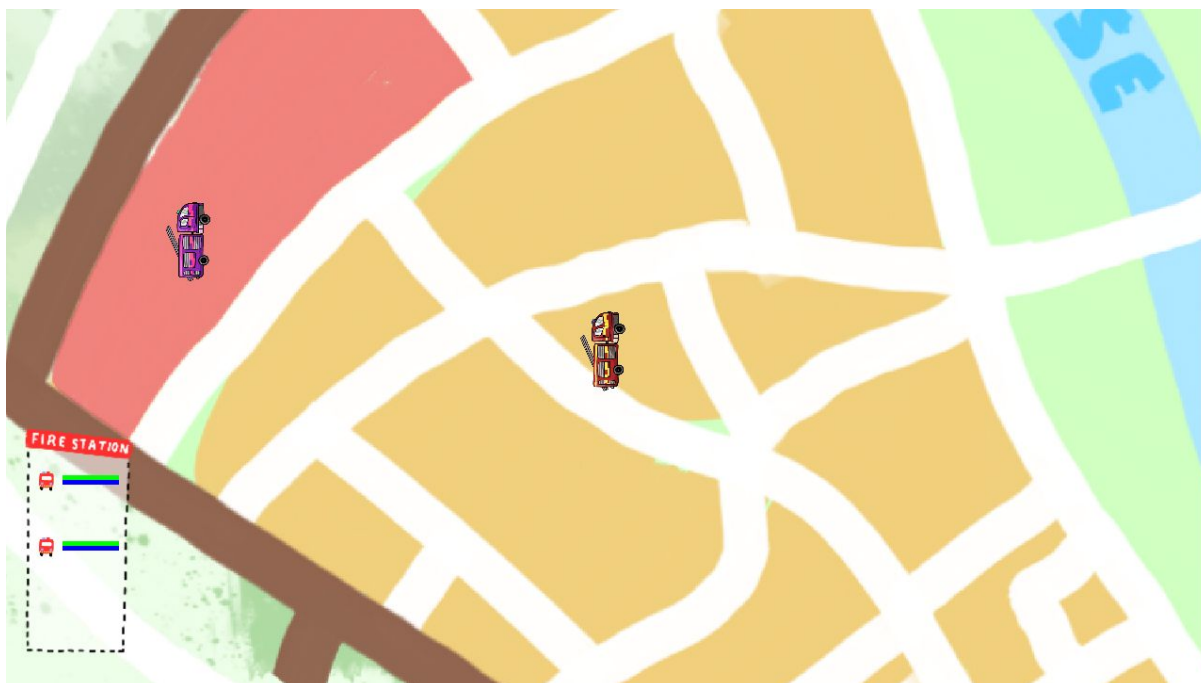


Figure 5A



Figure 5B



Figure 6A

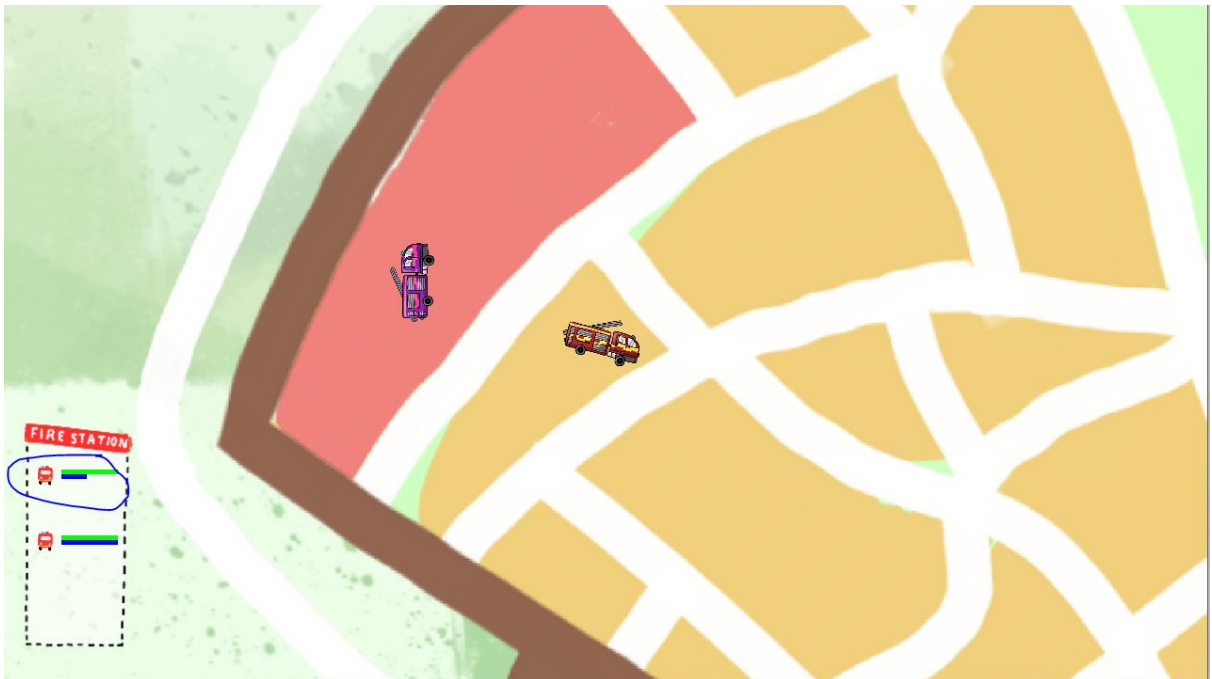


Figure 6B



Figure 7



Figure 8

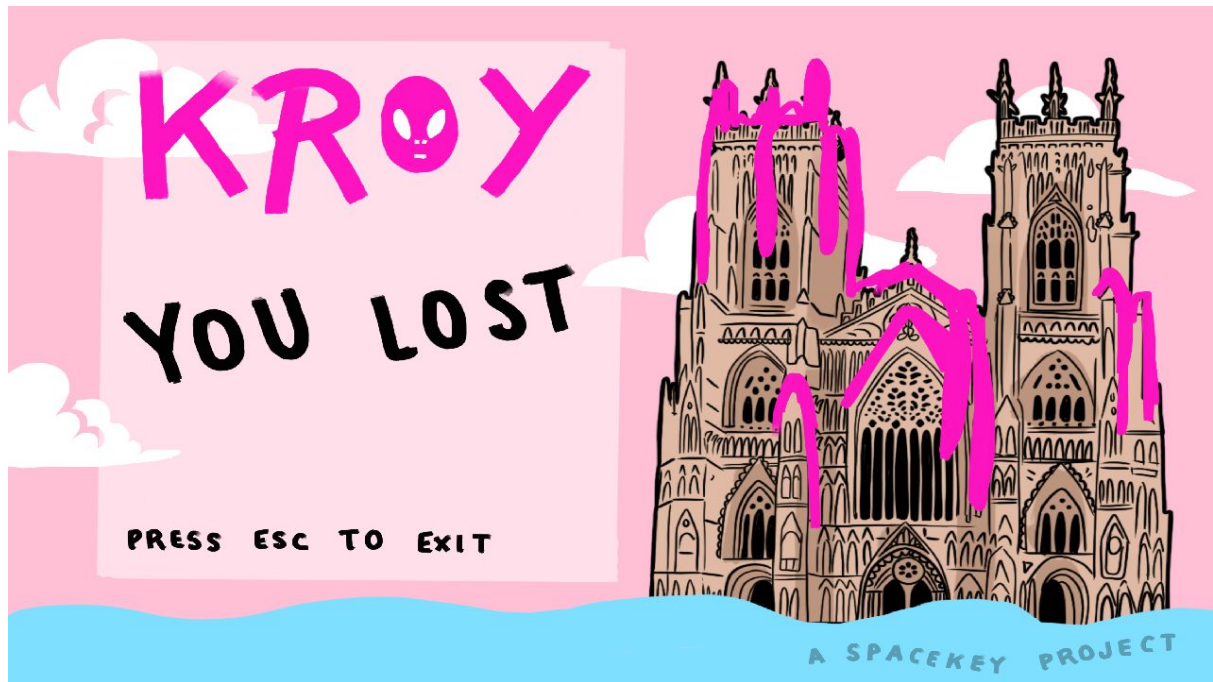
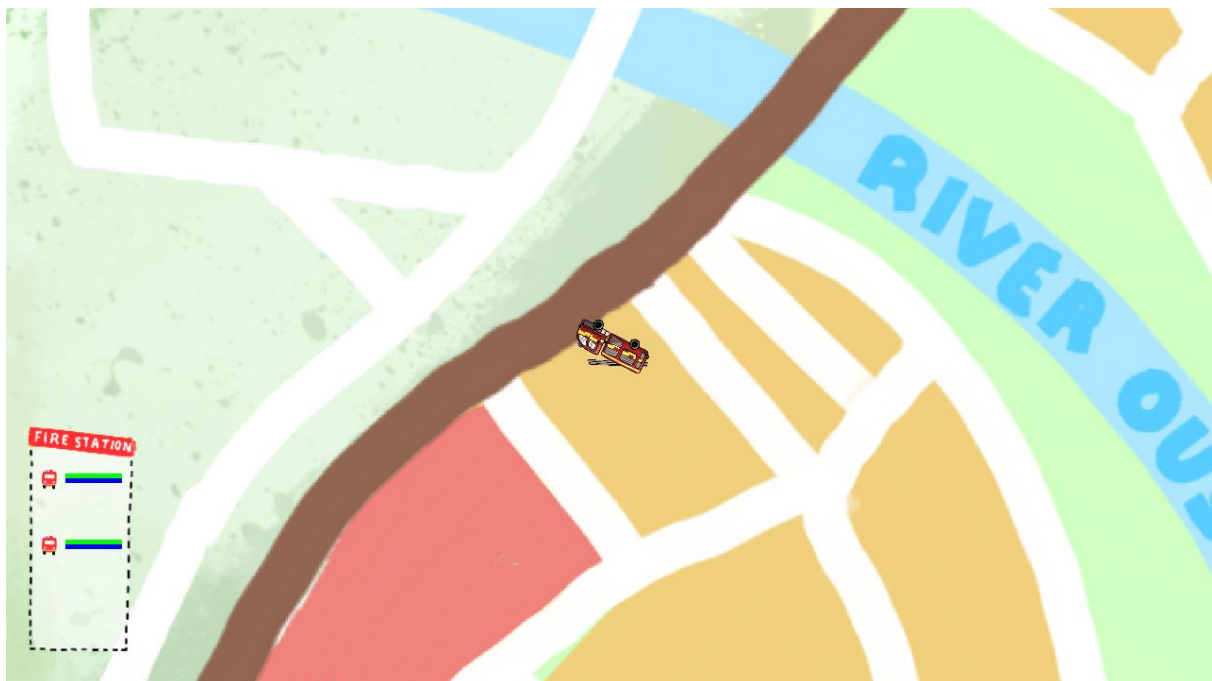


FIGURE 9



Acceptance test results

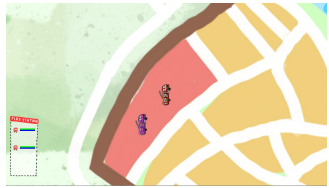
Functional Requirements test results:

TEST ID	REQUIREMENT ID	FIT CRITERION	RESULT	EVIDENCE
F_Map	FR_MAP	The map will be a simplified version of York city centre	Pass	We compared our map to the map of York, and it is a direct representation of York.
F_Shoot	FR_SHOOT	The fire engines will shoot water in the direction of driving and the water will continue to travel for 2 seconds	Pass	The water comes out the front end of the truck and does travel for 2 seconds when we timed it.
F_FShoot	FR_FORTSHOOT	The fortresses will shoot projectiles when a fire engine is in a range of 100 screen units and will continue to travel for 2 seconds	Pass	The fortresses do shoot projectiles which stay on screen for 2 seconds. The range is confirmed to be 100 screen units.

F_Gen	FR_REGEN	The fire engine's health is refilled at a specified rate of 60/second when it reaches the fire station	Pass	The rate of the fire engines health does increase at 60 units a second however it doesn't go past the maximum allocated amount.
F_Fill	FR_REFILL	The fire engine's water is refilled at a specified rate of 60/second when it reaches the fire station	Pass	The rate of the fire engines water supply does increase at 60 units a second however it doesn't go past the maximum allocated amount.
F_Arrow	FR_CONTROLS	The fire engine can be moved using the arrow keys on the keyboard	Pass	Pressing up, down, left and right arrow keys allows the user to control the fire engine.
F_Speed	FR_SPEED	The fire engine will move at a varied speed across the map from 0 to 100 pixels/second depending on the map beneath the truck. Slower on green and faster on roads	Pass	Going across grass slows down the fire truck as opposed to roads and buildings.
F_Stop	FR_COLLISION	The fire engines will not be able to drive through the fortresses or the city walls	Pass	Attempting to go through city walls will stop the fire truck in its tracks.
F_FHealth	FR_FORTRESSHEALTH	As the game progresses, the fortresses will increase in strength linearly 0.2 units/second	Fail	This has not been implemented yet and will be implemented at a later stage.
F_WS	FR_WIN	The system will end the game and present a win screen if all the fortresses have reached 0 health	Pass	After destroying all fortresses, the win screen does display.
F_LS	FR_LOSE	The system will end the game and present a lose screen if all the fire engines have 0 health	Pass	After depleting the health of all fire trucks, the lose screen does display.

Non-Functional Requirements test results:

TEST ID	REQUIREMENT ID	FIT CRITERION	RESULT	EVIDENCE
NF_Rep	NFR_REPEATABLE	The rules must not change each time the game is played	Pass	Each time you open the game nothing changes so the rules are constant.
NF_Time	NFR_TIME	The game must take no longer than 10 minutes to complete	Pass	Each member tested the game and it took under 10 minutes to complete.
NF_MTB Y	NFR_MAINTAINABILITY	There must be extensive documentation and the code should follow the Java naming conventions/coding style	Pass	All our code follows the java naming conventions and java coding style.
NF_Doc	NFR_DOCUMENTATION	Extensive documentation, specifying the class structure and code standards, and explaining the code functionality	Pass	All our code has documentation explaining what each specific part of the code does.
NF_Res	NFR_RESILIENCE	Whenever an error occurs, it will be caught and reported in an error log	Pass	We have had 0 crashes since the final release of the game after playing it multiple times. There are no errors in the official game release error log.
NF_Oper	NFR_OPERABILITY	A one page max tutorial page will be shown explaining how to play the game	Pass	The tutorial screen is less than 1 page in length. 
NF_User	NFR_USABILITY	Simple colour-scheme and gameplay which can be understood from	Pass	The colour scheme is very simple. We made sure to use not too many colours and simple colours, so the

		the NFR_OPERABILITY specified tutorial		game is easy to understand and play.
NF_GUI	NFR_GUI	The GUI will be simple and not take up more than 25% of the screen	Pass	We have a simple GUI in the game which makes the game look aesthetically pleasing and takes less than 25% of the screen. 
NF_Goals	NFR_GOALS	The user can destroy fortresses	Pass	To destroy a fortress, you just need to spray water at it.
NF_Var	NFR_VARIATION	The fortresses all have unique attack styles	Pass	All three fortresses have different attacks. A single shot, a spread of three and a spread of five. This means that there is variation in the game making it more enjoyable to play.
NF_RS	NFR_RUNSPEED	The game does not visibly lag as a result of processing	Pass	The game runs at 60fps

Constraint test results:

TEST ID	REQUIREMENT ID	FIT CRITERION	RESULT	EVIDENCE
C_PC	CR_DESKTOP	Must be able to run on a standard desktop PC	Pass	The final release of the game runs well on the computers in the computer science department.
C_Key	CR_CONTROLS	A keyboard is needed to play the game	Pass	Arrow keys can only be found on a keyboard therefore a keyboard is needed to play this game.
C_Legal	CR_COPYRIGHT	No copyright names or	Pass	All graphical components of the game were hand-drawn by team members, so nothing is of

		materials can be inside the game		copyrighted materials. We also made sure that the name of our game wasn't copyrighted either.
C_Time	CR_TIME	Game length must be no longer than ten minutes	Pass	Each team member played the game and it took less than 10 minutes for them to complete the game.
C_Cost	CR_COST	The libraries and software used must be free/open-source	Pass	LibGDX is a free open source software which is what we used to produce this game.
C_Jar	CR_EXEC	Must be able to be compiled into an executable	Pass	The game is in a jar file which can be run on any computer.
C_Java	CR_JAVA	The project must be written in Java	Pass	The game is written in the programming language of java.
C_Deadline	CR_DEADLINE	Each assessment has an unnegotiable deadline.	Pass	Deadlines were set by the computer science department for which we have no control over.