Testing Report

Team 24

Joe Cambridge
Joss Davis
Emily Dennison
Louise Evenden
Amber Hemsley
Josh McWilliam
Lianyu Zhao

a) Briefly summarise your testing method(s) and approach(es), explaining why these are appropriate for the project. (5 marks, ≤ 1 page)
b) Give a brief report on the actual tests, including statistics of what tests were run and what results were achieved, with a clear statement of any tests that are failed by the current implementation. If some tests failed, explain why these do not or cannot be passed and comment on what is needed to enable all tests to be passed. If no tests failed, comment on the completeness and correctness of your tests instead (12 marks, ≤ 3 pages).
c) Provide the precise URLs for the testing material on the website: this material should comprise the testing results and coverage report generated by your automated testing tooling, and descriptions of manual test-cases that you designed to test the parts of the code that could not be covered by your automated tests (5 marks).

a)

We used a combination of manual and automated testing in this project. Our automated tests were aimed at ensuring that codebase changes did not have unintended consequences, as this codebase is very complicated. Small changes or feature additions frequently required changing multiple java classes with poor or no documentation, making it easy to introduce errors. To allow for testing, we generated a fake GL20 library. This allowed us to instantiate the complex mixture of objects used to render the game. Without this library we ran into issues with the headless client which does not have GL20 by default. When the maploader tried to build lights, the game would crash. We tested the application headlessly, focusing on the relationships between objects and the user. We prioritised our testing based on things most likely to cause frustrating bugs for the user - for example, making sure that users can pick up and drop items. This ensures an enjoyable game experience that our customer would be able to either show off at an open day or commercialise into a game.

Our automated tests were carried out using JUnit and Mockito. This is a commonly used testing framework and was suitable for our project as it is well documented and information was available online to help us troubleshoot issues we faced. It also integrated well with Gradle and the test runner in Visual Studio Code, making testing quicker. Gradle generated a series of local HTML files graphically showing our test results, which made sharing results for our assessments on the website easy.

Our manual tests were heavily focused on the gameplay experience. A short playtest was carried out with a list of items to test. We wanted to ensure that users would enjoy playing the game. We tested visual elements that were difficult to test using a headless application, such as lighting and texture mapping. Although we could have found a way to test these headlessly, doing so would have taken too many resources and time and so manual testing was a better option. Manual testing also ensured that we could test the game for any unforeseen bugs our automated testing would not catch, helping ensure a better game.

b) (This can be drastically shortened, explain a few things but not in the level of detail of each test as we currently have)

We ran 30 automated tests and 6 manual tests.

The automated tests are listed below:

Test file and name	Test justification	Test result
testSaveHandler/testSave()	Test the game can be saved	FAILED
testSaveHandler/testLoad()	Test the game can be loaded	PASSED
testAssets/testAssetsPresent()	Tests that all of the required assets are present for the game	PASSED
testPiazzaPanic/test()	Tests the creation of a PiazzaPanic object, to instantiate the game	PASSED
testCookingComponent/testCooking()	Test that food has two stages of processing and changes what food it is after the end of those stages	PASSED
testOverCookingComponent/testOverCooking()	Check foods become spoiled if processing too long (e.g. burnt or overchopped) Note this is very extensive test!	PASSED
testPlayerComponent/instantiatePlayerComponent()	Check if we can instantiate the player component	PASSED
testPlayerComponent/testPlayerComponentAttributes()	Check the default attributes of the testPlayer component initialise correctly	PASSED
testCarryItemsSystem/testPlayerBringsItem()	Tests if an item follows the player it is being carried by.	PASSED
testCustomerAISystem/testReputationDamage()	Test that when an order isn't fulfilled within the time limit the reputation decreases by 1	PASSED
testCustomerAISystem/testMoneyReward()	Test that the player gets money for successfully fulfilling an order.	FAILED
testCustomerAISystem/testAiPathing()	Test that the AI pathfinding works properly	PASSED
testCustomerAISystem/testCustomerTakesOrder()	Test that the customer can take an order successfully, This includes testing that they recieve the item and that the aiobjective for pathfinding changes	PASSED

testCustomerAISystem/testCustomerRejectsOrder()	Test that the Customers can reject incorrect orders	PASSED
testCustomerAISystem/testDifficulty()	Test that gamers can change the difficulty settings.	PASSED
testPlayerInteraction/testPlayerInteractFoodStation()	Test that a player can interact with a Food Station successfully	PASSED
testPlayerInteraction/testPlayerInteractCounter()	Test that a player can pick up or put down food on a counter	PASSED
testPlayerInteraction/testPlayerServingStation()	Test that a plate combines food into a meal	PASSED
testPlayerInventorySystem/testPlayerPickUpItem()	Tests if a player can pick up an item from a countertop	PASSED
testPlayerInventorySystem/testPlayerPickUpOneItem()	Tests if a player can pick up one item of two from a countertop. This also by extension that the order of items on the countertop is preserved.	PASSED
testPlayerInventorySystem/testPlayerPickUpItemInventoryAlmostFull()	Tests if a player can pick up an item with just 1 inventory slot left.	PASSED
testPlayerInventorySystem/testPlayerPickUpItemInventoryFull()	Tests that a player can't pick up an item with a full inventory.	PASSED
testPlayerInventorySystem/testPlayerPickUpItemStationEmpty()	Tests that a player can't pick up an item from an empty station.	PASSED
testPlayerInventorySystem/testPlayerPutDownItem()	Tests if a player can put down an item.	PASSED
testPlayerInventorySystem/testPlayerPutDownOneItem()	Tests if a player puts down just one item when holding more. This also by extension checks that the order of items in the player's inventory is preserved.	PASSED
testPlayerInventorySystem/testPlayerPutDownItemStationAlmostFull()	Tests if a player can put down an item on an almost full station.	PASSED
testPlayerInventorySystem/testPlayerPutDownItemStationFull()	Tests if a player can put down an item on a full station.	PASSED
testPlayerInventorySystem/testChefRememberItems()	Test that a chef remebers the item it is holidng even when not selected	PASSED
testPlayerMovementSystem/PlayerMoveLeftRight()	Test the player can move left and right	PASSED
testPlayerMovementSystem/MoveUpDown()	Test the player can move up and down.	PASSED
powerupTests/testPowerupSpawnRate()	Tests that powerup spawn chance increases with time since last	PASSED

	frame and last spawn	
powerupTests/testPowerupExprire()	Tests that powerups expire after a given timeframe	PASSED

Comment on the completeness and correctness of your tests (and of the failed test).

The manual tests are listed below:

Test file and name	Test justification	Test result
Lighting system accurately updates camera location to player position and shadows are correctly drawn	Results are purely visual and cannot be effectively tested automatically	Passed
HUD disappears when you press H and reappears upon release.	Results are purely visual and cannot be effectively tested automatically with a headless client as we do not instantiate hud on our test client	Passed
Animations play to the user with a conceivably consistent framerate.	Purely performance based and difficult to test in a vacuum without all game systems and textures	Passed
The HUD accurately shows money and reputation	Visual	Passed
Customers queue at and pathfind to objective locations set in the level editor.	Issue relates to accurate translation from editor to world, can only effectively be done manually	Passed
Powerups can be collected via collision with the entities	Difficult too effectively simulate box2d in unit tests	Passed

While all tests passed this was largely a result of the component based architecture inherited from the previous group, however as this architecture made it difficult to test things within the context of the wider game, only in isolation, additionally as a result of ashleys system design many private functions required separate public functions to call those private functions to facilitate testing, this potentially results in a situation where our test assumes states are handled in a set order while the actual order in which events happen

Similarly due to the previous groups architecture it was not practical to actually initialise a full copy of the game for use in testing, instead we focussed on instantiating specific components and systems and manually recreating the the environment, this does allow us to test more consistently, without random or time based mechanics interfering with tests, it does open us up to the possibility of missing a system in our recreation resulting in altered or missing behaviour that would effect the results of the test

Similarly some tests correctly test the immediate result (such as difficulty setting the number that can spawn at once) while not testing the practical result, largely as a result of random chance related mechanics such as powerups and customer group, Additionally the tests that we wrote for testing the save system can conflict as

a result of sharing a file, this can result in the tests failing when they should pass as they create a race condition, this was however deemed to be a non-issue

c)

The testing results can be found here: https://jmm889901.github.io/ENG1/testing.html