Software Testing

Team 7: Bermuda Digital Entertainment

Prajwal Binnamangala
Joseph Sisson
Sebastian Sobczyk
Aymeric Goransson
David Ademola
CJ Donoghue

Introduction

The purpose of this section is to explain the way in which testing has been implemented, organised and designed such that it maximises the amount of code that can be automatically tested.

White/Open box testing

In order to test the various pieces of code, we are using the JUnit library and libGDX's Headless frontend. This should allow us to test any of the game code that does not require rendering. The JUnit test results are output to a html file and, by default, run with every build.

The automatic software tests are separated into the classes they are primarily designed to test. There is also a separate class for checking assets. This leads to the tests being split into the following classes:

- AssetTests - For checking that all assets required by the game are present
- GameObjectTests - For testing the base GameObject Class
- CollegeTests - For testing the College Class (which inherits GameObject)
- PlayerTests - For testing the Player Class (which inherits GameObject)
- SaveTests - For testing the SaveLoad Class
- BoatTests - For testing the Boat Class
- ScreenTests - For testing the GameScreen class.

Within these testing classes, there are both unit and integration tests. The unit tests only check individual sections of code, such as the initialisation of a class, the running of a single method, or the existence of a particular asset.
There are also integration tests that test several methods, sometimes from different classes, in sequence to ensure that the actual communication between them is correct.
None of these test classes are capable of testing every element of the game, given that the render and draw methods, for example, cannot be checked properly by the game running in a headless configuration. Instead, all of the easily testable methods are tested, and anything else will be checked manually via Black box testing.
If a method is inherited by another class (like updateHitBox() from GameObject to Player), it is not tested again, unless it is overridden, or anything related to it's function (like hitBox) has been altered by the subclass.

Black box/ System testing

We are using black box testing to find the gaps in the functionality of the software. It helps in giving an overview of how the software is performing. Using this form of testing helps us to gain the knowledge of the software issues at the users end. We carried out Black box testing after the development of the software to ensure that it will cover all of the requirements and to ensure that the entire game integrates correctly.
Black Box testing also helps to check the sections of code that cannot be properly unit tested in a headless program, such as render and draw methods. This makes black box testing particularly helpful in this project given that it is a game in which many of it's features are dependent on having a GUI and a person looking at it.

White Box Testing

In total 57 tests were run by JUnit, of which 56 passed.

The largest problem in implementing the test framework was that so many parts of the game had the GameScreen class passed to it. In some cases, it was possible to take the required data from the GameScreen individually, and pass it to the methods of other classes without having to pass the entire screen to it (and these sections could be tested).
In other cases, it would have required passing tens of variables to the method individually, which would have most likely resulted in having to rewrite a large amount of the code, for the sole purpose of making unit testing work correctly.

The one test that did fail was testScreenInit. This was a test that instantiated the GameScreen. The reason for it's failure was not because GameScreen (and YorkPirates which was passed to it as a parameter) did not work, but because within the headless testing setup, there is not enough support for the rendering methods.
We attempted to create a 'fake' GameScreen for testing purposes, but many of the methods requiring the GameScreen were heavily dependent on methods of the screen, not just the properties and may not have given a true representation of if the game actually worked in practice.

Test Results can be found at https://bermuda-digital-entertainment.github.io/test/

Black box testing

Below there is a traceability matrix that has 4 columns. The requirements given by the user and software project brief are divided into blocks. Each block has requirements that are associated with each other. Therefore, this represents boundary testing.
The first column represents the test ID. Every test block has a unique test case ID for its identification.
The second column is the software requirement ID. Every block in this column contains the ID's of all related software requirements. These requirement ID's are extracted from the Requirements documentation. All the software and user requirements that can be uniquely identified by the same test case ID are related as well.
The third column is of User requirements ID in which every block has the ID's of all related user requirements.
The last column indicates "Yes" if the tests on each block were successful and "No" if not.

Test case ID	Description	Software requirement ID	User requirement ID	Test successful
1	Software must lod within 30 secs and check if starting screen has a name input field and a start button	FR.START.SCRN FR.START.START FR.START.NAME NFR.LOAD_TIME	UR.START_SCRN UR.SCRN_NAME	Yes
2	Be able to see player ship all the time and move it within the boundary	FR.DISPLAY.SHIP FR.DISPLAY.CAM FR.FREEMOVE FR.DISPLAY.EDGE FR.BOUNDARY	UR.SEE_POS UR.UPDATE_POS	Yes
3	Simple game tutorial for new players	NFR.SIMPLICITY FR.TUTORIAL	UR.TUTORIAL	Yes
4	Player must be able to view other colleges, college info and their ships	FR.DISPLAY.CLG FR.CLG_INFO FR.DISPLAY.DOCK	UR.CLG_POS	Yes
5	Award points and gold after tasks	FR.AWRD.POINTS FR.AWRD.GOLD	UR.COLLECT_PNTS UR.COLLECT_LOOT	Yes
6	Check college health decrementing while being attacked by user	FR.ATTACKCURSOR FR.CLG_HEALTH	UR.ATK_CLG	Yes
7	Check for loot, points and mini map display	FR.DISPLAY.HUD	UR.VIEW_PNTS UR.VIEW_LOOT	Partially (No Mini Map)

8	Check if college is captured when its health is zero	FR.CLG_HEALTH FR.CLG_CONVERT	UR.CPTR_CLG	Yes
9	Check if the tasks are being displayed with increase in difficulty level	FR.OPTNL_TASKS FR.DIFFICULTY_LVL	UR.SEE_TASKS	Yes
10	Check if the user is able to collide and attack enemy colleges, ships and obstacles	FR.CLG_ATTACK FR.COLLISION	UR.ENEMY_SHIP UR.OBSTACLES	Yes
11	Check if the kill screen appears when 1) won, 2) lost, 3) restart or 4) finish the game before returning home. Be able to pause and save.	FR.KILL_SCRN	UR.RESTART_GAME UR.FINISH_GAME UR.SAVE_GAME UR.LOSE_GAME	Yes
12	Check if the shop works fine along with GUI.	FR.DISPLAY.GUI	UR.SHOP	Yes
13	Check if power ups do their job.		UR.POWER_UP	Yes
14	Test the game on 13” and 47” display	CR.RESOLUTION		Partial (Tested on 15” display)
15	Game must be playable using UK standard mouse and keyboard and on a machine with 4GB and higher specs	CR.LOW_SPEC		Yes
16	Gameplay max 10 min and stable throughout	NFR.GAME_TIME NFR.STABLE		Yes