Group 14 — Phase 3 Report
Raccoon Simulator

Alex Impert, Daniel Todd,
Ryan Wittkopf

April 2022

1 Introduction

The purpose of this report is to chronicle the testing phase of our game Raccoon Simulator.

2 Findings: Subject & Player

Testing subject, player and enemy (player and enemy inherit from subject) meant testing what in game is essentially the player character, and the enemies that target them. This primarily came down to movement updates and checking expected val.

1. **Subject**

   createSubject() This is just a test for our subject constructor. As our game *requires* a player, our test simply just asserts whether player is null or not.

   update() This function calls our more specific updates such as direction and movement. However it also contains branching for sprite animation flipping (based off the direction the user goes). As such, we used the Java Robot library to simulate key presses for movement and then checked to make sure the proper sprites were being loaded.

2. **Player**

   createPlayer() This constructor was tested similarly to that of Subject. However, player contains two members that dictate whether the character is moving and/or has escaped. Therefore we made sure these were false at construction as the player should not be moving and have escaped immediately.

   customUpdate() This function calls a timer update to tick down our clock. Additionally, it contains a condition that ends the game if this clock runs out. As such the test immediately drops the clock to -10, and the game expectantly ends.

   moveUpdate() The move update method handles the actual logic for moving a player on the screen by adjusting the character's position. To get branch coverage we compared the positions of the player with the position prior to a keypress from the robot. Additionally, a condition that checks if the player has stopped moving once the robot isn't simulating any keypresses.

   directionUpdate() The direction update function sets the player to *moving* and determines the direction. Since we know from the previous test that the direction switch works, we opted to instead to test when the conditionals *don't pass* as a big component of this method is making sure the player can only move on floor tiles (aka collision).

   collectObject(int index) The collect object method processes retrieval of items (including the trap.) As the function contains two conditions we opted on making the method return a boolean based off where the condition leads for maximum line coverage. After that, we had the player essentially walk over manually spawned items as a way to integration test how the character interacts with their environment. We checked to see if the object existed in our object list after retrieval (it should expectantly be deleted in its array).

3. **Enemy**

   Enemy() This constructor was also tested similarly to that of subject. However, here we also reference our list of enemies, making sure it is not null. We also test that the location of an enemy is correct.

   directionUpdate()

3 Findings: Managers & Loaders

Testing the management and loading classes differed a bit depending on the method but mostly involved checking that the core methods for these classes such as loadMap() for the mapLoader class.

1. **RaccoonGame**

   RaccoonGame() Testing of the main game class (RaccoonGame) led me to refactor the class quite a bit, including a major renovation of introducing a setupGame() method. While doing line coverage testing I noticed that there was a bit of an issue of how we initialized all of RaccoonGame's attributes in the constructor class and also initializing a JFrame in Main() instead of in RaccoonGame. I reworked the class so that initialization of a game consists of 4 method calls, 3 of which are executed in RaccoonGame's construction: setupScreen(), setupGame(), startGame(), and startThread().

   setupGame() Creating the setupGame() method allowed for a clean initialization of all of RaccoonGame's attributes. During testing I noticed that it was very difficult to locate how some of RaccoonGame's attributes were being initialized, so I brought them all into one clean method.

2. **Thread Management**

   doTick() During my testing of methods such as KeyHandler I noticed that we had a major issue with our testing structure. Every test required us to use a java.awt Robot to mimic keypresses and instantiate a new Thread. This caused a huge performance hit and even caused the tests to fail on different computers because the Robot's delay method caused our testing to be system dependent. The delay method was based off of milliseconds. But ideally we want our tests to be based off in-game ticks. I managed to repurpose the run() method from our RaccoonGame class into a new doTick() method in the ThreadManager class (a class designed for test-specific methods) so that we could use ticks in place of milliseconds for delays when testing.

3. **Managers**

   objectHandler() While attempting to test some methods from objectHandler() and I noticed that there was some major room for improvement to make this class more Object-Oriented-Friendly. I had initally been calling a lot of methods from the constructor of objectHandler() that should not have been called there. During my testing I noticed that this went against the flow of Object-Oriented programming because it was very difficult for me to properly unit test some features without calling in many other classes. This was a case of high cohesion and I managed to fix this by abstracting the constructor and calling these methods when objectHandler() is intialized. This works much better than the previous implementation.

4 Findings: Path Finding

Testing of enemy path finding algorithms, which are based off text documentation of the game map. This mainly consists of classes GraphMaker and TreeMaker. Due to the fact these classes are implementations of abstract data types, the nature of the tests are focused on expected values and behaviors.

1. **GraphMaker** The only variable stored in GraphMakerTest is a copy of RaccoonGame, so that it is accessible by all methods, as we do not need to modify the game instance.

   Node() Seeing as Node is simply a inner class of GraphMaker to store data about each block, testing for Node was done via integration tests. Almost every tested method in GraphMaker and TreeMaker additionally tests Node. Node never presented any issues.

GraphMaker() For my testing of the GraphMaker constructor, I made use of assertions to check that each variable had been assigned, and as such was not null. There were no issues here.

graphGenerate() & print() I tested graphGenerate() and print() together as all print does is scan through the map created by graphGenerate(). Any attempt to test graphGenerate() individually would be a waste of time, as it would have to implement the same logic as print(). As such I tested both together as an integration test. The inital issue was that print() would print to the terminal, and to test this more effectively, I refactored the code to instead return a string. After resolving the return type issue, I had issues with the formatting that the string would return, after long tedious work, I realized the issue was a format error in my reference string that was asserted against.

graphDirectionFill() Here I iterate through the map returned by our constructor, GraphMaker(), and check that each direction (up, down, left, and right) are correctly instantiated and assigned for every node. There were no issues here.

find() For the find method, I first iterate through our map generated by the graphGenerate() method. I then look for the current column and current row in the text file, and make sure that the returned Node has the correct coordinates. The only issue here was a mistake in test logic, where I forgot that indices in Java start at 0, and as such I was getting a failed assertion for a while until I figured it out.

2. **TreeMaker** The only variable stored in TreeMakerTest is a copy of RaccoonGame, so that it is accessible by all methods. This variable is changed each class, but the use of a testFactory makes keeping the variable as part of the class a viable option for saving memory.

TreeMaker() For the TreeMaker() constructor I test that each variable is correctly assigned. Not much more can be done for the constructor, as it is very abstract to allow for flexible adaptation. There were no issues with this test.

update() For the update() method, I test a path generated for each enemy at a specified location. First we test that the enemies' coordinates are correctly assigned as the root of the tree, as well as testing that the player's coordinates are correctly assigned as the playerNode of the tree. I also test that root, playerNode, and the path generated by the tree are all correctly instantiated. The issue here was the enemy would kill the player (by following the generated path) and crash the game before the test could complete. I fixed this by setting the enemy's speed to 0 in the testFactory().

print() & BFS() The print method is tested by iterating through all of the enemies within the game, and checking that their paths to the player are all correctly printed. Notice how this is also an integration test for BFS(), which is our adaptation of the breadth first search algorithm. This test had one big issue, where the enemies were not generating a path due to not being within range of the player. I fixed this by forcing each enemy to make a path from where they spawn. I decided to do this instead of changing the enemy spawn locations, as where they stand now, there are loads of obstacles between them and the player, as well as the fact the enemies are already evenly spread out. After this, there were no other issues with testing print() and BFS().

blockUpdate() To test blockUpdate, I simply called the testFactory, and checked that each enemy's coordinates were correctly rounded to the nearest block, as well as performing this same check on the player's coordinates. There were no issues here.