Phase 3 Report

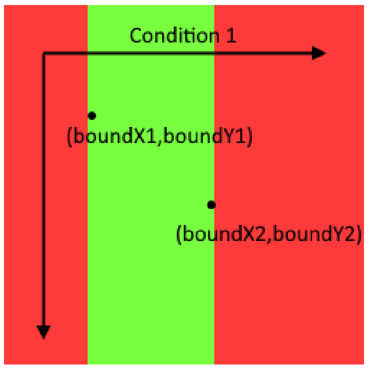
For this report, we chose to omit some explanations of smaller Junit tests, since they are generally pretty self explanatory. Instead, we try to focus on the discussion and explanation of how we designed specific tests for what we deemed to be the most important features of the game.

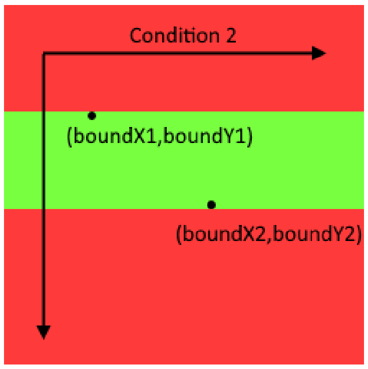
Collisions Between Characters, Enemies, Projectiles, and StaticEntities

Let's start our discussion with one of the most integral components of the game, collision. Although not as important as displaying the game, poor collision results in awkward play, so it is a feature we've identified as important and subjected to rigorous testing.

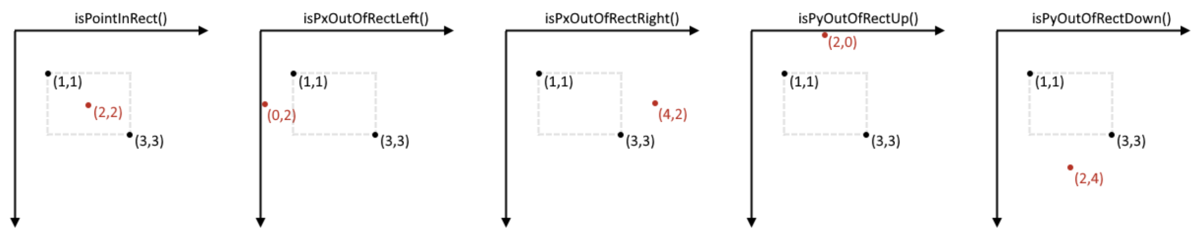
In the collision class there is a helper method named isPointInSquare (I'm aware is should be rectangle), that is utilised by all the other methods in the class. Although this method was initially private, it's so essential to the functionality of the other methods, that verifying that it is functioning as intended, significantly reduces the amount of work we must while testing the other methods in this class. Thus, we've made the decision to change it to public so it could be tested appropriately. Essentially, what this method does is, given two integer coordinates pX and pY, it returns true if this point is in a rectangle given by the integer bounds boundX1, boundY1, boundX2 and boundY2. In order to determine how this is done, we'll briefly provide some context to how coordinates are represented in our game. If we consider the point (x,y) in the plane, as we move to the right, the value of x will increase, and as we move down the value of y will increase.

isPointInSquare checks two conditions first it checks if the point pX, is within the bounds boundX1 and boundX2. If it is, then it checks if the point pY is within the bounds boundY1 and boundY2. If both conditions (which we will refer to as condition 1 and condition 2 respectively) are met, then the method returns true. We can illustrate each of these conditions for two arbitrary points as such,





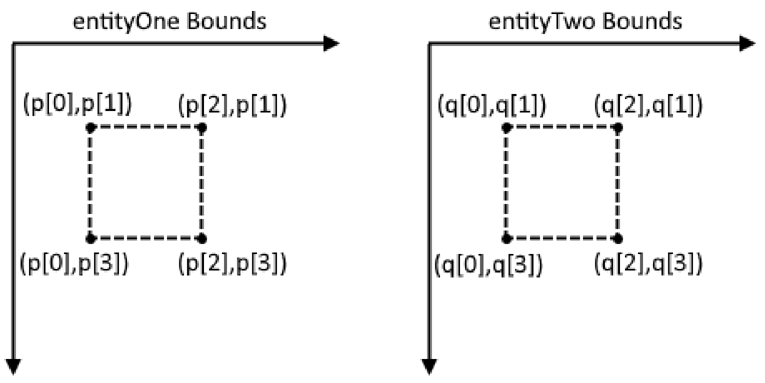
Where the green area represents the range of values of pX and pY for which condition 1 and condition 2 respectively will evaluate to true. We can easily see, that there are 3 cases each condition. Testing each combination of each of these cases would result in 9 separate test cases, which is a little excessive. So, instead of testing each nine, we divided the testing of this method into five general separate unit tests, isPointInRect(), isPxOutOfRectLeft(), isPxOutOfRectRight(), isPyOutOfRectUp(), isPyOutOfRectDown(). Each of these methods uses the same bounds to test pX and pY, boundX1 = 1, boundY1 = 1, boundX2 = 3, boundY2 = 3. The values used for pX and pY tested with each of these methods are shown in the diagrams below where the red point represents the point being tested.



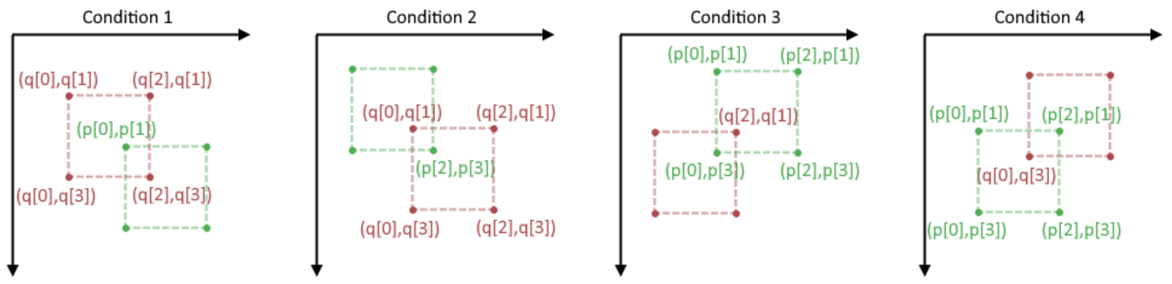
Running mvn test, we see all unit test are successful.

Although we didn't test each case for the prior method, let's just assume that it's running correctly. We have one other method in the class that checks if two objects are colliding (technically it's four separate methods overloaded, but they all function in the exact same way. So, without loss of generality, we will delegate the explanation to the first method which handles collisions between two entities).

The method, isColliding(), relies on four methods from the entity class (getHeight(), getWidth(), getX() and getY()) and one helper method (isPointInSquare()). The functionality of each method from entity will be omitted for the sake of brevity. The implementation of this function is a bit disorganized (especially in how the components are named), but I will explain the testing process using a more consistent naming scheme. First, the function creates two arrays of four elements, p and q, where each component defines the bounds for the entity as such,



Using our helper method, isPointInSqaure, we know that two entities are colliding if any point from entityOne is in the bounds defined by entityTwo (or vice versa). So, the method looks for the following four conditions to see if two entities are colliding, demonstrated in the diagrams below where the labeled vertices are the ones being tested in isPointInSquare (green = entityOne, red = entityTwo),



If any of the above conditions are met, the method will return true. Thus, when designing test to make sure all these conditions appropriately cause the method to return true, we artificially model each situation using the main character as entityOne, and an instance of enemy2 as entityTwo, and positioning them to match each condition, while accommodating for their width and height. A test is also included that verifies when none of the conditions are true then the method turns false. As you can see, all test pass.

Additionally, integration tests are performed between different types of entities in the class TestCollisionInteractions.java, to make sure the logic is behaving correctly when two entities collide with each other.

Images and Correctly Animating Entities While Moving

Having your graphics display correctly, is arguably the most important aspect of the game. Despite how well your game functions logically, it means nothing if your graphics aren't being displayed correctly. And while not essential to the function of the game, animation adds visual flare that enhances the quality of the game.

The initial challenge in developing the test code for images, was figuring out a way to compare two images for equality (i.e., if they are actually the same image). Although we can look at the screen and see if our image is being displayed correctly, when you perform an animation that is updating multiple times per second, issues can be imperceptible if you're not looking for them. So, it's important that we have a way to verify that what we are seeing, is actually what is supposed to happen. Thus, we added a new method equals(Image image) to Image.java designed specifically to aid in testing for equality between images. Although it's evident to humans if two images are the same, it's not so easy for computers. Essentially what equals() does is, given two images, scan each pixel, compare their RGB values, if they all match return true. The method does allow for a slight margin of error (e.g., if pixel1 has RGB values (142,122,254) and pixel2 has RGB values (141,122,255) we say they are identical), due to the fact that the method getSubImage() of the BufferedImage class can slightly modify certain RGB values. We subject equals to unit testing in ImageTest.java, where we test 3 conditions, testing two identical images, testing two images that are slightly similar, but distinct enough to be considered different, and two images that are different sizes. Since all test conditions pass, we felt comfortable enough in the validity of this method, to use it in testing for our animator class.

When testing animation, we will use an instance of entity. Basically, when an entitity is initialized, we need to give it a path. There should be four files (there are technically five, but the function of the fifth is not explained in the report) associated with this path, "path"+Down.png, "path"+Up.png, "path"+Left.png, "path"+Right.png. These images each contain four frames of the character being animated in a specific direction. Since each of the four directional images are processed in the exact same manner, let's delegate the discussion to "path"+Down.png.

In explaning the test code, we initialize an instance of Entity using the image path, "src/main/resources/character/mainCharacter", so when explaning tests pertaining to "path"+Down.png, we are referring to the file "mainCharacterDown.png" in the folder "src/main/resources/character/". Using mainCharacterDown.png we create an instance of the Animator class. The purpose of the Animator class is self explanatory, an animated image will return a different frame of the image based on the value of an internal counter "counter". Each frame of an image is divided into four subsections of mainCharacterDown.png, so the function getImage(boolean isMoving) will return one of the four frames of the image based on the value of counter. The value of counter will not change if the parameter isMoving is false, so if an entity is still, there image will just be a still frame. There is also another counter "subCounter" within the method, to delay counter from being increased every time getImage is called (If counter was updated everytime getImage was called, the image would be updated 60 times per second, which does not look smooth at all, instead it is updated 6 times a second, which looks a lot smoother).

Counter= 0
1
2
3

The value of counter updates every tenth time getImage() is called, so in order to artificially test the method, we call getImage() 0 times for frame 0, 11 times for frame 1, 21 times for frame 2, and 31 times for frame 3. Then, using the equals(Image image) method we described earlier, we are able to compare each frame to an actual single image of each frame. Initially all of these test output errors, we determined it was due to the way the program was subdividing the images. We fixed the error and now the tests are all functional.

Some Lighter Explanations of Smaller Tests

Let's now change the direction of the discussion to just briefly describe some of the lighter tests we perform. In the EntityTest.java we test to make sure that an entity is appropriate given an image for each possible direction it can face. We also test to make sure when an enemy is hit, it's graphic will briefly change to a frame where the enemy is red, we verify this using a similar method as discussed above.

I (**Rohan**) worked on testing all the enemy classes. I wrote 2 tests for enemy 2 to make sure that the enemy was moving and updating properly for the game to work. I wrote 3 tests for the Enemy list class to make sure that when spawning an enemy the push(), remove(), update() and IsEmpty() methods are working fine to make sure there is an existence of that enemy in the game. I wrote two codes for the Enemy class to make sure that the kill(); and hit(); methods are working fine so that the enemy is in fact getting erased when killed and the losing health once hit by the characters bullet. All the tests passed, and all the methods are working as needed for the game to perform properly.

My (**Alvin**) coding parts including Assets, Image, SpriteSheet, Map, staticEntity,Display, Loading, Game, GameState, Sate, floorTile, Tile, wallTile and Start. I wrote 5 test code use to test function: getImage( ), imageLoading( ), switchToString( ), switchToInt( ) and isSolid( ). The function imageLoading( ) is used to create an object can contain a picture and getImage( ) can return the picture contain with the object. I test these two functions because if these two function fail the game will not show anything after users click the start button. SwitchToString( ) and SwitchToInt( ) are use to loading the map information. If these two function fail there will only shows an empty window on the screen. isSolid( )is used to determine which tile can be passed, if this not work the character and monsters' moving area are not limited. I also add "/ MavenBuild " before all the resources path. Because we move all the stuff under MavenBuild folder. If I don't add that all the function need to use resource will not work on my computer.

Conclusion

We apologize if our Phase-3 was a little vague in some areas, and over specific in others. The reality is we’re all pretty new to testing, and this phase was less cut-and-dry than phase 2. We didn’t really know where to start or stop, and lack of face-to-face communication definitely inhibited our ability to communicate effectively. Speaking more personally I (**Jordan H.**), really underestimated the amount of time it can take to make test functions that are actually meaningful. While I feel like we covered a pretty substantial number of tests, there was a lot of functionality I would have liked to test further. Thanks for reading.