COMP20050 - Group 47 - Features
Sprint 3

Student Numbers: 22441636, 22450456, 22715709

In this sprint, we wanted to further develop our ray behaviour and implement all possible movement to our game logic. We also begin bringing the “game” aspect to our implementation through use of tracking score, announcing winner etc. While not originally forecasted in our project plan, we have now incorporated a single player game mode to our game. While the current implementation is strictly through the command line, our final goal for the next sprint is to translate our game to a graphical user interface to bring a better user experience however as of now all game logic and the “model” side of our project has been completed but we will continue to test and ensure that further improvement/development do not affect previous working code.

(A revised project plan has been included in this submission)

Implemented Features:

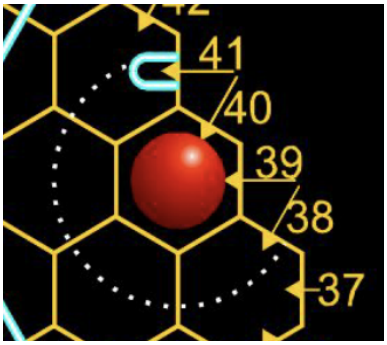
Ray:

While our previous ray implementation could handle all 60 degree deflections, this sprint we developed handling 120 degree deflections and also reflections for both cases where a ray encounters a circle of influence at the edge of the board and also when a ray encounters intersecting circles of influence in a certain position. While previously, the code for handling deflections was located in our board class, we migrated the code to within the ray class itself so with one argument (the orientation of circle of influence or the orientations of the intersecting circles of influence) a ray can handle its only deflection and figure out what its orientation should be after it encounters a circle of influence or intersecting circles of influence.

```
if (board[y][x] == null || board[y][x] instanceof RayTrail) {
    board[y][x] = new RayTrail(orientation);
    return false;
}
else if(board[y][x] instanceof CircleOfInfluence c){
    return r.deflectionLogic_CircleOfInfluence(c.getOrientation());
}
else if(board[y][x] instanceof IntersectingCircleOfInfluence i) {
    return r.deflectionLogic_IntersectingCircleOfInfluence(i);
}
return false;
```

When a ray moves along the board, it will check the contents of the hexagon. As seen above in the first if statement, no encounters just leads to a ray trail being placed on the board. Encounters with a circle of influence lead to the ray call its class method using the circle of influences orientation and finally if an intersecting circle of

influence if encountered it will execute a similar class method which too will result in the ray changing its class variable - “orientation”.
Our previous sprint had fully implemented deflections for 60 degrees ie. when a ray encounters a single circle of influence however in this sprint we also implemented the feature of a ray getting deflected when it hits a circle of influence at the edge of the board.



For this example, the ray enters at 41 however also exits at 41. Same would be true for a ray entering at 38 as it would also exit at 38.

```
Ray entered at 41
Reflected!
      - - - - - -
    - x x x x x -
  - x x x x x x -
- x x x x x x / +
 - x x x x x x | o -
- x x x x x x x \ / -
 - x x x x x x x x -
  - x x x x x x x -
    - x x x x x -
      - - - - - -
```

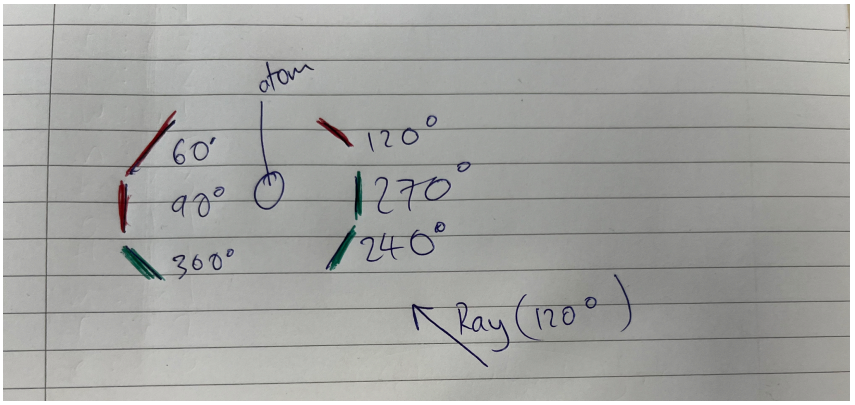
This is the same example but printed onto our temporary command line board view. The ray enters at 41 and then immediately gets reflected and exits at 41. As this is type “reflection” a purple ray marker is placed onto the board.

This behaviour also works for all other possible combinations of atom placements and ray entrance points as seen below with the another example in an atom being placed on the bottom left side of the board and a ray entering at 16 and immediately being reflected.

```
Ray entered at 16
Reflected!
      - - - - - -
    - x x x x x -
  - x x x x x x -
- x x x x x x x -
 - x x x x x x x -
- x x x x x x x x -
 - / \ x x x x x -
  - o | x x x x -
    + / x x x x -
      - x x x x -
        - - - - -
```

This feature was implemented in our `deflectionLogic_CircleOfInfluence` method which would figure out if the atom encountering a circle of influence should be reflected. To work this out we found that for each ray travelling in a certain orientation, it would have two corresponding circles of influence orientations which would cause it to be reflected. For the standard circle of influence reflection, a ray has 3 corresponding possible circles of influence orientations that it can hit - 2 for ±60 degree deflection and 1 for an absorption. This leaves 3 remaining circles of influence orientations which are impossible to hit as seen below. Each line is a “part”

of an entire circle of influence or what we simply call a circle of influence in our project. Green line represents a possible interaction and red represents impossible interaction (in normal case - when atom is not located at the edge of the board)



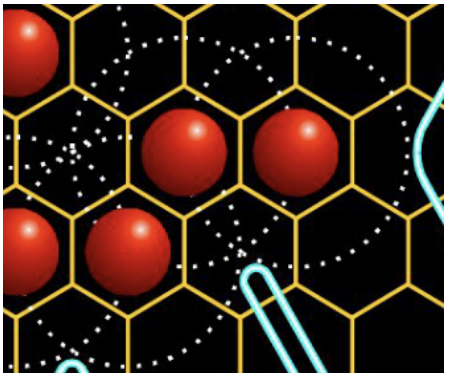
Ray travelling at 120 degree orientation cannot hit a circle of influence with orientations (90, 60 or 120). Thus we can deduce if it hits one of these usually “impossible” to hit orientations (90 or 120), the ray should be reflected. We found a general case for finding this out eg. a ray travelling with orientation 120 can never hit a circle of influence with the same orientation - 120 as seen in photo illustration above. Each ray also has one other possible orientation of a circle of influence that it cannot hit that results in reflection - for 120 degree ray, 90 is the other possibility. We then mapped these cases in our code which result in a ray being reflected when encountering a single circle of influence.

```java
private boolean reflection(int coiOrientation) {
    if (this.orientation == coiOrientation) return true;

    return switch (this.orientation) {
        case 0 -> coiOrientation == 120 || coiOrientation == 240;
        case 60, 300 -> coiOrientation == 270;
        case 120, 240 -> coiOrientation == 90;
        case 180 -> coiOrientation == 60 || coiOrientation == 300;
        default -> false;
    };
}
```

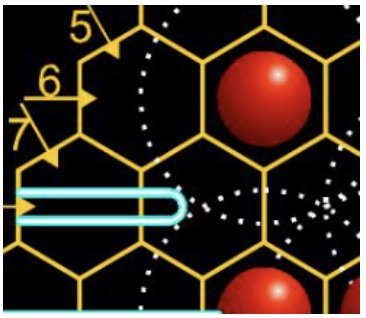
Above is the function which returns true if a ray should be immediately reflected and if returns true, we call another function; flipOrientation() which will simply add or subtract 180 from the ray's current orientation which also became useful in handling intersecting circles of influence reflections.

For handling deflections and reflections caused by intersecting circles of influence, we found that based on the rays orientation and whether or not certain circles of influence orientations were included in the list of orientations contained in a single hexagon, we could figure out what the ray should do. We divided the fate of a ray

when encountering intersecting circles of influence into two groups, a reflection or 120 degree deflection.



While this photo shows a reflection where circles of influence intersect, it does not mean that 3 intersecting circles of influence result in a reflection as seen below where 2 circles of influence also produce a reflection.



We found that even if a third part of the circle of influence was added to the second photo, the outcome would not change - leading us to conclude that the most important pieces of the circle of influence were the ones created by the outer atoms (not the atom between). This led us to check the array list of circles of influence for just two distinct numbers which allow us to determine whether or not a ray should be reflected. We have two class methods in our IntersectingCircleOfInfluence class called, horizontalReflection() and diagonalReflection() - diagonal referring to ray orientations 60, 120, 240 and 300 and horizontal referring to 0 and 180. For “horizontal” reflections we found that when two circles of influence orientations sum to make 360, a reflection should occur. For example in the above second image, the top atom bottom left circle of influence has orientation 300 while the bottom atoms top left has orientation 60 which sum to 360. This works on the opposite side also as seen below.

Orientations in this instance being 120 and 240, also summing to 360.

To handle diagonal reflections, we found that certain combinations of numbers result in a reflection.

```
public boolean diagonalReflection(){
    return (orientations.contains(270) && (orientations.contains(300) || orientations.contains(60)))
            || (orientations.contains(90) && (orientations.contains(240) || orientations.contains(120)));
}
```

As there are only four possible combinations of a ray travelling diagonally and getting reflected and that it is only possible for it to encounter 1 combination of orientations

we did not have to also check the rays orientation only the circles of influence it encounters. For example below, two rays are travelling diagonally and as when they hit the intersecting circles of influence and for each case, true is returned by our diagonalReflection() method, both orientations get flipped.

To handle the 120 degree reflections in the case the ray is travelling horizontally (0 or 180 orientation), we only had to figure out whether the ray should plus or minus 120 from its orientation as our above code has already figured out whether the ray should be reflected or not, and if not, means it must have a 120 degree reflection (as it is encountering an intersecting circle of influence).

```
if (this.getOrientation() == 180 && intersectingCircleOfInfluence.getOrientations().contains(120)) {
    minus120();
} else if (this.getOrientation() == 180 && intersectingCircleOfInfluence.getOrientations().contains(240)) {
    plus120();
} else if (this.getOrientation() == 0 && intersectingCircleOfInfluence.getOrientations().contains(60)) {
    plus120();
} else if (this.getOrientation() == 0 && intersectingCircleOfInfluence.getOrientations().contains(300)) {
    minus120();
}
```

We found the above screenshot of code will correctly update the rays orientation through checking the rays orientation and checking what kind of orientations are included in the intersecting circles of influence.

To handle the case of diagonally travelling rays, we found it important to find out what combination of orientations exist in the arraylist of circle of influences

If they did not include what we call 90 degrees and 270 degrees, through a simple mathematical equation and the fact that the rays' new orientation will be one of the two intersecting circles of influence orientation, we could easily find out the new ray orientation. In the case of this screenshot, the top ray initially enters from the left and has orientation 300, when it hits the two circles of influence/intersection, its new orientation would be off the part which is not ±180 of the ray's current orientation. In this case, the two included parts have orientations 120 and 60, belonging to the left and right atom respectively. As 120 + 180 is equal to the current orientation of 300,

we set the rays new orientation to that of the complimenting piece of circle of influence, in this case, 60 degrees. This method effectively calculates the new orientation for all combinations but works on the assumption that the list of circles of influence is of size two which in this board game works as if there is no possible combination of three circles of influence which result in a 120 degree deflection as all combinations lead to a reflection.

```
if (this.getOrientation() == 120) {
    minus120();
} else if (this.getOrientation() == 60) {
    plus120();
} else if (this.getOrientation() == 300) {
    minus120();
} else if (this.getOrientation() == 240) {
    plus120();
}
```

To finally handle the possibility of the list of circles of influence including 90 or 270, this screenshot shows how we effectively calculated the new orientation as this is the final possibility of what could happen as we have previously accounted for every other possibility.

Example of ray experiencing 120 degree deflection

Example of using the new complex ray logic is included in the testing section of this document.

Implementing the game:

In this project we prioritised initially creating all of the games logic and ensuring that all features were included. In previous sprints we included a “controller” aspect in the project to allow for user input however it did not fully simulate the final game with two players both utilising both roles and eventually deducing a winner. In this sprint we worked on allowing for a setter to place atoms, letting the experimenter send rays and then eventually guessing the location of the atoms. We then replayed the exact same steps but having swapped the roles of both players.

```
public void guessAtom(int x, int y) {
    if (!(board.getBoardPosition(x, y) instanceof Atom)) getExperimenter().updateScore( n: 5);
    else getExperimenter().correctAtom();
}
```

This screenshot shows how if an atom is correctly guessed, the experimenter will update their number of correctly guessed atoms and if incorrectly guessed, 5 will be added to their score. We also keep track of a player's previously guessed atoms and do not allow for them to guess atoms in the same position.

In updating the score with every sent ray, we similarly have a function which will find out if a ray exits the board, if so adding 2 to the experimenter's score and otherwise only adding 1 (absorption).

As previously implemented functions of sending rays return true if a ray is absorbed, this easily lets us update the score accordingly.

```
public void sendRay(int input) {
    if (board.sendRay(input)) {
        getExperimenter().updateScore( n: 1);
    }
    else {
        getExperimenter().updateScore( n: 2);
    }
}
```

View:

In creating the game part of our project, we also added new methods to our view class (command line interface for now) which at the beginning will display options to the user of what game mode to play

```
================================================================================
Welcome to Black Box Plus by Cian, Lloyd and Shlok (demo version v3 - (terminal))
================================================================================
Please input a number to select an option from the list below:
--------------------------------------------------------------------------------
1. Play single player game
2. Play 2 player game
3. Quit
```

Above is the main menu (which will eventually be translated to a GUI)

```
============================================================
Cian won!
============================================================
Final scores - Cian: 1 | Lloyd: 2
============================================================
Statistics:
************************************************************
Player 1 - Cian | 1 rays sent | 1 correctly guessed atoms
------------------------------------------------------------
Player 2 - Lloyd | 1 rays sent | 1 correctly guessed atoms
------------------------------------------------------------
```

Above is a mock end game screen which displays the statistics of the game and announces a winner.

As mentioned above, a single player game mode was also added to our implementation. For now, simply 6 randomly selected atoms are placed on the board and the player sends rays and attempts to locate all 6 atoms, similarly to the two player game however in this implementation, only the setters functionality is implemented only allowing for a user to be the experimenter. While it is not included in our project plan, we may still experiment in bringing a full game to the single player mode, having the player also able to play as setter and in some form simulate the role of the experimenter through code. As of now the single player mode will end

after 1 round and displays the players stats and how many rays/correctly guessed atoms they had.

Controller:

As we are using the command line to take user input, it can still be quite unreliable and produces unknown results, eg. new line characters not being registered. While for now this does not affect the running of the project significantly we have not explicitly handled the scarce occurrences of buffer errors. This is as we have previously mentioned that we are hoping to fully develop an interactive GUI and taking input from the command line is not going to be included in our final implementation.

(if when entering input and after pressing the enter key, a new line appears in the command line and input is not registered, please press backspace and hit enter again - this will resolve issue for current implementation)

Testing:

As mentioned in our project plan, all features we specified that we would implement were tested for.

120 Degree Deflections:

Firstly, the 120-degree deflection was tested for. Six atoms were placed on the board and rays were sent, striking at an angle of 120 degrees. The expected inputs and outputs were compared for each case as well as the expected deflection type.

Below are two screenshots of the testing the 120 degree reflection from all possible angles of ray orientations. Below screenshots of the code also include what the board looks like in order to help visualise the deflections.

```
@Test
void test120deflection() {
    Board b = new Board();
    GameView view = new GameView(b);
    b.placeAtom( x: 3, y: 5);
    b.placeAtom( x: 4, y: 5);
    b.placeAtom( x: 5, y: 4);
    b.placeAtom( x: 6, y: 4);
    b.placeAtom( x: 6, y: 6);
    b.placeAtom( x: 7, y: 6);

    b.sendRay( input: 24);
    b.sendRay( input: 53);
    b.sendRay( input: 30);

    Ray ray = b.getSentRays().get(0);
    assertEquals(ray.getInput(),  actual: 24);
    assertEquals(ray.getOutput(),  actual: 17);
    assertEquals(ray.getDeflectionType(), actual: 120);

    Ray ray2 = b.getSentRays().get(1);
    assertEquals(ray2.getInput(),  actual: 53);
    assertEquals(ray2.getOutput(),  actual: 48);
    assertEquals(ray2.getDeflectionType(), actual: 120);

    Ray ray3 = b.getSentRays().get(2);
    assertEquals(ray3.getInput(),  actual: 30);
    assertEquals(ray3.getOutput(),  actual: 25);
    assertEquals(ray3.getDeflectionType(), actual: 120);
    view.printEntireBoard();

}
```

```
@Test
void test120deflection2() {
    Board b = new Board();
    GameView view = new GameView(b);

    b.placeAtom( x: 6, y: 3);
    b.placeAtom( x: 5, y: 5);
    b.placeAtom( x: 5, y: 4);
    b.placeAtom( x: 6, y: 4);

    b.sendRay( input: 37);
    b.sendRay( input: 17);
    b.sendRay( input: 6);
    b.sendRay( input: 46);

    Ray ray = b.getSentRays().get(0);
    assertEquals(ray.getInput(),  actual: 37);
    assertEquals(ray.getOutput(),  actual: 30);
    assertEquals(ray.getDeflectionType(), actual: 120);

    Ray ray2 = b.getSentRays().get(1);
    assertEquals(ray2.getInput(),  actual: 17);
    assertEquals(ray2.getOutput(),  actual: 10);
    assertEquals(ray2.getDeflectionType(), actual: 120);

    Ray ray3 = b.getSentRays().get(2);
    assertEquals(ray3.getInput(),  actual: 6);
    assertEquals(ray3.getOutput(),  actual: 1);
    assertEquals(ray3.getDeflectionType(), actual: 120);
```

The above tests helped us conclude that for all possible directions and combinations of intersecting circles of influence, ray will behave as required.

180 Degree Reflection:

The next instance of ray deflection tested for was the 180-degree reflection, occurring when the ray encounters two atoms in certain positions which produce a specific intersecting circles of influence. Again, the atoms placed had rays fired at them, with input, output and deflection type values compared.

```
@Test
void test180deflection() {
    Board b = new Board();
    GameView view = new GameView(b);

    b.placeAtom( x: 9, y: 1);
    b.placeAtom( x: 5, y: 2);
    b.placeAtom( x: 6, y: 5);
    b.placeAtom( x: 4, y: 4);
    b.placeAtom( x: 1, y: 9);
    b.placeAtom( x: 5, y: 7);

    b.sendRay( input: 12);
    b.sendRay( input: 6);
    b.sendRay( input: 35);
    b.sendRay( input: 41);

    Ray ray = b.getSentRays().get(0);
    assertEquals(ray.getInput(), actual: 12);
    assertEquals(ray.getOutput(), actual: 12);
    assertEquals(ray.getDeflectionType(), actual: 180);

    Ray ray2 = b.getSentRays().get(1);
    assertEquals(ray2.getInput(), actual: 6);
    assertEquals(ray2.getOutput(), actual: 6);
    assertEquals(ray2.getDeflectionType(), actual: 180);

    Ray ray3 = b.getSentRays().get(2);
    assertEquals(ray3.getInput(), actual: 35);
    assertEquals(ray3.getOutput(), actual: 35);
    assertEquals(ray3.getDeflectionType(), actual: 180);
```

Once again included is the program output with the specified values used in the unit test. This time, the purple ray indicates the reflection.

```
@Test
void test180deflection2() {
    Board b = new Board();
    GameView view = new GameView(b);

    b.placeAtom( x: 5, y: 4);
    b.placeAtom( x: 3, y: 5);
    b.placeAtom( x: 6, y: 5);
    b.placeAtom( x: 3, y: 8);
    b.placeAtom( x: 2, y: 7);
    b.placeAtom( x: 5, y: 7);


    b.sendRay( input: 19);
    b.sendRay( input: 26);
    b.sendRay( input: 46);
    b.sendRay( input: 3);

    Ray ray = b.getSentRays().get(0);
    assertEquals(ray.getInput(), actual: 19);
    assertEquals(ray.getOutput(), actual: 19);
    assertEquals(ray.getDeflectionType(), actual: 180);

    Ray ray2 = b.getSentRays().get(1);
    assertEquals(ray2.getInput(), actual: 26);
    assertEquals(ray2.getOutput(), actual: 26);
    assertEquals(ray2.getDeflectionType(), actual: 180);

    Ray ray3 = b.getSentRays().get(2);
    assertEquals(ray3.getInput(), actual: 46);
    assertEquals(ray3.getOutput(), actual: 46);
```

Above are two examples of rays being reflected by intersecting circles of influence, which include all possible combinations. By performing these tests we can see that for every combination of ray orientations and intersecting circles of influence, our ray will behave as expected.

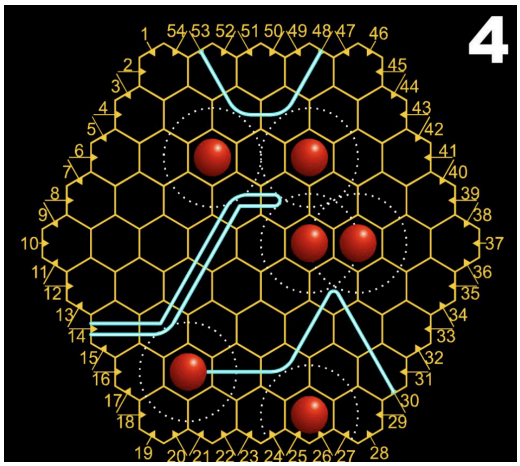
Reflected atom (edge of board):

The case where an atom is placed at the edge of the box had to be tested for.

```
@Test
void testAtomEdge() {
    Board b = new Board();
    GameView view = new GameView(b);

    b.placeAtom( x: 2, y: 4);
    b.placeAtom( x: 9, y: 4);
    b.placeAtom( x: 5, y: 1);
    b.placeAtom( x: 9, y: 1);
    b.placeAtom( x: 5, y: 9);
    b.placeAtom( x: 1, y: 9);

    b.sendRay( input: 6);
    b.sendRay( input: 41);
    b.sendRay( input: 3);
    b.sendRay( input: 44);
    b.sendRay( input: 26);
    b.sendRay( input: 17);

    Ray ray = b.getSentRays().get(0);
    assertEquals(ray.getInput(), actual: 6);
    assertEquals(ray.getOutput(), actual: 6);

    Ray ray2 = b.getSentRays().get(1);
    assertEquals(ray2.getInput(), actual: 41);
    assertEquals(ray2.getOutput(), actual: 41);

    Ray ray3 = b.getSentRays().get(2);
    assertEquals(ray3.getInput(), actual: 3);
    assertEquals(ray3.getOutput(), actual: 3);

    Ray ray4 = b.getSentRays().get(3);
    assertEquals(ray4.getInput(), actual: 44);
    assertEquals(ray4.getOutput(), actual: 44);
```

As demonstrated here by our program using the same inputs, the ray is reflected before it can pass any further. As this is a reflection, it was once again represented by a purple ray marker. Again the input and output values were compared here plus the deflection type

With this sprint completing ray and circle of influence logic, we also wanted to use examples given in the assignment document as tests against our program. Firstly tried to recreate the below example, included in the software project brief.



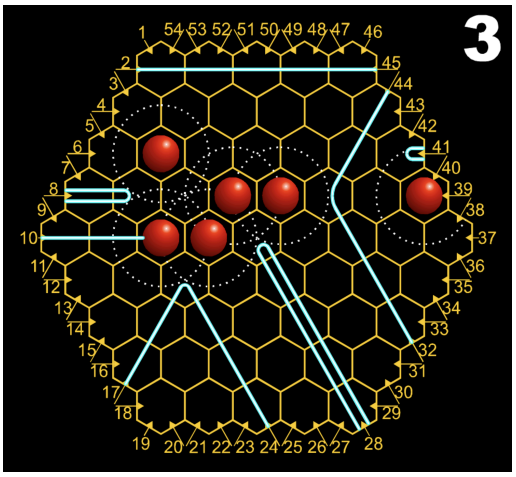

```
@Test
void testAdvancedRayPath() {
    Board b = new Board();
    b.placeAtom( x: 5, y: 3);
    b.placeAtom( x: 7, y: 3);
    b.placeAtom( x: 6, y: 5);
    b.placeAtom( x: 7, y: 5);
    b.placeAtom( x: 2, y: 8);
    b.placeAtom( x: 4, y: 9);

    b.sendRay( input: 48);
    b.sendRay( input: 14);
    b.sendRay( input: 30);

    Ray ray = b.getSentRays().get(0);
    assertEquals(ray.getInput(), actual: 48);
    assertEquals(ray.getOutput(), actual: 53);

    Ray ray2 = b.getSentRays().get(1);
    assertEquals(ray2.getInput(), actual: 14);
    assertEquals(ray2.getOutput(), actual: 14);

    Ray ray3 = b.getSentRays().get(2);
    assertEquals(ray3.getInput(), actual: 30);
    assertEquals(ray3.getOutput(), actual: -1);
}
```

Above is the provided board in the brief compared to our working board. As seen, all rays behave correctly and provide correct corresponding ray markers, green - absorbed, purple - reflected and blue 60 degree reflection.

We also provide the test used to ensure the correct results, ensuring the ray enters and exits at the correct points.

Here the first ray underwent two deflections emerging at 53. After initially undergoing two deflections, the second ray experiences a reflection and then a final two deflections before emerging once again at 14. Lastly, the final ray underwent 120 followed by 60-degree deflections and then a final absorption (output = -1).

Below is the second example provided in the project brief, again we provide the image included and also the board our project produced with similar placements and inputs.




```
@Test
void testAdvancedRayPath2() {
    Board b = new Board();
    GameView view = new GameView(b);

    b.placeAtom( x: 4, y: 3);
    b.placeAtom( x: 5, y: 4);
    b.placeAtom( x: 6, y: 4);
    b.placeAtom( x: 3, y: 5);
    b.placeAtom( x: 4, y: 5);
    b.placeAtom( x: 9, y: 4);

    b.sendRay( input: 8);
    b.sendRay( input: 2);
    b.sendRay( input: 10);
    b.sendRay( input: 41);
    b.sendRay( input: 32);
    b.sendRay( input: 28);
    b.sendRay( input: 17);


    Ray ray = b.getSentRays().get(0);
    assertEquals(ray.getInput(),  actual: 8);
    assertEquals(ray.getOutput(),  actual: 8);
    assertEquals(ray.getDeflectionType(), actual: 180);

    Ray ray2 = b.getSentRays().get(3);
    assertEquals(ray2.getInput(),  actual: 41);
    assertEquals(ray2.getOutput(),  actual: 41);
    assertEquals(ray2.getDeflectionType(), actual: 180);
```

Also provided is code used in a test where we ensure the rays behave correctly.

We also performed a test for the random allocation of six atoms which we mentioned in our project plan. This test checked whether or not six atoms were placed correctly

```
@Test
void testSinglePlayer() {
    Game game = new Game();
    game.singlePlayerSetAtoms();
    assertEquals(game.getBoard().getNumAtomsPlaced(),  actual: 6);
}
```

Finally to test the ‘game’ part of the system, we simulated where a player could place atoms and then the location of these atoms were guessed. The score was kept track of and was then compared using assertEquals();

Below is an example of one of our simulations to ensure the correct score was produced

```
@Test
void testGameScore() {
    Player player = new Player( playerName: "Test Player",  isSetter: false);
    Game game = new Game(player);

    game.getBoard().placeAtom( x: 5,  y: 5);
    game.getBoard().placeAtom( x: 7, y: 5);
    game.getBoard().placeAtom( x: 8, y: 1);
    game.getBoard().placeAtom( x: 6, y: 6);
    game.getBoard().placeAtom( x: 1, y: 5);
    game.getBoard().placeAtom( x: 4, y: 2);

    game.sendRay( input: 29);
    game.sendRay( input: 39);
    game.sendRay( input: 46);
    game.sendRay( input: 3);
    game.sendRay( input: 5);
    game.sendRay( input: 18);

    game.guessAtom( x: 7,  y: 7);
    game.guessAtom( x: 8,  y: 3);
    game.guessAtom( x: 9,  y: 4);
    game.guessAtom( x: 6,  y: 6);
    game.guessAtom( x: 4,  y: 2);
    game.guessAtom( x: 6,  y: 1);

    assertEquals(game.getExperimenter().getScore(),  actual: 30);
}
```

With all tests passing and our project seeming to work as intended, we can mark off that sprint 3 has been fully completed and that all implemented features have been tested and are working well.