COMP20050 - Group 47 - Features
Sprint 4

Student Numbers: 22441636, 22450456, 22715709

In this final sprint, we wanted to advance our previous TUI and create an interactive GUI which we believe will add to the game experience and improve the user experience. We also developed our single player game mode from last sprint which now creates an “AIPlayer” to play both setter and experimenter roles however this is an extra feature, special to our implementation of the game.

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

Implemented Features:

View Package:

The main aspect of the game we wanted to translate over to a graphic user interface was the main board picture - including hexagons, atoms, circles of influence, ray trails and also ray markers.

Displaying the board:

As we used the “Java Swing” API to create a graphical user interface, it gave us the opportunity to draw hexagons onto the screen to represent the board. Through some research and trial and error, we found that the following code would print a hexagon shape to the screen.

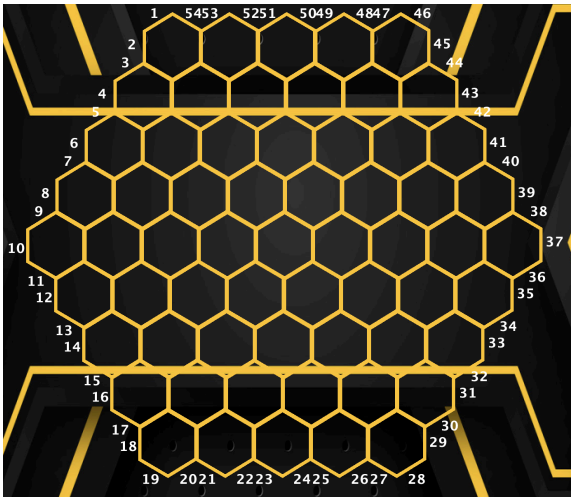
```
for (int i = 0; i < 6; i++) {
    double angle = 2 * Math.PI / 6 * (i + 0.5);
    int xOff = (int) (Math.cos(angle) * side);
    int yOff = (int) (Math.sin(angle) * side);
    if (i == 0) {
        path.moveTo( x: startX + xOff,  y: startY + yOff);
    } else {
        path.lineTo( x: startX + xOff,  y: startY + yOff);
    }
}
path.closePath();
g2.draw(path);
```

As seen, we do not just draw the hexagon but we also create a “path” which is vital in attaching each hexagon drawn onto the board with a corresponding coordinate which our board would understand and vice versa.

We then add each “HexagonPath” to a list of “HexagonPath” objects which simply store a hexagon path and its corresponding coordinates which align with our board.

```
private final List<HexagonPath> hexagonPaths = new ArrayList<>();
```

For then drawing the board, we simply iterate through our original board[][] data structure and for each valid hexagon ie. not “NullHex” (explained in previous reports), a hexagon graphic was drawn to the screen. To create a perfect hexagonal board however proved quite difficult as we had to ensure that each hexagon was perfectly aligned which resulted in trial and error with calculating the x and y value for each hexagon which would be used to dictate where it was printed to on the screen but after some attempts to make it as aligned as possible, we were left with this board being printed to our screen.



As seen above we were also able to print the corresponding number for each ray input around the edge of the board. Similarly to printing the hexagons and tracking its corresponding coordinates, we created a numberAreas list which held a number and its corresponding location on our screen.

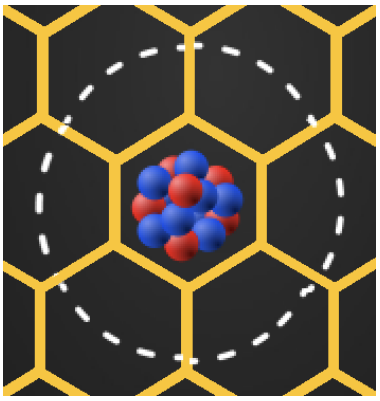
```
private final List<NumberArea> numberAreas = new ArrayList<>();
```

Each number area would have a path on our board and a corresponding number which would be printed to our screen. Again to print the numbers in the correct position on the screen involved some trial and error in getting each number to print in the correct position but after ensuring each was printed correctly resulted in the numbers appearing as seen in the above screenshot.

These were the two things we needed to translate to the GUI initially to allow us to then begin printing the current state of the game to the screen for the user to see but were also important in allowing the user to interact with the game, which will be discussed later.

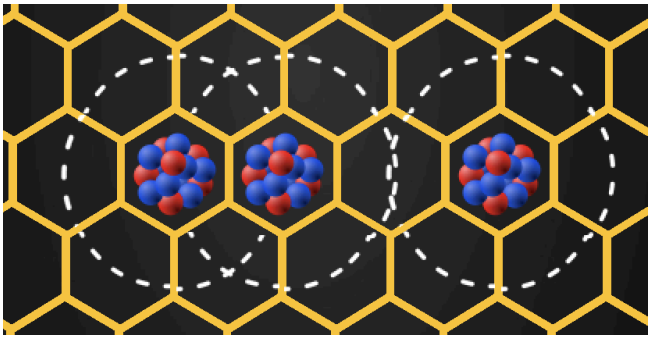
In our previous implementations, we displayed the board through iterating through our board[][] data structure which holds all current items on our board and simply printing a different ascii character for each different object/item on the board. In essence, the exact same thing occurs in our GUIGameBoard class. We again,

iterate through the current state of the board and print a graphic to the screen depending on if something is present on the board or not, the same way we had in our text based interface.



Seen here is our atom placed onto our board and its accompanying circle of influence which is represented as 6 “parts” of a circle of influence as has been previously defined in our implementation. Each part of a circle of influence has an orientation and as a result we can print its corresponding image associated with that orientation like had done previously in our text based interface.

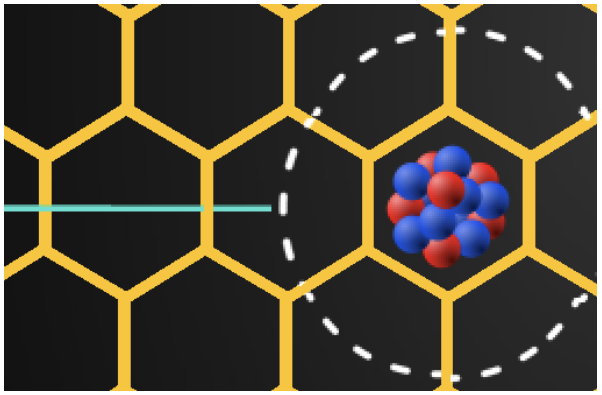
Seen here is what we call a “60 degree” circle of influence graphic which is printed at the top left of an atom. These parts all come together to create a full circle of influence.

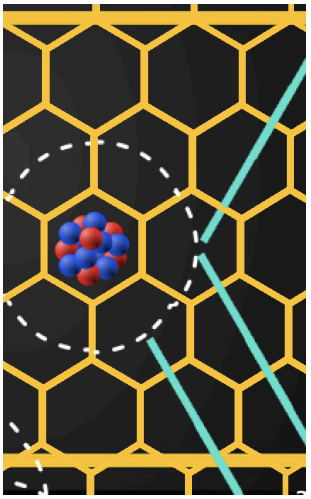


Shown here is how we represent intersecting circles of influence. Before in our text based interface an “x” was printed to the screen as an abstract way of representing circles of influence intersecting but now in our GUI, if a position on a board contains an intersecting circle of influence object, we cycle through its list of circles of influence and simply print each orientation to the screen as seen above where on the left, two atoms have intersecting circles of influence above and below and on the right where two atoms have an intersection between them. This graphical representation did not hinder our board in any way and results in the same game logic. Below shows the main idea in displaying our intersecting circles of influence to the screen.

```
else if (currentBoard.getBoardPosition(col, row) instanceof IntersectingCircleOfInfluence is
    for(int i = 0; i < is.getCircleOfInfluences().size(); i++){
        printCircleOfInfluence(is.getCircleOfInfluence(i), g2, x, y);
    }
}
```

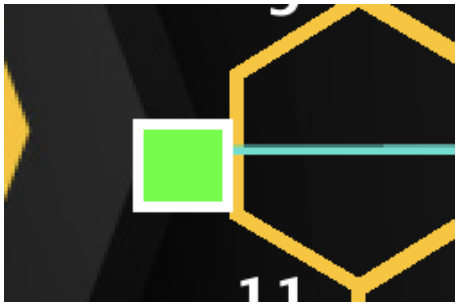
Similarly for printing our ray paths to the board, the coordinate will contain an instance of a “RayTrails” object and will then cycle through each “RayTrail” and print the corresponding graphic associated with each orientation of a ray trail as previously done in our text based interface.





As seen above for each hexagon, if a raytrails object is found, it prints the corresponding trail to the screen which represents a full ray path. However we found that our ray trail would only print inside the hexagon it was present in, it didn’t exactly show the ray deflecting/absorbing however to get around this, as seen in the second image specifically, each raytrail graphic extends from centre to centre which better represents the deflection process.

To finally represent our ray markers, the numberPaths which we discussed earlier became vital as they included both an input/output number and also a path on the screen. In our board class, we updated it to have a list of placed ray markers which we cycle through when painting the board to the screen and place the correct ray marker in its corresponding numbers path on the screen. Our ray marker class was also updated to now include a java swing “Color” instance variable. We cycle through the list of placed ray markers and then print it to its corresponding location on the screen.

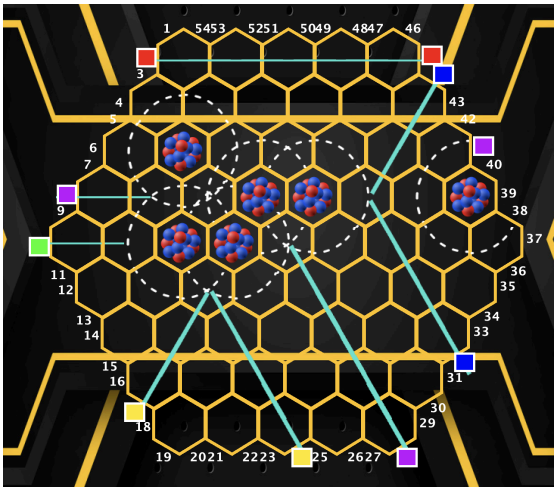


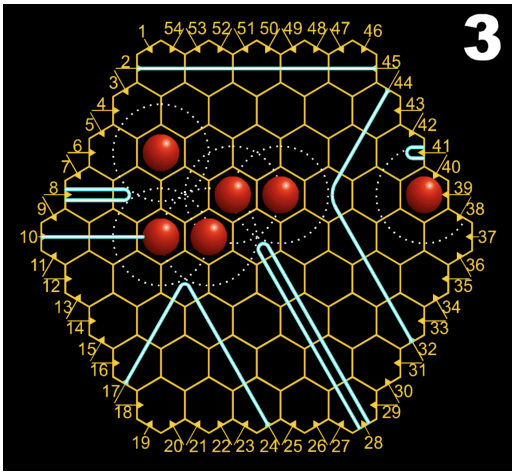
Example of “green” reflected marker

Example of “blue” 60 degree deflection marker

As seen in the above screen shots, the paths are replaced with a “ray marker” to accurately represent a present ray marker.

After fully representing the game board, a full game picture looks like the following in our implementation. Below includes both the board given in the assignment brief alongside our board display.



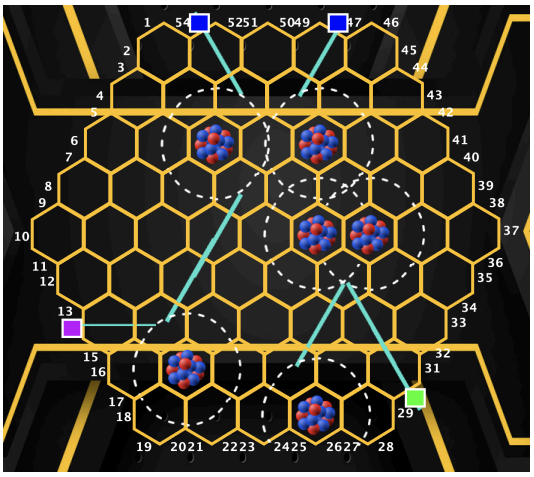


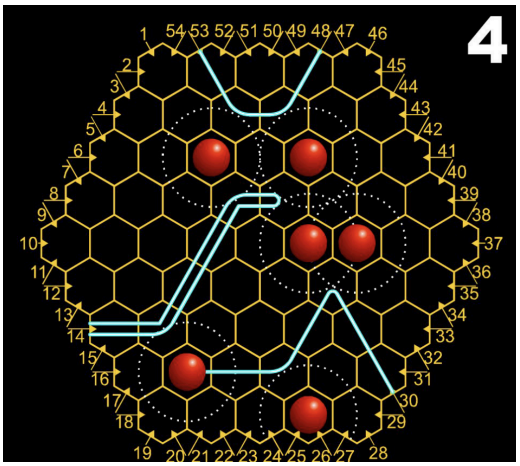
Ray markers description:

Red: no interaction, Blue: 60 degree deflection, Yellow: 120 degree deflection, Purple: reflection, Green: Absorption

(Note on rail trails)

As the way our board is represented to the screen is done through printing each individual hexagons contents and in our previous implementations we done this by placing objects at certain coordinates on our board[][] data structure, atoms and circles of influence will not get overridden by a ray trail which is placed on the board and as a result in instances where rays enter a field of atoms and the ray takes many deflections, you will not be able to fully see the rays path once it starts moving from a hexagon with a circle of influence to another with a circle of influence as it does not overwrite the contents of the hexagon but simply traverses the board and takes on different behaviour depending on the contents of a hexagon. As seen below, you cannot see the placed ray trails in hexagons which include a circle of influence; however this is simply only related to the view side of our implementation and does not say anything for the logic side as rays will behave correctly but may not be fully represented on the finished board screen.





Above shows that all ray behaviour is correct however its entire path is not shown. The ray at the top enters at 53 and experiences a 60 degree deflection and then another 60 degree deflection and exits the board, placing a blue ray marker which corresponds to a ray experiencing a 60 degree deflection. The bottom left ray enters at 14 and takes a 60 degree deflection and then one reflection which results in a purple ray marker as seen above. Finally the last ray enters, experiences a 120 degree deflection, then a 60 degree deflection finally followed by getting absorbed resulting in a green ray marker as seen above.

Interactive GUI:

As mentioned above, each hexagon and each number in which rays are sent in from contain a path on our screen and because of that, when things like mouse clicks occur, we can simply check if a path contains the point in which the mouse was clicked. Through this we are able to get either a coordinate in the case of clicking a hexagon, or an input number in the case of clicking one of the numbers around the edge of the board.

In keeping with the MVC architecture, our GUI does not interact with the board at all and does not call any functions which add/remove things from the board but instead we created an interface which has several methods for the various click situations which can take place on our board. Our main game class, which is our main controller, implements this interface and its methods. Below is an example of when a hexagon is clicked, resulting in an atom being placed/removed.

```
if (game.getBoard().getBoardPosition(hexPath.col, hexPath.row) instanceof Atom) {
    listener.onAtomRemoved(hexPath.col, hexPath.row);
} else {
    listener.onAtomPlaced(hexPath.col, hexPath.row);
}
```

The onAtomPlace() or onAtomRemoved() methods are then called with the corresponding paths col and row variable (a position on our board) in our listener, which is the game

```
@Override
public void onAtomPlaced(int x, int y) {
    placeAtom(x, y);
    guiView.refreshBoard();
    stateManager.updateGameState();
}
```

Shown here is the implemented method which then calls our actual placeAtom() method which was previously implemented in previous sprints. This separation of concerns helped us adhere to the MVC architecture.

Similarly we implemented methods for handling sending rays and also moving from one game state to another which is done through clicking a button on the screen.

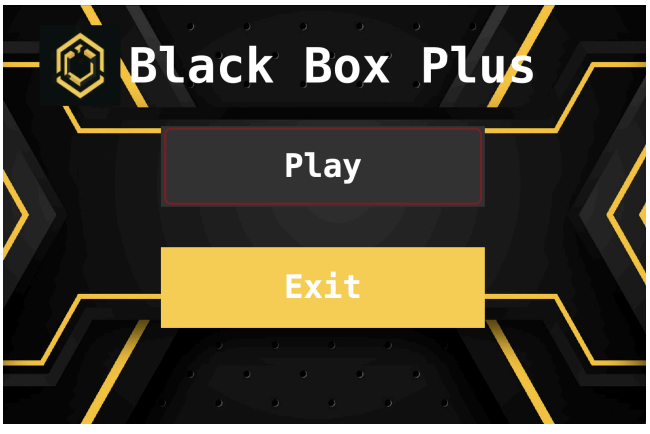
For handling the guessing aspect of the game we created another instance of the board class to allow the user to place and also view the atoms they have currently guessed on the board. We ensure to compare the guessing board to the main game board to calculate the score of the player.

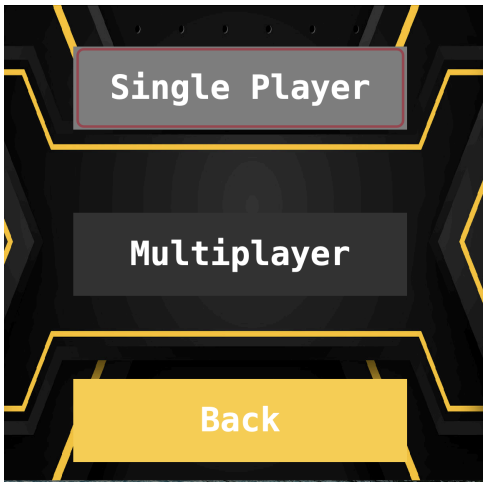
Handling Game States:

As we had migrated our game to a GUI, we could no longer rely on for loops for user input and keeping track of the current state of the game so instead our game takes on a finite set of states in any one round. They are represented by enums and are crucial in defining the picture of the game at any time. For example in our GUI class, once the game reaches the state of “SENDING_RAYS”, the board's “isVisible” boolean turns to false resulting in the experimenter only being able to see the ray markers and nothing else. Another example being when the game reaches the state of “GAME_OVER” the entire board becomes visible and all mouse clicking on to the board become disabled. This helped us merge the idea of our controller and view together and ensure that the game followed the correct state flow

Menu:

In order to make our game more user friendly we added a main menu into our game which is once again through Java Swing. The initial Menu (JPanel) opens and gives you the option to “Play” or “Exit”. Which is located in the GUIMenu class within View sticking with MVC architecture.



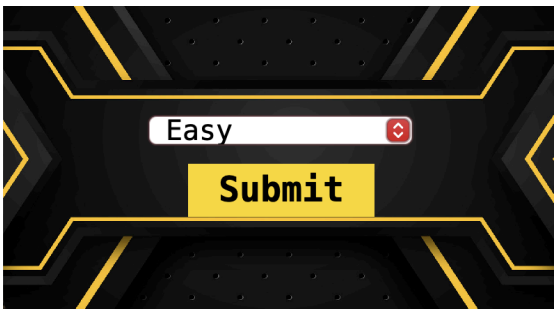


Based on the user selection it moves onto the next screen Which gives the option to which game mode they want to play - Single Player *(extra features, mentioned later)* or Multiplayer. The PlaceComponents Method holds the code for this Menu with a separate method to style the buttons to standardise the buttons as we were going for the retro feel when designing the GUI.

Then based on which button the user clicks it takes them to a screen which they should input their names as part of the game.

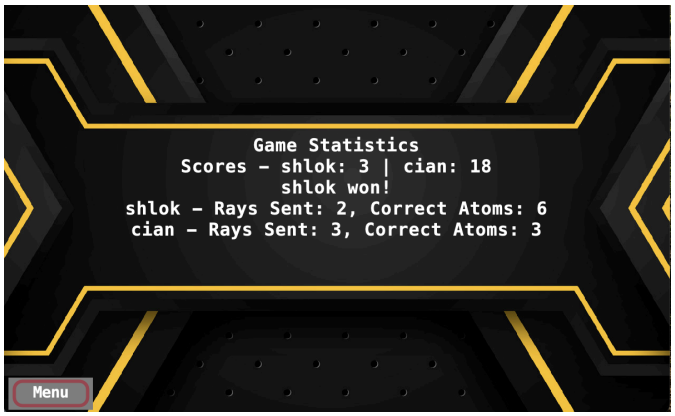
The input is taken through a class called GUI_UserInput. Input is taken via JTextfield which allows user input through the Swing library. And based on whether it is 2 players or 1 player it will take the required number of names.

After Names are input it goes directly to the main game screen unless it is Single Player it takes you to a screen where you select which difficulty you should select this is also done in the GUI_UserInput class via GetAIDifficulty() method. *(extra features - mentioned later)*



After this screen it goes to the main game screen where the game is played and of course if it is multiplayer then the difficulty screen is bypassed as there will be 2 players.

One last thing implemented is the end screen which prints the results based on the winner.



The score is based on the number of atoms guessed incorrectly and number of ray markers on the board at the end of the round. For each ray marker placed on the board +1 is added to your score and for each atom guessed incorrectly plus 5 is added to your score so the player with the highest score loses the game as it means they got more atoms wrong/sent more rays.

From these results we can see “shlok” has won as he has a score of 3 with 2 rays sent meaning 1 ray was absorbed / reflected and 1 ray was deflected or went straight through and he guessed all atoms correctly. Cian had a score of 18 with +15 for guessing 3 atoms incorrectly and +3 for 3 ray markers. These numbers are calculated through communication of our model and controller in sticking with MVC architecture and finally sent to the view to display the final scores/statistics.

There is also a button which enables the player to go back to the menu from the end screen in case they want to play the game again.

Additional Features (not specified in project brief):

Music/Sound:

While we have implemented all features required in the brief of the project, we also took the opportunity to implement some extra features not specified to further enhance the experience of our implementation.

Firstly, sound effects and a sound track were added to the game in an attempt to create the full experience of the BlackBoxPlus game. Sound effects are played on certain button toggles like, placing/removing an atom, sending a ray and also entering a username and starting the game. The sound effects were small little additions which we believe enhance the user experience. A sound track was also created and plays in the background as a game is ongoing. Music is loaded in through the GameMusic class which contains methods to start and stop music and also play each sound effect.

Samples used in music and as sound effects are all royalty free and from www.splice.com

Single Player:

In our previous sprint 3, a single player mode was added to our game which let a user play 1 round against a setter which randomly allocated atoms but in this sprint, this feature was developed and now includes the ability to both send rays and place atoms.

The implementation was quite simple as we just wanted to focus on trying to create the illusion of another player playing. The implementation includes a new class AIPlayer which extends our normal player class but includes a new instance variable which is its difficulty and also some class methods. Depending on the difficulty selected, the “AIPlayer” will do different things. Below includes examples of what different difficulties will do:

Easy (difficulty rating 3):
- Shoots between 6 - 15 rays in random locations
- Guesses 1 atom correctly but the other 5 will be random guesses

Medium (difficulty rating 2):
- Shoots between 4 - 10 rays in random locations
- Guesses 2 atoms correctly but also has a 30% chance of guessing the other 4 correctly

Hard (difficulty rating 1):
- Shoots between 4 - 10 rays in random locations
- Guesses 4 atoms correctly but also has a 30% chance of guessing the other 2 correctly

The randomness included in this class comes from simple randomly generated numbers to make it seem as though an actual person is also playing. While this game mode was not required nor a prime example of AI integration to a game, we believe that it was a unique addition to the implementation and allowed us to further enhance our programming skills.

Testing:

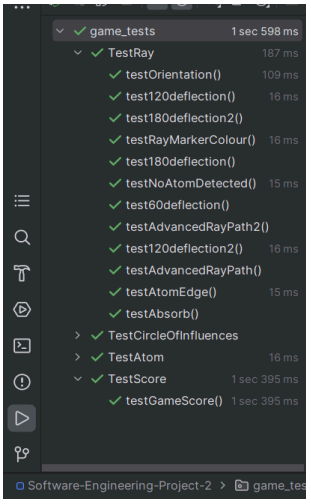
As before in previous sprints, our main objective was more so towards the internal game logic rather than the TUI/GUI, we have a collection of previous unit tests which we ran before previous submissions, ensuring that all logic was correct.

In sprint one we developed the board and allowed an atom and its subsequent circle of influence/intersecting circle of influence to be placed onto the board. This was all tested for through checking what our board data structure contained and following the development of the GUI, all tests regarding sprint 1 features still pass.

In sprint two we developed the ray object and began integrating some simple ray paths along the board. Again this was all tested for in that submission through ensuring rays enter and exit the board correctly. Again all tests still pass which make us confident that all logic regarding simple ray movement/ray markers is still correct

In sprint three finally, we finished developing complex ray paths and all integrated scoring and the “game” aspect. Again we had many tests ensuring correct ray behaviour and also tests to make sure that each player had the correct score. Again all previously created tests still pass, allowing us to conclude that all internal game logic is correct.

Seen here are all previous tests passing



In this sprint as we primarily developed our GUI which was also interactive, ie. was able to handle inputs but would then pass those inputs to the main controller to then pass to the model. As stated above we implemented an interface in order to help separate concerns so below are some examples of tests we created to specifically test our controller aspect and ensure that all information passed to the controller was correctly executed

```
@Test
void testOnAtomPlace() {
    Game g = new Game();
    GUIGameBoard gb = new GUIGameBoard(g);

    gb.listener.onAtomPlaced( col: 4,  row: 5);

    assertTrue(g.getBoard().getBoardPosition( x: 4, y: 5) instanceof Atom);
    assertTrue(g.getBoard().getBoardPosition( x: 4, y: 6) instanceof CircleOfInfluence);
    assertTrue(g.getBoard().getBoardPosition( x: 3, y: 6) instanceof CircleOfInfluence);
    assertTrue(g.getBoard().getBoardPosition( x: 5, y: 5) instanceof CircleOfInfluence);
    assertTrue(g.getBoard().getBoardPosition( x: 3, y: 5) instanceof CircleOfInfluence);
    assertTrue(g.getBoard().getBoardPosition( x: 4, y: 4) instanceof CircleOfInfluence);
    assertTrue(g.getBoard().getBoardPosition( x: 5, y: 4) instanceof CircleOfInfluence);
}
```

As before we tested placing atoms through just placing an atom directly onto the board however to fully test the controller, we call the method to let the controller know to place an atom onto the model. As seen here we asserted that the everything was present on the board

We also conducted similar tests in sending rays and ensuring that our controller correctly handles requests to send rays. Below is an example of one of these tests

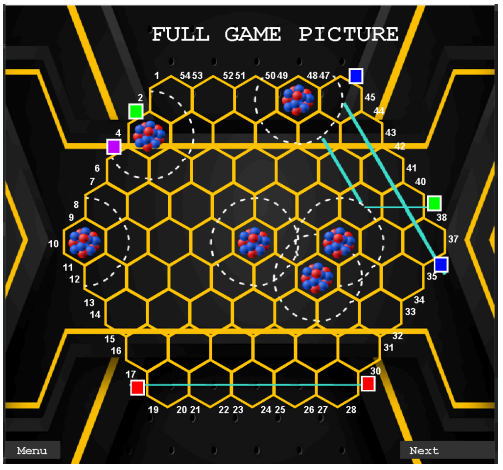
```
@Test
void testOnSendRay() {
    Game g = new Game();
    GUIGameBoard gb = new GUIGameBoard(g);

    gb.listener.onRaySent( number: 10);

    Ray ray = g.getBoard().getSentRays().get(0);
    assertEquals(ray.getInput(), actual: 2);
    assertEquals(ray.getOutput(), actual: 45);
}
```

Again we are not directly sending the ray but instead calling our listener - (controller, to execute the sendRay() method

We ensured that all interactions with the controller were correct which led us to finally just having to test our view component.

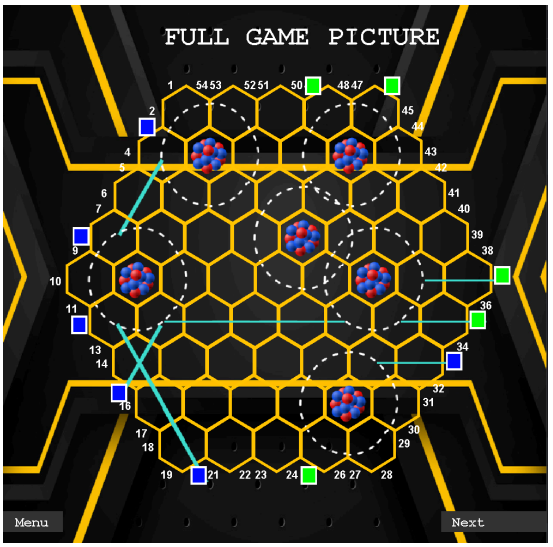
For this we ensured that all contents from the model were successfully translated to the live game board. As we know all internal logic is correct this made testing much more accessible as we simply had to manually validate content of the board was correctly displayed.

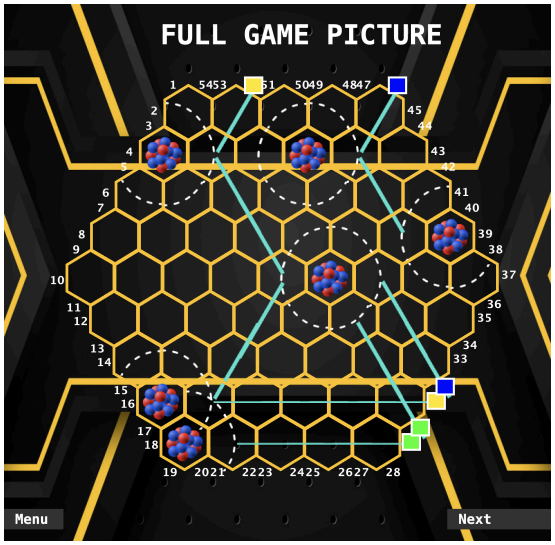
Below are some examples of manual testing we conducted to ensure that all atom placement, circle of influence/intersecting circle of influence, ray path and also ray markers were translated to the screen from the main board[][] data structure.



As seen here, everything displays correctly and everything is visible on the screen.

We conducted various examples of the development of the GUI and were constantly updating and ensuring that everything displays as expected





With these various examples, we were content in believing that our view correctly translates all model logic/state to the screen.

Again as mentioned above, due to the nature of how our board stored content and the picture of the current game, sometimes we cannot override more important components which lead to ray paths not being visible in certain situations but after concluding testing, we knew that all ray logic is in fact correct.

In the above screenshot, notably the ray emerging from the blue ray marker firstly hits one circle of influence, changing its direction directly into another which again changes its direction. While the internal change is not visible, the ray emerges from the expected position and this is also true for all possibilities.

With all implementation and testing complete for test 4, we are happy to mark the GUI development as complete.