Breakdown:

This sprint was dedicated to the implementation of the ray functionality, which progressed remarkably well as our initial conceptualization aligned perfectly with its practical execution. Due to this we ended up digging into some of the sprint 3 objectives. Consequently, this has led us to strategically reorganizing the upcoming two sprints. This adjustment ensures that we remain on track, effectively using our current momentum to enhance future development.

Overall sprint 2 was very successful as we completed all objectives in the original and more.

Revised Sprints

Sprint 2

Objectives	Tasks	Results
From original sprint 2: - Continue on core game mechanics - Ray inputs - Ray reaction to atoms - Ray visibility **Sprint 3 features completed:** - Enhancing the ray mechanics **Also:** -Main menu	**From original sprint 2:** - Add the feature for the user to input the ray. - Add the feature when a ray does not meet with any atoms. - Add the feature when a ray comes in contact with an atom and returns. - Add the feature when a ray comes in contact with an atom and reflects with an angle of 60 degrees. **Sprint 3 features completed:** - Add the feature that when the ray comes into contact with 1< atoms, it gets reflected at 120 degrees. - Add the feature that if a ray comes In contact with an atom at the edge of the board it is reflected. **Also:** -create main menu	**From original sprint 2:** - Can now input a ray. (using mouse click) - Ray can be reflected if coming in contact with an atom. - Can see the path of the ray for testing purposes. **Sprint 3 features completed:** -Most abnormal cases complete. **Also:** -Created main menu

See “Project_Plan_2.pdf” for more details.

Implementation:

-Ray Class:

```
public class Ray{
    2 usages
    private final Point entryPoint;//axial coord of hexagons from which is enters

    3 usages
    private Point direction;//direction of ray movement;

    2 usages
    private Point exitPoint;
    10 usages  artemkuc44
    public Ray(Point entryPoint,Point direction){
        if(direction.x < -1 || direction.y<-1 || direction.x > 1 || direction.y>1){
            throw new IllegalArgumentException("direction out of range");
        }
        this.entryPoint = entryPoint;
        this.direction = direction;
    }
```

Stores entry point and exit point (will be used for ray markers later).

Each ray has a direction for movement, this is used to calculate the next step of the ray.

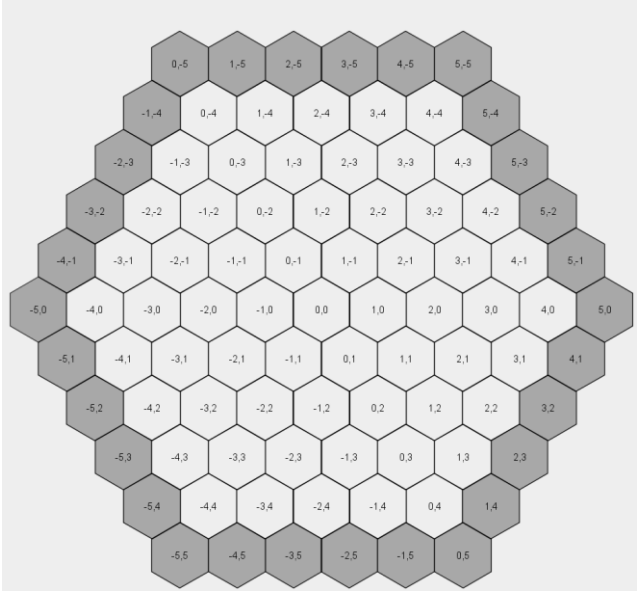
```
public static final Point[] DIRECTIONS = new Point[] {//Directions array used to compute circular dependency
        new Point( x: 0,  y: 1), new Point( x: 0,  y: -1), new Point( x: -1,  y: 0),
        new Point( x: 1,  y: 0), new Point( x: -1,  y: 1), new Point( x: 1,  y: -1)
};
```

Similar logic is used to calculate the atom circular dependency "neighbours" (each row vector from array shown above is added on to the atom to calculate all neighbouring hexagons), which has now been adjusted in the atom class to map the direction that was used to create it.

```
private final HashMap<Point,Point> neighbours;
```

Now by creating a ray with an entry point along the border (outlined in grey) and a direction the ray can now move.

eg. Ray ray = new Ray(new Point(5,0),new Point(-1,0));, will create a ray that starts at 5,0 pointing in the "left direction" specified by the (-1,0) row vector.



The moveRay method then moves the ray and sets the exitPoint.

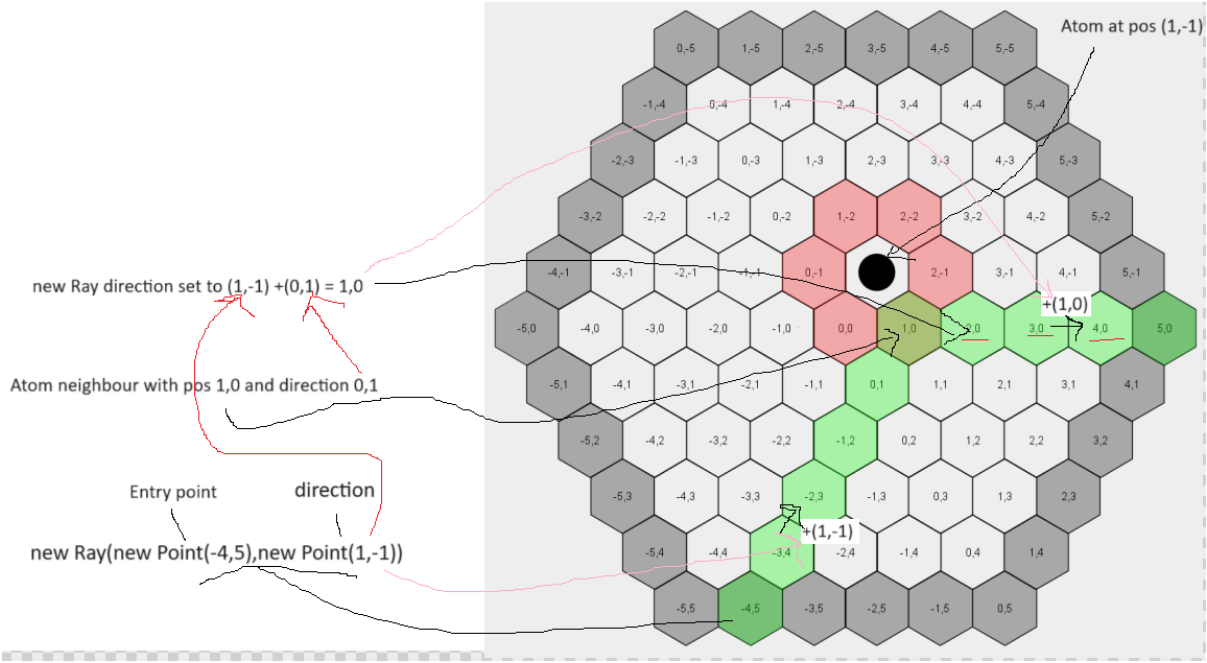
moveRay method in hexBoard class:
This is where the magic happens,

-A point is created to mark the current position of the ray.
-A while loop prevents the ray from leaving the board.

-checks if the entryPoint is a neighbour, if so it deflects back setting the exitPoint equal to the entryPoint

-else if checks if the current position is a neighbour, if so it calculates the new rays deflected direction by simply adding the directional vectors of the ray and the neighbour. This accurately calculates all deflections. If there are shared circle of influences between multiple atom all of the directions will be added to the ray direction deflecting it correctly.

This was the plan all along as we used directional vectors to create the atoms neighbours, which we used to our advantage when calculating deflection.




```
//Point currPoint = ray.getEntryPoint();//very intereseting case where entry point was being refrenced and altered despite being final
Point currPoint = new Point(ray.getEntryPoint().x,ray.getEntryPoint().y);//new point needed to be initialised to removed any reference to entry point
System.out.println("entry point" + ray.getEntryPoint());

while(hexCoordinates.contains(currPoint) || (borderHex.contains(currPoint))){
    for(Atom atom:atoms){//traverse atom array
        if(atom.getNeighbours().containsKey(ray.getEntryPoint())){//checks for deflection with circle of influence on border
            ray.setExitPoint(ray.getEntryPoint());
            System.out.println("exit point" + ray.getExitPoint());
            return;
        }
        else if(atom.getNeighbours().containsKey(currPoint)){
            // Retrieve the direction from the atom to the Neighbour (which is the key's value)
            Point neighbourDirection = atom.getNeighbours().get(currPoint);
            // Add directions
            ray.setDirection(new Point( x: ray.getDirection().x + neighbourDirection.x,  y: ray.getDirection().y + neighbourDirection.y));
        }
    }
    if(rayMovement.contains(currPoint)){
        rayMovement.remove(currPoint);
    }else{
        rayMovement.add(new Point(currPoint.x,currPoint.y));
    }
    currPoint.x += ray.getDirection().x;
    currPoint.y += ray.getDirection().y;
}
currPoint.x -= ray.getDirection().x;
currPoint.y -= ray.getDirection().y;
ray.setExitPoint(currPoint);
repaint();
```

User inputting rays:

We came to a simple solution for the user to be able to input the rays, similarly to the black box square game the user would simply click a border hexagon from which to send the ray, the side closest to the clicked point is used to create a ray with that direction.

closestSide helper method in hexBoard class:

A fairly simple helper method which takes the clicked point (in pixel coords) and finds the closest hexagon to it in order to calculate and return the direction by finding the difference between the clicked border hexagon and the closest internal hexagon to it.

As of now it traverses the entire array of internal hex coordinates, which works but is inefficient. A solution to traverse only the edge internal hexagons will be worked on in sprint 4.

```
public Point closestSide(Point clickedPoint){//returns direction of ray movement
    double min = 1000;//needed to initialise to something
    Point minPoint = clickedPoint;//needed to initialise to something
    for(Point internalPoint:hexCoordinates){//for all hexCoordinates points (inefficient)//TODO make more efficient
        Point pixelInternal = axialToPixel(internalPoint.x, internalPoint.y);//convert to pixel(needed for distance)
        if(clickedPoint.distance(pixelInternal) < min){
            min = clickedPoint.distance(pixelInternal);
            minPoint = pixelInternal;
        }
    }
    Point clickedAxial = pixelToAxial(clickedPoint.x, clickedPoint.y);
    Point minPointAxial = pixelToAxial(minPoint.x, minPoint.y);

    return (new Point( x: minPointAxial.x - clickedAxial.x,  y: minPointAxial.y - clickedAxial.y));//returns direction
}
```

In the mouse listener:

```
else if(borderHex.contains(hexCoord)){
    Ray ray = new Ray(hexCoord,closestSide(clickedPoint));
    moveRay(ray);
}
```

In the paint component (for visualisation of ray movement):

```
//draw rays
for(Point point: rayMovement){
    Point point1 = axialToPixel(point.x, point.y); // Convert axial to pixel coordinates
    g2d.setColor(new Color( r: 0,  g: 255,  b: 0,  a: 75)); // Semi-transparent red for highlighting
    g2d.fill(createHexagon(point1.x, point1.y));
    g2d.setColor(Color.BLACK); // Reset color for drawing other elements
}
```

Array used to store the rays movements used in paint component of hexBoard:

```
private ArrayList<Point> rayMovement = new ArrayList<>();//array of points crossed in rays path
```

Main Menu Screen:

For main menu screen, I created a Jframe calling it Blackbox +.

```
JPanel titlePanel = new JPanel();
titlePanel.setLayout(new BorderLayout());
JLabel titleLabel = new JLabel( text: "BLACK BOX +", SwingConstants.CENTER); //putting the title in the centre
titleLabel.setFont(new Font(titleLabel.getFont().getName(), Font.BOLD,  size: 50));
titlePanel.add(titleLabel, BorderLayout.NORTH);
```

Above code is about the title label which acts as our logo. I moved the title label from the actual game onto the main menu as I felt it would be more suited to there. The last 2 lines above, “setFont” and “add” designs the position and look of the title.

```
JPanel mainMenuPanel = new JPanel(new GridBagLayout()); //creating panel for mm
JButton startButton = new JButton( text: "Click to Start");
JButton rulesButton = new JButton( text: "Rules");
```

Created the actual main menu panel calling it “mainMenuPanel”, also adding two buttons which will be used to navigate to rules or the actual game.

```
startButton.setFont(new Font( name: "Arial", Font.BOLD,  size: 20));
startButton.setPreferredSize(new Dimension( width: 200,  height: 100)); //dimensions for button

rulesButton.setFont(new Font( name: "Arial", Font.BOLD,  size: 20));
rulesButton.setPreferredSize(new Dimension( width: 200,  height: 100));
```

Above is setting the font for both buttons and also creating the dimensions for the button (how big they are).

```
GridBagConstraints gbcStart = new GridBagConstraints(); //css basically for starting button
gbcStart.gridwidth = GridBagConstraints.REMAINDER; //skips line
gbcStart.fill = GridBagConstraints.HORIZONTAL;
gbcStart.insets = new Insets( top: 0,  left: 0,  bottom: 20,  right: 0); //padding for where the button is.


//same thing as above for rules button.
GridBagConstraints gbcRules = new GridBagConstraints();
gbcRules.gridwidth = GridBagConstraints.REMAINDER;
gbcRules.fill = GridBagConstraints.HORIZONTAL;
gbcRules.insets = new Insets( top: 20,  left: 0,  bottom: 0,  right: 0);
```

GridBagConstraints is used to position the buttons on the screen. “GridWidth” is used to skip a line so the buttons will be on different y axis positions. “Insets” is essentially padding for the middle of the screen as to why start and rules button are opposite in values as I added a space between them.

```
frame.setLayout(new BorderLayout());
frame.add(titlePanel, BorderLayout.NORTH); //adding title panel up north.
frame.add(mainMenuPanel, BorderLayout.CENTER); //everything centre
frame.setLocationRelativeTo(null); //makes it so when launching, it launches in the middle of the screen.
frame.setVisible(true); //makes the whole thing viewable.
```

This code here is about how it is launched, as I added “setLocationRelativeTo” to make the screen centre when launching for easier use. Also made everything centre from the “BorderLayout.CENTER”.

```
jamie6084 +1
startButton.addActionListener(new ActionListener() { //when start button is pressed...
    jamie6084 +1
    @Override
    public void actionPerformed(ActionEvent e) {

        frame.getContentPane().removeAll(); //when its pressed, removes everything on screen


        HexBoard hexPanel = new HexBoard();


        frame.add(hexPanel, BorderLayout.CENTER); //adds the hex panel.


        frame.validate(); //validates
        frame.repaint(); //painting
    }
```

When the start button is pressed it will run the following code. First it removes everything on the frame as it will be switching frames using the “removeAll()” function, Creates new hex panel and simply adds it. Also made the hex panel centre.

```
jamie6084
rulesButton.addActionListener(new ActionListener() { //when rules button is pressed,,,,
    jamie6084
    @Override
    public void actionPerformed(ActionEvent e) {
        RulesScreen rulesFrame = new RulesScreen();
        rulesFrame.setVisible(true); //making it viewable.
    }
});
```

Above is the code that will run when the rules button is pressed. It simply makes the RulesScreen object and makes it viewable using “setVisible”. There is no need to remove everything like when launching hex panel as the rules is just a pop up.

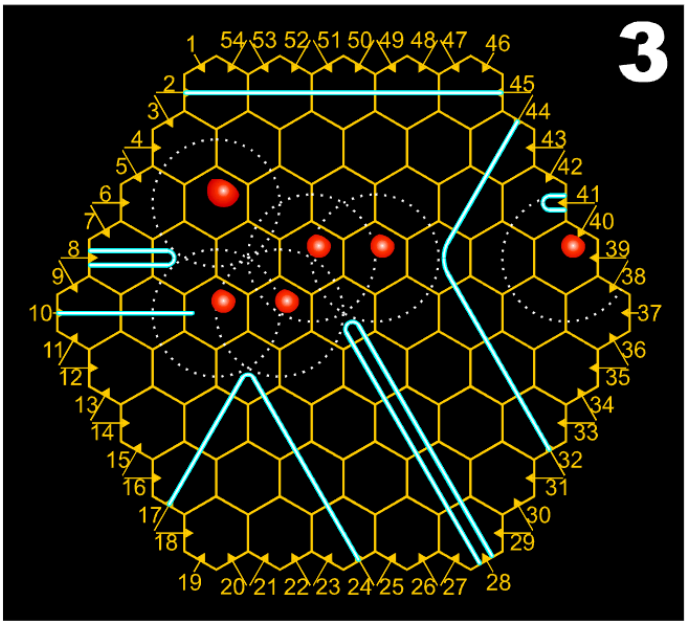
Rules Screen:

```
JPanel rulesPanel = new JPanel(); // Creating new JPanel
rulesPanel.setLayout(new BoxLayout(rulesPanel, BoxLayout.Y_AXIS)); // Makes it so it's like a list in html, goe

for (String rule : ruleTexts) { //goes through every rule in the String array
    JLabel ruleLabel = new JLabel(rule);
    ruleLabel.setBorder(BorderFactory.createEmptyBorder( top: 5,  left: 5,  bottom: 5,  right: 5)); //css for it
    rulesPanel.add(ruleLabel);
}
```

In this code it creates a new JPanel, the “setLayout” is used to create the rules in list format as it essentially skips a line every time a new one is implemented. Created for loop to iterate through each rule. Converts the rules into JLabel with “setBorder” designing where it is going to be placed on the JPanel. Last line it adds it to the panel.

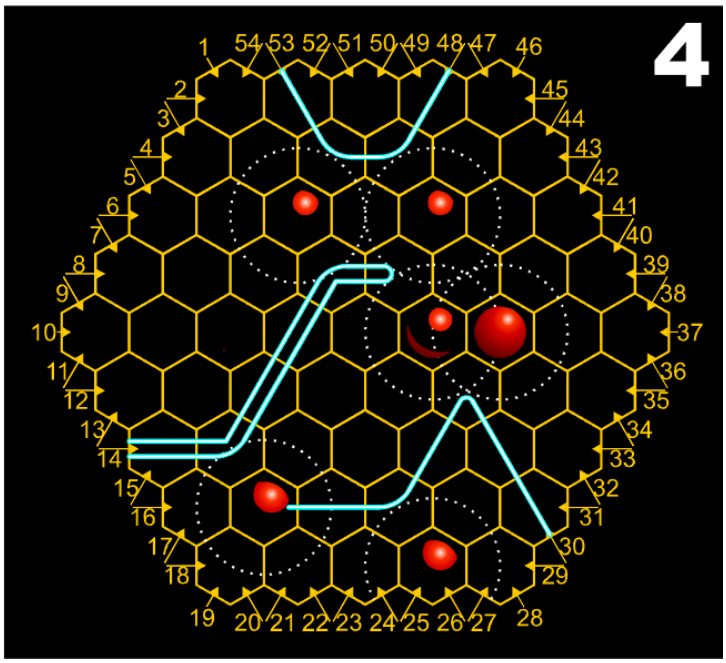
Unit Testing:

Added test cases recreating the 2 diagrams (3&4) in the “Black Box+ Rules” document. Allows to test entry and exit points for rays depending on the atom positions.




```java
@Test
void testRayDeflection3(){
    //recreates diagram number 3 from Black Box+ Rules pdf.
    Atom atom1 = new Atom(new Point( x: -2, y: 0));
    Atom atom2 = new Atom(new Point( x: -1, y: 0));
    Atom atom3 = new Atom(new Point( x: 0, y: -1));
    Atom atom4 = new Atom(new Point( x: 1, y: -1));
    Atom atom5 = new Atom(new Point( x: 4, y: -1));
    Atom atom6 = new Atom(new Point( x: -1, y: -2));

    hexGridPanel.atoms.add(atom1);
    hexGridPanel.atoms.add(atom2);
    hexGridPanel.atoms.add(atom3);
    hexGridPanel.atoms.add(atom4);
    hexGridPanel.atoms.add(atom5);
    hexGridPanel.atoms.add(atom6);

    Ray ray10 = new Ray(new Point( x: -5, y: 0),new Point( x: 1, y: 0));
    Ray ray24 = new Ray(new Point( x: -2, y: 5),new Point( x: 0, y: -1));
    Ray ray28 = new Ray(new Point( x: 0, y: 5),new Point( x: 0, y: -1));
    Ray ray32 = new Ray(new Point( x: 2, y: 3),new Point( x: 0, y: -1));
    Ray ray41 = new Ray(new Point( x: 5, y: -2),new Point( x: -1, y: 0));
    Ray ray8 = new Ray(new Point( x: -4, y: -1),new Point( x: 1, y: 0));

    hexGridPanel.moveRay(ray10);
    hexGridPanel.moveRay(ray24);
    hexGridPanel.moveRay(ray28);
    hexGridPanel.moveRay(ray32);
    hexGridPanel.moveRay(ray41);
    hexGridPanel.moveRay(ray8);
```

```java
assertEquals(new Point( x: -5, y: 4),ray24.getExitPoint());
assertEquals(ray28.getEntryPoint(),ray28.getExitPoint());
assertEquals(new Point( x: 5, y: -4),ray32.getExitPoint());
assertEquals(ray41.getEntryPoint(),ray41.getExitPoint());
assertEquals(ray8.getEntryPoint(),ray8.getExitPoint());

//assertEquals(new Point(-3,0),ray10.getExitPoint());//absorbtion
```




```
@Test
void testRayDeflection4(){
    //recreates number 4 from Black Box+ Rules pdf
    Atom atom7 = new Atom(new Point( x: -3, y: 3));//bottom left
    Atom atom8 = new Atom(new Point( x: -1, y: 4));//bottom right
    Atom atom9 = new Atom(new Point( x: 2, y: 0));//middle right
    Atom atom10 = new Atom(new Point( x: 1, y: 0));//middle left
    Atom atom11 = new Atom(new Point( x: 2, y: -2));//top right
    Atom atom12 = new Atom(new Point( x: 0, y: -2));//top left

    hexGridPanel.atoms.add(atom7);
    hexGridPanel.atoms.add(atom8);
    hexGridPanel.atoms.add(atom9);
    hexGridPanel.atoms.add(atom10);
    hexGridPanel.atoms.add(atom11);
    hexGridPanel.atoms.add(atom12);


    Ray ray30 = new Ray(new Point( x: 1, y: 4),new Point( x: 0, y: -1));
    Ray ray14 = new Ray(new Point( x: -5, y: 2),new Point( x: 1, y: 0));
    Ray ray48 = new Ray(new Point( x: 4, y: -5),new Point( x: -1, y: 1));

    hexGridPanel.moveRay(ray30);
    hexGridPanel.moveRay(ray14);
    hexGridPanel.moveRay(ray48);

    //assertEquals(new Point(-2,3),ray30.getExitPoint());//absorbtion
    assertEquals(ray14.getEntryPoint(),ray14.getExitPoint());//reflection

    assertEquals(new Point( x: 1, y: -5),ray48.getExitPoint());
}
```

TODO:

-Figure out absorption case, as of now a direct hit is reflected straight back which is incorrect and needs to be accounted for when dealing with different ray markers in game, this is was in sprint 3 originally.

-The rest of sprint 3.

Sprint 3 (revised)

Objectives	Tasks	Results
From original sprint 3: - Tidy up on any loose ends of ray reflection. **New Objectives:** -Ray exit and entry markers -Rules for colours/symbols of markers -Game modes -Calculating score	**From original sprint 3:** -Add feature where a direct hit is absorbed **New Tasks:** -Find a way to show the type of entry/exit markers -Add feature to show ray entry and exit markers. -Add feature to play multiplayer/single player -Calculate and Display Score	**From original sprint 3:** -completed ray mechanics **New Results:** -completed ray markers -completed game modes -completed score calculation/ Display