Documentation

Htet Wai Yan

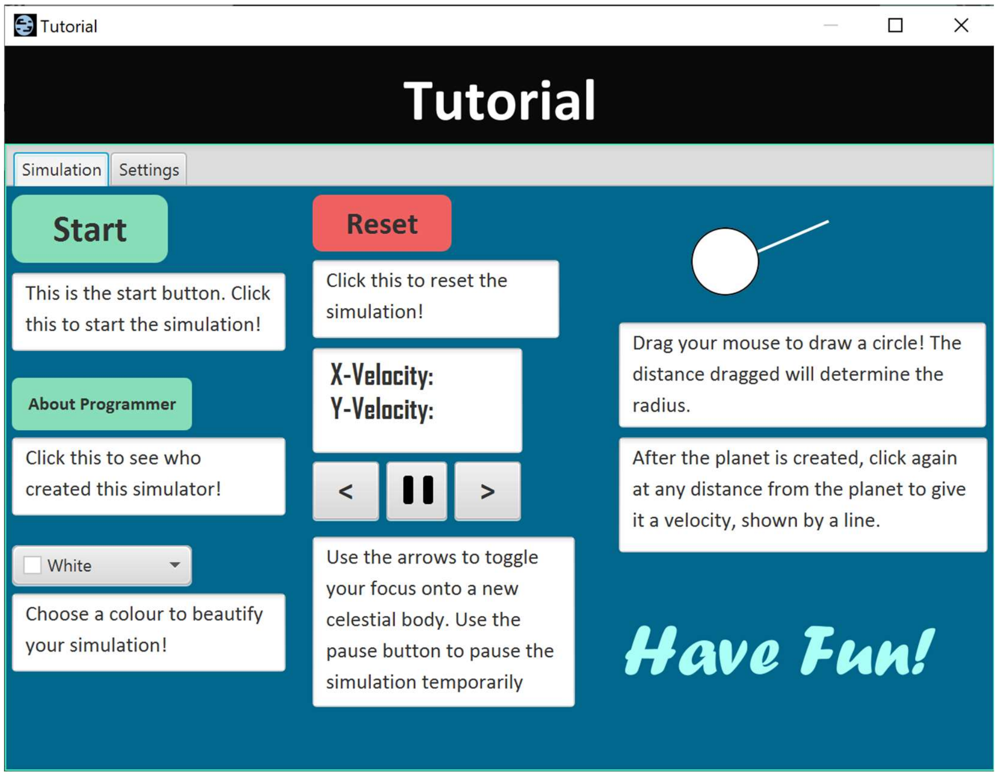
<u>**Overview of project**</u>

This project is a simulator, simulating the movement of celestial objects, like planets and stars, all in a computer screen. The simulator allows users to add in their own planets of variable size and velocity and observe the simulation move the planet about, obeying laws of physics like the gravity forces between two celestial bodies, and the velocity of the object. Predicting movement of multiple bodies influenced by each other's gravitational fields is complex in nature and difficult to imagine and visualize, but this project allows for the visual aspect of this complex problem to be seen clearly.

Who is this project aimed at, or, in other words, who is the target audience? The target audience is children aged 9-16 years old, the prime age where children begin to allow their imagination to flow and become curious about the universe they live in. A popular subject that captures the curiosity of many is the subject of physics in space, or astronomy. Specifically, when these children learn about the solar system and the Earth's orbital for the first time, they being to ponder even more deeply about the beauty of physics in space. Ideally, they would like to be able to satiate their curiosity, something that can be achieved by using this project. It accommodates their imaginative nature and curiosity, at the same time being an educational resource for them in their journey of learning about the strange physics and scientific laws that control our universe.

<u>**User guide**</u>

When the program is launched, a start screen will be seen. This start screen comprises of the title of the program, "SolarSim", a few images in the form of planets to beautify the start screen. The start screen also comprises of a "Tutorial" button, which guides users in the usage of this program, and an "About Programmer" button, which, when clicked, shows the profile of the programmer who built this program. The Tutorial button will open up a window comprising of two tabs, one giving the users a tutorial of the Simulation itself, and one of the Settings page.


Tutorial

Tutorial

This is the start button. Click this to start the simulation!

Click this to reset the simulation!

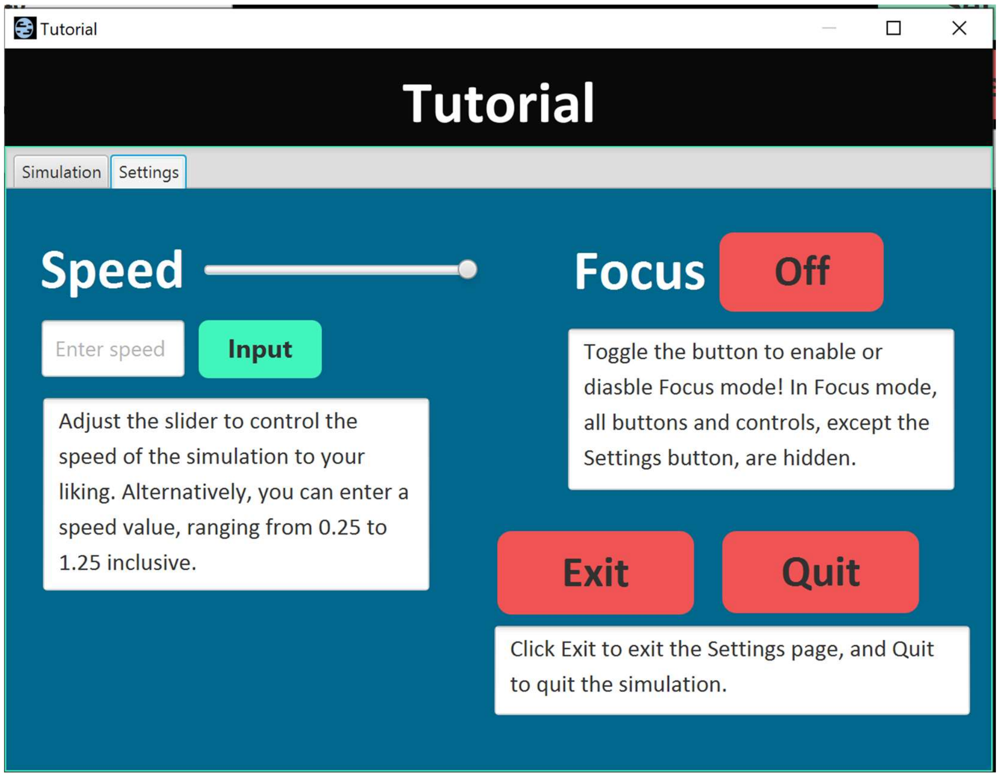
Drag your mouse to draw a circle! The distance dragged will determine the radius.

Click this to see who created this simulator!

After the planet is created, click again at any distance from the planet to give it a velocity, shown by a line.

Choose a colour to beautify your simulation!

Use the arrows to toggle your focus onto a new celestial body. Use the pause button to pause the simulation temporarily


Tutorial

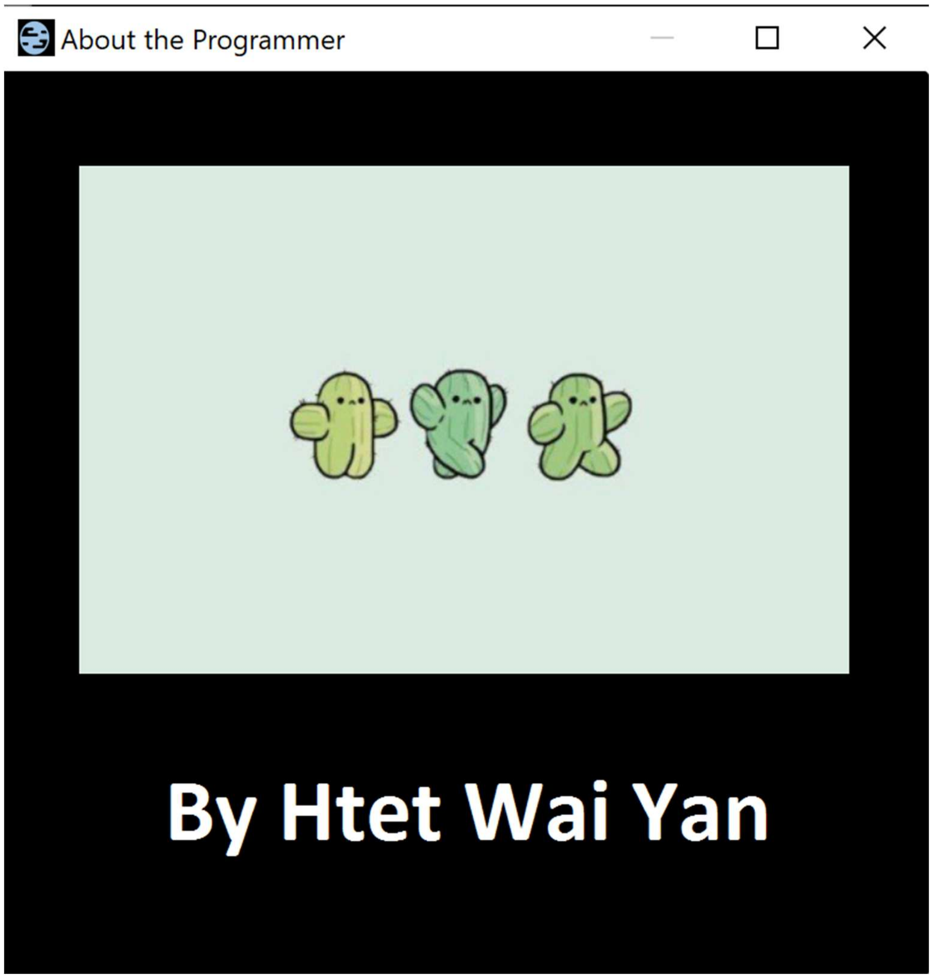
Tutorial

Adjust the slider to control the speed of the simulation to your liking. Alternatively, you can enter a speed value, ranging from 0.25 to 1.25 inclusive.

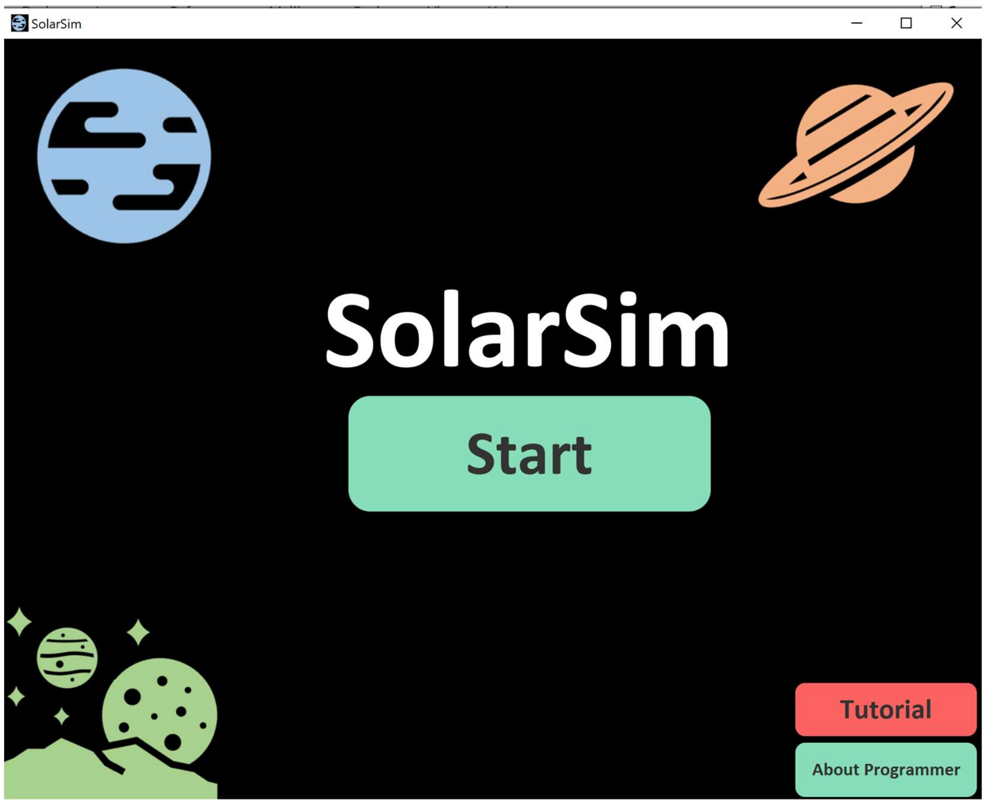
Toggle the button to enable or diasble Focus mode! In Focus mode, all buttons and controls, except the Settings button, are hidden.

Click Exit to exit the Settings page, and Quit to quit the simulation.

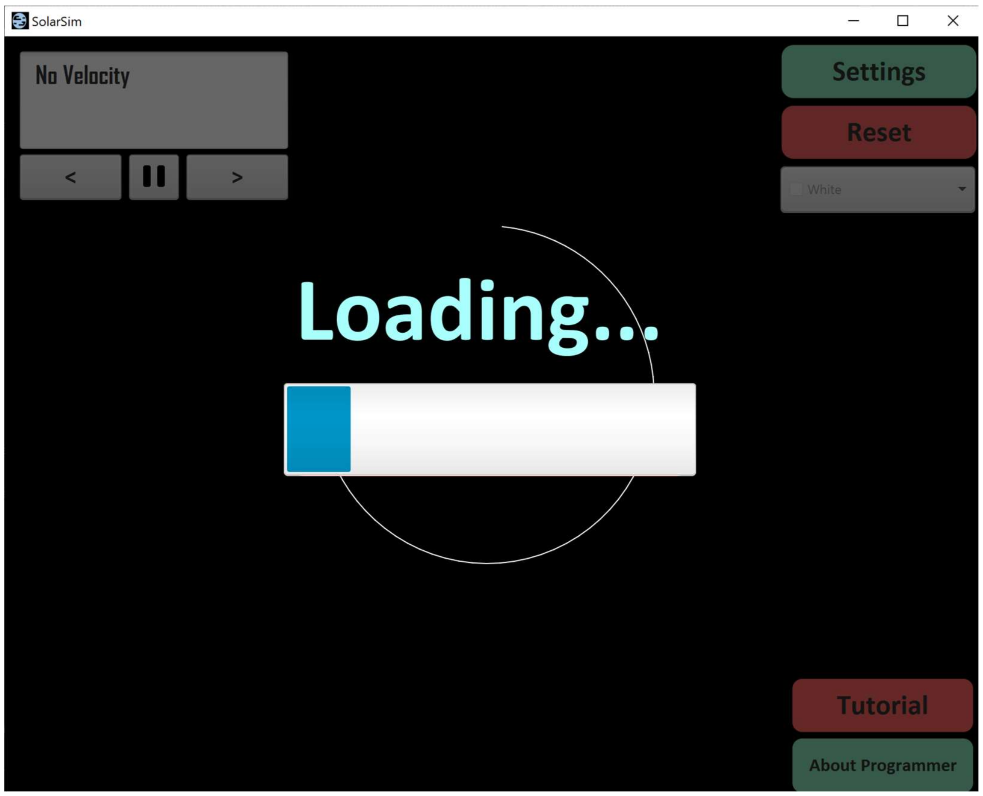
The About Programmer button leads to the following page.



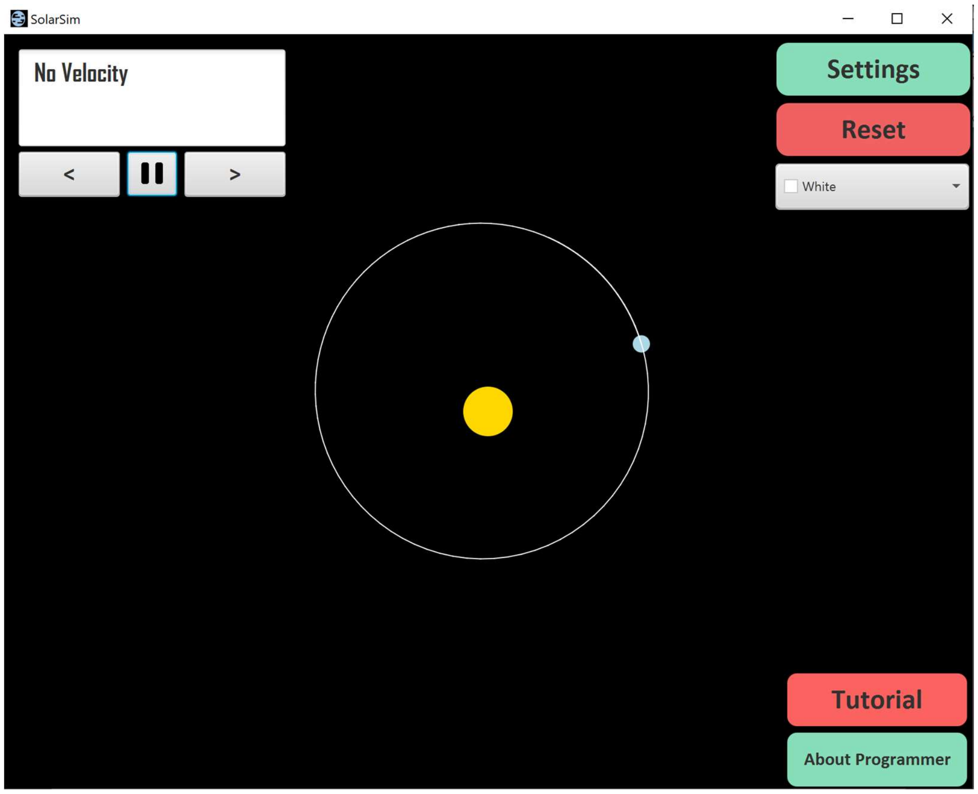
Most notably, there is a “Start” button in the centre of the screen, which, when clicked, will start the simulation, taking the users to another screen.



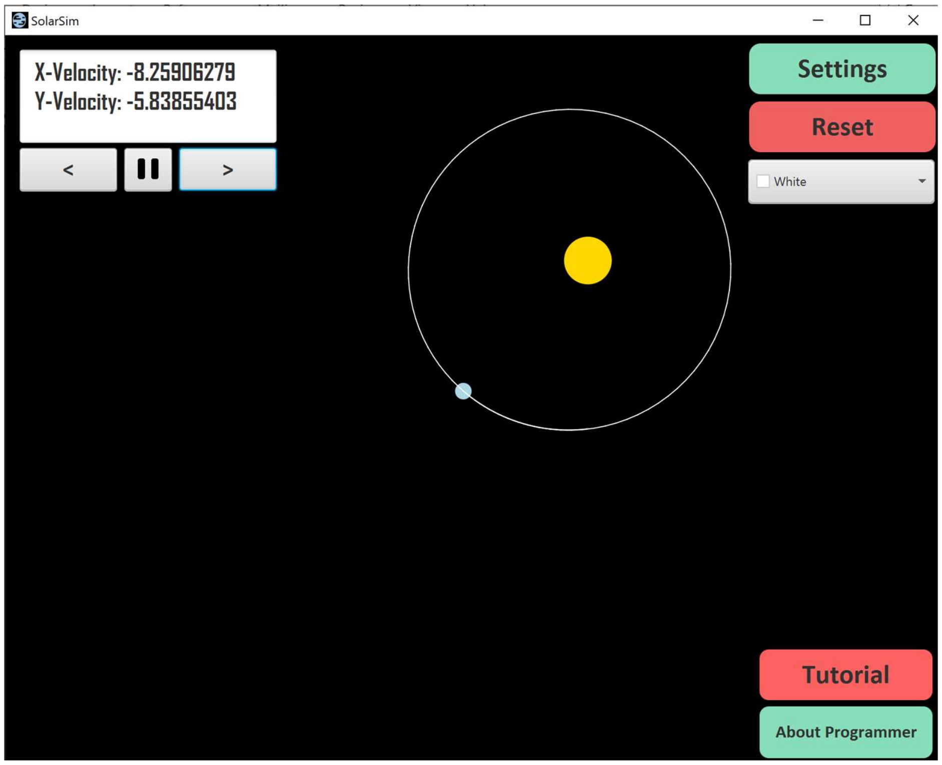
Upon clicking the "Start" button, a splash screen, or a loading screen, will appear, taking a few seconds to load.



Once the splash screen is complete and has finished loading, the main page will be shown, with, by default, a small blue planet orbiting around a fixed, immoveable star object.



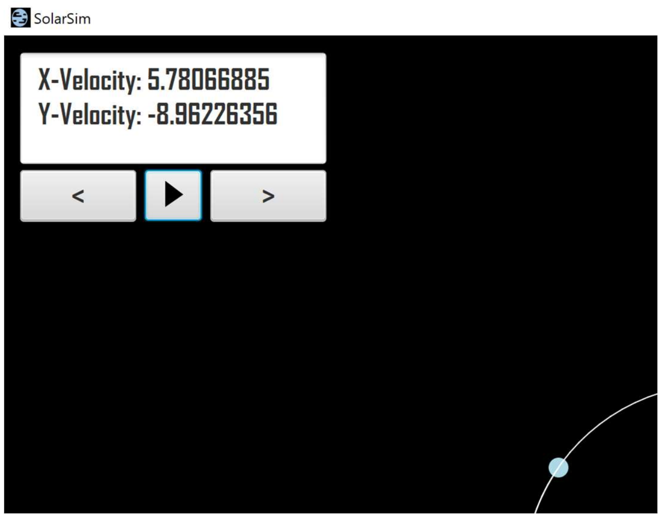
On the main page, at the top left-hand corner, there is a panel, showing a "screen" and a few control buttons below it. The leftward pointing and rightward pointing buttons will allow the user to cycle through the celestial objects currently added, by default being focused on the star in the centre. This is why the screen currently shows "No Velocity", as the star is fixed and does not have velocity.



X-Velocity: -8.25906279
Y-Velocity: -5.83855403

When the rightward or leftward button is clicked (rightward was clicked here), the camera shifts its focus onto the small blue planet, additionally, displaying its X and Y velocities in the panel screen.

The button in the centre of the two left and right buttons is the pause/play button. It enables the user to toggle between a state of pausing and playing the simulation. When the button is clicked while the simulation is playing, the button changes into one with a triangular Play symbol on it, which, when clicked, would resume the simulation and revert the button to its original state.

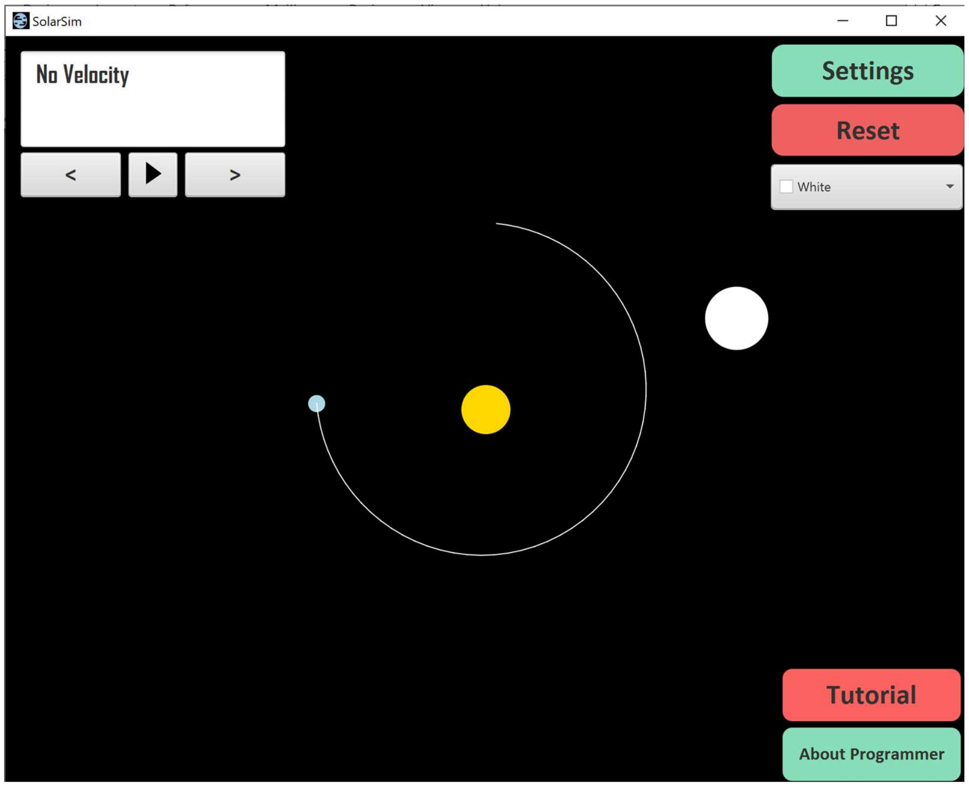


X-Velocity: 5.78066885
Y-Velocity: -8.96226356

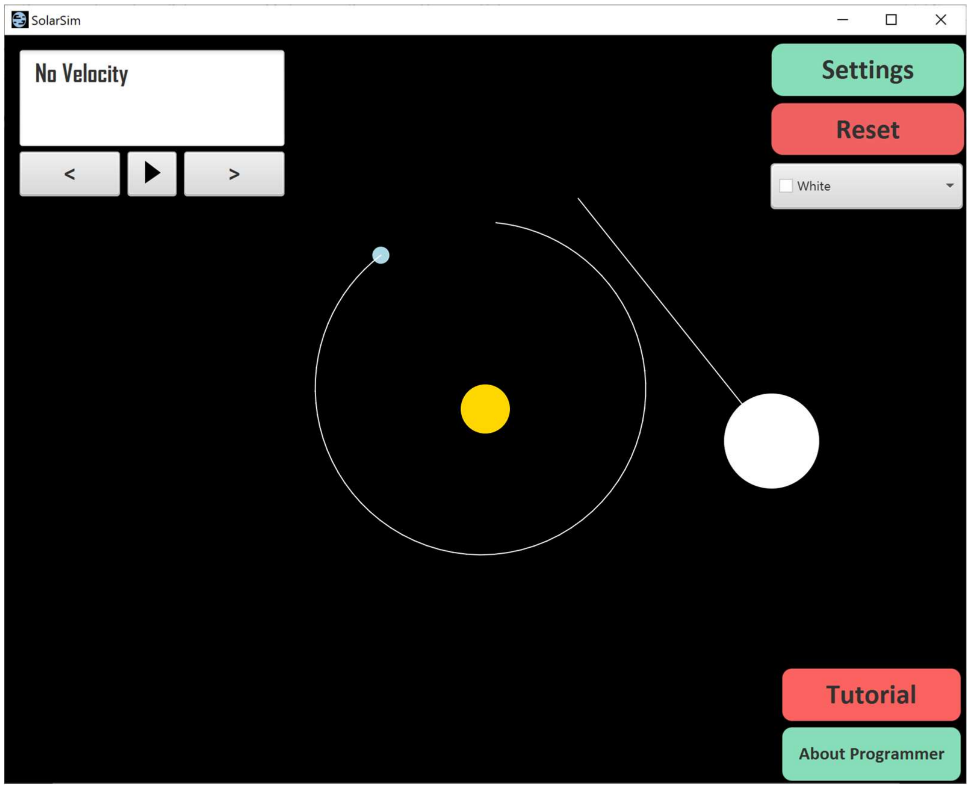
Still, the Tutorial and About Programmer buttons are featured in the main page, which have been discussed earlier already.

In the main simulation, several things can be done to add new celestial bodies. When the user clicks and hold their left or primary mouse button, a new planet would be created. This planet's radius would be set to the distance between the initial point of the user's click and the point at which the mouse button is released. Note that the mass of the planet, which affects its movements and how it exerts "forces" on the other celestial bodies, is scaled depending on the size, which depends on the radius.

In the below image, the primary mouse button was clicked at the centre of the bright white circle near the right of the window, and then dragged to the circumference of the bright white circle, creating the bright white circle as shown below, or a planet.



Upon releasing the primary mouse button, a line is drawn between the user's mouse and the centre of the planet. This line determines the planet's initial velocity, with its angle determining the planet's velocity direction and the length determining the magnitude of the velocity.

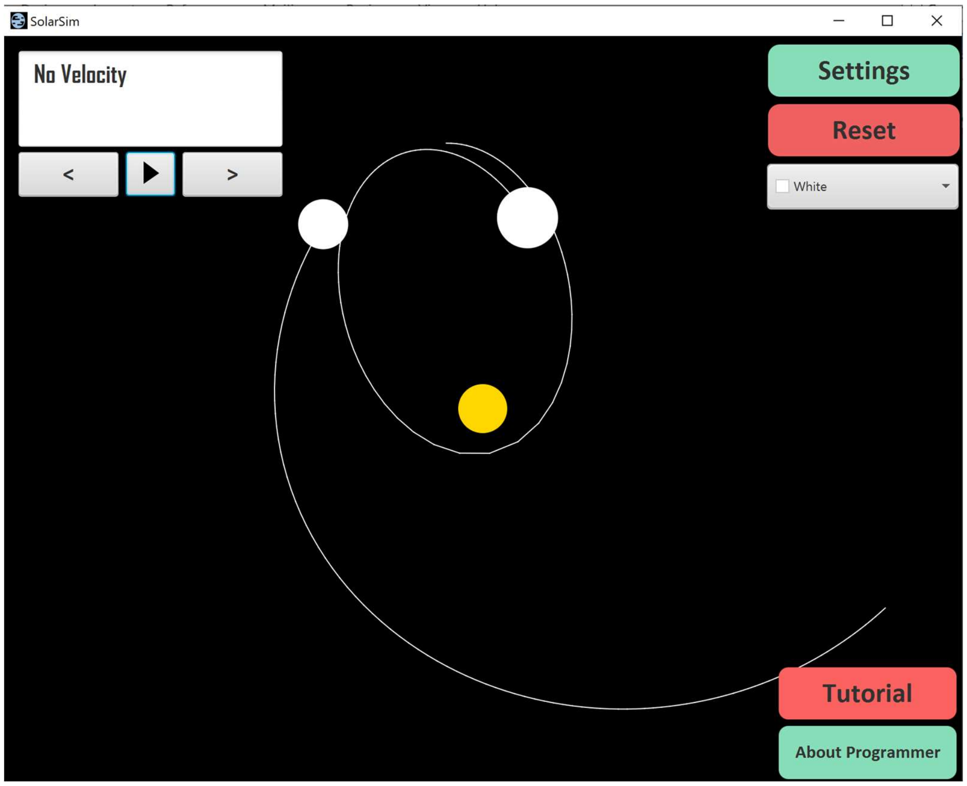


In the example shown above, the velocity line is directly aimed at the top of the window, and spans approximately 3 times the length of the planet itself.

Note that during this entire process of creating a planet, the simulation is in a paused state, reflected by the Pause/Play button in the panel.

Once the planet is released, it is allowed to move and influence the movement and orbit of the other planets. This process of the planet's movement is tracked by a trail that follows behind the planet, matching the planet's path, and disappearing after a while.

An example is shown below.



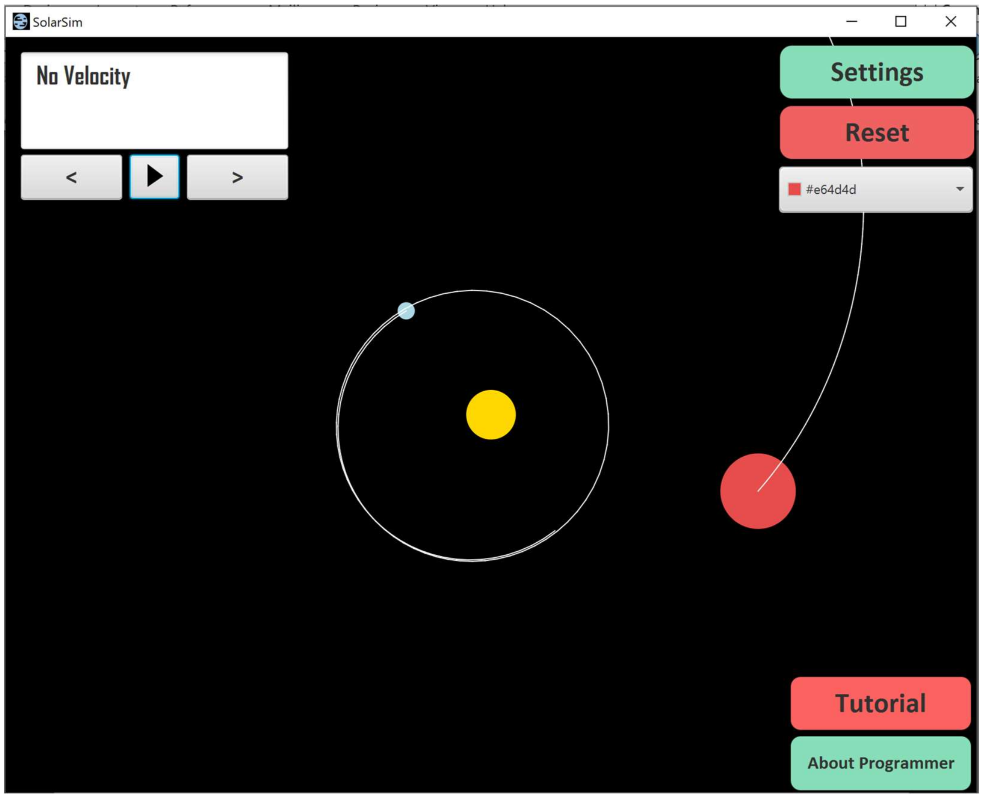
On the top right of the simulation, three controls can be seen. From the bottom to the top, these are:

1. A color picker. The color picker allows the users to select a color, which is then shown on the color picker, and reflected when the user creates a new planet. The new planet will be of the color chosen by the user in the color picker.

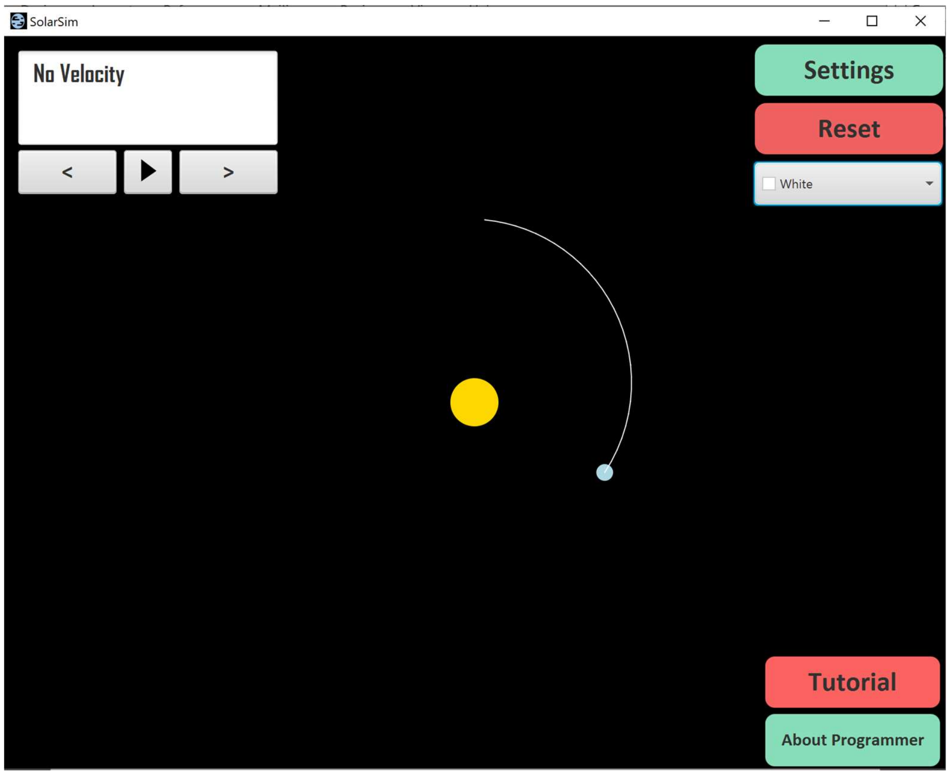
In the example below, the vivid red color is picked in the color picker, represented by the hex value #e64d4d.

This is reflected in the newly created planet orbiting around the star.

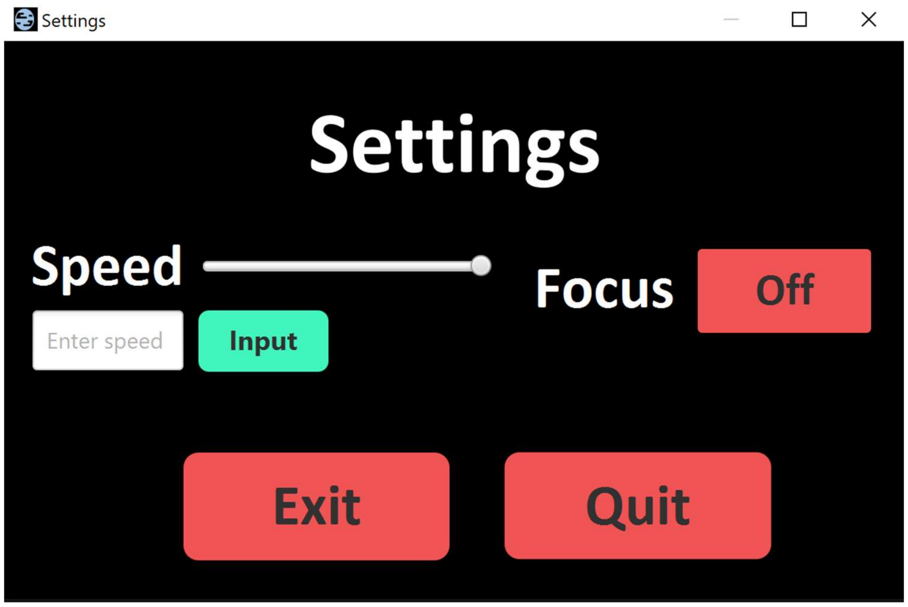
Note that by default, the color white is selected.



2. A vivid red Reset button. This button, when clicked, reverts the simulation back to its original state, or the same state the user will observe when they launch the simulation.

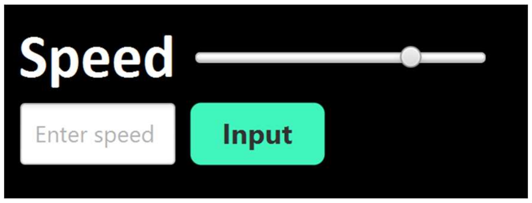


3. A green Settings button. This button, when clicked, will open up a Settings page.

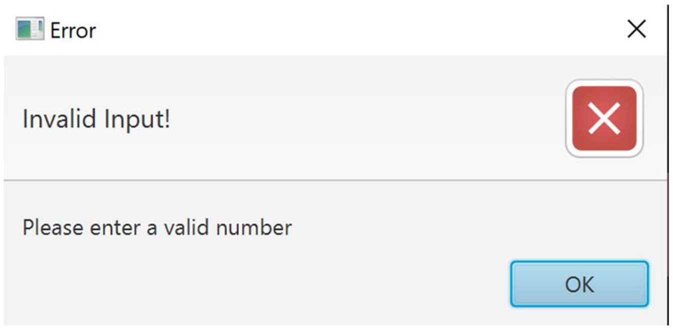


At the left side, a "Speed" label along with a slider, set to the maximum value, can be observed. This slider allows the user to adjust the speed of the simulation, or how fast the celestial bodies move about in the simulation. However, for a more controlled adjustment, the simulation also offers a text field, prompting the user to enter a speed value. This speed value must be of a number format. Additionally, the number must be between the values of 0.25 and 1.25, inclusive. Examples of valid text inputs include 1 and 1.21. Upon entering the input, the user must click the green "Input" button next to the text field. If the user enters a valid value, this change will be reflected in the slider above.

In the following example, an input value of 1 is inputted. This clears the text field and adjusts the slider accordingly.



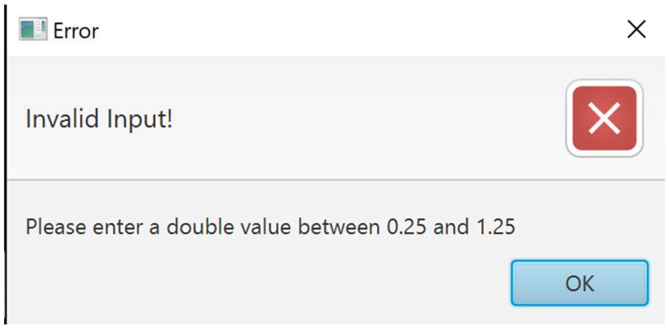
If the user enters an invalid text that is not a number, the following error message is shown.



Invalid Input!

Please enter a valid number

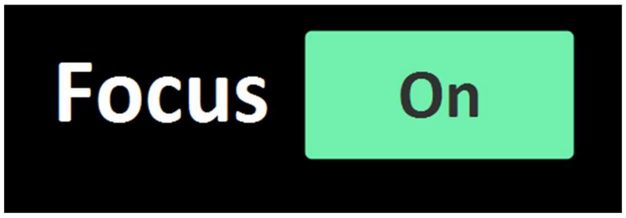
If the user does enter a valid number, but it is not within the range of 0.25 and 1.25 inclusive, the following error message is shown.



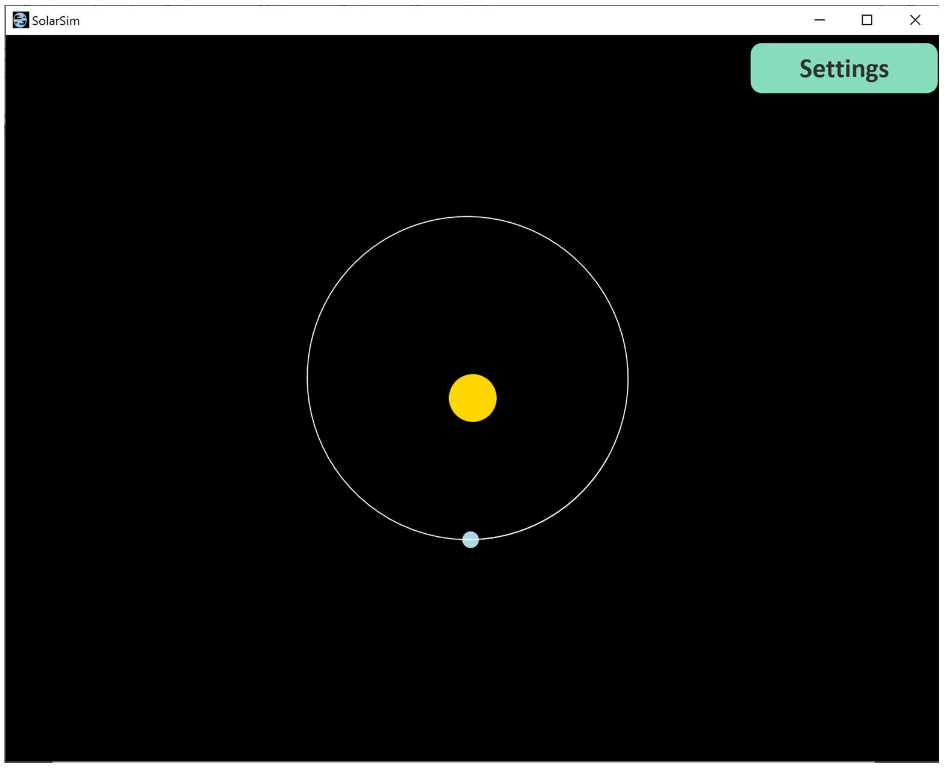
Invalid Input!

Please enter a double value between 0.25 and 1.25

On the right side of the window, a “Focus” labels, along with a toggle button next to it can be seen. This toggle button toggles the simulation between being in a “Focus state” and not being in a “Focus state”, which is reflected by the button. By default, the simulation is not in a “Focus state”. When the button is clicked, however, the button changes to a green color and has the text “On”, indicating that the simulation is now in a “Focus state”.

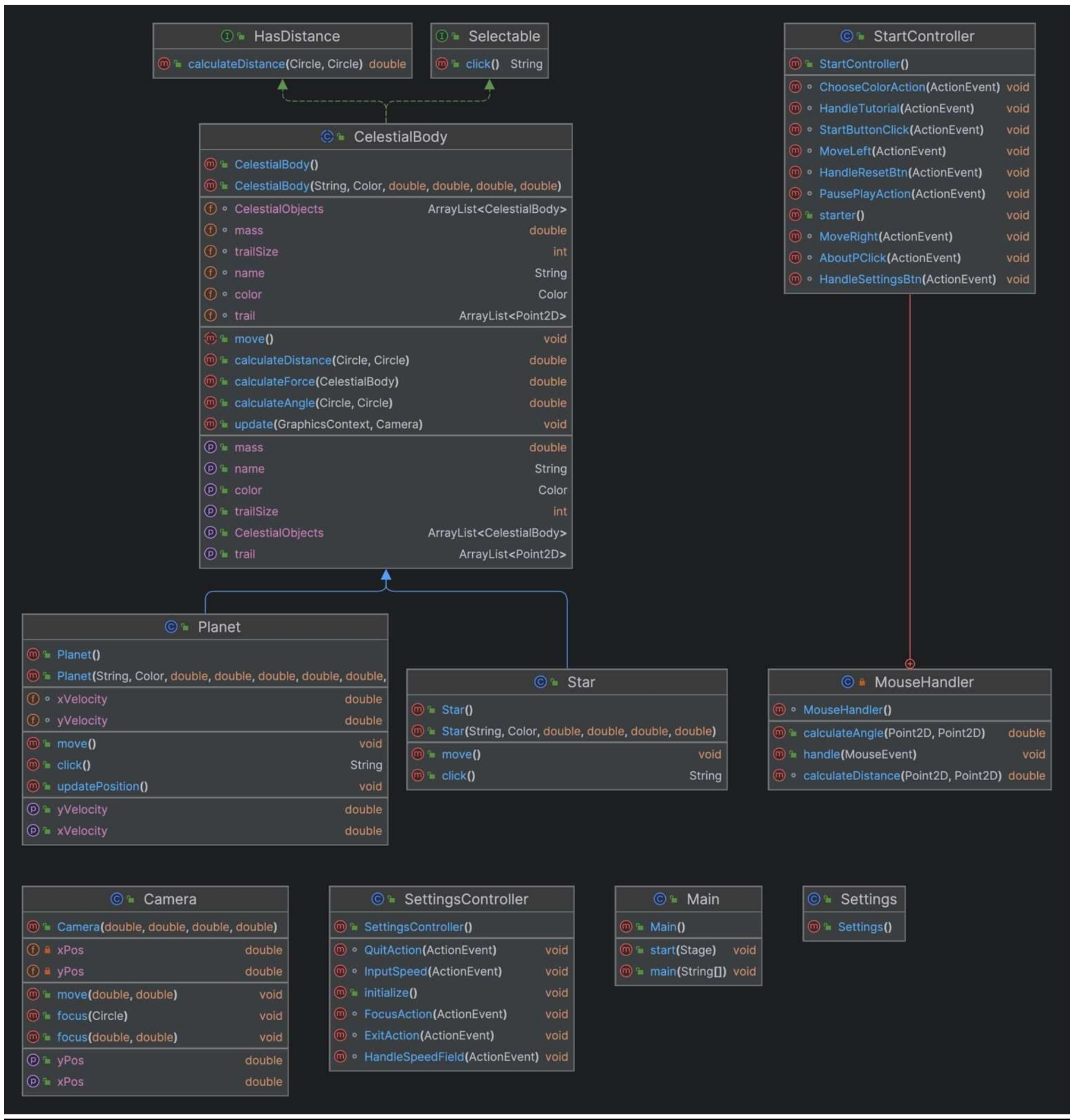


This also makes some changes to the main window, namely, all the buttons but the Settings button becomes invisible, allowing the user to focus on the movement of the celestial bodies.



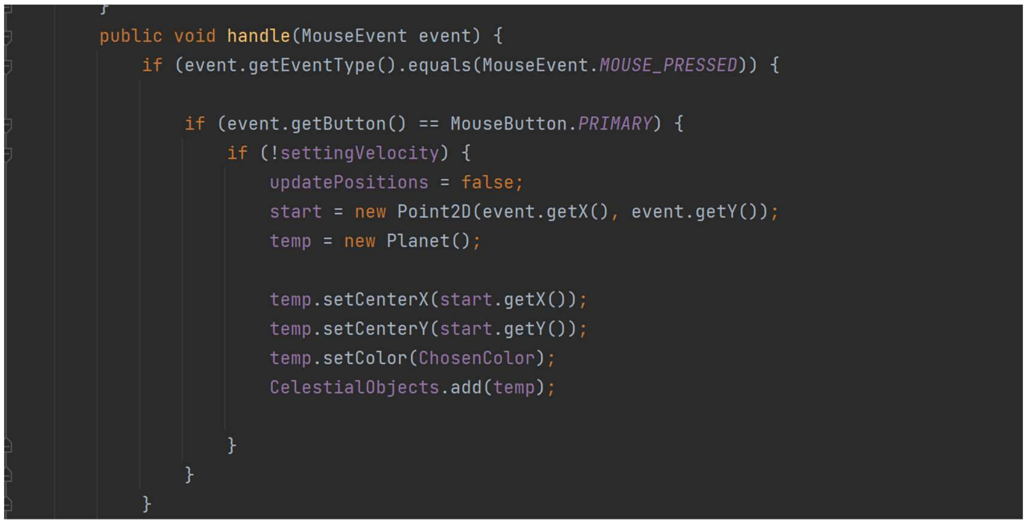
At the bottom, there are two buttons colored a vivid red. These are the “Exit” and “Quit” buttons. When the “Exit” button is clicked, the system closes the Settings page and returns the user to the main window, while the “Quit” button simply closes the program.

Class design


HasDistance
calculateDistance(Circle, Circle) double
Selectable
click() String
StartController
StartController()
ChooseColorAction(ActionEvent) void
HandleTutorial(ActionEvent) void
StartButtonClick(ActionEvent) void
MoveLeft(ActionEvent) void
HandleResetBtn(ActionEvent) void
PausePlayAction(ActionEvent) void
starter() void
MoveRight(ActionEvent) void
AboutPClick(ActionEvent) void
HandleSettingsBtn(ActionEvent) void
CelestialBody
CelestialBody()
CelestialBody(String, Color, double, double, double, double)
CelestialObjects ArrayList<CelestialBody>
mass double
trailSize int
name String
color Color
trail ArrayList<Point2D>
move() void
calculateDistance(Circle, Circle) double
calculateForce(CelestialBody) double
calculateAngle(Circle, Circle) double
update(GraphicsContext, Camera) void
mass double
name String
color Color
trailSize int
CelestialObjects ArrayList<CelestialBody>
trail ArrayList<Point2D>
Planet
Planet()
Planet(String, Color, double, double, double, double, double,
xVelocity double
yVelocity double
move() void
click() String
updatePosition() void
yVelocity double
xVelocity double
Star
Star()
Star(String, Color, double, double, double, double)
move() void
click() String
MouseHandler
MouseHandler()
calculateAngle(Point2D, Point2D) double
handle(MouseEvent) void
calculateDistance(Point2D, Point2D) double
Camera
Camera(double, double, double, double)
xPos double
yPos double
move(double, double) void
focus(Circle) void
focus(double, double) void
yPos double
xPos double
SettingsController
SettingsController()
QuitAction(ActionEvent) void
InputSpeed(ActionEvent) void
initialize() void
FocusAction(ActionEvent) void
ExitAction(ActionEvent) void
HandleSpeedField(ActionEvent) void
Main
Main()
start(Stage) void
main(String[]) void
Settings
Settings()

Explanation of code

Key functionality 1: Ability to draw a new celestial body and set its velocity.



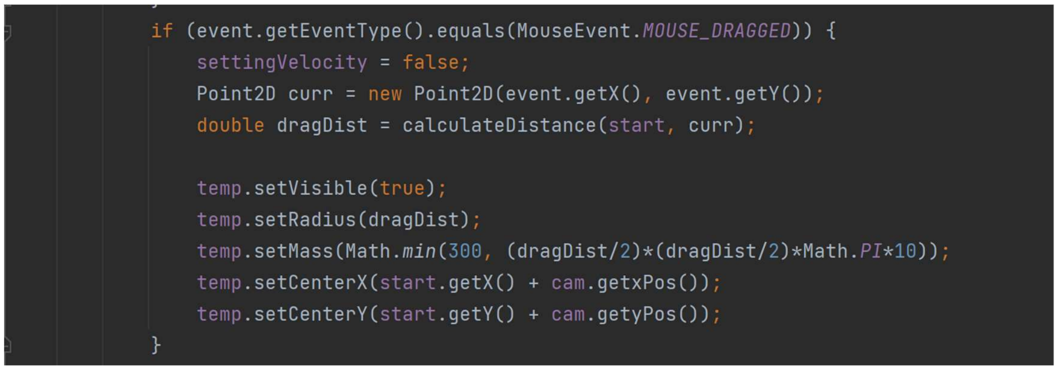
```
    public void handle(MouseEvent event) {
        if (event.getEventType().equals(MouseEvent.MOUSE_PRESSED)) {

            if (event.getButton() == MouseButton.PRIMARY) {
                if (!settingVelocity) {
                    updatePositions = false;
                    start = new Point2D(event.getX(), event.getY());
                    temp = new Planet();

                    temp.setCenterX(start.getX());
                    temp.setCenterY(start.getY());
                    temp.setColor(ChosenColor);
                    CelestialObjects.add(temp);

                }
            }
        }
```

A Mouse Handler is implemented, with different code for different moue actions.

When the mouse button is pressed and the button pressed is the primary one, the program sets the Boolean updatePositions to false, pausing the simulation.

It initializes a starting point based on the cursor's current positions, and initializes temp to a new Planet object, setting it's centre to the point where the mouse button is clicked and its color to the color picked by the user, and adding the planet object to the ArrayList CelestialObjects.

All this is done only when the Boolean value settingVelocity is false, indicating that the user is current creating a new planet, not setting the velocity of another one.



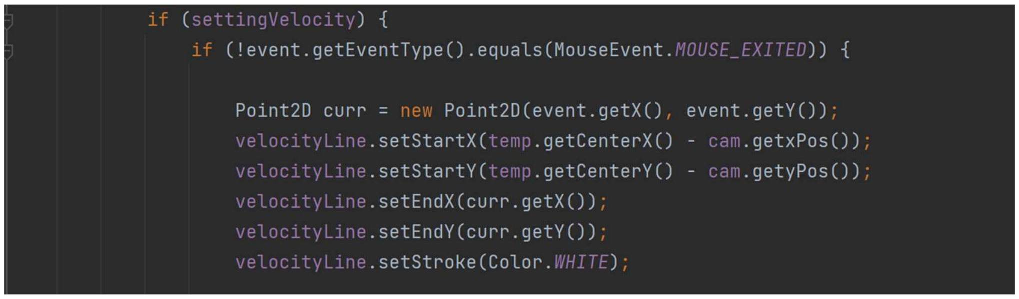
```
        if (event.getEventType().equals(MouseEvent.MOUSE_DRAGGED)) {
            settingVelocity = false;
            Point2D curr = new Point2D(event.getX(), event.getY());
            double dragDist = calculateDistance(start, curr);

            temp.setVisible(true);
            temp.setRadius(dragDist);
            temp.setMass(Math.min(300, (dragDist/2)*(dragDist/2)*Math.PI*10));
            temp.setCenterX(start.getX() + cam.getxPos());
            temp.setCenterY(start.getY() + cam.getyPos());
        }
```

Upon clicking to start creating the planet, the program checks if the user is dragging the mouse cursor about. If they are, the program sets the Boolean value settingVelocity to false to prevent the program from setting the velocity and putting the user in a state of only creating and adjusting the size of a new planet. This is done by initializing a Point2D object curr which is where the mouse cursor currently is.

The temp object is set to be visible, and its radius is set to the distance between the centre of the object and the current mouse cursor position.

Its mass is set to be scaled accordingly to the radius, with a upper limit.

The next two lines allows the program to display the Planet object the user is creating at the position on the window that the user intends, by offsetting the centre of the object by the X Position of the camera and the Y Position of the camera, as the camera constantly moves to stay centred on the celestial objects.



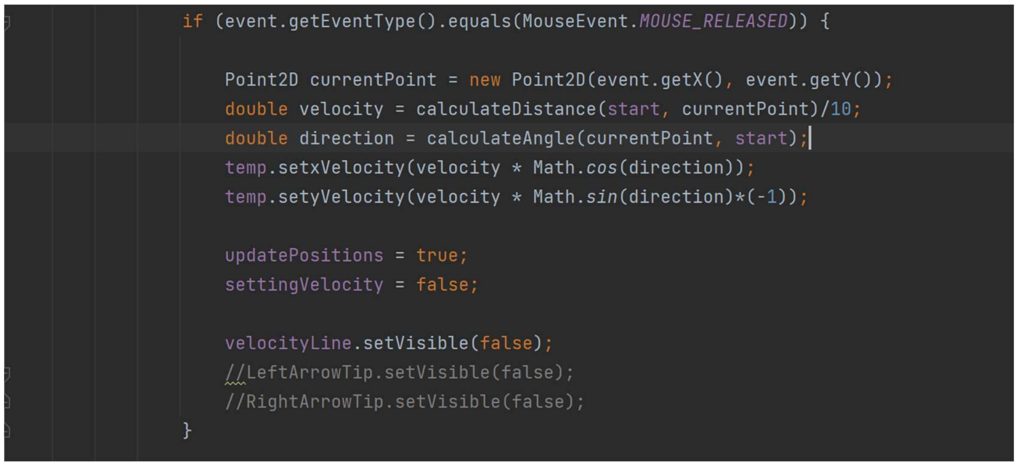
```
if (settingVelocity) {
    if (!event.getEventType().equals(MouseEvent.MOUSE_EXITED)) {

        Point2D curr = new Point2D(event.getX(), event.getY());
        velocityLine.setStartX(temp.getCenterX() - cam.getxPos());
        velocityLine.setStartY(temp.getCenterY() - cam.getyPos());
        velocityLine.setEndX(curr.getX());
        velocityLine.setEndY(curr.getY());
        velocityLine.setStroke(Color.WHITE);
```

Under the same handle() method of the MouseHandler, if the Boolean Value settingVelocity is set to be true, the program understands that the user is currently in the process of setting a velocity for a planet.

The program thus first checks that the mouse is not exited from the window, and initializes a new point curr that corresponds to the current position of the cursor.

The program then adjusts an already initialized Line object called velocityLine, setting its starting position to be the centre of the planet object earlier created, and its end position to be the cursor's current position.

The program also sets the color of the line to be white, allowing the user to see the line clearly, as it is contrasted against the black background.



```java
if (event.getEventType().equals(MouseEvent.MOUSE_RELEASED)) {

    Point2D currentPoint = new Point2D(event.getX(), event.getY());
    double velocity = calculateDistance(start, currentPoint)/10;
    double direction = calculateAngle(currentPoint, start);
    temp.setxVelocity(velocity * Math.cos(direction));
    temp.setyVelocity(velocity * Math.sin(direction)*(-1));

    updatePositions = true;
    settingVelocity = false;

    velocityLine.setVisible(false);
    //LeftArrowTip.setVisible(false);
    //RightArrowTip.setVisible(false);
}
```

Under the same If conditional block, the program contains a handler for if the mouse is released while the velocity is being set. This confirms the velocity that the user has set and applies this velocity to the planet. It does this by initializing a new Point2D object currentPoint, at the point where the mouse button is released.

A double value velocity is then calculated, based on the distance between the starting point, or the centre of the planet and the end point, or the point at which the user has clicked the cursor again, then divided by 10.

The direction is also determined in a similar way, using the calculateAngle() function and inputting both the start and end points. The calculateAngle() function utilises trigonometry to calculate the angle of the velocity.

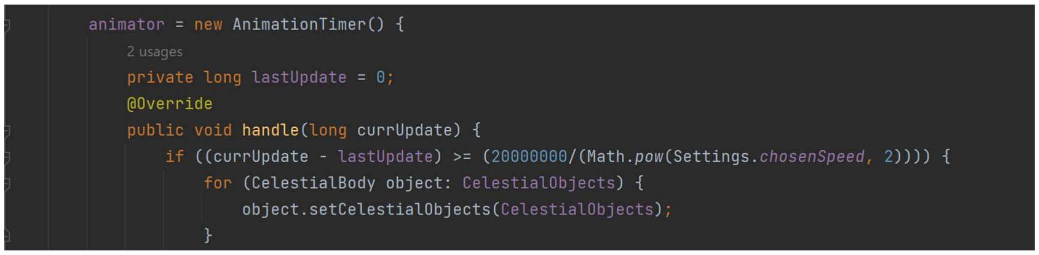
This velocity is then applied to the temp object of type Planet, applying the x component of the velocity by multiplying the velocity's magnitude by the cosine of the direction and then applying the y component of the velocity, by multiplying the magnitude of the velocity by the sine of the direction and multiplying the value by -1.

Under this If conditional block, the Boolean value updatePositions is set again to be true, to indicate that the user has finished setting the velocity, resuming the simulation.

The Boolean value settingVelocity is also set to false, indicating that the user has finished setting the velocity.

Finally, the velocityLine is set to no longer be visible again, as the user has finished setting the velocity and the velocityLine is no longer needed.

Key functionality 2: Planets simulate movement under the influence of the gravitational fields.

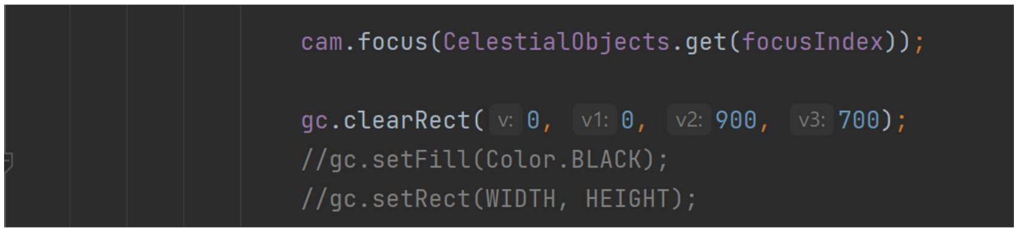


```java
animator = new AnimationTimer() {
    2 usages
    private long lastUpdate = 0;
    @Override
    public void handle(long currUpdate) {
        if ((currUpdate - lastUpdate) >= (20000000/(Math.pow(Settings.chosenSpeed, 2)))) {
            for (CelestialBody object: CelestialObjects) {
                object.setCelestialObjects(CelestialObjects);
            }
```

An AnimationTimer() object is initialized, called animator. This animator updates the simulation based on the time. The handle() method takes in a long value, which represents the current time in long value. The difference between this value and the lastUpdate value, which was the time of the previous update, is used to update the simulation.

If this difference is larger than or equals to a certain value determined by the selected speed of the simulation, the animation driving the simulation is updated, starting by setting the CelestialObjects ArrayList in every object that is currently in the simulation to be the ArrayList that represents the objects in the simulation.

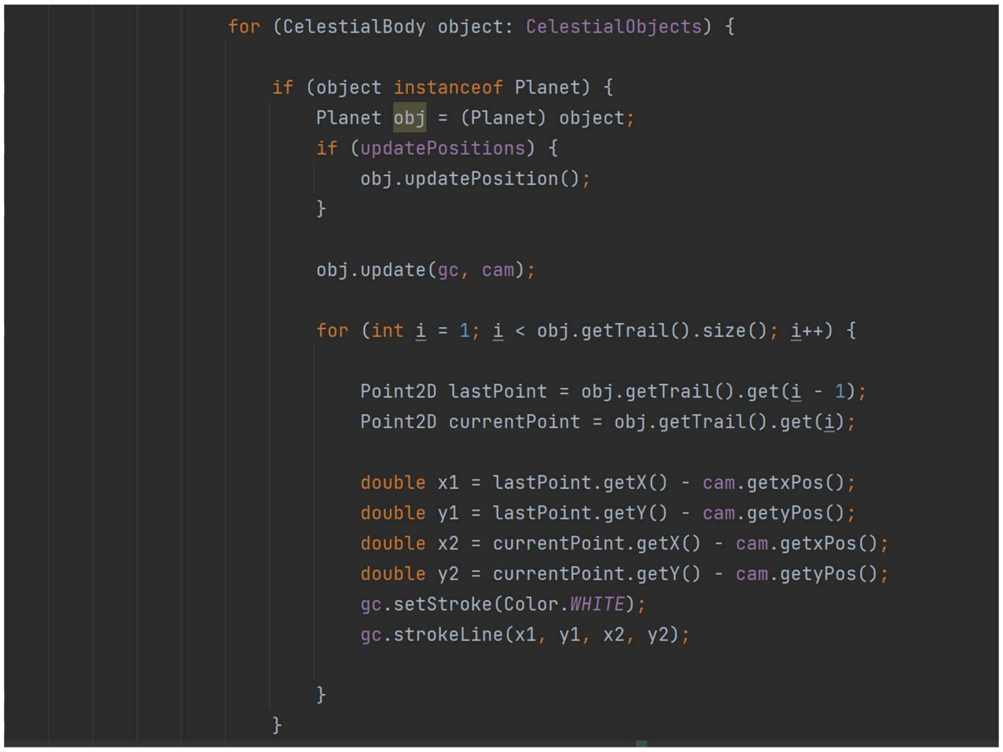
This allows all the objects to understand that there are other objects in the simulation, allowing them to simulative movement under the influence of these objects.



```java
cam.focus(CelestialObjects.get(focusIndex));

gc.clearRect( v: 0,  v1: 0,  v2: 900,  v3: 700);
//gc.setFill(Color.BLACK);
//gc.setRect(WIDTH, HEIGHT);
```

Under the same If conditional block, the Camera object cam is set to focus on the CelestialObject object that is at the index focusIndex in the CelestialObjects ArrayList. This focusIndex can be changed by the user, and is initially set to 0.

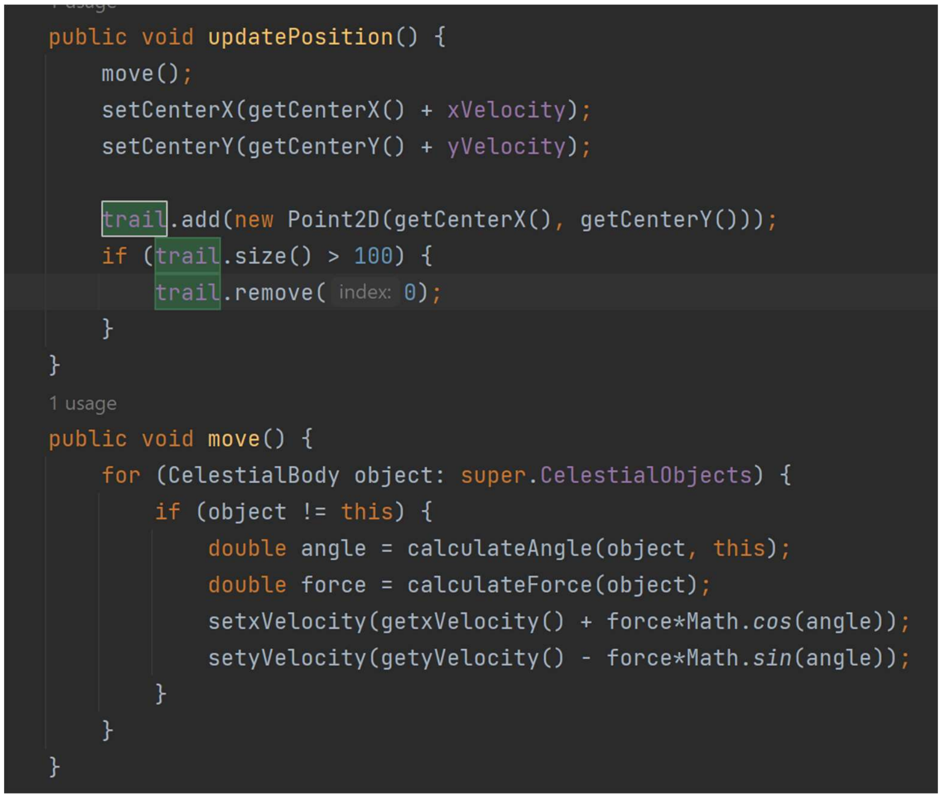
gc is a GraphicsContext object, which allows the program to change and set elements, like the black background of the program. gc.clearRect() clears all the elements in the window, like the CelestialObject. This allows for the redrawing of the elements to reflect the simulation's updates.



```
for (CelestialBody object: CelestialObjects) {

    if (object instanceof Planet) {
        Planet obj = (Planet) object;
        if (updatePositions) {
            obj.updatePosition();
        }

        obj.update(gc, cam);

        for (int i = 1; i < obj.getTrail().size(); i++) {

            Point2D lastPoint = obj.getTrail().get(i - 1);
            Point2D currentPoint = obj.getTrail().get(i);

            double x1 = lastPoint.getX() - cam.getxPos();
            double y1 = lastPoint.getY() - cam.getyPos();
            double x2 = currentPoint.getX() - cam.getxPos();
            double y2 = currentPoint.getY() - cam.getyPos();
            gc.setStroke(Color.WHITE);
            gc.strokeLine(x1, y1, x2, y2);

        }
    }
```

The program then cycles through the CelestialObjects ArrayList. For each CelestialObject object in the ArrayList, it checks whether the object is an instance of the Planet class, which indicates that the object is not fixed and can move, unlike a Star object.

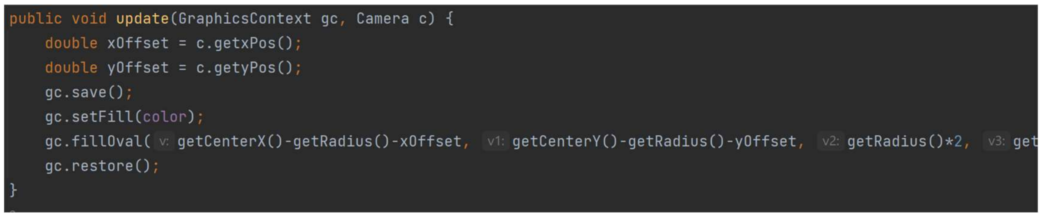
The program then updates the position of the object by calling the updatePosition() instance method of the Planet class, provided that the Boolean value updatePositions is true, indicating that the simulation is not in a paused state.



```
public void updatePosition() {
    move();
    setCenterX(getCenterX() + xVelocity);
    setCenterY(getCenterY() + yVelocity);

    trail.add(new Point2D(getCenterX(), getCenterY()));
    if (trail.size() > 100) {
        trail.remove( index: 0);
    }
}
1 usage
public void move() {
    for (CelestialBody object: super.CelestialObjects) {
        if (object != this) {
            double angle = calculateAngle(object, this);
            double force = calculateForce(object);
            setxVelocity(getxVelocity() + force*Math.cos(angle));
            setyVelocity(getyVelocity() - force*Math.sin(angle));
        }
    }
}
```

The updatePosition() method is shown briefly above. Notable things are that it relies on a move() method, which calculates the angle and force in comparison to the other CelestialObject objects, and adjusts its own velocity accordingly.

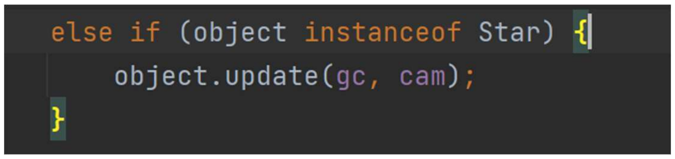
Back to the main controller, the program calls update(), with GraphicsContext Camera objects as input parameters. The update() method simply draws the planet object into the GraphicsContext, using the Camera to position it properly.



```
public void update(GraphicsContext gc, Camera c) {
    double xOffset = c.getxPos();
    double yOffset = c.getyPos();
    gc.save();
    gc.setFill(color);
    gc.fillOval( v: getCenterX()-getRadius()-xOffset,  v1: getCenterY()-getRadius()-yOffset,  v2: getRadius()*2,  v3: get
    gc.restore();
}
```

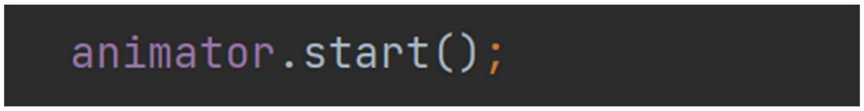
The program then cycles through the getTrail() of the object, which returns an ArrayList of Point2D objects, indicating the previous points of the object, and thus using it to create a path on the GraphicsContext, by drawing a white line between each point of the trail.

The Camera object is used to compute and position the x and y values of the points properly.



```java
else if (object instanceof Star) {
    object.update(gc, cam);
}
```

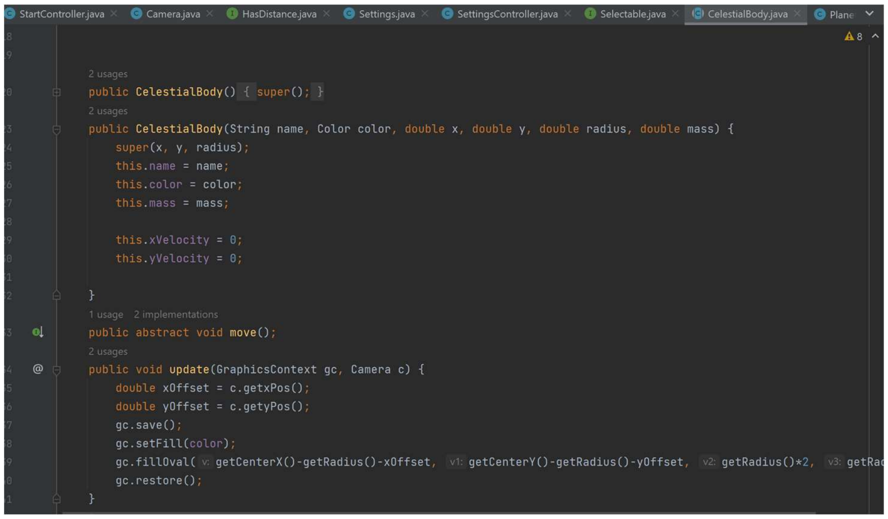
Otherwise, if the object is an instance of the Star class, the update() method to update the GraphicsContext is simply called, and the updatePosition() method is not called, as the Star objects do not move and thus the Star class does not have an updatePosition() instance method.



```java
animator.start();
```

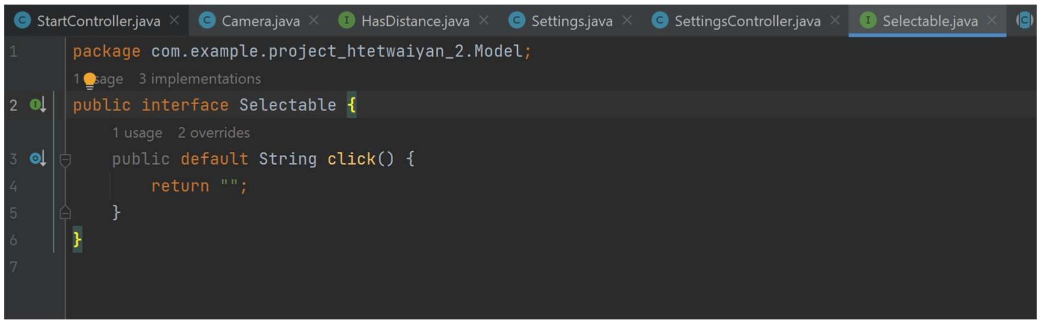
The animator's start() method is then called to start the animator, playing the simulation.

Minimum requirements



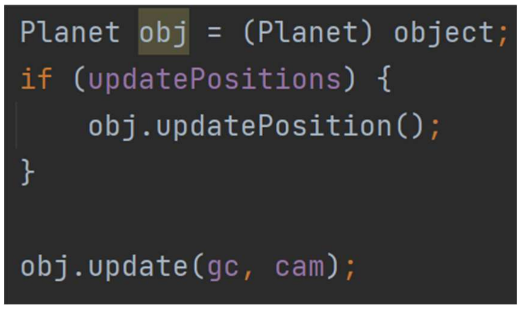
```java
    2 usages
    public CelestialBody() { super(); }
    2 usages
    public CelestialBody(String name, Color color, double x, double y, double radius, double mass) {
        super(x, y, radius);
        this.name = name;
        this.color = color;
        this.mass = mass;

        this.xVelocity = 0;
        this.yVelocity = 0;

    }
    1 usage   2 implementations
    public abstract void move();
    2 usages
    public void update(GraphicsContext gc, Camera c) {
        double xOffset = c.getxPos();
        double yOffset = c.getyPos();
        gc.save();
        gc.setFill(color);
        gc.fillOval( v: getCenterX()-getRadius()-xOffset,  v1: getCenterY()-getRadius()-yOffset,  v2: getRadius()*2,  v3: getRa
        gc.restore();
    }
```

CelestialBody is an abstract class not already in the Java library, and it contains the abstract method move(). The CelestialBody class is thoroughly used in this project.



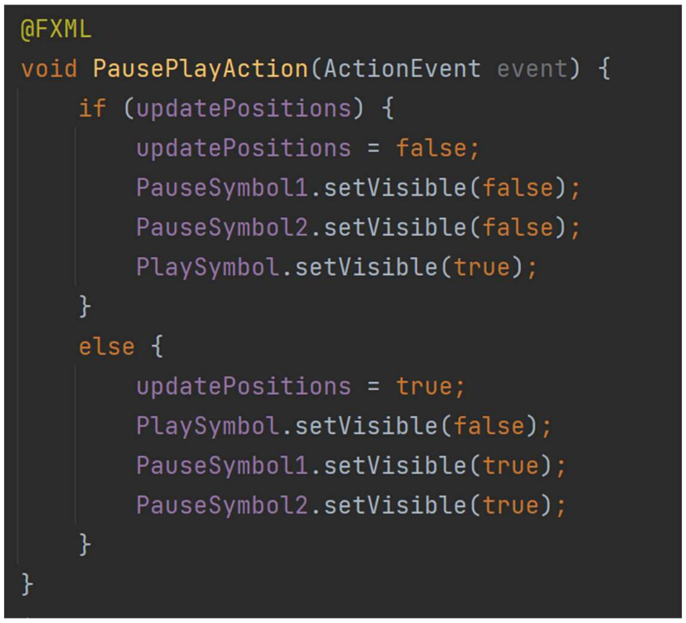
```java
package com.example.project_htetwaiyan_2.Model;
public interface Selectable {
    public default String click() {
        return "";
    }
}
```

Selectable is an interface not already in the Java library, and contains the method click(). The Selectable interface is thoroughly used in this project.



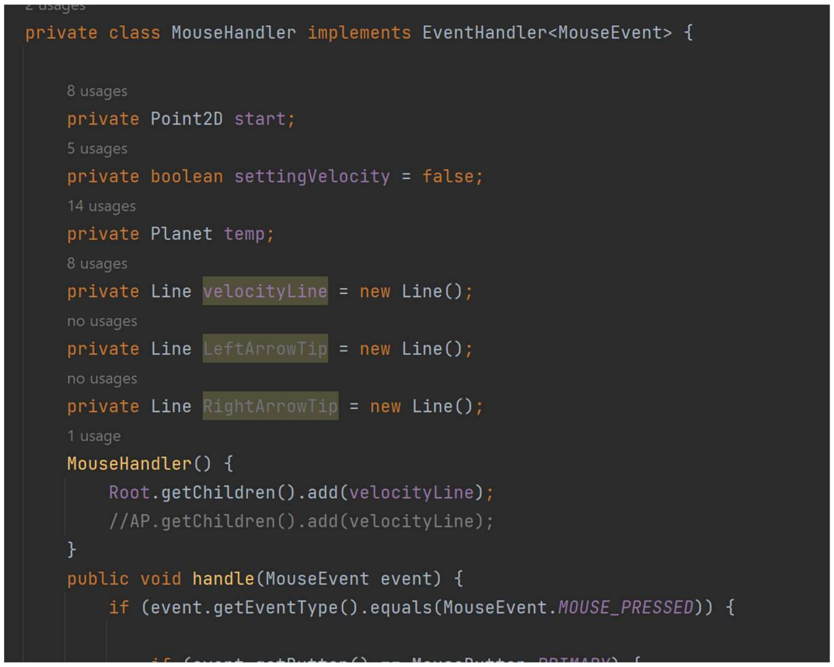
```java
Planet obj = (Planet) object;
if (updatePositions) {
    obj.updatePosition();
}

obj.update(gc, cam);
```

The update() method is used on Planet objects, and the update() method is only seen in the abstract class CelestialBody that Planet extends from. This is an example of inheritance, where the update() method is inherited.



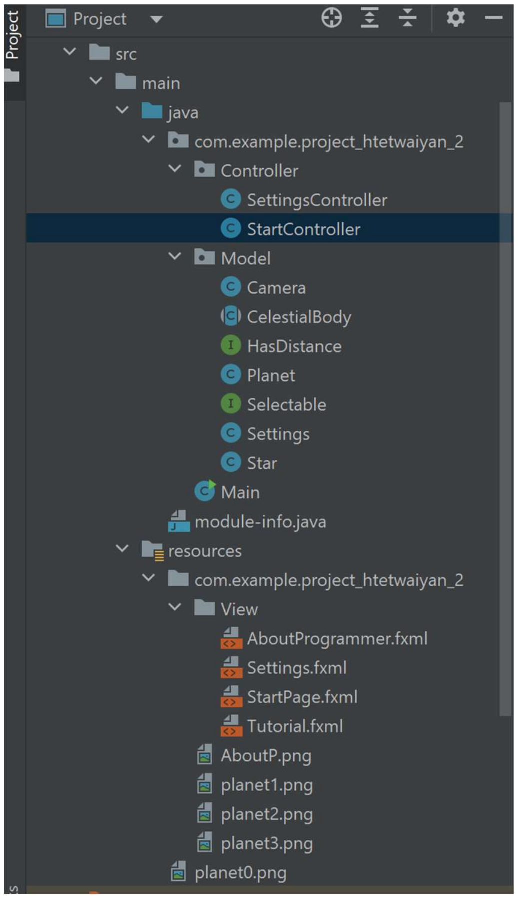
```java
@FXML
void PausePlayAction(ActionEvent event) {
    if (updatePositions) {
        updatePositions = false;
        PauseSymbol1.setVisible(false);
        PauseSymbol2.setVisible(false);
        PlaySymbol.setVisible(true);
    }
    else {
        updatePositions = true;
        PlaySymbol.setVisible(false);
        PauseSymbol1.setVisible(true);
        PauseSymbol2.setVisible(true);
    }
}
```

The function shown on the left is linked to a button in FXML, and upon clicking the button, which is an event, the simulation pauses or resumes the animation. This is a clear example of event handling on User Interface components.



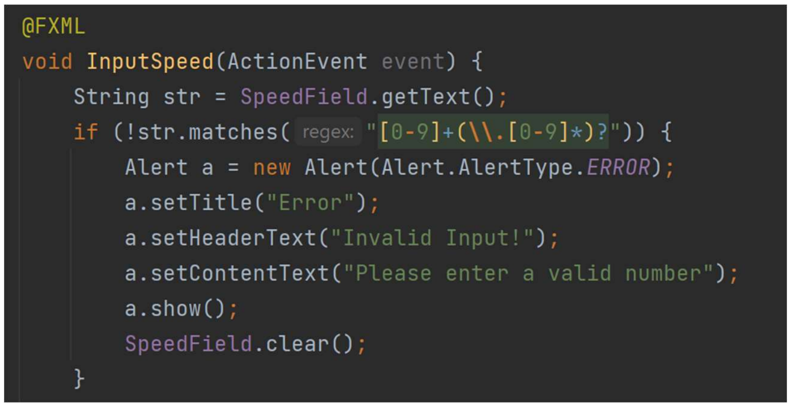
```
private class MouseHandler implements EventHandler<MouseEvent> {

    8 usages
    private Point2D start;
    5 usages
    private boolean settingVelocity = false;
    14 usages
    private Planet temp;
    8 usages
    private Line velocityLine = new Line();
    no usages
    private Line LeftArrowTip = new Line();
    no usages
    private Line RightArrowTip = new Line();
    1 usage
    MouseHandler() {
        Root.getChildren().add(velocityLine);
        //AP.getChildren().add(velocityLine);
    }
    public void handle(MouseEvent event) {
        if (event.getEventType().equals(MouseEvent.MOUSE_PRESSED)) {
```

A class MouseHandler shown above that implements EventHandler is used to handle mouse events from the user on the User Interface. This is a clear example of mouse interactivity from the user.



```
Project
src
  main
    java
      com.example.project_htetwaiyan_2
        Controller
          SettingsController
          StartController
        Model
          Camera
          CelestialBody
          HasDistance
          Planet
          Selectable
          Settings
          Star
        Main
      module-info.java
    resources
      com.example.project_htetwaiyan_2
        View
          AboutProgrammer.fxml
          Settings.fxml
          StartPage.fxml
          Tutorial.fxml
        AboutP.png
        planet1.png
        planet2.png
        planet3.png
      planet0.png
```

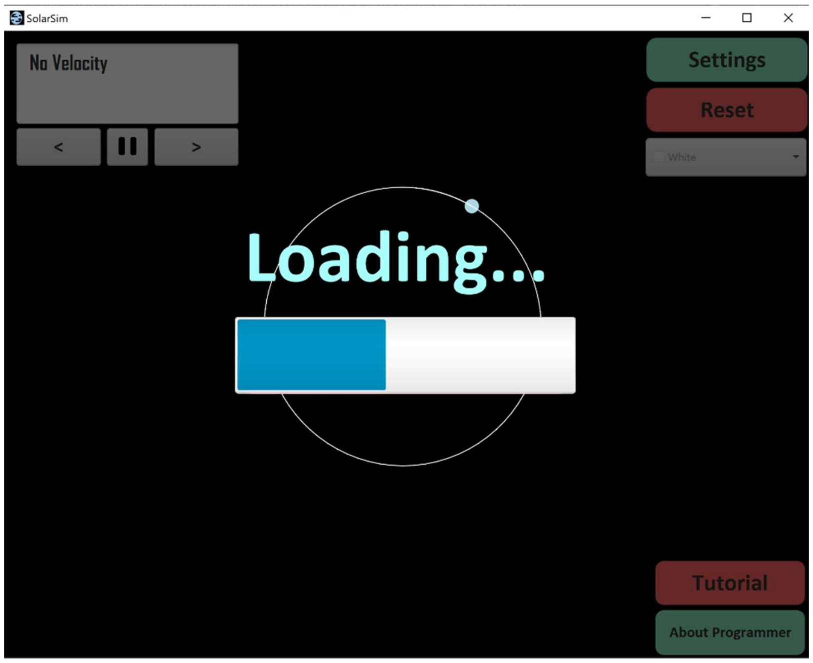
The controller classes, model classes, and FXML files are clearly separated under packages titled Controller, Model, and View. This is a clear implementation of the MVC (Model-View-Controller) structure.



```
@FXML
void InputSpeed(ActionEvent event) {
    String str = SpeedField.getText();
    if (!str.matches( regex: "[0-9]+(\\.[0-9]*)?")) {
        Alert a = new Alert(Alert.AlertType.ERROR);
        a.setTitle("Error");
        a.setHeaderText("Invalid Input!");
        a.setContentText("Please enter a valid number");
        a.show();
        SpeedField.clear();
    }
```

Here, in the controller class that controls the Settings page, a button event includes regex for checking of the input. This regex checks for a proper number, that either comes in the form of only digits or digits followed by a more digits separated only by a single decimal.

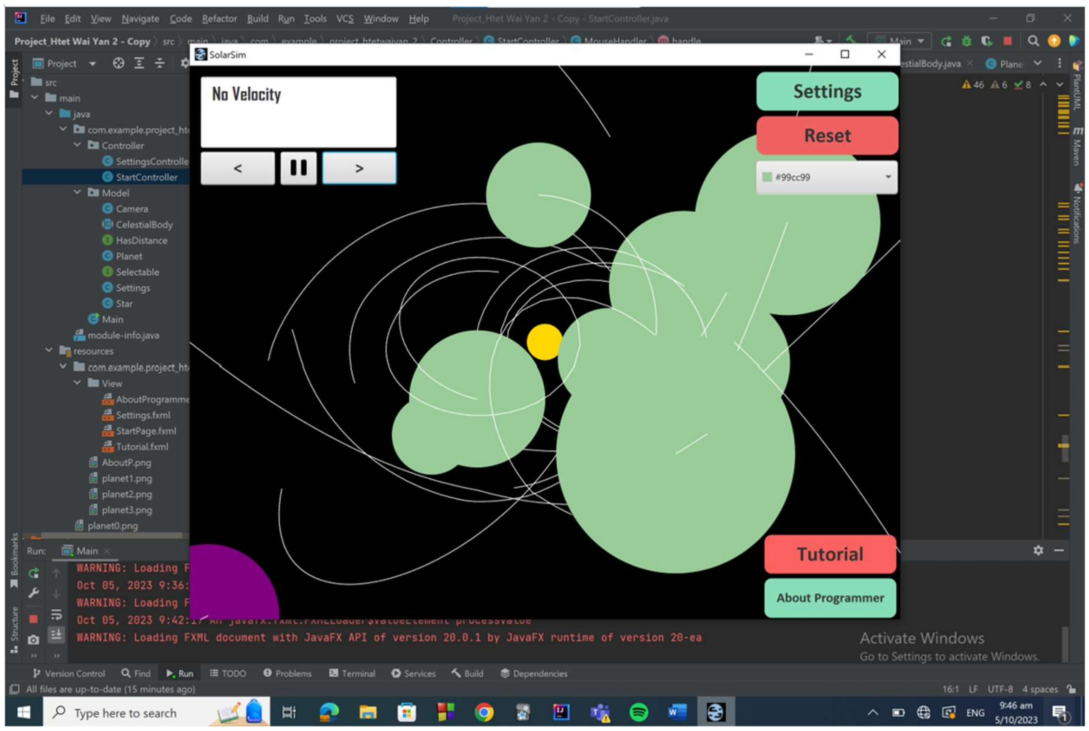
Here, regex is clearly used in validation checking.



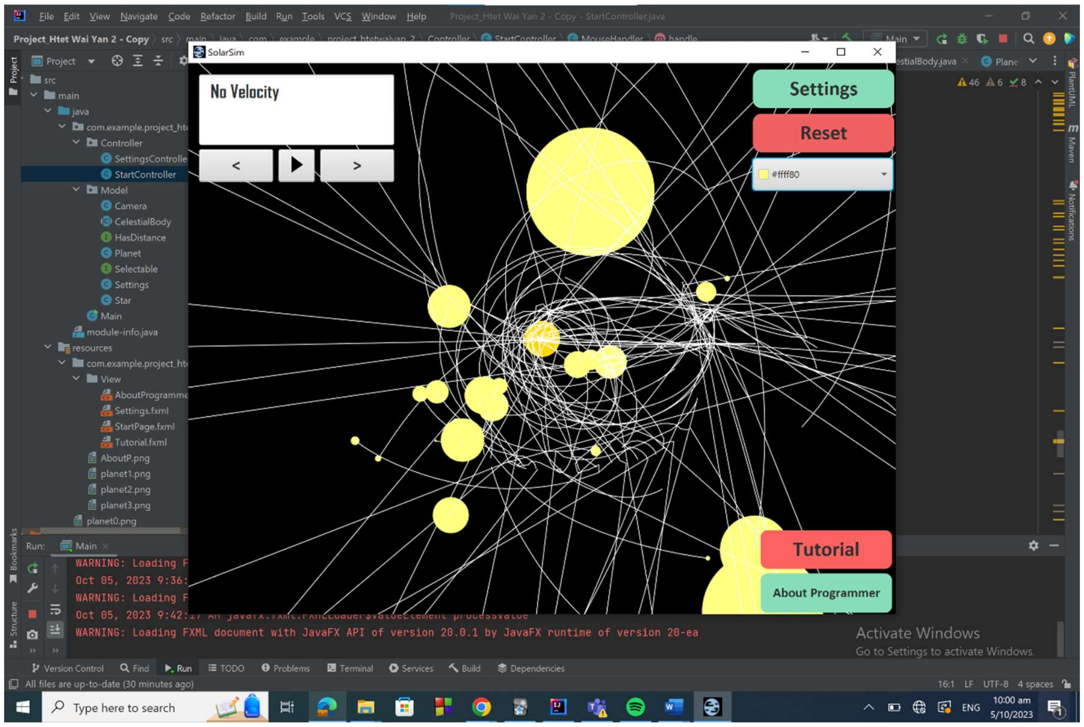
Here, a splash screen is clearly shown when the simulation is started. It includes a blue circle rotating about a label that says “Loading…” and a progress bar that updates with a range of random values. The components of the main window are shown as well, dimmed.

Testing Methodology and Results

Black box testing was done on the simulation effectively, by describing the simulation to a friend who proceeded to attempt to “crash” the program by adding tons of planets. Nonetheless, the simulation was still able to simulate the movement of all the planets properly.



```
WARNING: Loading F
Oct 05, 2023 9:36:
WARNING: Loading F
Oct 05, 2023 9:42:17 AM javafx.fxml.FXMLLoader$ValueElement processValue
WARNING: Loading FXML document with JavaFX API of version 20.0.1 by JavaFX runtime of version 20-ea
```

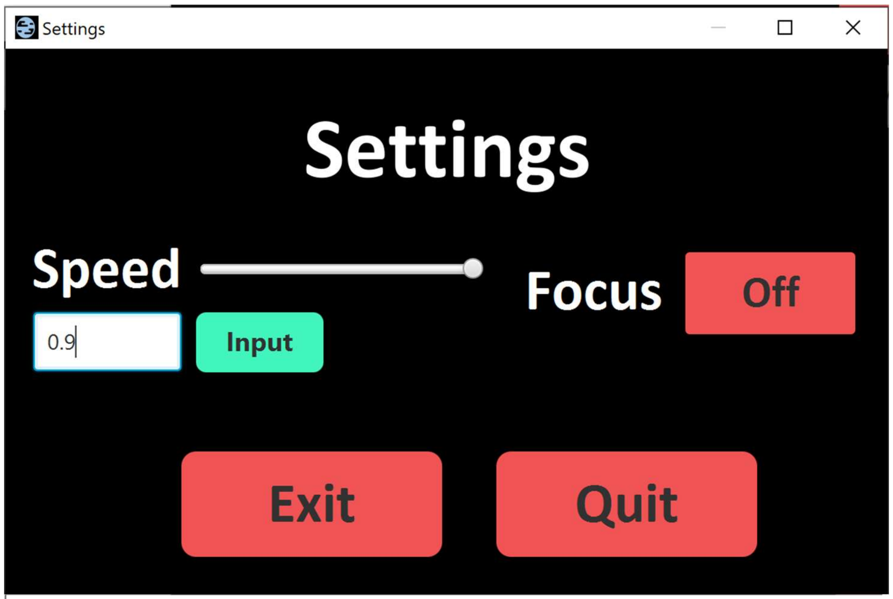


```
WARNING: Loading F
Oct 05, 2023 9:36:
WARNING: Loading F
Oct 05, 2023 9:42:17 AM javafx.fxml.FXMLLoader$ValueElement processValue
WARNING: Loading FXML document with JavaFX API of version 20.0.1 by JavaFX runtime of version 20-ea
```

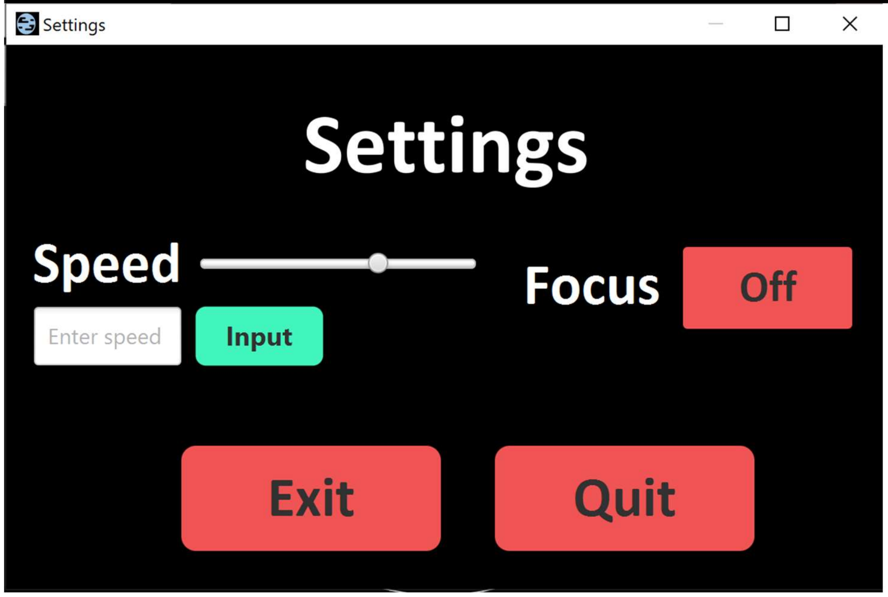
In the above example, the friend spent about 30 minutes continuously adding planets, estimating that there would be about 400 planets in motion at the same time. Nonetheless, the program continued to run well.

This test was a test of the program being able to handle several hundred instances of Planet objects being animated without crashing or producing too much lag. The program was able to continue animating all these objects.

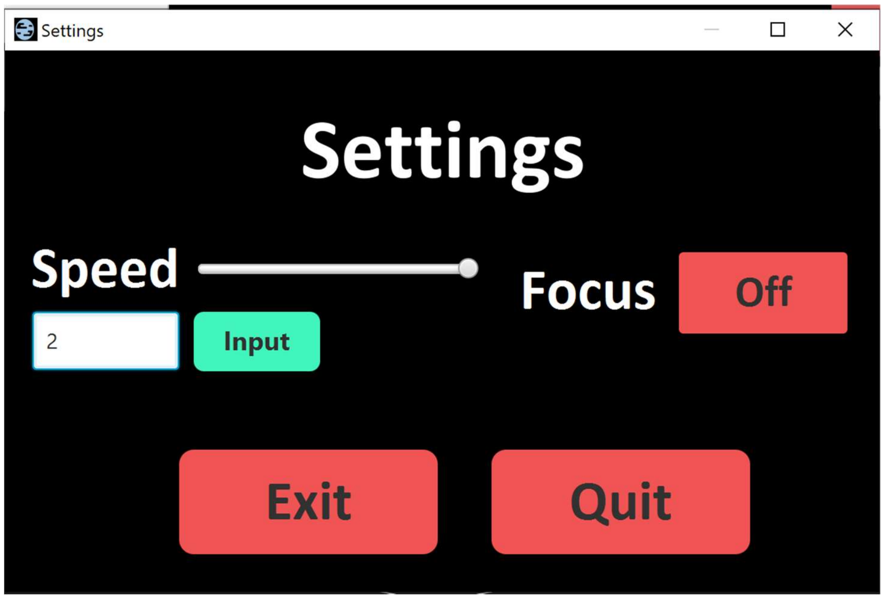
This was a stress test case.

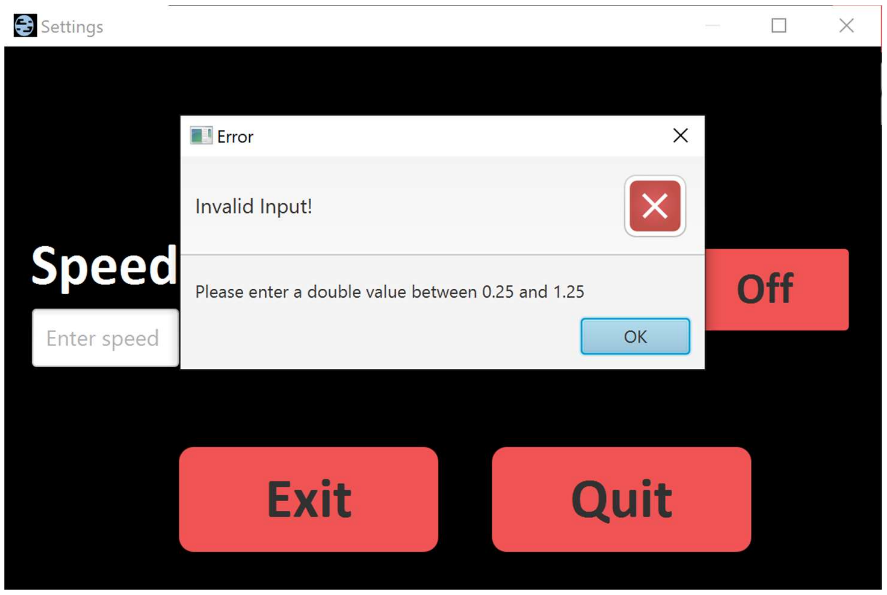


0.9



In the test above, knowing that the Text Field to input a custom speed only allowed for values ranging from 0.25 to 1.25 inclusive, a test case was conducted to ensure that this was working as intended. 0.9, a valid value was entered in the first test. It was expected to work without errors, and it did. The speed slider automatically adjusted to the value 0.9, and in the backend, a variable was updated to be 0.9 as well.





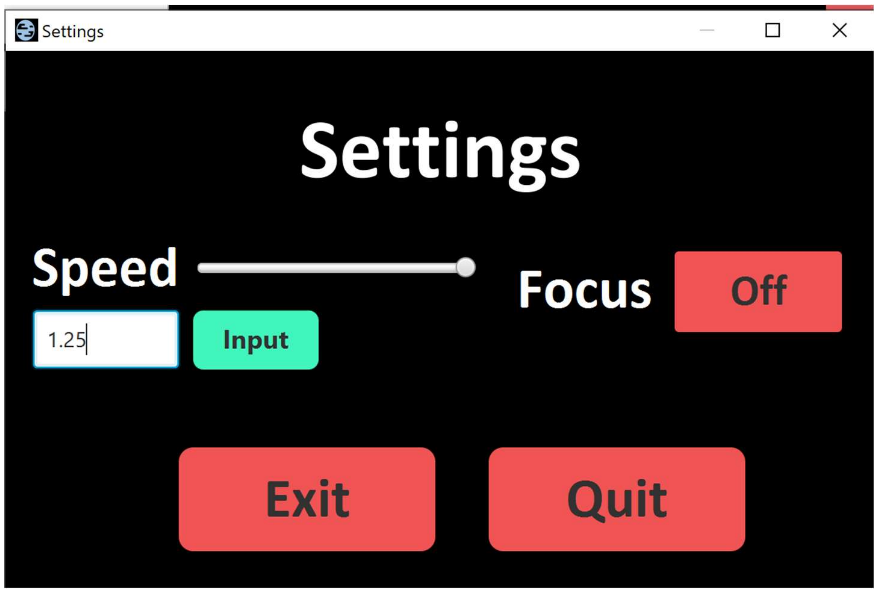
Invalid Input!

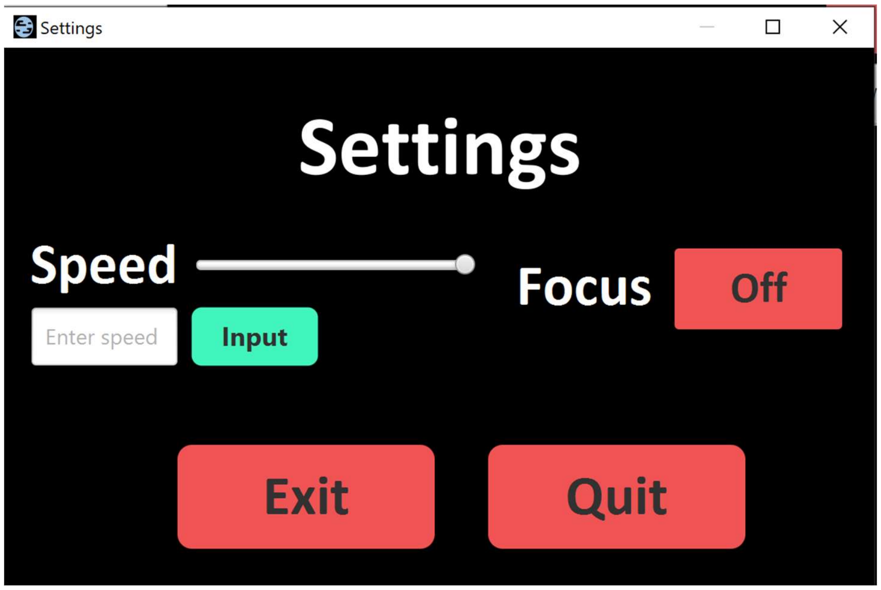
Please enter a double value between 0.25 and 1.25

In the second test, a value of 2 was inputted, and the program was expected to show an alert dialog box. Indeed, the alert dialog box with the right message and type popped up.

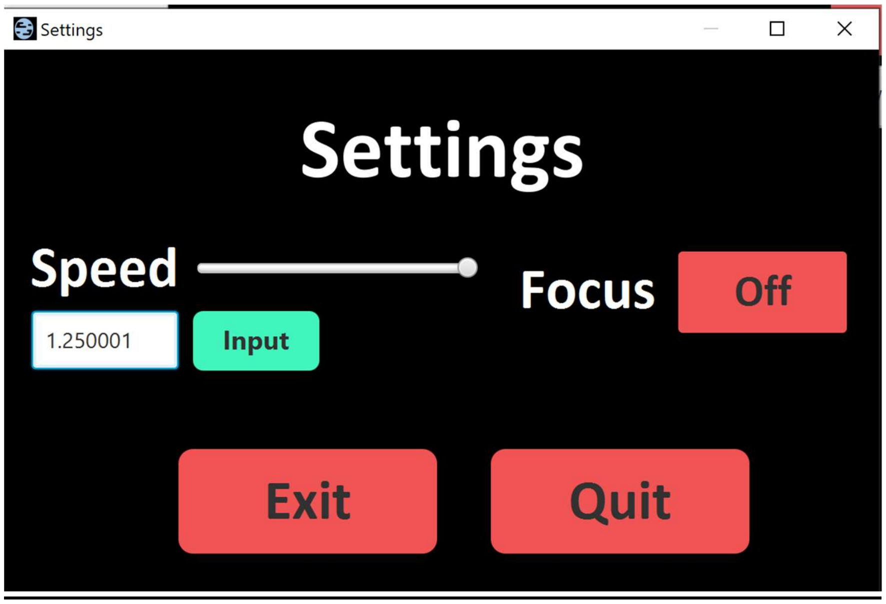
These two tests were tests of correct input validation checking.

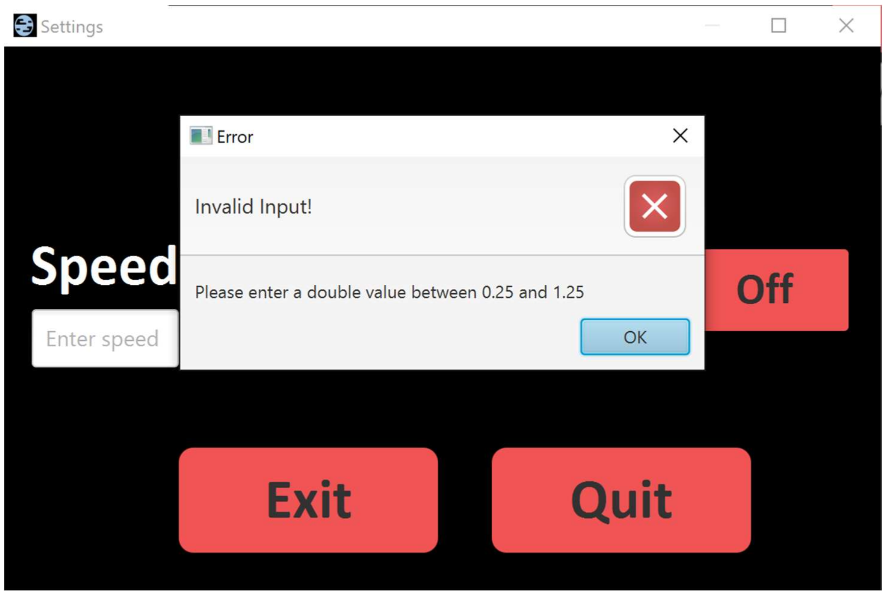
These two tests are the Correct Use Test Case and Incorrect Use Test Case.





In the above test, knowing that the text field only accepted inputs between 0.25 and 1.25 inclusive, a value of 1.25 was inputted. This value, as shown above, was accepted by the program, as was the expected result.





Invalid Input!

Please enter a double value between 0.25 and 1.25

Then, a value of 1.250001 was inputted, which is barely over the boundary of accepted inputs. The program handles this correctly, showing the appropriate alert dialog.

This was the Boundary Value Test Case.

Reflections

Through doing the project, I learnt how to create a “camera” in JavaFX that could follow one object about, and switch its focus to another object. This idea proved to be very useful for my project especially since my project is based off a simulation, and in simulations with animations, a camera that can dynamically move about is very useful. I had initially thought that being able to implement a working, dynamic camera in JavaFX would not be possible, but since learning that objects drawn and animated in JavaFX are not just limited to being active in the window, and can continue to be animated and drawn, even outside the window. This feature was also what allowed me to learn and implement a camera that shifts its focus based on the coordinate system in JavaFX.

Furthermore, I gained a greater understanding of “physics in JavaFX” through this project. This project required me to implement working physics in a simulation, which required me to utilize various mathematical concepts such as trigonometry to calculate the proper angle for the velocity.

However, I did face some difficulties in that simulating what was effectively an N-Body simulation in JavaFX was quite complex and I had to spend some time thinking about how I was going to implement it properly. However, once I had planned it all out properly, the physics side of this project was not as much of a difficulty as I had initially anticipated.

With more time given, I could have been able to make the simulation much more scientifically accurate, which would be more beneficial in educating curious individuals about the physics of orbiting celestial bodies. This more scientifically accurate simulation would be able to account for much more factors to produce a more accurate simulation, beyond just allowing curious individuals to visualize the movement of celestial bodies under the influence of other celestial bodies.

This project could also be made more user friendly, allowing users to “save” their simulations that they have built and export saved simulations to visualize in this project.

Citations

IntelliJ Ultimate, for the producing of the UML Diagram (https://www.jetbrains.com/idea/buy/?section=commercial&billing=yearly)

Oracle Docs, Graphics Context tutorial (https://docs.oracle.com/javase/8/javafx/api/javafx/scene/canvas/GraphicsContext.html)

IntelliJ IDEA Community Edition 2022.3, for the designing and coding of the program (https://www.jetbrains.com/edu-products/download/other-IIE.html)

OBS Studio, 64 bit, for the recording of the video presentation (https://obsproject.com/download)

Microsoft Word, version 2309, for the writing of the documentation

Wang Yifan, for being one of my project's black box testers

CS3233 OOP II and CS3231 OOP I notes, for guidance in this project