420-204-RE

Vanier College

THE SEA MEN

An educative simulation game using kinematics concepts.

Project by

Rayane Bouafia, Oussama Bouhenchir and Philippe Nikolov

Work presented to Parth Shah

2022-05-03

CONTENTS

User Interface 3

Menu Screen 3

Level Selection Screen 3

Shop Screen 4

In-Game 4

Implementation Part-1: Algorithm and Application Logic 6

Projectile Movement 6

Collision Logic 8

Implementation Part-2: Integration of Individual Parts 9

Implementation Part-3: Project Reports 11

- Project Description 11
- Wireframes 11
- Diagrams 14
  - Use Case Diagram 14
  - Class Diagram 15
- Program Features and Screenshots 16
- Team Responsibilities 16
- Challenges 17

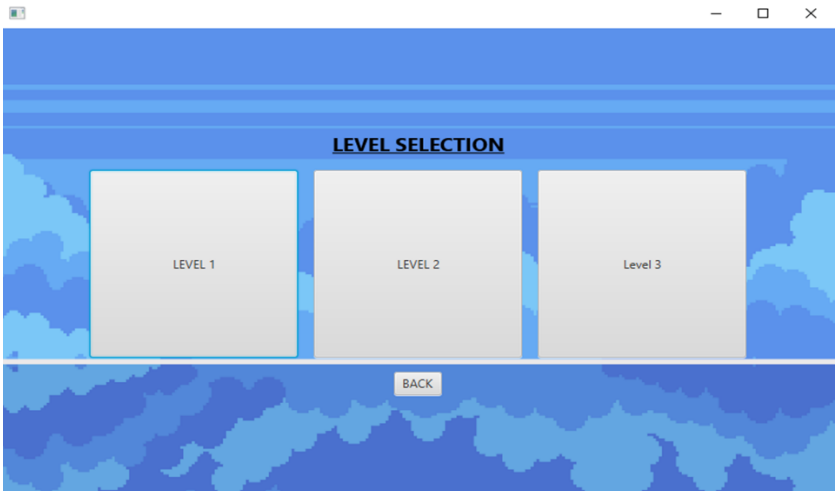
User Interface

Menu Screen

First, we have the main menu screen which appears when you open the game. There are two options here: Play and Options. Pressing the play button brings you to the level selection screen. Pressing the options button brings up the options screen.

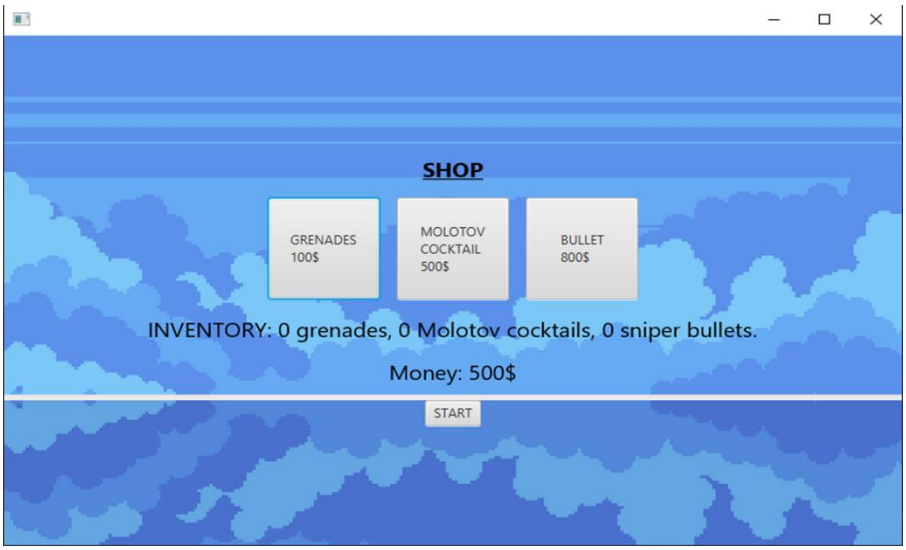
Level Selection Screen

In the level selection screen, we can either pick a level or go back to the main menu by pressing the back button. At first, only level 1 is available to the player, the other two level buttons are disabled. You can only access the next level by beating the level before it.



Shop Screen

In the shop, you can buy items to use in-game. You simply click on the item's icon to buy it. The player's money and inventory are managed by playerBankAndInventory.java. There are three items you can buy: grenades, Molotov cocktails and bullets. Each projectile has different properties. The grenades do more damage, the Molotov cocktails have a bigger hitbox and do less damage, and the bullet goes in a straight line because we gave it a high initial velocity and is more precise than the other projectiles, but it is also more expensive. The shop is there to add a level of strategy to the game, as players now must strategically manage their money to win. The player can press the "Start" button at any time to start the game on the level they selected.



In-Game

In-game, a screen shows the player's ship on the left and the enemy's ship on the right, separated by a long distance. There is a vector that starts at the player's position and ends at the user's cursor, following it. The user's cursor determines the initial velocity and initial angle of the shot. When the player clicks the mouse, the projectile is launched according to the laws of projectile motion. The goal is to hit the enemies until they have zero health points.

(Example from level 1)

(Example from level 3)

Implementation Part-1: Algorithm and Application Logic

This section explains the logic of the in-game part of the application, explanations on all classes and their utilities in Implementation Part-2.

Projectile Movement

- When moving the mouse, a vector starting from the player ship and ending to the cursor indicates the starting velocity of the projectile. This is done using a mouse movement event handler that constantly activates the code refreshing the vector when the mouse is moved.

```
root.setOnMouseMoved((MouseEvent cursorLocation) -> {

                if (cursorLocation.getSceneX() <= 175 && cursorLocation.getSceneY() >= 175)
{

…
```

As you can see from this code, we added a limit to how far the user can move his mouse and edit the initial velocity line, it would be excessive and unbalanced for the user to be able to shoot at an extremely high velocity.

When the player clicks (shoots), an event handler is used, and the x and y components of this vector are stored and then used to calculate the trajectory. The animation of the projectile is coded within this event handler.

We used the class Timeline for animating the parabolic trajectory of the projectile. Using keyframes that represent the starting and final coordinates of the projectile (using equation 1 and 2 for X to find the final coordinates) and an interpolator that constantly calculates the updated position in the y-axis using the kinematics equations.

In this interpolator we implement equation 2 for Y in a curve() method.

```
KeyValue xKV = new KeyValue(c.centerXProperty(), 75+xDistance); //xdistance is the calculated range of the projectile, to find its final coordinate in x.

KeyValue yKV = new KeyValue(c.centerYProperty(), 100, new Interpolator() {

                    @Override

                    protected double curve(double t) {

                    return -4*t*t + yComponent/10*t;

            }

    });

KeyFrame xKF = new KeyFrame(Duration.millis(2000), xKV);

KeyFrame yKF = new KeyFrame(Duration.millis(2000), yKV);

//c is the projectile
```

Here is an idea of the use of the keyframes and interpolator. The logic works perfectly, but we still must tweak the scaling of values like the acceleration and the velocity to further to balance the game better.

For different projectiles, the velocity is scaled differently.

The keyframes are then added to the TimeLine which is played.

X	**Y**
I. $v_x = v_{0x} + a_x t$	I. $v_y = v_{0y} + a_y t$
II. $x = x_0 + v_{0x} t + \frac{a_x}{2} t^2$	II. $y = y_0 + v_{0y} t + \frac{a_y}{2} t^2$
III. $v_x^{\,2} = v_{0x}^2 + 2a_x(x - x_0)$	III. $v_y^{\,2} = v_{0y}^2 + 2a_y(y - y_0)$

Collision Logic

Using If…Then statements, if the projectile hitbox coordinates are within the enemy or player hitbox coordinates (hitboxes are simply a defined range of x and y coordinates for each level), collision is detected, and the enemy loses HP.

Example:

If (c.centerXProperty() >= enemy1.getX() && c.centerYProperty >= enemy1.getY() ...

Each projectile has different properties. The normal projectiles are infinite and free but have the smallest hitbox and damage. The grenades have a bigger hitbox and deal more damage. The Molotov cocktail deals less damage but has the biggest hitbox. The bullet is the most precise.

When an enemy is hit, a health point bar appears and shows the enemy's remaining health.When all enemies' health points are depleted, the level is passed. Originally, we wanted to have a more extensive collision system that included rebound using vectors, etc. More on this in Challenges, below.

Implementation Part-2: Integration of Individual Parts

The project is divided into multiple Java classes.

SceneController.java exists to create all our scenes as public methods that return the created scene to the main class when called. This is the solution we found to pass the various scenes through the main class and make them reusable. For example, if we want to change the scene to the level selection screen in the main class, we will write:
stage.setScene(SceneController.levelSelection());

ButtonDatabase.java exists to store the different buttons on all scenes as private static variables that are used in the main class via getters, making them usable at any stage of the runtime. Declaring the buttons in the SceneController class led to being unable to make event handlers in the main class because the buttons were limited to that class. Also, storing them in a class meant we could not edit their properties (.prefSize() etc.) (since they're not in a method). We solved this issue by editing their properties every time a getter is used on them., ex:

```
public static Button getOptionButton() {

        ImageView options = new
ImageView("https://cdn.discordapp.com/attachments/898709510154227752/972641206527995935/options-export.png");

    optionButton.setGraphic(options);

    return optionButton;

  }
```

Project.java is our main class. Using event handlers, such as this:

```
    ButtonDatabase.getLevel3Button().setOnAction(new EventHandler<ActionEvent>() {

        @Override

        public void handle(ActionEvent event) {
```

```
            SceneController.selectedLevel = 3;
            stage.setScene(SceneController.shop());
            stage.show();
        }
    });
```

This example code handles the event that the user selects the level 3 button in the level selection menu, this will change the scene to the shop and store the selected level as an integer for future reference when the player presses the start button. The in-game level scenes are the only ones that are not in the scene controller class, they are all made in the main class, because they are constantly being updated, whereas the other ones are not. All logic and algorithm implementation is done within this class with the use of event handlers.

playerBankAndInventory.java is where we store and manage transactions of the player's money, and the items in his inventory. All this data is kept static, which means it resets every time we run the program. When an item is purchased, money from the static class is deducted and one item is added. There are three items the user can buy: grenades, Molotov cocktails and bullets. Since the game is short and meant to be beaten in one sitting, a saving feature, or any kind of database were not considered or added.

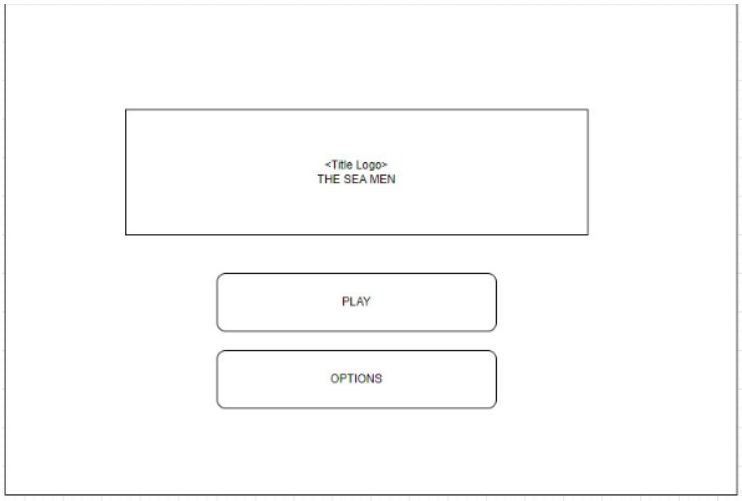
Implementation Part-3: Project Reports

Project Description

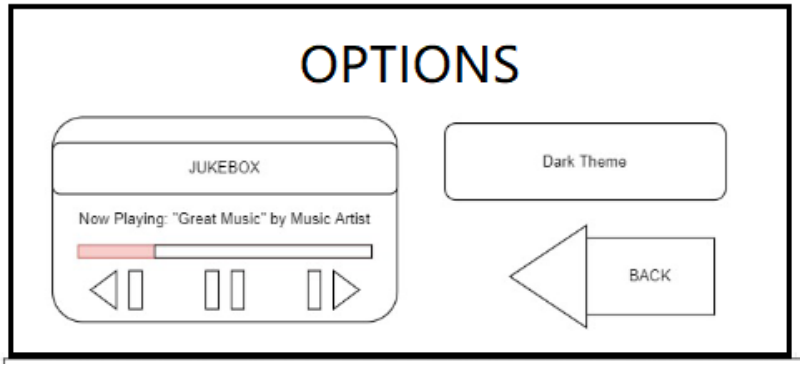
Our integrative project is an educative game/physics simulation.
The project is called “The Sea Men.” It is a game where the player must shoot cannonballs accurately to defeat the enemies on the other side of the map. It uses concepts of projectile motion and kinematics to simulate a real-life projectile.

Wireframes

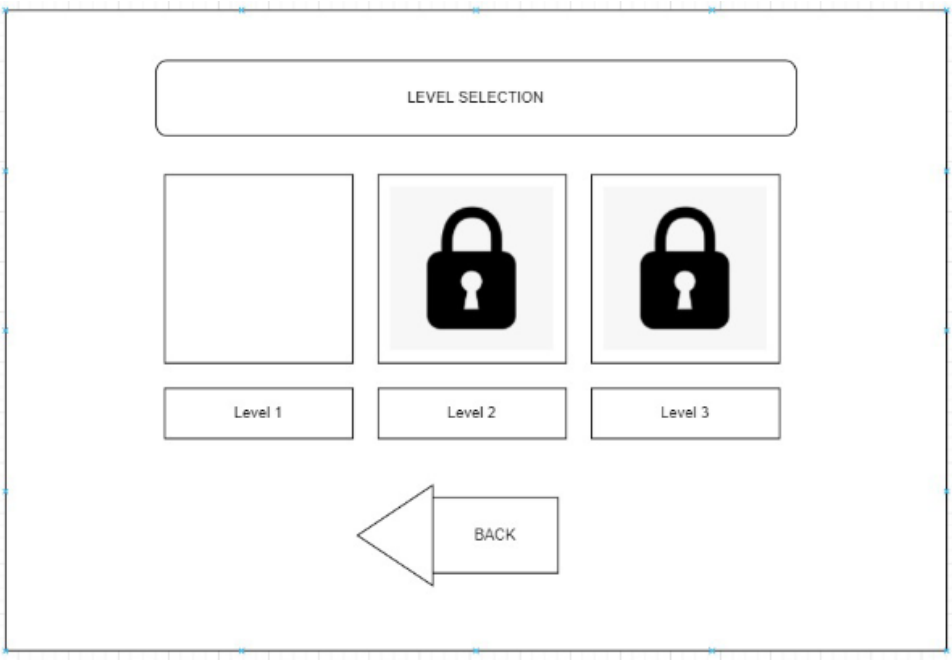
Menu screen:



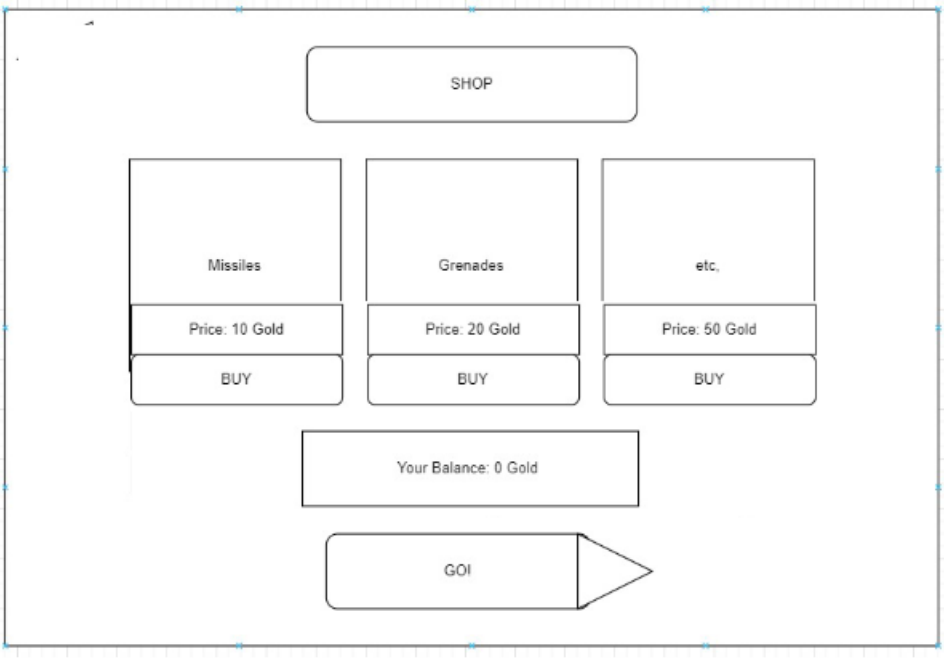
Options screen:



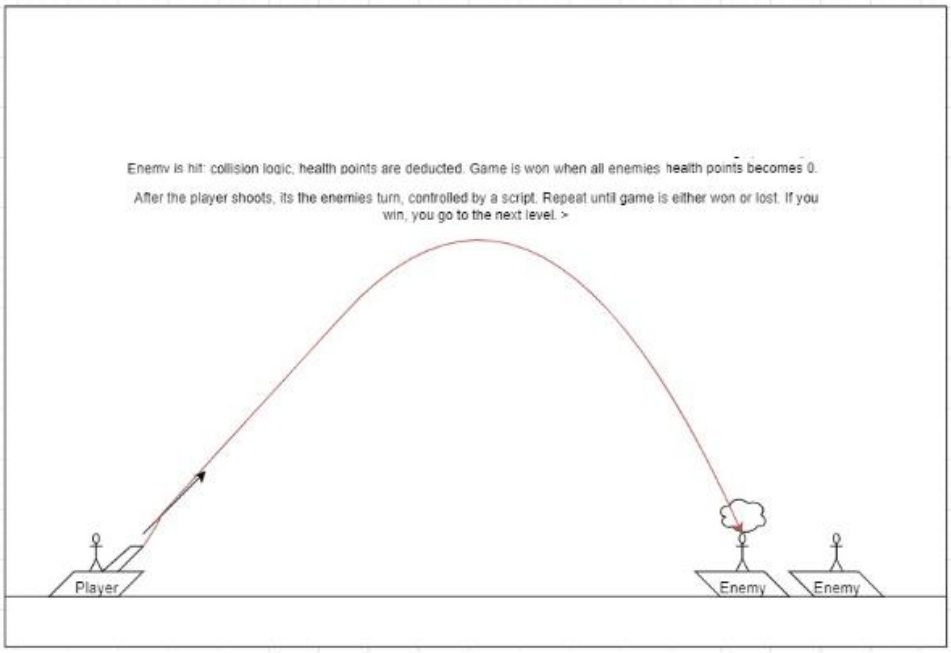
Level Selection Screen:



Shop screen:

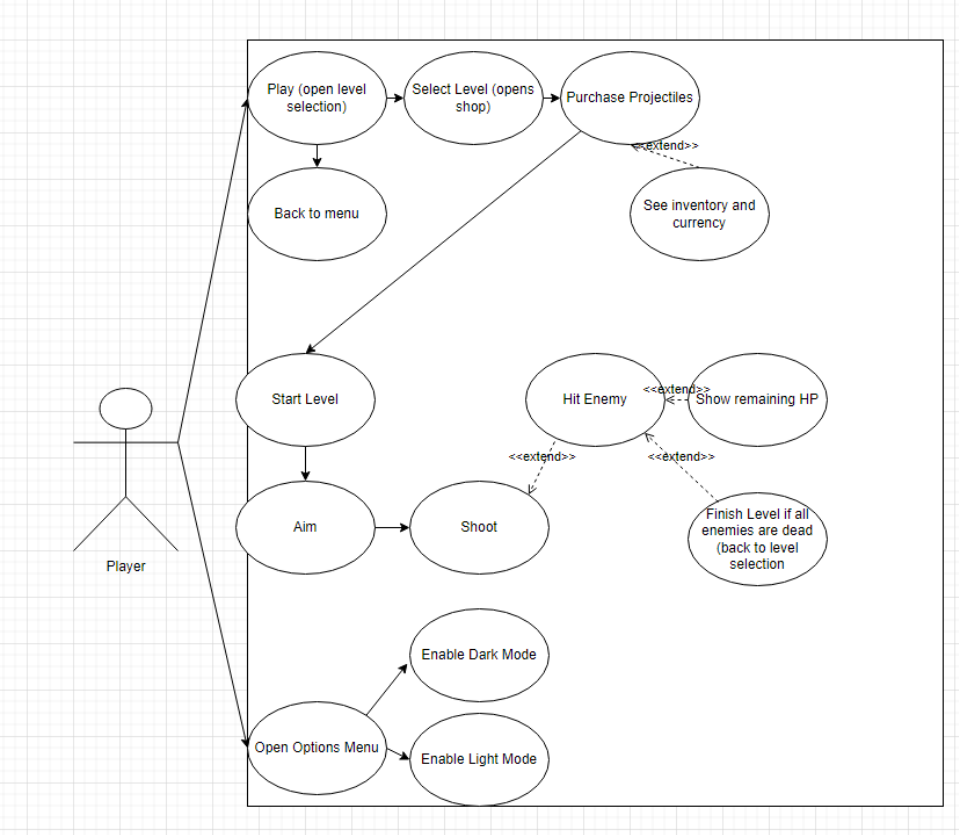


In-game screen:

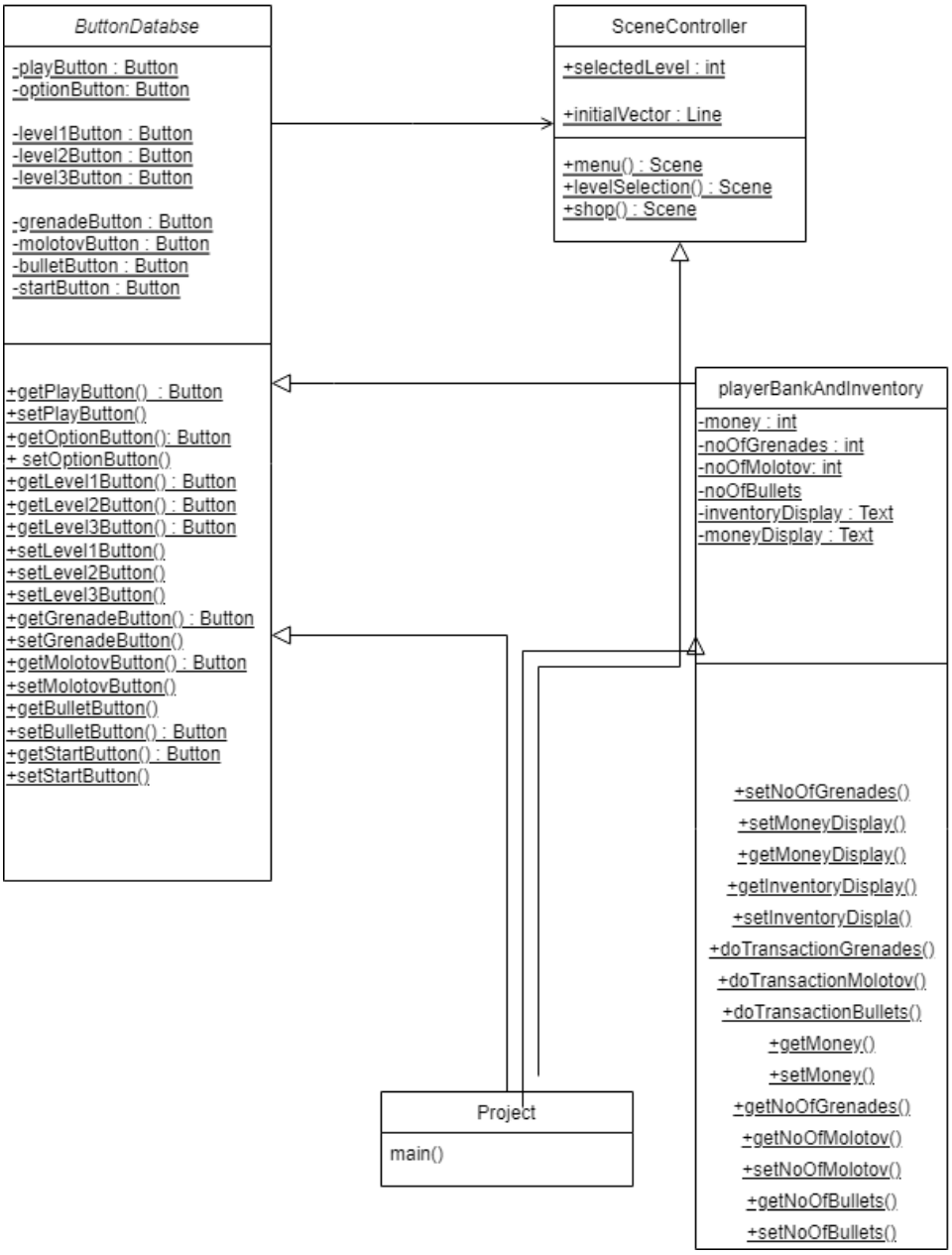


Diagrams

Use Case Diagram



Class Diagram



Program Features and Screenshots

For screenshots and description of our features, you can see them above in the User Interface section and Implementation section.

Team Responsibilities

Philippe: The application logic, such as the SceneController and the ButtonDatabase classes, as well as the event Handlers in the main class. Done with the help of Rayane.

Oussama and Rayane: Design of all scenes

Philippe: in-game physics implementation (parabolic trajectories)

Team effort: collision detection system.

Rayane: all inventory and currency related implementations.

Rayane: everything art-related, player and enemy sprites, backgrounds, and button art designs.

A big part of the project was done together at school, so everyone had a part to play in every aspect of the project, the above names are simply the ones who played the biggest role in achieving the parts.

Daniel: Collision physics, camera movement, music, and sound effects.

Challenges

Let us first discuss coding challenges.
One of the first challenges we encountered was changing scenes within the game and making them reusable to have working “Back” buttons. This issue was solved by implementing a SceneController class. Full explanation in Implementation Part-2 above.

Another challenge we faced was that buttons were often inaccessible, so we made a class called ButtonDatabase and had buttons be private static variables accessible by other classes via getters. This way, buttons were accessible by every class at any time during runtime.

Implementing the kinematics was difficult. JavaFX does not have the most adequate animation resources. After trying multiple different methods using while loops or ChangeListeners, we decided to use Timeline. Then, we struggled to find a constant to scale the velocity of the projectiles appropriately and struggled to translate real-world kinematics into in-game kinematics and having it be functional.

Aside from the coding challenges we encountered throughout this project, one of the main challenges that we faced was that one of our team members disappeared without warning anyone and has been completely unreachable since. This threw off a lot of the plans we had made for the project. Despite this handicap, we were able to cope and complete the game and do everything we aimed to achieve, but we were forced to cut some additional content. This is a challenge that affected the final product the most. We were forced to take out functionalities such as having a camera that follows the cannonballs, as well as a proper collision system with knockback, rebound, etc. (those are features that our missing teammate promised to implement).

In conclusion, we think we did an excellent job when it came to surmounting challenges. Some of our problems stemmed from the fact that JavaFX is not built for the kind of program we were trying to make. If we had to retry it, we would code our project in Python using pygame. This would have made a lot of things easier, but we would have had to learn Python and not everyone in the team knew Python and used it in the past.