Requirements

1	Design and Cohesion (30%)	Excellent use of classes, each class is used only for a sensible use and has good cohesion (right amount of sensible responsibilities)
2	Constants and Enumerated Types (10%)	Excellent use of enumerated types and enums to represent all values that do not change in the code.
3	Input (10%)	User input is completed and all actions are implemented
4	Display (10%)	All required drawable objects are implemented in code by specifying the coordinates that individual pieces should be drawn at
5	Menu (10%)	Game contains a main menu, hall of fame display (high scores) and an info screen showing the controls for the game
6	High Scores (5%)	Previous high scores are loaded from a file and any new high scores are saved in the file when a game is completed
7	Motion (10%)	Player motion implemented perfectly
8	Asteroids (5%)	New asteroids are created and they are given new directions and speeds based on the direction and speed of the original and a random element.
9	Alien (5%)	The alien ship appears and follows a (relatively) random path while shooting at the player.
10	Hyperspace (5%)	Hyperspace puts player in new location that is guaranteed to be safe (right now)

Customization

Below are constants used in the game, which could be set to customize the game.

```
public static final double pWidth = 1200, pHeight = 800;
2 usages
public static final int initialLives = 3;
1 usage
public static final double alienShipFiringInterval = 2, alienShipInterval = 20000; // Frequency of alienShip firing and generation
1 usage
private static final int largeAsteroidScore = 900, mediumAsteroidScore = 600, smallAsteroidScore = 300, alienShipScore = 1000, bonusLifeScore = 25000;
```

Fig 1. Setting in Game.java

```
private static final int bulletLimit = 50; // Maximum number of bullets on the screen at the same time
6 usages
private static final double playerMaxSpeed = 35; // Maximum speed for the player
1 usage
public static final long playerFiringInterval = 300; // Delay between shots in milliseconds
```

Fig 2. Setting in Player.java

```
public static final double largeRatio = 5, mediumRatio = 3, smallRatio = 1;
```

Fig 3. Setting in Asteroid.java

Draft of Drawing

The game sets asterid, player, and alienship based on the classic Asteroid game, which is similar to the picture below.

Fig 4. Source: https://6502disassembly.com/va-asteroids/

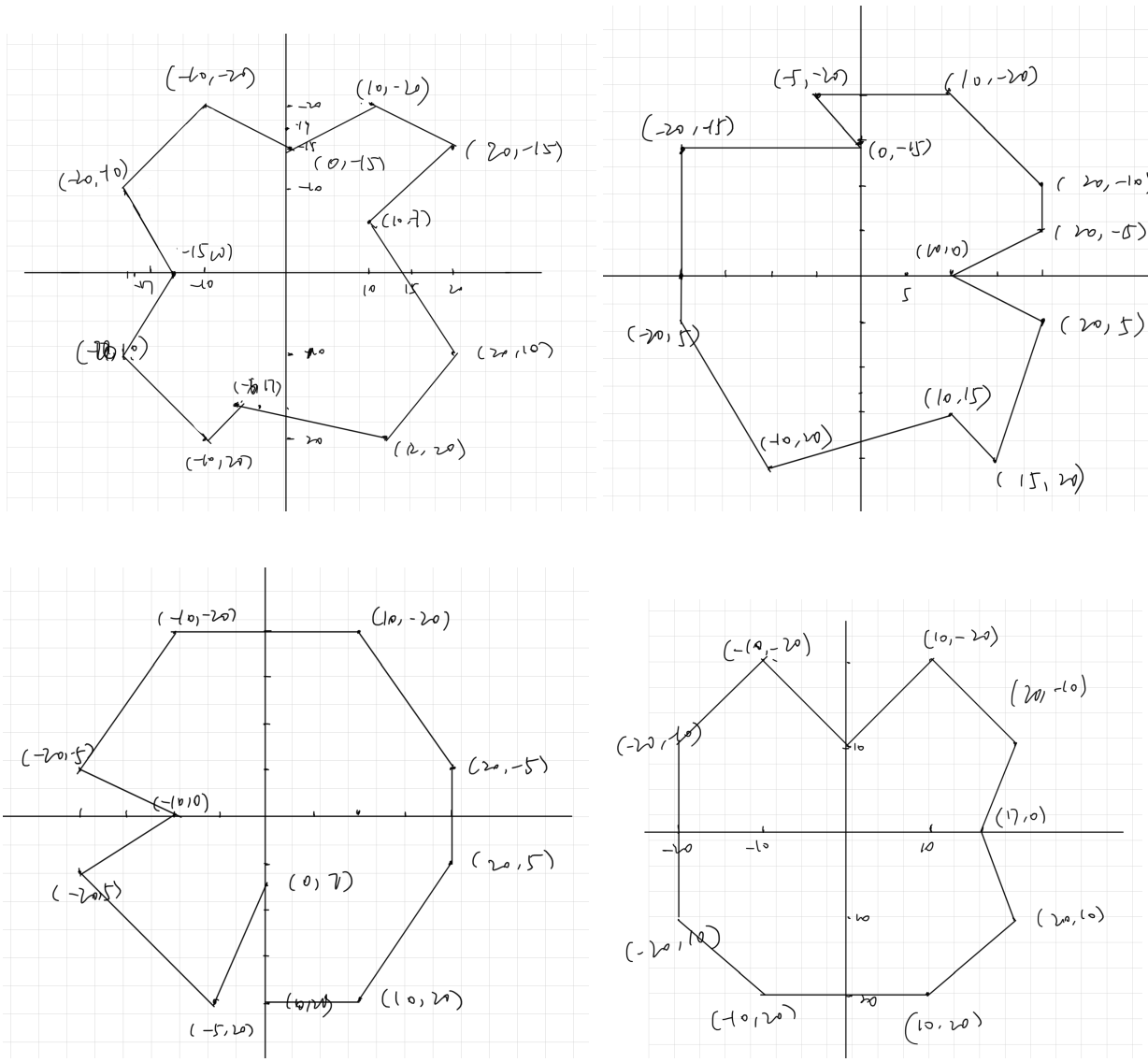


Fig 5. Design of Asteroids

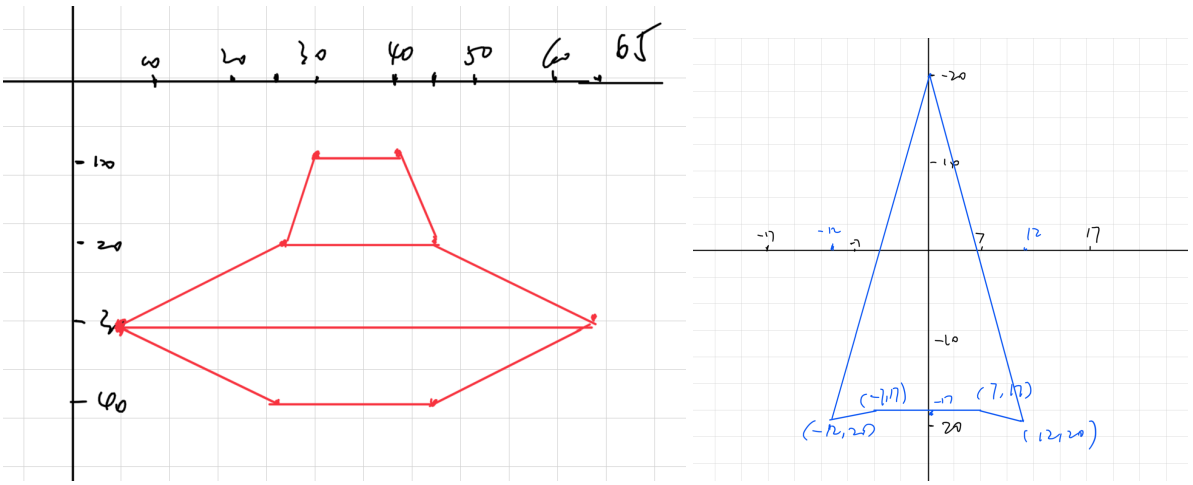


Fig 6. Design of Alien ship and Player

Explanation of the Project

The game has 7 classes, with the Game class as the main entrance of the game. The SpaceObject is the base class for all characters.

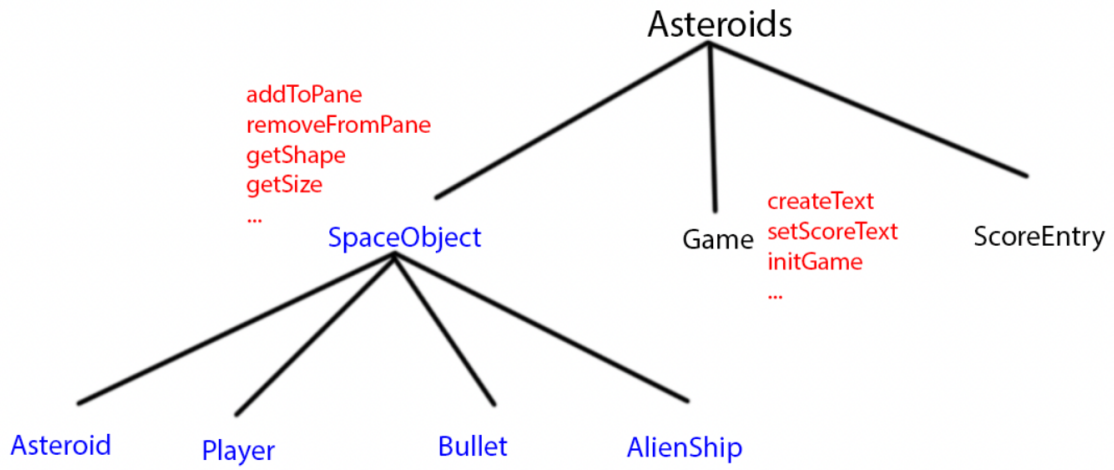


Fig 6. Design of classes

In the game, all characters (Asteroid, Player, Bullet, and AlienShip) have an `update` method, which will update their positions based on their speed (which will influence the dx and dy so that determines the velocity of the object). When a character moves into the edge of the screen, it will appear on the opposite side of the screen moving in the same direction.

The player's movement is limited by the "*`playerMaxSpeed`*", which means that when the user applies thrust to the player, it will increase speed up to the maximum speed and maintain that velocity.

The game also sets a bullet limit so that when there are *`bulletLimit`* number of bullets on the screen, the player cannot fire.

The player has the method `getSafeCoordintaes` to generate safe pairs *`safeDistance`* away from the player. This function helps generate asteroids and alien ships at a safe distance away from the player.

```
public Pair<Double, Double> getSafeCoordinates(double safeDistance) {
    double x, y;
    do {
        x = Math.random() * Game.pWidth;
        y = Math.random() * Game.pHeight;
    } while (!isSafe(x, y, this.getShape().getLayoutX(), this.getShape().getLayoutY(), safeDistance));
    return new Pair<>(x, y);
}
```

```
private void generateLevelAsteroids(int level, List<Asteroid> asteroids, Player player, Pane pane) {
    int asteroidNumber = Math.min(level, 7) + (int)Math.log(level);
    // Generate asteroids for the level
    for (int i = 0; i < asteroidNumber; i++) {
        // Ensure the asteroid is generated at a safe distance from the player
        Pair<Double, Double> safeCoordinates = player.getSafeCoordinates(200);
        double x = safeCoordinates.getKey();
        double y = safeCoordinates.getValue();
```

```
private void spawnAlienShip(Player player, Pane pane) {
    // Ensure the alien ship is generated at a safe distance from the player
    Pair<Double, Double> safeCoordinates = player.getSafeCoordinates(200);
    double sx = safeCoordinates.getKey();
    double sy = safeCoordinates.getValue();
```

The player has a `hyperspaceJump` method, which will transfer the player to a safe place immediately on the screen.

```
public void hyperspaceJump(List<Asteroid> asteroids, AlienShip alienShip, List<Bullet> alienShipBullets) {
    // Generate a random position for the player
    double a, b;
    boolean safeToJump;
    do {
        a = Math.random() * Game.pWidth;
        b = Math.random() * Game.pHeight;
        safeToJump = safeAsteroid(a, b, asteroids, safeDistance: 200);

        if (alienShip != null) {
            double alienDistance = Math.sqrt(Math.pow(a - alienShip.getX(), 2) + Math.pow(b - alienShip.getY(), 2));
            if (alienDistance < 200) {
                safeToJump = false;
            }
        }

        if (alienShipBullets != null) {
            for (Bullet bullet : alienShipBullets) {
                double bulletDistance = Math.sqrt(Math.pow(a - bullet.getX(), 2) + Math.pow(b - bullet.getY(), 2));
                if (bulletDistance < 100) {
                    safeToJump = false;
                    break;
                }
            }
        }
    } while (!safeToJump);
```

The alien ship has the ability to generate at intervals longer than "*alienShipInterval* " milliseconds and can fire at the player every "*alienShipFiringInterval* " second.

If the player's score surpasses the "*bonusLifeScore*" threshold, the player will receive an extra life and their score will be deducted by the "*bonusLifeScore* " amount. The game indicators will be updated accordingly.

Upon the player's death, they will be prompted to input their name into a window. The program will then save the player's record to the local file "highScores.txt". When the game is reopened, the program will read the file and display the top 10 records.

```
private void saveScores() {
    try {
        File file = new File( pathname: "highScores.txt");
        if (!file.exists()) {
            file.createNewFile();
        }
        FileWriter fileWriter = new FileWriter(file);

        for (ScoreEntry entry : highScores) {
            fileWriter.write( str: entry.getName() + "," + entry.getScore() + "\n");
        }

        fileWriter.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
private void loadScores() {
    highScores = new ArrayList<>();
    try {
        File scoreFile = new File( pathname: "highScores.txt");
        System.out.println("High scores file location: " + scoreFile.getAbsolutePath());
        if (scoreFile.exists()) {
            BufferedReader reader = new BufferedReader(new FileReader(scoreFile));
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split( regex: ",");
                highScores.add(new ScoreEntry(parts[0], Integer.parseInt(parts[1])));
            }
            reader.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

After losing a life, the player will respawn at the center of the screen and will flicker for 3 seconds. The player is invulnerable during this time.

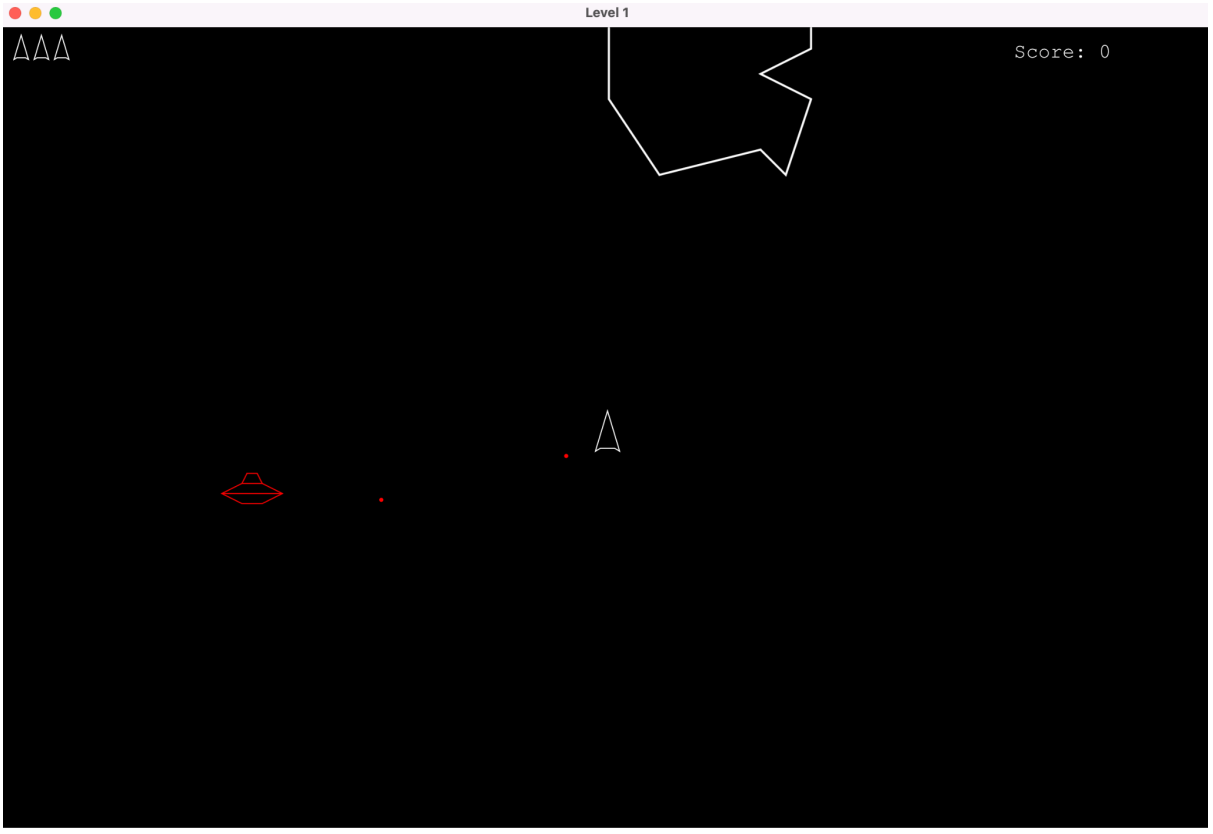
Screenshots of Panes of the game

1. Main menu

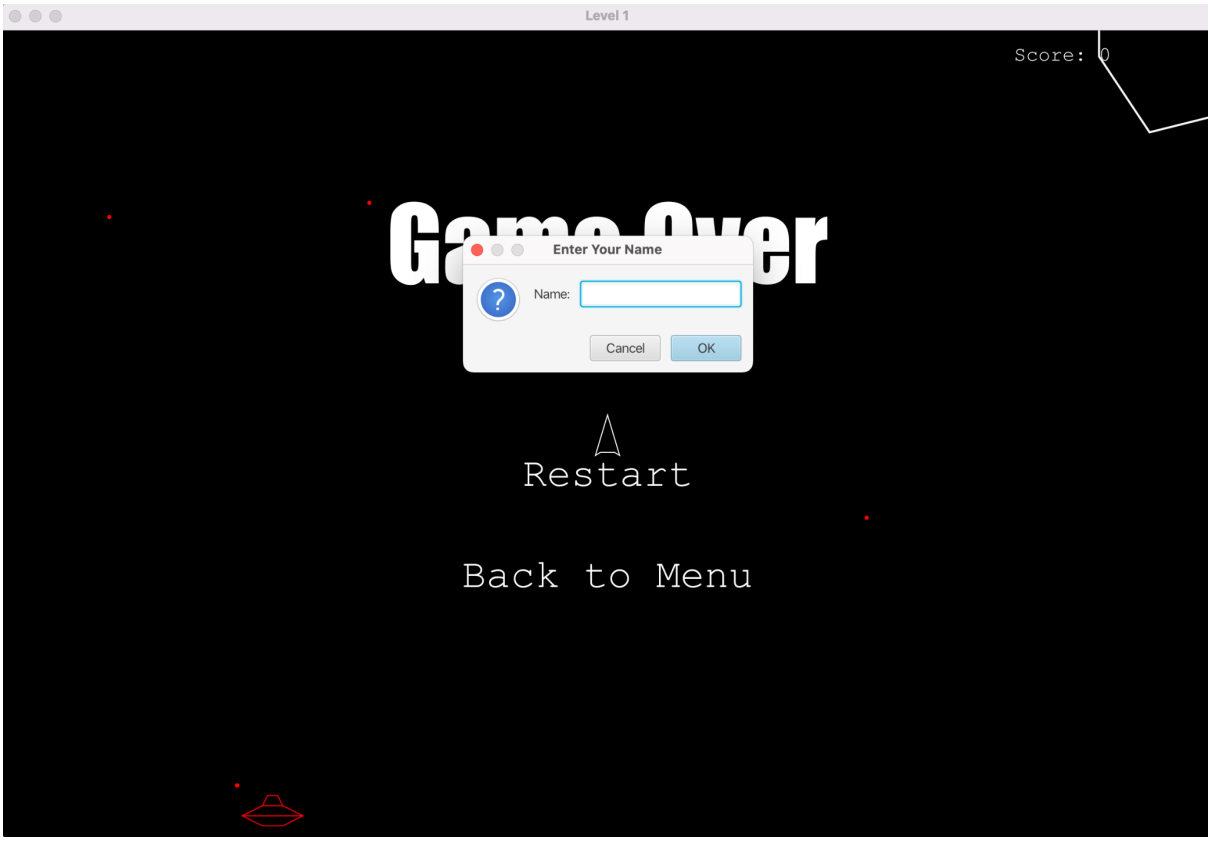
2. High Score Pane

3. Pause Pane (press ESC)

4. New Game Pane



5. Enter Score Pane



6. Controls Pane