VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Object-Oriented Programming
IT069IU

FINAL REPORT
MSc:Nguyen Trung Nghia

TOPIC: PLANE AND ROCKET

BY GROUP OOP – MEMBER LIST

NGUYEN HUY HOANG	ITCSIU21061	Leader
NGUYEN DU NHAN	ITDSIU22140	Member
NGUYEN KHOI NGUYEN	ITITSB23006	Member

TABLE OF CONTENTS

ABSTRACT

This project presents the development of "Plane and Rocket," a 2D space shooter game implemented in Java. The game features a player-controlled spaceship that must navigate through space while avoiding and destroying incoming enemy rockets. The primary objective is to achieve the highest possible score by destroying enemy rockets while staying alive.

CHAPTER 1: INTRODUCTION

1.1 Overview of Space Shooter Games

Space shooter games are a classic genre where players control a spacecraft, navigating through space to destroy enemy ships and avoid obstacles. Known for their fast-paced action, these games often feature power-ups, scoring systems, and increasingly challenging enemy waves. Originating from arcade hits like "Space Invaders," the genre has evolved with advancements in graphics and gameplay mechanics, maintaining a strong presence in gaming culture.

1.2 About Our Game

"Plane and Rocket" is a 2D space shooter game where players control a spaceship navigating through a starry backdrop. The objective is to survive and score points by destroying incoming enemy rockets. Players can rotate their ship, speed up, and shoot bullets to fend off waves of rockets. The game features dynamic difficulty, with rocket speed and quantity increasing as the player's score rises. Visual and audio effects enhance the immersive experience, while a health system adds strategic depth, challenging players to balance offense and defense.

1.3 Developer team

MMT team is a Object-Oriented Programming project team. We have 3 members

team come from International University:

Name - Github username	UID	Contribute
Nguyen Du Nhan - DuNhan1930	ITDSIU22140	Write report Design program Draw class diagram Tester
Nguyen Huy Hoang - Hageon3110	ITCSIU21061	Write report Design program Draw class diagram Fix Bug
Nguyen Khoi Nguyen - khoinguyen2k5	ITITSB23006	Write report Design programme Draw class diagram Fix Bug

1.4 Reference

Image from DJ-Raven/gmae-image

Tutorial game 2D development Java Graphics2D Game - YouTube

Design pattern Refactoring and Design Patterns

CHAPTER 2: SOFTWARE REQUIREMENTS

2.1 Development Environment

Java Development Kit (JDK): The game is developed using Java, requiring the JDK for compiling and running the application. Java provides a robust platform for building cross-platform applications with its extensive libraries and tools.

IDE (IntelliJ IDEA): IntelliJ IDEA is used as the integrated development environment, offering powerful code editing, debugging, and project

management features. It enhances productivity with its intelligent code completion and refactoring tools.

Graphics and Sound Libraries: The game utilizes Java's built-in libraries for graphics rendering and sound playback. These libraries handle 2D graphics drawing and audio management, essential for creating an engaging gaming experience.

2.2 Functional Requirements

Player Controls and Movement: The game must allow players to control the spaceship using keyboard inputs, enabling rotation, acceleration, and shooting actions.

Enemy Spawning System: Rockets should spawn at regular intervals from both sides of the screen, increasing in number and speed as the player's score increases.

Collision Detection: The game must accurately detect collisions between bullets and rockets, as well as between rockets and the player's spaceship, to trigger appropriate responses like damage or destruction.

Scoring System: Players earn points for destroying rockets. The scoring system should track and display the player's current score on the screen.

Sound Effects: The game should include sound effects for shooting, hitting, and destroying rockets to enhance the immersive experience.

Visual Effects: Explosions and other visual effects should be implemented to provide feedback for actions like rocket destruction and player damage.

2.3 Non-Functional Requirements

Performance Requirements: The game should run smoothly at a consistent frame rate, minimizing lag and ensuring responsive controls.

Graphics Quality: Visual elements should be clear and aesthetically pleasing, with smooth animations and transitions to enhance the gaming experience.

User Interface Responsiveness: The game's UI should be intuitive and responsive, providing clear feedback to player actions and ensuring easy navigation through menus

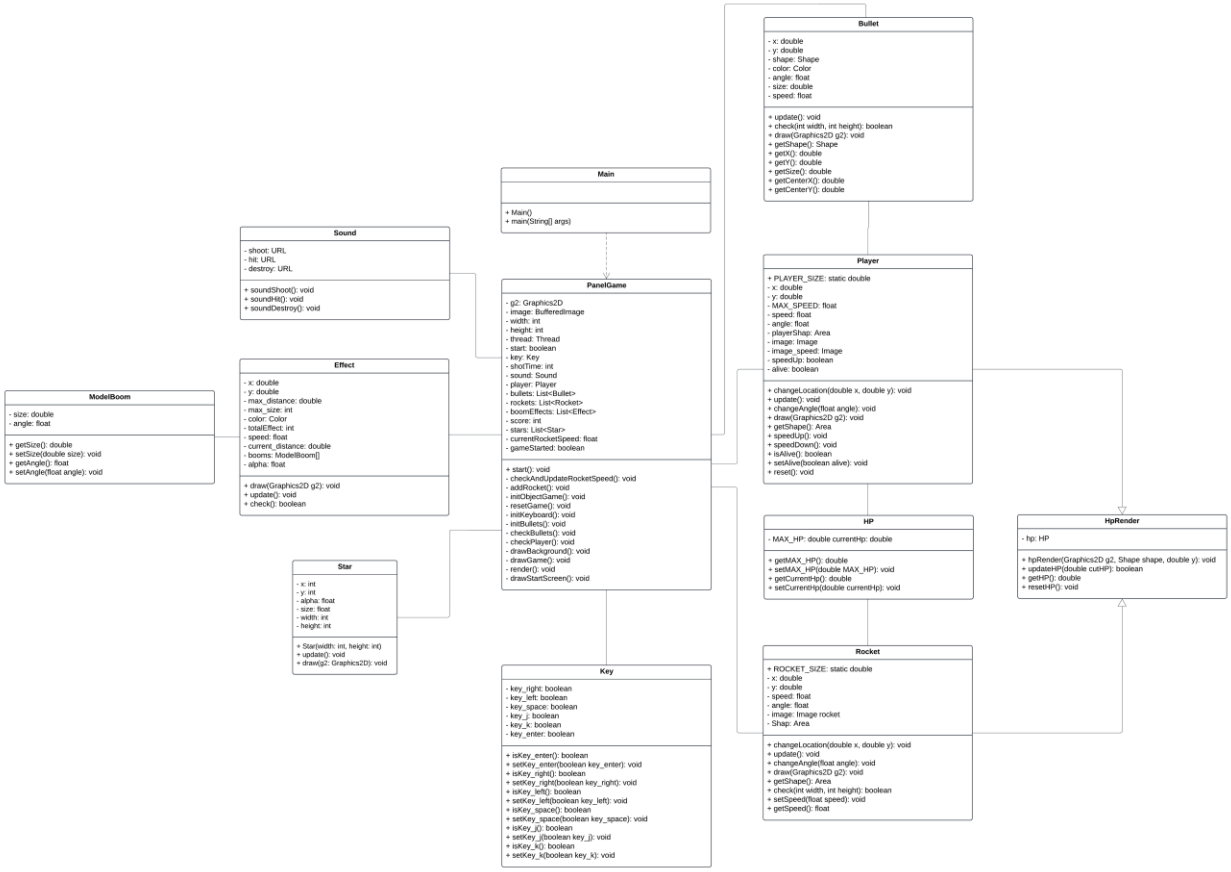
and game states.

Game Difficulty Balance: The difficulty should scale appropriately with the player's progress, maintaining a challenging yet achievable experience to keep players engaged.

CHAPTER 3: DESIGN & IMPLEMENTATION

3.1 Design

a. UML:



3.2 Implementation

a. Player.java

```
public Player() {  1 usage
    // Call the superclass constructor with initial HP values
    super(new HP( MAX_HP: 50,  currentHp: 50));
```

In the constructor of the Player class, the superclass (HpRender) constructor is called with initial health points (HP) set to 50 for both the current and maximum HP. This sets up the player's health system.

```
public void update() {  1 usage
    x += Math.cos(Math.toRadians(angle)) * speed; // Move along the X-axis
    y += Math.sin(Math.toRadians(angle)) * speed; // Move along the Y-axis
}
```

The **Player** class manages the player's movement, speed, and appearance. To update the player's position, the **update()** method is used, which calculates the new coordinates based on the current speed and angle. The movement is determined using trigonometric functions, specifically **Math.cos()** for the x-axis and **Math.sin()** for the y-axis, converting the angle to radians. This ensures that the player moves in the correct direction.

```
public void changeAngle(float angle) {  1 usage
    if (angle < 0) {
        angle = 359;
    } else if (angle > 359) {
        angle = 0;
    }
    this.angle = angle;
}
```

To change the direction the player is facing, the **changeAngle(float angle)** method is utilized. It adjusts the player's angle while ensuring it stays within the range of 0 to 359 degrees, thus allowing the player to rotate seamlessly in the game world.

The player's appearance is drawn with the **draw(Graphics2D g2**) method. This method applies a transformation to the graphics context, positioning and rotating the player according to its current coordinates and angle. It also switches between two different images based on whether the player is speeding up or not, determined by the speedUp flag.

```
public void draw(Graphics2D g2) {  1 usage
    AffineTransform oldTransform = g2.getTransform();
    g2.translate(x, y);
    AffineTransform tran = new AffineTransform();
    tran.rotate(Math.toRadians(angle + 45),  anchorx: PLAYER_SIZE / 2,
    g2.drawImage(speedUp ? image_speed : image, tran,  obs: null);
    hpRender(g2, getShape(), y);
    g2.setTransform(oldTransform);
```

Additionally, the player's health is displayed using the hpRender() method.

```
public void speedUp() {  no usages
    speedUp = true;
    if (speed > MAX_SPEED) {
        speed = MAX_SPEED;
    } else {
        speed += 0.01f;
    }
}
```

```
public void speedDown() {  1 usage
    speedUp = false;
    if (speed <= 0) {
        speed = 0;
    } else {
        speed -= 0.003f;
    }
}
```

To control the player's speed, the ***speedUp()*** and ***speedDown()*** methods are provided. When called, **speedUp()** gradually increases the player's speed, ensuring it does not exceed the maximum speed (MAX_SPEED), while **speedDown()** decreases the speed, eventually halting the player's movement if necessary.

```
public boolean isAlive() { return alive; }

public void setAlive(boolean alive) { this.alive = alive; }

public void reset() {  1 usage
    alive = true;
    resetHP();
    angle = 0;
    speed = 0;
}
```

Finally, the **isAlive()** method checks if the player is alive, and **the reset()** method resets the player's state—position, speed, and health—when required, such as after the player is defeated or when restarting the game.

b. Rocket.java

```
public Rocket() {  2 usages
    super(new HP( MAX_HP: 20,  currentHp: 20));
    this.image = new ImageIcon(getClass().getResource( name: "/Image/rocket.png")).getImage();
    Path2D p = new Path2D.Double();
```

In the Rocket class, the constructor initializes the rocket's properties, including its initial position, speed, and shape. The super constructor is called with initial health points (HP) set to 20. The rocket's shape is defined using a Path2D object, which outlines the rocket's design.

```
public void changeLocation(double x, double y) {  2 usages
    this.x = x;
    this.y = y;
}

public void update() {  1 usage
    x += Math.cos(Math.toRadians(angle)) * speed;
    y += Math.sin(Math.toRadians(angle)) * speed;
}
```

The rocket's position can be updated with the **changeLocation()** method, and its movement is controlled in the **update()** method using basic trigonometry, where the x and y positions are updated based on the angle and speed.

To change the rocket's direction, the changeAngle() method adjusts the angle within the 0 to 359-degree range.

```
public void changeAngle(float angle) {
    if (angle < 0) {
        angle = 359;
    } else if (angle > 359) {
        angle = 0;
    }
    this.angle = angle;
}
```

```
public void draw(Graphics2D g2) {  1 usage
```

The draw() method is responsible for rendering the rocket on the screen, rotating it based on its angle and drawing the health bar.

```
public Area getShape() {  4 usages
    AffineTransform afx = new AffineTransform();
    afx.translate(x, y);
    afx.rotate(Math.toRadians(angle),  anchorx: ROCKET_SIZE / 2,  anchory: ROCKET_SIZE / 2
    return new Area(afx.createTransformedShape(rocketShap));
}
public boolean check(int width, int height) {  1 usage
    Rectangle size = getShape().getBounds();
    if (x <= -size.getWidth() || y < -size.getHeight() || x > width || y > height) {
        return false;
    } else {
        return true;
    }
}
```

The getShape() method provides the rocket's shape as an Area object, while the check() method verifies if the rocket is within the screen boundaries.

c. Bullet.java

```
public Bullet(double x, double y, float angle, double size, float speed) {
```

In the Bullet class, a bullet is represented as a small triangle that moves in the direction of its given angle. The constructor initializes the bullet's position (x, y), size, speed, and angle. The bullet's shape is defined using a Path2D object, creating a triangle that represents the bullet.

```
Path2D triangle = new Path2D.Double();
triangle.moveTo( x: 0,  y: 0);
triangle.lineTo( x: size/2, size);
triangle.lineTo(size,  y: 0);        //
triangle.closePath();                //
```

```
public void update() {  1 usage
    x += Math.cos(Math.toRadians(angle)) * speed;
    y += Math.sin(Math.toRadians(angle)) * speed;
}
```

```
public boolean check(int width, int height) {  1 usage
    if (x <= -size || y < -size || x > width || y > height) {
        return false;
    } else {
        return true;
```

The update() method moves the bullet based on its speed and angle, calculated using

trigonometric functions (cosine for the x-axis and sine for the y-axis). The check() method verifies if the bullet is still within the screen boundaries based on its current position and size.

```
    public void draw(Graphics2D g2) {  1 usage
```

The **draw()** method renders the bullet on the screen, rotating the triangle shape based on the bullet's travel direction. The **getShape()** method provides the bullet's shape as an Area object, which is rotated and translated to its current position. Additional getter methods like **getX(), getY(), getSize(), getCenterX()**, and **getCenterY()** provide access to the bullet's position, size, and center coordinates.

d. BulletFactory.java

```
public class BulletFactory {  no usages
    public static Bullet createBullet(double x, double y, float angle, double size, float speed) {
        return new Bullet(x, y, angle, size, speed);
    }
```

The ***BulletFactory*** class is a simple factory design pattern implementation that provides a method to create instances of the Bullet class. The **createBullet()** method accepts the position (x, y), angle, size, and speed as parameters, and returns a new Bullet object with those properties. This approach helps to separate the creation logic of bullets from the rest of the game logic, making it easier to manage and maintain bullet creation in the game.

e. HP.java

Create a class to store the MAX_HP and currentHp variables.

```
1   package Object;
2
3   public class HP {
4       private double MAX_HP;
5       private double currentHp;
6
7       public double getMAX_HP() {
8           return MAX_HP;
9       }
10
11      public void setMAX_HP(double MAX_HP) {
12          this.MAX_HP = MAX_HP;
13      }
14
15      public double getCurrentHp() {
16          return currentHp;
17      }
18
19      public void setCurrentHp(double currentHp) {
20          this.currentHp = currentHp;
21      }
22
23      public HP(double MAX_HP, double currentHp) {
24          this.MAX_HP = MAX_HP;
25          this.currentHp = currentHp;
26      }
27
28      public HP() {
29      }
30  }
```

f. HpRender.java

Create class HpRender

```
8  public class HpRender {
9      private final HP hp;
10
11     public HpRender(HP hp) {
12         this.hp = hp;
13     }
14
15     protected void hpRender(Graphics2D g2, Shape shape, double y) {
16         if (hp.getCurrentHp() != hp.getMAX_HP()) {
17             double hpY = shape.getBounds().getY() - y - 10;
18             g2.setColor(new Color(70, 70, 70));
19             g2.fill(new Rectangle2D.Double(0, hpY, Player.PLAYER_SIZE, 2));
20             g2.setColor(new Color(253, 91, 91));
21             double hpSize = hp.getCurrentHp() / hp.getMAX_HP() * Player.PLAYER_SIZE;
22             g2.fill(new Rectangle2D.Double(0, hpY, hpSize, 2));
23         }
24     }
25
26     public boolean updateHP(double cutHP) {
27         hp.setCurrentHp(hp.getCurrentHp() - cutHP);
28         return hp.getCurrentHp() > 0;
29     }
30
31     public double getHP() {
32         return hp.getCurrentHp();
33     }
34
35     public void resetHP() {
36         hp.setCurrentHp(hp.getMAX_HP());
37     }
38 }
```

Create Hp variablea and the constructer.

Initialize method hpRender to render the hp bar. Set condition to only draw when the hp is not max.

```
    protected void hpRender(Graphics2D g2, Shape shape, double y) {
        if (hp.getCurrentHp() != hp.getMAX_HP()) {
            double hpY = shape.getBounds().getY() - y - 10;
            g2.setColor(new Color(70, 70, 70));
            g2.fill(new Rectangle2D.Double(0, hpY, Player.PLAYER_SIZE, 2));
            g2.setColor(new Color(253, 91, 91));
            double hpSize = hp.getCurrentHp() / hp.getMAX_HP() * Player.PLAYER_SIZE;
            g2.fill(new Rectangle2D.Double(0, hpY, hpSize, 2));
```

We will call g2.setColor and g2.fill to draw the health bar. HpY is used to represent the total health of object, while the hpSize will indicate the current health of object.

After that, we will call method update() to update the current health. Create the getter method for the currentHp, then create the resetHp() method to reset the current health back to full.

g. ModelBoom.java

```java
package Object;

public class ModelBoom {
    private double size;
    private float angle;

    public double getSize() {
        return size;
    }

    public void setSize(double size) {
        this.size = size;
    }

    public float getAngle() {
        return angle;
    }

    public void setAngle(float angle) {
        this.angle = angle;
    }

    public ModelBoom(double size, float angle) {
        this.size = size;
        this.angle = angle;
    }

    public ModelBoom() {
    }
}
```

This class is used to store size and angle variable.

h. Effect.java

Create class Effect to store double x, y variable, max_distance, max_size, color, total_effect, speed, current_distance, model_boom boom[]. This will be used for creating the explosion effect.

```java
package Object;

import java.awt.AlphaComposite;
import java.awt.Color;
import java.awt.Composite;
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import java.util.Random;

public class Effect {
    private final double x;
    private final double y;
    private final double max_distance;
    private final int max_size;
    private final Color color;
    private final int totalEffect;
    private final float speed;
    private double current_distance;
    private ModelBoom booms[];
    private float alpha = 1f;

    public Effect(double x, double y, int totalEffect, int max_size, double max_distance, float speed, Color color) {
        this.x = x;
        this.y = y;
        this.totalEffect = totalEffect;
        this.max_size = max_size;
        this.max_distance = max_distance;
        this.speed = speed;
        this.color = color;
        createRandom();
    }
```

Create constructor to set the value for the attributes.

```
    private void createRandom() {
        booms = new ModelBoom[totalEffect];
        float per = 360f / totalEffect;
        Random ran = new Random();
        for (int i = 1; i <= totalEffect; i++) {
            int r = ran.nextInt((int) per) + 1;
            int boomSize = ran.nextInt(max_size) + 1;
            float angle = i * per + r;
            booms[i - 1] = new ModelBoom(boomSize, angle);
```

Using the createRandom method to randomly set the size and angle of the explosion particle. Then initialize it in the constructor.

```
11  public class Effect {
46      public void draw(Graphics2D g2) {
47          AffineTransform oldTransform = g2.getTransform();
48          Composite oldComposite = g2.getComposite();
49          g2.setColor(color);
50          g2.translate(x, y);
51          for (ModelBoom b : booms) {
52              double bx = Math.cos(Math.toRadians(b.getAngle())) * current_distance;
53              double by = Math.sin(Math.toRadians(b.getAngle())) * current_distance;
54              double boomSize = b.getSize();
55              double space = boomSize / 2;
56              if (current_distance >= max_distance - (max_distance * 0.7f)) {
57                  alpha = (float) ((max_distance - current_distance) / (max_distance * 0.7f));
58              }
59              if (alpha > 1) {
60                  alpha = 1;
61              } else if (alpha < 0) {
62                  alpha = 0;
63              }
64              g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, alpha));
65              g2.fill(new Rectangle2D.Double(bx - space, by - space, boomSize, boomSize));
66          }
67          g2.setComposite(oldComposite);
68          g2.setTransform(oldTransform);
69      }
```

Initialze the method draw to draw the effect, then initialize the update() to make the particle fly, then use check() method to check if the particle has reach the max distance

```
71      public void update() {
72          current_distance += speed;
73      }
74
75      public boolean check() {
76          return current_distance < max_distance;
```

i. Star.java

We use Star class to create the star sparkling on the screen

```
 9  public class Star {
10      // Star's position coordinates on the screen
11      int x, y;
12      // Star's transparency/opacity level (0.0 to 1.0)
13      float alpha = 1.0f;
14      // Star's size in pixels
15      float size;
16      // Screen dimensions
17      private int width;
18      private int height;
19
20      /**
21       * Creates a new star with random position and size
22       * Position is within game screen bounds
23       * Size ranges from 1-3 pixels
24       */
25      public Star(int width, int height) {
26          this.width = width;
27          this.height = height;
28          Random rand = new Random();
29          x = rand.nextInt(width);
30          y = rand.nextInt(height);
31          size = rand.nextFloat() * 2 + 1;
32      }
33
```

Create a constructor to get the variable. The x, y is the location of randomly placed stars. Initialize the update() method to set the transparency, making the twinkling effect.

```
38        public void update() {
39            alpha = (float) (0.5f + Math.random() * 0.5f);
40        }
41
```

Lastly, initialize the draw() method to draw the star.

```
47        public void draw(Graphics2D g2) {
48            g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, alpha));
49            g2.setColor(Color.WHITE);
50            g2.fill(new Rectangle2D.Double(x, y, size, size));
51        }
52    }
```

j. Sound.java

```
public Sound() {  2 usages
    this.shoot = this.getClass().getClassLoader().getResource( name: "Object/Sound/shoot.wav");
    this.hit = this.getClass().getClassLoader().getResource( name: "Object/Sound/hit.wav");
    this.destroy = this.getClass().getClassLoader().getResource( name: "Object/Sound/destroy.wav");
}
```

- **Sound Resources**:

The Sound class is responsible for managing and playing sound effects in the game. It follows the Singleton pattern to ensure only one instance of the class is used throughout the game. The constructor loads sound files (shoot.wav, hit.wav, destroy.wav) from the resources directory using the ClassLoader.

```
private void play(URL url) {  3 usages
    try {
        AudioInputStream audioIn = AudioSystem.getAudioInputStream(url);
        Clip clip = AudioSystem.getClip();
        clip.open(audioIn);
        clip.addLineListener(new LineListener() {
            @Override
            public void update(LineEvent event) {
                if (event.getType() == LineEvent.Type.STOP) {
                    clip.close();
                }
            }
        });
        audioIn.close();
        clip.start();
    } catch (IOException | LineUnavailableException | UnsupportedAudioFileException e) {
        System.err.println(e);
    }
}
}
```

The class provides methods like soundShoot(), soundHit(), and soundDestroy() to play these sound effects by invoking the private play() method.

```
clip.addLineListener(new LineListener() {
    @Override
    public void update(LineEvent event) {
        if (event.getType() == LineEvent.Type.STOP) {
            clip.close();
        }
```

- **Playing Sounds:**

This method handles opening the audio file, creating a clip, and playing the sound. A *LineListene*r is used to close the clip once the sound finishes playing, ensuring efficient resource management.

k. Key.java

The Key class serves as the input handling component for the game system. It manages keyboard states through boolean flags and provides a clean interface for other game components to access these states.

Movement Controls

The movement system utilizes three primary controls:

- `key_right`: Controls rightward rotation (D key)
- `key_left`: Controls leftward rotation (A key)
- `key_space`: Controls speed acceleration (Spacebar)

```
// Movement controls
private boolean key_right;    // D key - rotate right
private boolean key_left;     // A key - rotate left
private boolean key_space;    // Spacebar - speed up
```

Getters and setters for Movement controls in the game:

```
public boolean isKey_right() {
    return key_right;
}

public void setKey_right(boolean key_right) {
    this.key_right = key_right;
}
```

```
public boolean isKey_left() {
   return key_left;
}

public void setKey_left(boolean key_left) {
   this.key_left = key_left;
}

public boolean isKey_space() {
   return key_space;
}

public void setKey_space(boolean key_space) {
   this.key_space = key_space;
}
```

Weapon Controls

The combat system implements two weapon controls:

- key_j: Triggers small bullet firing
- key_k: Triggers big bullet firing

```
// Weapon controls
private boolean key_j;        // J key - fire small bullet
private boolean key_k;        // K key - fire big bullet
```

Getters and setters for Weapons controls in the game:

```
public boolean isKey_j() {
   return key_j;
}

public void setKey_j(boolean key_j) {
   this.key_j = key_j;
}

public boolean isKey_k() {
   return key_k;
}
```

```
public void setKey_k(boolean key_k) {
   this.key_k = key_k;
}
```

System Controls

Game system functionality is controlled through:

- `key_enter`: Manages game start/restart
- `key_r`: Handles high score reset

```
// System controls
private boolean key_enter;    // Enter key - start/restart game
private boolean key_r;        // R key - reset high score
```

Getters and setters for System controls in the game:

```
public boolean isKey_enter() {
   return key_enter;
}

public void setKey_enter(boolean key_enter) {
   this.key_enter = key_enter;
}

public boolean isKey_r() {
   return key_r;
}

public void setKey_r(boolean key_r) {
   this.key_r = key_r;
}
```

l. PanelGame.java

The PanelGame class is the core component of a 2D space shooter game, extending JComponent. It manages all game elements including the player's spaceship, enemy rockets, bullets, and collision detection. The class handles game rendering at 60 FPS, user input for ship control, and maintains game states including score tracking and difficulty progression. It has many features such as a star-filled background, explosion effects, and

persistent high score storage.

```
public class PanelGame extends JComponent {
    // Graphics variables
    private Graphics2D g2;
    private BufferedImage image;

    // Game objects
    private Player player;
    private LinkedList<Bullet> bullets;
    private List<Rocket> rockets;
    private List<Effect> boomEffects;

    // Game state variables
    private boolean gameStarted = false;
    private int score = 0;
    private int highestScore = 0;
}
```

- FPS = 60 means the game aims to update 60 times per second.
- 1,000,000,000 represents one second in nanoseconds (1 billion nanoseconds = 1 second).
- TARGET_TIME = 1,000,000,000 / 60 = 16,666,666 nanoseconds per frame.

This value is used to control the game loop timing:

```
while (start) {
    long startTime = System.nanoTime();  // Start time of frame

    // Game update and render code here

    long time = System.nanoTime() - startTime;  // Time taken for frame
    if (time < TARGET_TIME) {
        // If frame completed too quickly, sleep for remaining time
        long sleep = (TARGET_TIME - time) / 1000000;  // Convert to milliseconds
        sleep(sleep);
    }
}
```

Benefits:

- Consistent game speed across different computers
- Prevents the game from running too fast
- Reduces CPU usage by not rendering more frames than needed
- Provides smooth animation and gameplay

Game Setup and Initialization:

- **public void start()**

```
public void start() {
    // 1. Setup Phase
    loadHighScore(); // Load saved high score
    width = getWidth(); // Get window dimensions
    height = getHeight();

    // 2. Graphics Setup
    // Create drawing buffer
    image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
    g2 = image.createGraphics(); // Get graphics context
    // Enable smooth graphics
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
RenderingHints.VALUE_INTERPOLATION_BILINEAR);

    initStars(); // Setup background stars

    // 3. Create Game Loop Thread
    thread = new Thread(() -> {
        while (start) {
            long startTime = System.nanoTime(); // Start frame timing

            drawBackground(); // Draw background
            if (gameStarted) {
                drawGame(); // Draw game if playing
            } else {
                drawStartScreen(); // Draw menu if not started
```

```
            }
            render(); // Display frame

            // Control frame rate (60 FPS)
            long time = System.nanoTime() - startTime;
            if (time < TARGET_TIME) {
                long sleep = (TARGET_TIME - time) / 1000000;
                sleep(sleep);
            }
        }
    });

    // 4. Initialize and Start
    initObjectGame(); // Setup game objects
    initKeyboard(); // Setup controls
    initBullets(); // Setup bullet system
    thread.start(); // Start game loop
}
```

- **private void initObjectGame()**

```
private void initObjectGame() {
   // 1. Initialize game sound system
   sound = new Sound();

   // 2. Create player object and set starting position
   player = new Player();
   player.changeLocation(150, 150);

   // 3. Initialize empty lists for rockets and explosion effects
   rockets = new ArrayList<>();
   boomEffects = new ArrayList<>();

   // 4. Create and start a new thread for rocket spawning
   new Thread(new Runnable() {
       @Override
       public void run() {
           while (start) {
```

```
                // Only spawn rockets if the game has started
                if (gameStarted) {
                    addRocket();
                }
                // Wait 3 seconds before next spawn
                sleep(3000);
            }
        }
    }).start(); // Launch the thread immediately
```

- **private void initKeyboard()**

1. **Keyboard Setup**

- **Initialize key handler**

```
key = new Key();
```

- **Request input focus**

```
requestFocus();
```

- **Setup key listeners**

```
addKeyListener(new KeyAdapter() {}
```

2. **Key Press Handling**

```
public void keyPressed(KeyEvent e) {}
```

- **Pre-game controls (ENTER to start, R to reset)**

```
// Check if the game has not started
if (!gameStarted) {
   // Start the game when the Enter key is pressed
   if (e.getKeyCode() == KeyEvent.VK_ENTER) {
       gameStarted = true;
       return;
   }
   // Reset the high score when the 'R' key is pressed from the start screen
   else if (e.getKeyCode() == KeyEvent.VK_R) {
       resetHighScore();
   }
}
```

- **In-game controls (movement, shooting)**

```
if (gameStarted) {
```

```
    // Handle different key presses
    if (e.getKeyCode() == KeyEvent.VK_A) {
        key.setKey_left(true); // A key - Rotate ship left
    }
    else if (e.getKeyCode() == KeyEvent.VK_D) {
        key.setKey_right(true); // D key - Rotate ship right
    }
    else if (e.getKeyCode() == KeyEvent.VK_SPACE) {
        key.setKey_space(true); // Spacebar - Increase ship speed
    }
    else if (e.getKeyCode() == KeyEvent.VK_J) {
        key.setKey_j(true); // J key - Fire small bullet
    }
    else if (e.getKeyCode() == KeyEvent.VK_K) {
        key.setKey_k(true); // K key - Fire big bullet
    }
    else if (e.getKeyCode() == KeyEvent.VK_ENTER) {
        key.setKey_enter(true); // Enter key - Game control
    }
```

3. Key Release Handling

- **Reset key states when released**

```
if (gameStarted) {
    // Reset key states when keys are released
    if (e.getKeyCode() == KeyEvent.VK_A) {
        key.setKey_left(false); // A key released - Stop rotating left
    }
    else if (e.getKeyCode() == KeyEvent.VK_D) {
        key.setKey_right(false); // D key released - Stop rotating right
    }
    else if (e.getKeyCode() == KeyEvent.VK_SPACE) {
        key.setKey_space(false); // Spacebar released - Stop speed boost
    }
    else if (e.getKeyCode() == KeyEvent.VK_J) {
        key.setKey_j(false); // J key released - Stop small bullet firing
    }
    else if (e.getKeyCode() == KeyEvent.VK_K) {
```

```
        key.setKey_k(false); // K key released - Stop big bullet firing
    }
    else if (e.getKeyCode() == KeyEvent.VK_ENTER) {
        key.setKey_enter(false); // Enter key released
    }
    else if (e.getKeyCode() == KeyEvent.VK_R) {
        key.setKey_r(false); // R key released
    }
```

4. Game Loop Thread

- **Player movement and rotation**

```
new Thread(new Runnable() {
    @Override
    public void run() {
        float s = 0.5f;  // Rotation speed constant
        while (start) {
            if (player.isAlive()) {
                // 1. Handle Player Rotation
                float angle = player.getAngle();
                if (key.isKey_left()) angle -= s;      // Rotate left
                if (key.isKey_right()) angle += s;     // Rotate right
```

- **Shooting mechanics**

```
 // 2. Handle Shooting
 if (key.isKey_j() || key.isKey_k()) {
     if (shotTime == 0) {  // Control fire rate
         // Create appropriate bullet type
         if (key.isKey_j()) {
             // Create and add a standard bullet
             bullets.addFirst(BulletFactory.createStandardBullet(player));
         } else {
             // Create and add a big bullet
             bullets.addFirst(BulletFactory.createBigBullet(player));
         }
         sound.soundShoot();
     }
     // Manage shot timing (15 frame delay)
```

```
      shotTime++;
      if (shotTime == 15) shotTime = 0;
  } else {
      shotTime = 0;
  }
```

The use of the Factory Pattern (BulletFactory) in this code provides some key benefits. This pattern hides the complexity of bullet creation by centralizing initialization logic in one place. It not only makes the code more maintainable and extensible (such as adding new bullet types) but also ensures consistency in bullet creation.

- **Speed control and Player Update**

```
  // 3. Handle Speed
  if (key.isKey_space()) {
      player.speedUp();
  } else {
      player.speedDown();
  }
  // Update Player
  player.update();
  player.changeAngle(angle);

} else {  // Player is dead
  if (key.isKey_enter()) {
      resetGame();
  }
```

- **Rocket updates and collisions**

```
// 5. Iterate through all rockets in the game
for (int i = 0; i < rockets.size(); i++) {
   Rocket rocket = rockets.get(i);
   if (rocket != null) {
       rocket.update(); // Update rocket position

       if (!rocket.check(width, height)) {
           rockets.remove(rocket); // Remove if out of bounds
```

```
        } else {
            if (player.isAlive()) {
                checkPlayer(rocket); // Check collision with player
            }
        }
}
```

- **Runs every 5ms**

```
sleep(5);  // Control update rate
```

- **private void initBullets()**

The initBullets() method manages bullets and explosion effects in the game:

```
private void initBullets() {
   // Bullet System Setup
   // Create empty bullet collection
   bullets = new LinkedList<>();
   // Thread Creation
   // Start continuous management thread
   new Thread(new Runnable() {
       @Override
       public void run() {
           while (start) {
               // Bullet Management Loop
               for (int i = 0; i < bullets.size(); i++) {
                   Bullet bullet = bullets.get(i);
                   if (bullet != null) {
                       // Update bullet positions
                       bullet.update();

                       // Check for collisions
                       checkBullets(bullet);

                       // Remove out-of-bounds bullets
                       if (!bullet.check(width, height)) {
                           bullets.remove(bullet);
                       }
                   } else {
```

```
                    // Remove null bullets
                    bullets.remove(bullet);
                }
            }
            // Effect Management Loop
            for (int i = 0; i < boomEffects.size(); i++) {
                Effect boomEffect = boomEffects.get(i);
                if (boomEffect != null) {
                    // Update explosion animations
                    boomEffect.update();

                    // Remove completed effects
                    if (!boomEffect.check()) {
                        boomEffects.remove(boomEffect);
                    }
                } else {
                    // Remove null effects
                    boomEffects.remove(boomEffect);
                }
            }
            // Timing Control
            // 1ms sleep between updates
            sleep(1);
        }
    }
}).start();  // Launch thread immediately
```

- **private void initStars()**

```
private void initStars() {
   // Clear existing stars from collection
   stars.clear();

   // Create random number generator
   Random rand = new Random();

   // Generate random number of stars (between 100 and 199)
   int numStars = rand.nextInt(100) + 100;
```

```
    // Create and add new stars
    for (int i = 0; i < numStars; i++) {
        // Each star positioned within screen bounds
        stars.add(new Star(width, height));
    }
```

Game State Management:

- **private void resetGame()**

The resetGame method resets the game to its initial state by clearing all game objects (rockets and bullets), resetting the player's position and properties, returning the score to zero, and bringing the game back to the start screen.

```
private void resetGame() {
    // Reset game score
    score = 0;

    // Reset rocket speed to initial value
    currentRocketSpeed = 0.3f;

    // Clear all game objects
    rockets.clear(); // Remove all rockets
    bullets.clear(); // Remove all bullets

    // Reset player position and state
    player.changeLocation(150, 150); // Move player to starting position
    player.reset(); // Reset player properties

    // Return to start screen
    gameStarted = false;
}
```

- **private void updateHighScore()**

This method checks and updates the highest score when the current score exceeds the previous record.

```
private void updateHighScore() {
    // Check if current score is higher than the highest score
```

```
    if (score > highestScore) {
        // Update the highest score with current score
        highestScore = score;

        // Save the new high score to persistent storage
        saveHighScore();
    }
```

- **private void saveHighScore()**

This method saves the current highest score to a text file for persistent storage.

```
private void saveHighScore() {
    //Create a BufferedWriter with auto-closing using try-with-resources
    try (BufferedWriter writer = new BufferedWriter(
            new FileWriter("src/Component/highscore.txt"))) {

        // Convert highestScore to String and write to file
        writer.write(String.valueOf(highestScore));

    } catch (IOException e) {
        // Print stack trace if file operation fails
        e.printStackTrace();
    }
```

The method saves a single high score value, because `FileWriter` is used in default mode (not append mode) and each write operation completely overwrites the previous file content. Therefore, it only writes `highestScore` value and previous score is always replaced.

- **private void loadHighScore()**

This method retrieves the previously saved highest score from the text file when the game starts.

```
private void loadHighScore() {
    // Create a BufferedReader with auto-closing using try-with-resources
    try (BufferedReader reader = new BufferedReader(
            new FileReader("src/Component/highscore.txt"))) {

        // Read the first line from file
```

```
        String line = reader.readLine();

        // Convert string to integer if line exists
        if (line != null) {
            highestScore = Integer.parseInt(line);
        }

    } catch (IOException e) {
        // Print stack trace if file operation fails
        e.printStackTrace();
    }
```

- **private void resetHighScore()**

This method resets the highest score to zero and saves the reset value to the file.

```
private void resetHighScore() {
    // Reset highest score to zero
    highestScore = 0;

    // Save the reset score to file
    saveHighScore();
```

Rendering and Graphics:

- **private void drawBackground()**

This method creates a space-like atmosphere by drawing the game background with a gradient effect and animated stars.

```
private void drawBackground() {
     // 1. Create gradient effect from dark blue to lighter blue
    GradientPaint gp = new GradientPaint(
        0, 0, new Color(0, 0, 20),      // Start color (very dark blue)
        0, height, new Color(20, 20, 40) // End color (slightly lighter blue)
    );
    g2.setPaint(gp);
    g2.fillRect(0, 0, width, height);    // Fill entire screen

    // 2. Render animated stars
    for (Star star : stars) {
        star.update();  // Update star position
```

```
        star.draw(g2);  // Draw star on screen
    }

    // 3. Reset transparency settings
    g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 1.0f));
```

The background system uses some Graphics2D components: GradientPaint creates a smooth transition between colors, Color objects define specific RGB values for the space theme, and AlphaComposite handles transparency levels in the rendering.

The drawing process employs three main methods: setPaint() applies the gradient effect to the graphics context, fillRect() creates the full background rectangle, and setComposite() manages the transparency settings for proper layering.

The game elements work together where Star objects provide the animated elements, screen dimensions (width and height) determine the drawing area, and the Graphics2D context (g2) serves as the canvas for all rendering operations.

- **private void drawGame()**

This method handles the rendering of all game objects, including the player character, projectiles (bullets), enemy rockets, visual effects, and user interface elements.

```
private void drawGame() {
    // 1. Player Rendering
    if (player.isAlive()) {
        player.draw(g2);
    }

    // 2. Game Objects Rendering
    // Draw bullets
    for (int i = 0; i < bullets.size(); i++) {
        Bullet bullet = bullets.get(i);
        if (bullet != null) bullet.draw(g2);
    }

    // Draw rockets (enemies)
    for (int i = 0; i < rockets.size(); i++) {
        Rocket rocket = rockets.get(i);
```

```
        if (rocket != null) rocket.draw(g2);
    }

    // Draw explosion effects
    for (int i = 0; i < boomEffects.size(); i++) {
        Effect boomEffect = boomEffects.get(i);
        if (boomEffect != null) boomEffect.draw(g2);
    }

    // 3. UI Rendering
    // Score display
    g2.setColor(Color.WHITE);
    g2.setFont(getFont().deriveFont(Font.BOLD, 15f));
    g2.drawString("Score : " + score, 10, 20);
    g2.drawString("Highest Score : " + highestScore, 10, 40);

    // 4. Game Over Screen
    if (!player.isAlive()) {
        updateHighScore();
        // Center-aligned text calculations and rendering
        // ... game over text and continue prompt
    }
```

The rendering system utilizes some Graphics Components: Graphics2D (g2) serves as the main rendering context, Color handles text coloring, Font manages text styling and sizing, and FontMetrics provides precise text measurement and positioning capabilities.

The system employs essential Drawing Methods where draw() handles the rendering of game objects, drawString() manages text display, and deriveFont() allows for dynamic font property modifications.

Various Game Elements work together in the rendering process, including the player object, collections of game objects (bullets, rockets, and effects), score tracking system, and a game over screen with informative text.

The UI Calculations ensure precise positioning where FontMetrics handles text positioning, Rectangle2D manages text boundary calculations, and center alignment

calculations ensure properly centered text elements on the screen.

- **private void drawStartScreen()**

This method creates a start screen with centered text elements and clear control instructions:

```
private void drawStartScreen() {
   // 1. Basic Setup
   g2.setColor(Color.WHITE);

   // 2. Title Rendering
   g2.setFont(getFont().deriveFont(Font.BOLD, 50f));
   String title = "PLANE AND ROCKET";
   FontMetrics fm = g2.getFontMetrics();
   int titleX = (width - fm.stringWidth(title)) / 2; // Center horizontally
   g2.drawString(title, titleX, height / 3); // Position at top third

   // 3. High Score Display
   g2.setFont(getFont().deriveFont(Font.PLAIN, 20f));
   String highScoreText = "Highest Score: " + highestScore;
   fm = g2.getFontMetrics();
   int highScoreX = (width - fm.stringWidth(highScoreText)) / 2;
   g2.drawString(highScoreText, highScoreX, height / 3 + fm.getHeight() + 10);

   // 4. Control Instructions
   String[] controls = {
       "Controls:",
       "A/D - Rotate ship",
       "SPACE - Speed up",
       "J - Small bullet",
       "K - Big bullet",
       "",
       "Press ENTER to start"
   };

   // Draw each line centered
   fm = g2.getFontMetrics();
```

```
    int y = height / 2;
    for (String line : controls) {
        int x = (width - fm.stringWidth(line)) / 2;
        g2.drawString(line, x, y);
        y += fm.getHeight() + 10; // Move down for next line
    }
```

- **private void render()**

This method implements double buffering by drawing the pre-rendered image to the screen, preventing flickering and ensuring smooth graphics display. The `dispose()` call is crucial for proper resource management:

```
private void render() {
    // Get graphics context from component
    Graphics g = getGraphics();

    // Draw buffered image to screen
    g.drawImage(image, 0, 0, null);

    // Clean up graphics resources
    g.dispose();
```

The `dispose()` method releases graphics resources and system memory, preventing memory leaks and maintaining stable game performance.

m. Main.java

The Main class extends JFrame to create the main game window. It serves as the entry point and container for all game components.

```
public class Main extends JFrame {
    public Main() {
        init();
}
```

Window Setup:

```
private void init() {
    // Set the title of the application window to "Plane and Rocket - OOP Project".
    setTitle("Plane and Rocket - OOP Project");
```

```
    // Set the size of the window to 1366x768 pixels.
    setSize(1366, 768);

    setLocationRelativeTo(null);
    // Center the window on the screen.

    // Prevent the user from resizing the window.
    setResizable(false);

    // Set the default close operation to exit the application.
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Game Panel Integration:

```
    // Use BorderLayout as the layout manager for the window.
    setLayout(new BorderLayout());

    // Create an instance of PanelGame, which might be part of the game interface.
    PanelGame panelGame = new PanelGame();

    // Add the panelGame to the main window.
    add(panelGame);
```

Game Start Mechanism:

```
    // Add a WindowListener to listen for the window opening event.
    // When the window is opened, call the start() method of panelGame.
    addWindowListener(new WindowAdapter() {
        @Override
        public void windowOpened(WindowEvent e) {
            panelGame.start();
        }
    });
```

Application Launch:

```
public static void main(String[] args) {
    Main main = new Main();
    main.setVisible(true);
}
```

CHAPTER 4: FINAL APP GAME

4.1 Source code (link github):

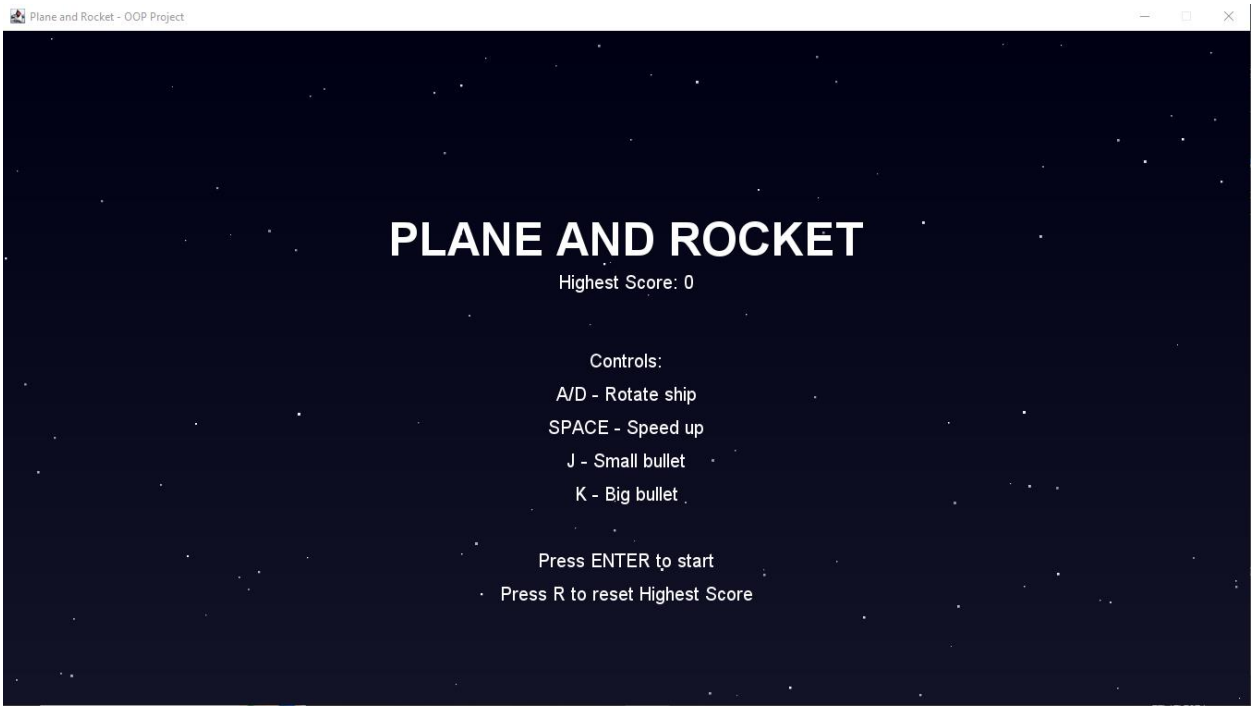
DuNhan1930/OOP-Project

4.2 Demo video:

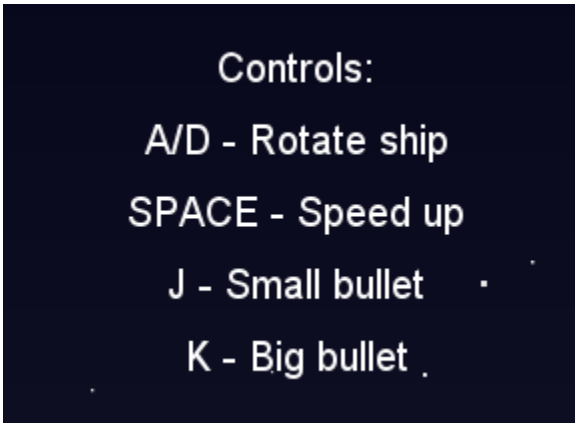
Plane and Rocket - OOP Project.mp4 - Google Drive

4.3 Instruction:

a. Main Menu



b. How to Play:

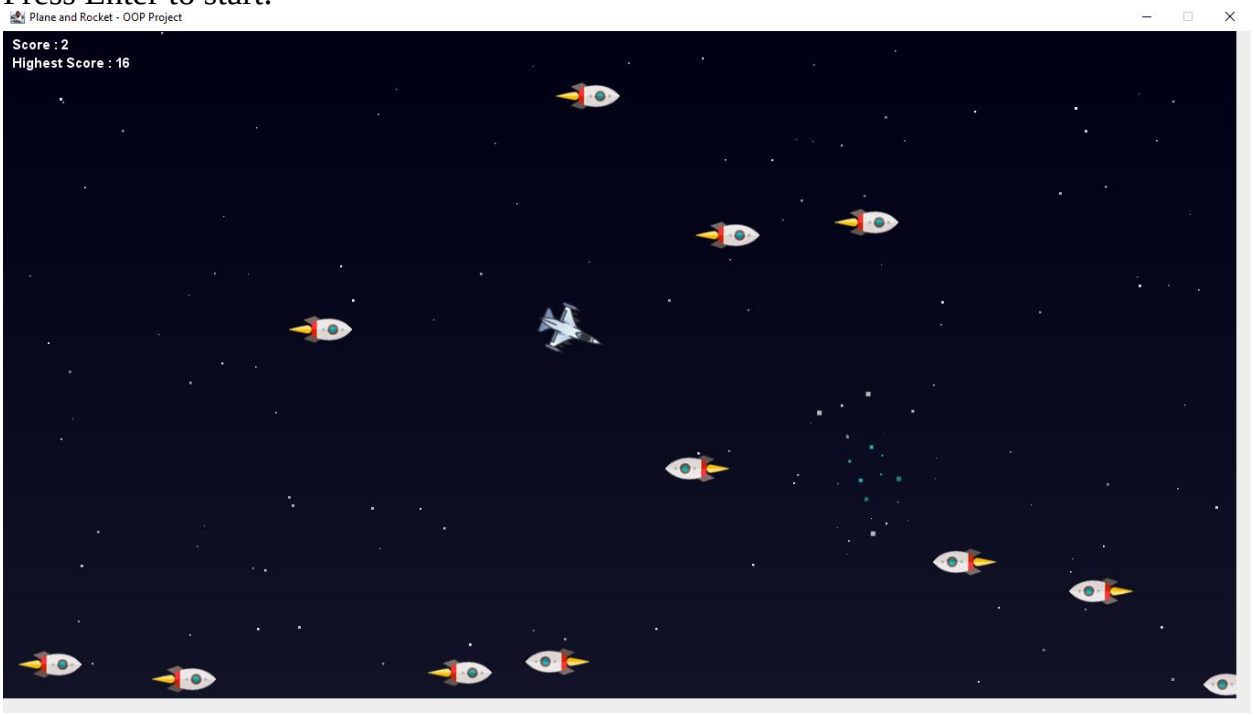


c. Reset Score

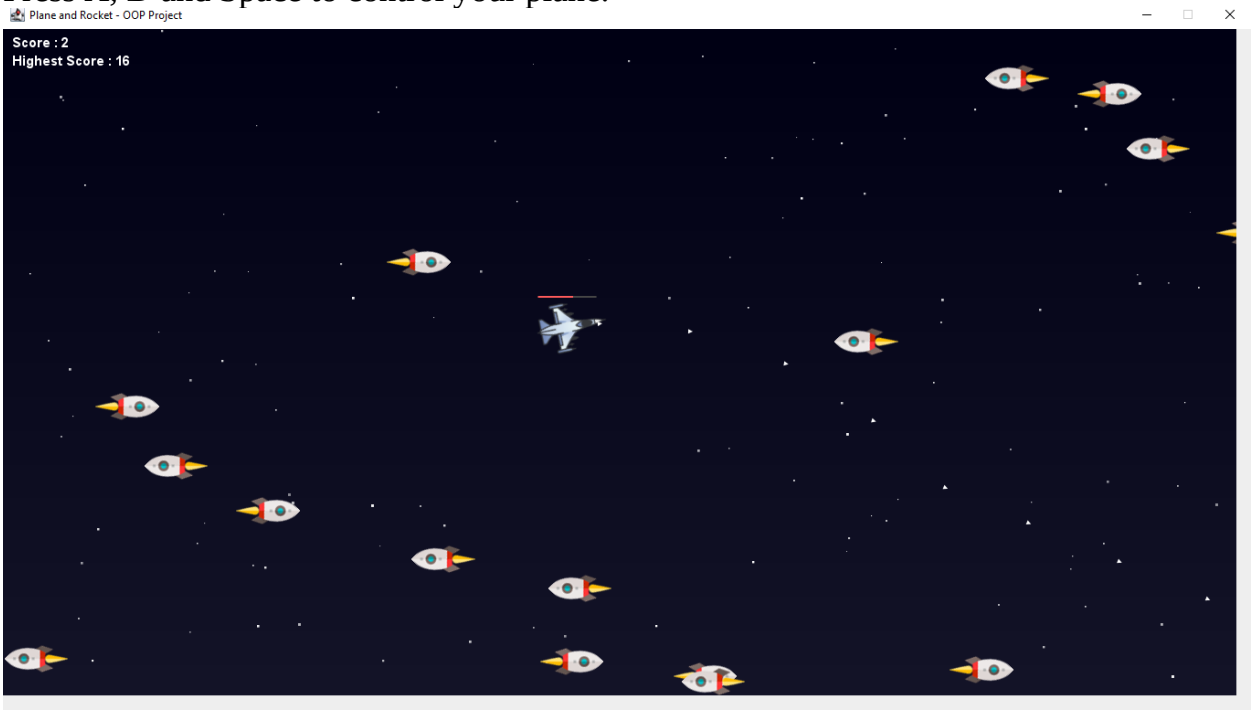
Challenge your friends to beat your high score, and use the 'R' key to reset the highest score when you want to start a fresh competition.

d. Let's Play

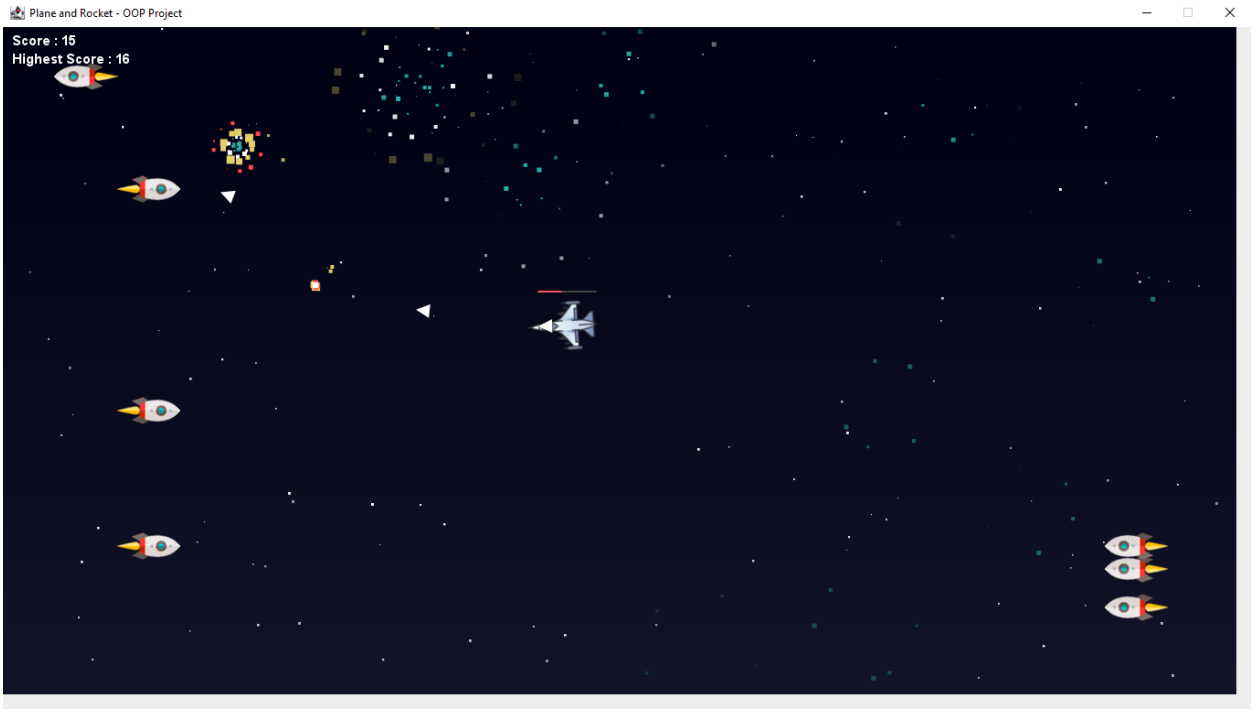
Press Enter to start.



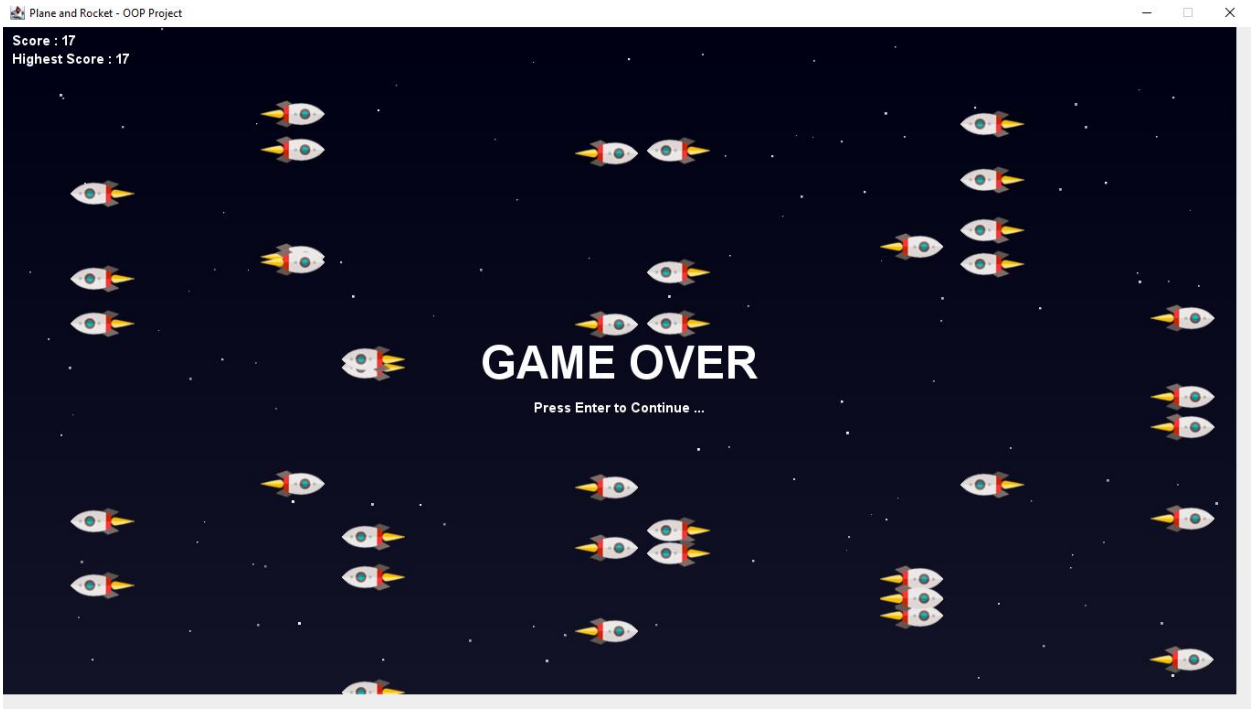
Press A, D and Space to control your plane.



Press 'J' to fire standard bullets for quick attacks, and 'K' to launch powerful big bullets against enemy rockets.



Your mission is to survive in the hostile space environment for as long as possible, dodging enemy fire while taking down threats.



When your ship is destroyed, simply press 'Enter' to immediately jump back into action and try again!

CHAPTER 5: EXPERIENCE

In this game project, the ultimate goal was to create an engaging and interactive experience through the interplay of planes, rockets, and bullets. After weeks of development, testing, and iteration, we have come to a significant realization:

A game's essence lies not just in its mechanics, but in the seamless integration of all its elements.

The process of building this game highlighted the importance of balance between technical implementation and the overall player experience. Beyond programming, we delved into crafting precise movement mechanics, dynamic interactions between game objects, and immersive visual effects. Every component, from player controls to sound effects, contributes to the fluidity and enjoyment of the game.

Working on this project allowed us to put theoretical knowledge into practice. We refined our understanding of trigonometric calculations for object movement, object-oriented programming principles, and collision detection techniques. Debugging and optimizing various elements presented challenges that required creative problem-solving, reinforcing our technical skills and encouraging independent research outside the classroom.

Through this endeavor, we also learned the value of designing with the player in mind. Creating intuitive controls, dynamic interactions, and visually appealing animations was just as important as writing clean, efficient code.

Ultimately, this project underscored the necessity of continuous learning and adaptability in Computer Science. The field is vast and rapidly changing, demanding not only technical expertise but also creativity and a deep understanding of user experience.

Our team remains committed to improving and expanding this game, treating it as the foundation for future projects. With each iteration, we aim to create more polished and innovative games, further bridging the gap between technical development and player satisfaction.