JavaFX Galaga Project

By: Kritika Bissessur, Jameel Edoo, Yassir Hossan Buksh

To: Mr Ramoth Khadimoullah

2024

17th March

Table of Contents

1 Abstract

The JavaFX Galaga project aimed to develop a modern version of the classic Galaga arcade game using JavaFX technology. This summary encapsulates the key highlights and outcomes of the project.

The project set out to recreate the iconic gameplay of Galaga while leveraging the capabilities of JavaFX for graphics and user interface design. Throughout the development process, the team adopted an Agile methodology, facilitating iterative development cycles and continuous feedback. Collaboration was streamlined through the use of GitHub, enabling team members to easily share code, track changes, and manage project tasks.

Furthermore, key features of the JavaFX Galaga game included faithful recreation of the original gameplay mechanics, intuitive user interface elements, and modern graphics. Despite encountering challenges such as integrating complex gameplay mechanics and optimizing performance, the team successfully delivered a fully functional game.

As a result, the project achieved significant milestones, including the completion of core gameplay mechanics, implementation of advanced features, and successful testing across multiple platforms. Looking ahead, there are opportunities for future enhancements such as additional levels, multiplayer functionality, and performance optimization.

In conclusion, the JavaFX Galaga project represents a successful endeavor in modernizing a classic arcade game using JavaFX technology. The project demonstrated effective teamwork, technical proficiency, and creativity in game development, paving the way for engaging gaming experiences in the future.

2 Introduction

GalagaRacerFX is a thrilling 3D space racing game that pays homage to the classic Pseudo-Racers of yesteryears. Developed as a culmination of our efforts in an Object-Oriented Programming (OOP) class, this project represents our passion for gaming and our commitment to mastering programming principles in a practical setting. Inspired by the timeless appeal of retro arcade games, GalagaRacerFX offers players an exhilarating journey through space, blending nostalgia with modern technology to create an immersive racing experience like no other.

At its core, GalagaRacerFX aims to capture the essence of classic space racing games while infusing it with contemporary elements to delight both seasoned gamers and newcomers alike. Drawing inspiration from iconic titles of the past, we've crafted a game that combines the thrill of high-speed racing with the charm of retro aesthetics. Players are invited to navigate through challenging lanes, dodge obstacles, and compete against opponents in a visually stunning 3D environment that evokes a sense of nostalgia while pushing the boundaries of modern game development.

Through meticulous attention to detail and a deep understanding of Object-Oriented Programming principles, we've strived to create a game that not only entertains but also educates. Each aspect of GalagaRacerFX, from its intuitive controls to its dynamic gameplay mechanics, serves as a testament to our dedication to excellence in game design and development. As you embark on your journey through space in GalagaRacerFX, we invite you to experience the thrill of retro-inspired racing brought to life in a modern, JavaFX-powered gaming experience.

3 Technology stack

In crafting GalagaRacerFX, we meticulously curated a technology stack to infuse life into our game and offer players an immersive experience. At the heart of our development lies JavaFX, a choice made for its versatility, user-friendly nature, and robust support for creating captivating graphical user interfaces (GUIs). JavaFX, being a modern and feature-rich framework, equips developers with an array of tools and components, making it the perfect fit for a dynamic game like GalagaRacerFX.

Java, our language of choice and game mechanics, seamlessly integrates with JavaFX. Leveraging Java's object-oriented paradigm and extensive libraries, we bring to life the core functionalities of the game, from player controls to collision detection and game physics. The synergy between Java and JavaFX ensures a smooth and responsive gaming experience that meets the expectations of today's players.

To augment our development process, we harness the power of external libraries and tools, enriching our game with features beyond the scope of JavaFX and Java alone. These supplementary resources include libraries for 3D graphics rendering, audio playback, and input handling, all of which enhance the depth and interactivity of GalagaRacerFX.

In essence, our technology stack is a carefully curated ensemble, tailored to the unique requirements and aspirations of GalagaRacerFX. By combining the strengths of JavaFX, Java, and complementary libraries, we deliver a polished and feature-rich gaming experience that seamlessly merges the nostalgia of classic space racing games with the advancements of modern technology.

4 Design and analysis

4.1 Analysis

In order to fully comprehend the purposes and specifications of each component, we thoroughly examine every aspect of the game. For the camera system to follow the movements of the spaceship in a dynamic manner, tracking must be done smoothly. For a smooth player experience, issues like occlusion and dynamic level design must be resolved. Since the title screen is where the game begins, layout, button design, and user interaction flow must all be carefully considered. In a similar vein, careful consideration of layout design, font choice, and information hierarchy is necessary for the static interface, which includes the HUD and pause screen, in order to convey the game state.

The spaceship's design encompasses movement physics, collision detection, and player controls. It's essential to balance responsiveness with realism to provide an immersive piloting experience. Additionally, the lane design, with its alternating pillars and environmental hazards, adds depth to the gameplay. Procedural generation techniques and dynamic elements contribute to replay ability and strategic depth, requiring careful planning and implementation.

4.2 Design

In the design phase, we focus on shaping the fundamental aspects of the game to ensure a cohesive and engaging player experience. This involves defining the game concept, mechanics, environments, and user interface elements.

Next, we delve into environment design, where we conceptualize the game world, levels, and interactive elements. This involves sketching out layouts, considering visual aesthetics, and creating plans for environmental storytelling.

The user interface (UI) design is crucial for providing players with intuitive controls and relevant information. We design menus, HUD elements, buttons, and interactive components to ensure clarity, accessibility, and seamless navigation throughout the game.

Overall, the design phase lays the foundation for the game's development, guiding decisions on art direction, gameplay mechanics, and player interactions. It sets the stage for a coherent and immersive gaming experience that resonates with the target audience.

4.3 Implementation

Implementation involves translating design concepts into functional code, leveraging JavaFX's capabilities for rendering graphics and handling user input. Camera scripts are coded using object-oriented principles, encapsulating functionality and promoting code reusability.

Title screen and static interface implementation involves creating JavaFX nodes and managing scene transitions based on user actions. Button actions are bound to event listeners or command patterns, enabling seamless navigation between game states. Spaceship implementation focuses on movement physics and collision detection, ensuring smooth and responsive controls. Lane generation algorithms handle procedural lane creation and dynamic element placement, providing a diverse and challenging gameplay experience for players.

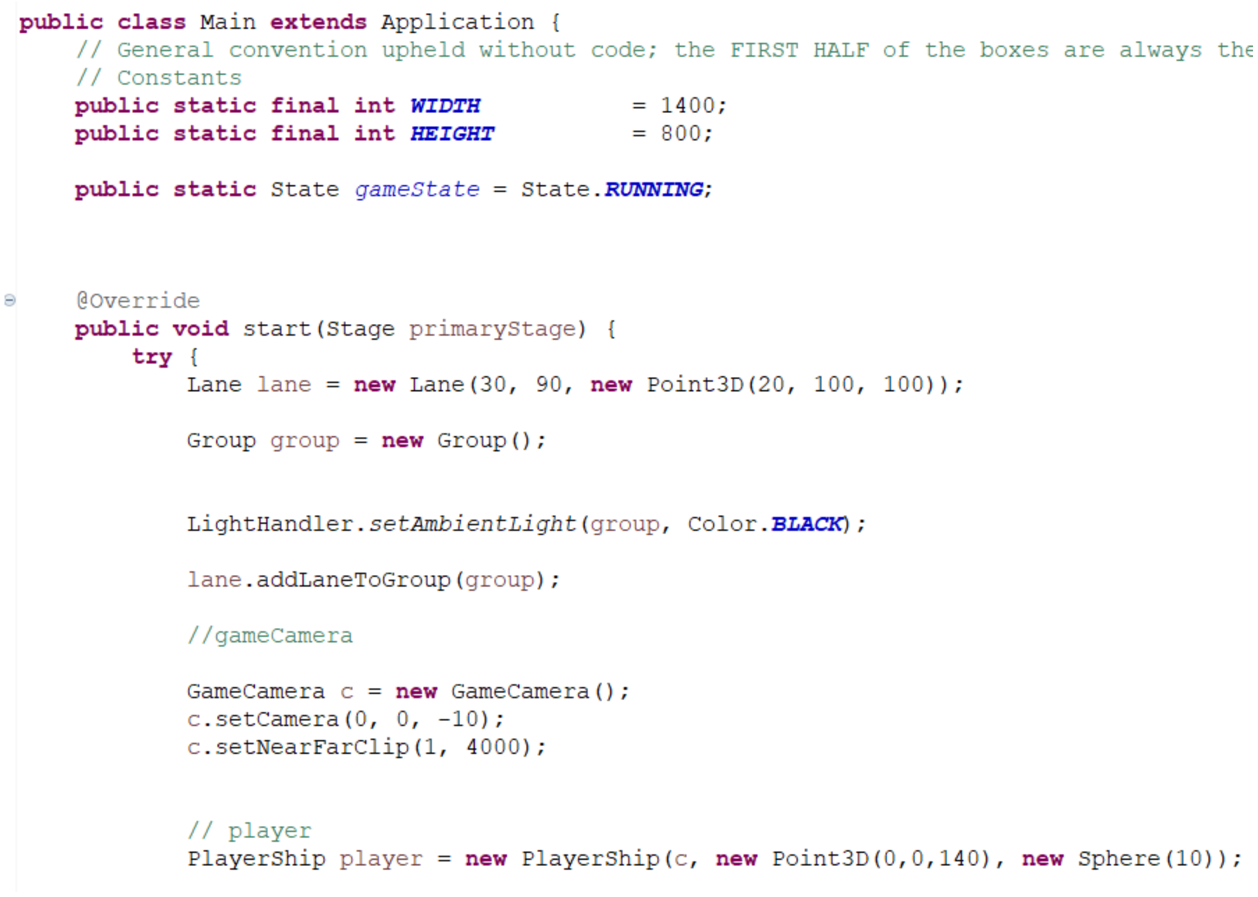
5 Conclusion

In conclusion, 'Galaga Racer' represents a unique fusion of classic arcade elements from the 'Galaga' franchise with the exhilarating experience of futuristic racing. By blending nostalgic charm with modern gameplay mechanics, the game offers players an engaging and immersive journey through space. Its captivating visuals, challenging races, and familiar alien adversaries ensure an enjoyable experience for both long-time fans and newcomers to the series. 'Galaga Racer' stands as a testament to the enduring appeal of retro gaming, reimagined for contemporary audiences.

6 Appendix

6.1 Code explanation

6.1.1 Main.java



```java
public class Main extends Application {
    // General convention upheld without code; the FIRST HALF of the boxes are always the
    // Constants
    public static final int WIDTH          = 1400;
    public static final int HEIGHT         = 800;

    public static State gameState = State.RUNNING;



    @Override
    public void start(Stage primaryStage) {
        try {
            Lane lane = new Lane(30, 90, new Point3D(20, 100, 100));

            Group group = new Group();


            LightHandler.setAmbientLight(group, Color.BLACK);

            lane.addLaneToGroup(group);

            //gameCamera

            GameCamera c = new GameCamera();
            c.setCamera(0, 0, -10);
            c.setNearFarClip(1, 4000);


            // player
            PlayerShip player = new PlayerShip(c, new Point3D(0,0,140), new Sphere(10));
```

1. PlayerShip Object Creation:

- PlayerShip is a class representing the player's ship in the game.

- An instance of "PlayerShip" is created using the constructor "new PlayerShip(...)".

- The constructor requires three parameters:

  - "c": This parameter represents the "GameCamera" object ("c") that controls the viewpoint of the game. The player's ship is associated with this camera to determine its position and orientation within the game world.

  - "new Point3D(0,0,140)": This parameter specifies the initial position of the player's ship in the game world. It's a "Point3D" object representing a point in 3D space with coordinates (x=0, y=0, z=140). This means the ship starts at the center of the game world with a z-coordinate of 140 units, which likely corresponds to its initial depth along the racing track.

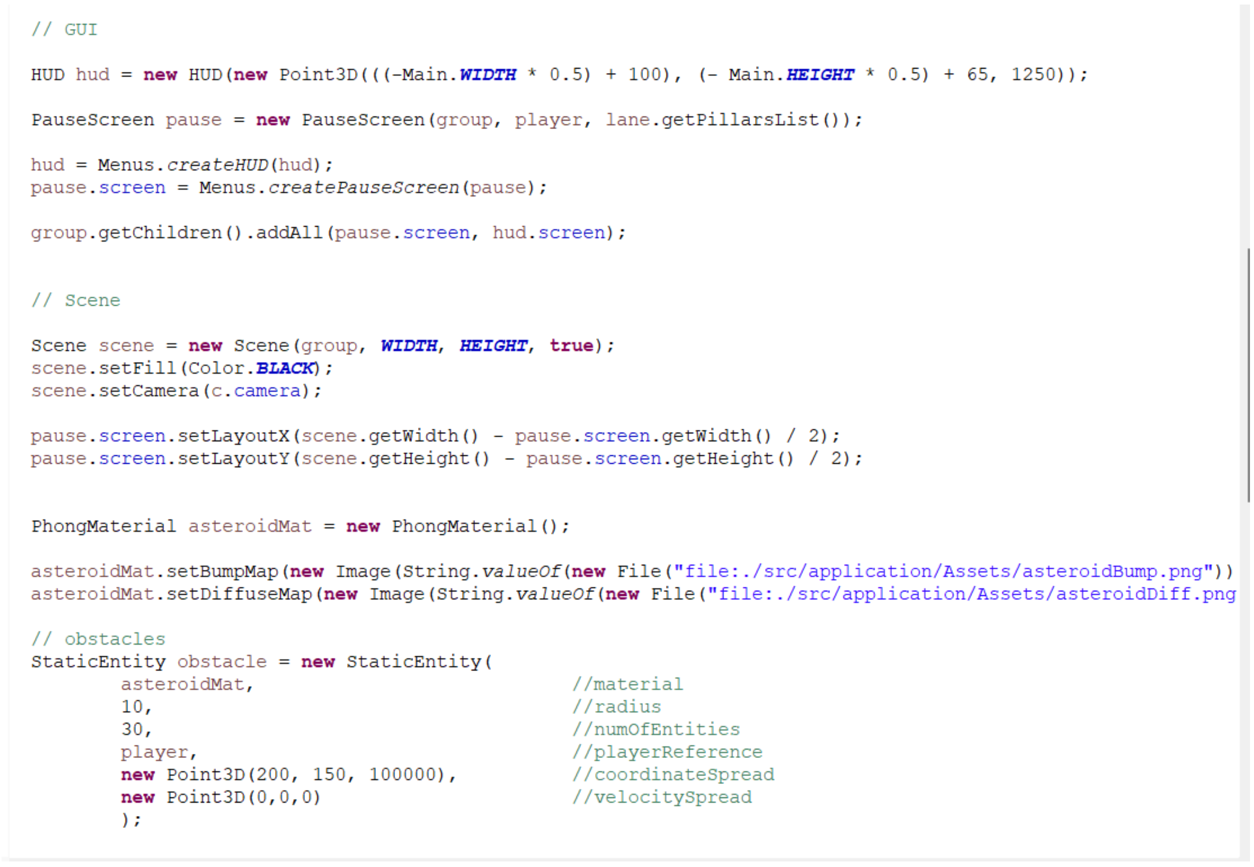
- "new Sphere(10)": This parameter defines the visual representation of the player's ship. It's a "Sphere" object with a radius of 10 units, indicating that the ship's model is spherical in shape with a diameter of 20 units.

2. Initialization:

- Once the "PlayerShip" object is created, it's ready to be used in the game.

- The ship's properties, such as its position, orientation, and visual representation, are set based on the provided parameters.

- In the game loop or elsewhere in the code, the "PlayerShip" object will likely be updated and rendered to reflect the player's interactions and movements within the game world.



```
// GUI

HUD hud = new HUD(new Point3D(((-Main.WIDTH * 0.5) + 100), (- Main.HEIGHT * 0.5) + 65, 1250));

PauseScreen pause = new PauseScreen(group, player, lane.getPillarsList());

hud = Menus.createHUD(hud);
pause.screen = Menus.createPauseScreen(pause);

group.getChildren().addAll(pause.screen, hud.screen);


// Scene

Scene scene = new Scene(group, WIDTH, HEIGHT, true);
scene.setFill(Color.BLACK);
scene.setCamera(c.camera);

pause.screen.setLayoutX(scene.getWidth() - pause.screen.getWidth() / 2);
pause.screen.setLayoutY(scene.getHeight() - pause.screen.getHeight() / 2);


PhongMaterial asteroidMat = new PhongMaterial();

asteroidMat.setBumpMap(new Image(String.valueOf(new File("file:./src/application/Assets/asteroidBump.png"))
asteroidMat.setDiffuseMap(new Image(String.valueOf(new File("file:./src/application/Assets/asteroidDiff.png

// obstacles
StaticEntity obstacle = new StaticEntity(
        asteroidMat,                            //material
        10,                                     //radius
        30,                                     //numOfEntities
        player,                                 //playerReference
        new Point3D(200, 150, 100000),          //coordinateSpread
        new Point3D(0,0,0)                      //velocitySpread
        );
```

6.1.2 GUI

```
HUD hud = new HUD(new Point3D(((-Main.WIDTH * 0.5) + 100), (- Main.HEIGHT * 0.5) + 65, 1250));


PauseScreen pause = new PauseScreen(group, player, lane.getPillarsList());


hud = Menus.createHUD(hud);

pause.screen = Menus.createPauseScreen(pause);


group.getChildren().addAll(pause.screen, hud.screen);
"""
```

1. HUD and Pause Screen Initialization:

- "HUD" and "PauseScreen" are classes representing the heads-up display and pause screen, respectively.

- An instance of "HUD" is created with the constructor "new HUD(...)", passing a "Point3D" object as a parameter. This point represents the initial position of the HUD in 3D space.

- Similarly, an instance of "PauseScreen" is created with the constructor "new PauseScreen(...)". It takes three parameters: a "Group" object ("group"), a "PlayerShip" object ("player"), and a list of pillars ("lane.getPillarsList()").

- The "HUD" and "PauseScreen" objects are then customized using the "Menus" class, which provides methods to create and customize the HUD and pause screen interfaces.

- Finally, both the pause screen and HUD are added to the "group" (a "Group" object representing the scene's root node) using the "addAll" method.

2. Scene Initialization:

- A new "Scene" object is created with the "group" as its root node and the specified dimensions ("WIDTH" and "HEIGHT").

- The scene's background color is set to black using "scene.setFill(Color.BLACK)".

- The camera for the scene is set to the "GameCamera" object's camera ("c.camera") to control the viewpoint during gameplay.

3. Pause Screen Layout:

- The layout of the pause screen is adjusted to center it within the scene's dimensions. This is achieved by setting its layout coordinates ("setLayoutX" and "setLayoutY") relative to the scene's width and height.

4. Obstacle Initialization:

- A "PhongMaterial" object ("asteroidMat") is created to define the material properties of the obstacles.

- Images representing the bump map and diffuse map of the material are loaded from files using the "Image" class and set to the material using "setBumpMap" and "setDiffuseMap".

- An instance of "StaticEntity" is created to represent the obstacles. It takes several parameters, including the material ("asteroidMat"), radius, number of entities, player reference ("player"), coordinate spread, and velocity spread. These parameters determine the properties and behavior of the obstacles in the game world.

6.1.3 star.java

```java
// star particle effect
StaticEntity star_particles = new StaticEntity(
        new PhongMaterial(Color.WHITE),         //material
        0.4,                                    //radius
        700,                                    //numOfEntities
        player,                                 //playerReference
        new Point3D(500, 500, 100000),          //coordinateSpread
        new Point3D(0,0,0)                      //velocitySpread
        );

group.getChildren().add(obstacle.getEntityGroup());
group.getChildren().add(star_particles.getEntityGroup());

player.setCameraOffset(new Point3D(0, -20, -150));

group.getChildren().add(player.getShipModel());


ControlShip controller = new ControlShip(player, group, hud, pause, scene, 10, 50.0, 0.05);

LightHandler.addLightInstance(group, Color.WHITE, new Point3D(0,0,-100));


LightInstance headlight = LightHandler.addLightInstance(group, Color.WHITE,  player.getCurrPosition() );
LightHandler.bindLightToObject(headlight, player.getShipModel(), new Point3D(0,0,5000));

group.getChildren().add(controller.particleGroup);

// UI Offset
Point3D UI_Offset = new Point3D((-WIDTH * 1.5) + 605, (-HEIGHT * 1.5) + 280, 500);

LightHandler.getAllLightSources(primaryStage);
controller.startGameLoop(lane, UI_Offset, c, obstacle, star_particles);
```

1. Star Particle Effect Initialization:

- A "StaticEntity" representing the star particle effect is created.

- It uses a "PhongMaterial" with the color set to "WHITE" to represent the appearance of the particles.

- The parameters passed to the "StaticEntity" constructor include the material, radius (0.4 units), number of entities (700 particles), player reference ("player"), coordinate spread (500, 500, 100000), and velocity spread (0, 0, 0).

- The star particle effect is added to the "group" by including its entity group in the scene.

2. Player Configuration:

- The camera offset for the player is set to adjust the viewpoint relative to the player's ship. This offset is applied to the camera's position to provide a suitable perspective during gameplay.

- The player's ship model is added to the "group".

3. Control Ship:

- An instance of "ControlShip" is created, responsible for controlling the player's ship during gameplay.

- It takes several parameters, including the player object, group, HUD, pause screen, scene, speed, turning speed, and acceleration.

4. Lights:

- Light instances are added to the scene to enhance the visual effects.

- One light is positioned at (0,0,-100) in the scene, while another is bound to the player's ship model at its current position with an offset of (0,0,5000).

5. User Interface Offset:

- An offset is defined for the user interface elements to position them correctly within the game window.

6.1.4 Title screen

```java
// Create the root group for the title screen
Group titleScreenGroup = new Group();

// Add the "title-screen" class to the title screen group
titleScreenGroup.getStyleClass().add("title-screen");

// Set black background for the title screen
titleScreenGroup.setStyle("-fx-background-color: black;");

// Create falling stars
createFallingStars(titleScreenGroup);
```

```java
Image titleImage = new Image("file:./src/application/Assets/galrace.jpeg");
ImageView imageView = new ImageView(titleImage);
imageView.setFitWidth(titleImage.getWidth());
imageView.setFitHeight(titleImage.getHeight());
imageView.setLayoutX((WIDTH - titleImage.getWidth()) / 2);
imageView.setLayoutY((HEIGHT - titleImage.getHeight()) / 2);

// Add the image to the title screen group
titleScreenGroup.getChildren().add(imageView);

// Create the "Start" button
Button startButton = new Button("Start");
startButton.setStyle("-fx-font-size: 20px; -fx-background-color: #333333; -fx-text-fill: white;");
startButton.setOnAction(event -> startGame(primaryStage));

// Position the button in the center of the screen
startButton.setLayoutX((WIDTH - 120) / 2);
startButton.setLayoutY((HEIGHT + imageView.getLayoutY() + imageView.getBoundsInLocal().getHeight()) / 2); // Adjust Y position
```

1. Create the root group for the title screen:

- "Group titleScreenGroup = new Group();": This creates a new instance of the "Group" class named "titleScreenGroup", which will serve as the root node for the title screen content.

2. Add the "title-screen" class to the title screen group:

- "titleScreenGroup.getStyleClass().add("title-screen");": This adds the CSS class name "title-screen" to the "titleScreenGroup". This class can be used to apply specific styling to elements within this group.

3. Set black background for the title screen:

- "titleScreenGroup.setStyle("-fx-background-color: black;");": This sets the background color of the "titleScreenGroup" to black using inline CSS styling.

4. Create falling stars:

- "createFallingStars(titleScreenGroup);": This calls a method "createFallingStars()" to generate falling stars within the "titleScreenGroup".

6. Add the image to the title screen group:

- "titleScreenGroup.getChildren().add(imageView);": This adds the "imageView" (containing the title image) to the "titleScreenGroup", so it will be displayed on the title screen.

7. Create the "Start" button:

- "Button startButton = new Button("Start");": Creates a new "Button" object with the text "Start".

- "startButton.setStyle("-fx-font-size: 20px; -fx-background-color: #333333; -fx-text-fill: white;");": Applies styling to the button, including font size, background color, and text color.

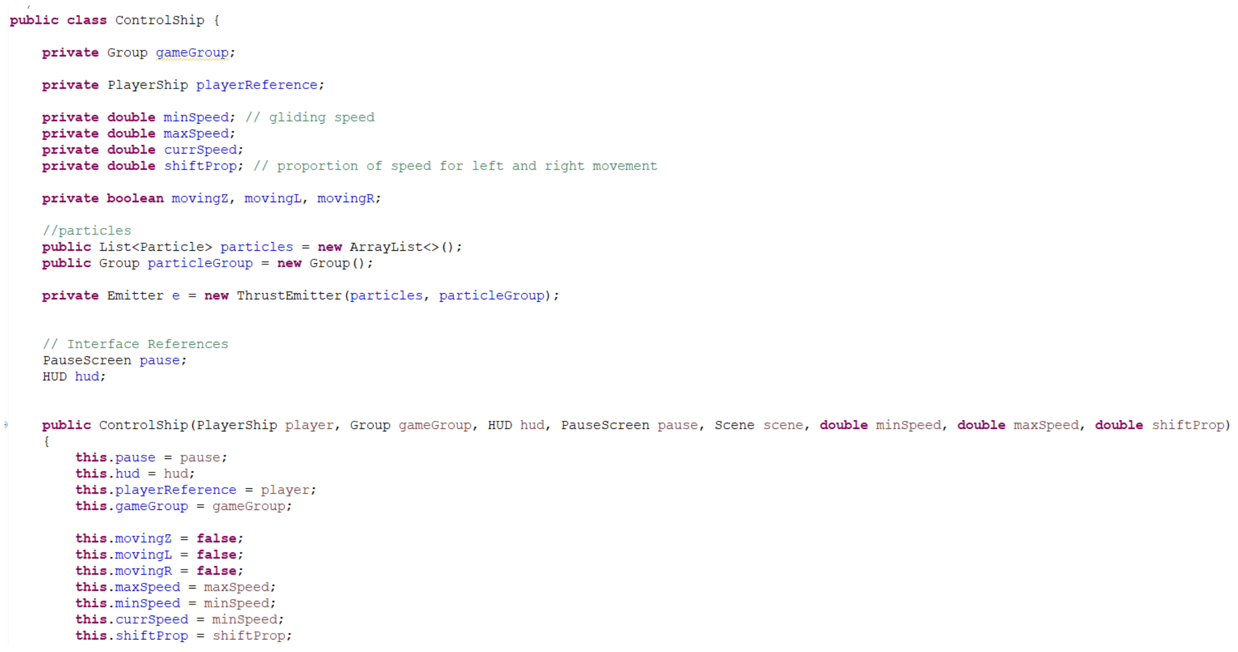
- "startButton.setOnAction(event -> startGame(primaryStage));": Sets an action event handler for the button, which calls the "startGame()" method when clicked.

8. Position the button in the center of the screen:

- "startButton.setLayoutX((WIDTH - 120) / 2);": Calculates the X-coordinate for the position of the button at the center of the screen horizontally.

- "startButton.setLayoutY((HEIGHT + imageView.getLayoutY() + imageView.getBoundsInLocal().getHeight()) / 2);": Calculates the Y-coordinate for the position of the button below the title image, ensuring it is vertically centered.

6.1.5 ControlShip.java



```java
public class ControlShip {

    private Group gameGroup;

    private PlayerShip playerReference;

    private double minSpeed; // gliding speed
    private double maxSpeed;
    private double currSpeed;
    private double shiftProp; // proportion of speed for left and right movement

    private boolean movingZ, movingL, movingR;

    //particles
    public List<Particle> particles = new ArrayList<>();
    public Group particleGroup = new Group();

    private Emitter e = new ThrustEmitter(particles, particleGroup);


    // Interface References
    PauseScreen pause;
    HUD hud;


    public ControlShip(PlayerShip player, Group gameGroup, HUD hud, PauseScreen pause, Scene scene, double minSpeed, double maxSpeed, double shiftProp)
    {
        this.pause = pause;
        this.hud = hud;
        this.playerReference = player;
        this.gameGroup = gameGroup;

        this.movingZ = false;
        this.movingL = false;
        this.movingR = false;
        this.maxSpeed = maxSpeed;
        this.minSpeed = minSpeed;
        this.currSpeed = minSpeed;
        this.shiftProp = shiftProp;
```

1. Instance Variables:

- "private Group gameGroup": This variable holds a reference to the main "Group" of the game scene, where game objects are added.

- "private PlayerShip playerReference": This variable holds a reference to the player's ship.

- "private double minSpeed": This variable represents the minimum speed of the player's ship (gliding speed).

- "private double maxSpeed": This variable represents the maximum speed of the player's ship.

- "private double currSpeed": This variable holds the current speed of the player's ship.

- "private double shiftProp": This variable represents the proportion of speed used for left and right movement.

- "private boolean movingZ, movingL, movingR": These variables indicate whether the player's ship is moving along the Z-axis (forward/backward) and whether it's moving left or right.

2. Particle Effects:

- "public List<Particle> particles": This list holds particles emitted by the ship.

- "public Group particleGroup": This "Group" contains all particle objects, allowing them to be managed collectively.

- "private Emitter e = new ThrustEmitter(particles, particleGroup)": This line initializes an emitter ("e") responsible for emitting particles. Here, it's specifically a thrust emitter ("ThrustEmitter"), which emits particles representing thrust from the ship's engines.

3. Interface References:

- "PauseScreen pause": This variable holds a reference to the pause screen interface.

- "HUD hud": This variable holds a reference to the heads-up display (HUD) interface.

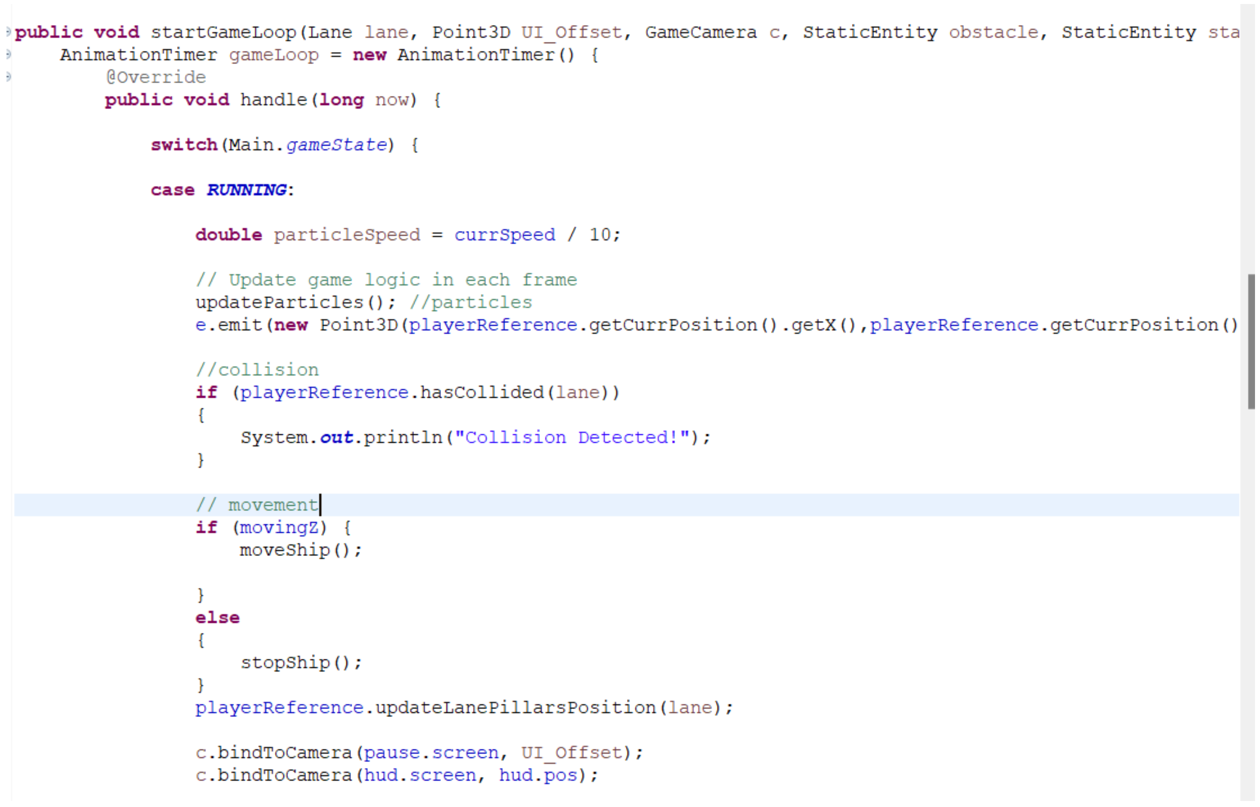
4. Constructor:

- The constructor "ControlShip" initializes a new instance of the "ControlShip" class.

- It takes several parameters: "PlayerShip" object ("player"), the main "Group" of the game ("gameGroup"), HUD ("hud"), pause screen ("pause"), the scene ("scene"), minimum speed ("minSpeed"), maximum speed ("maxSpeed"), and shift proportion ("shiftProp").

- The constructor sets the instance variables accordingly, initializes the movement flags ("movingZ", "movingL", "movingR"), and sets the initial speed of the player's ship ("currSpeed").

6.1.6 Spaceship



```
public void startGameLoop(Lane lane, Point3D UI_Offset, GameCamera c, StaticEntity obstacle, StaticEntity sta
    AnimationTimer gameLoop = new AnimationTimer() {
        @Override
        public void handle(long now) {

            switch(Main.gameState) {

            case RUNNING:

                double particleSpeed = currSpeed / 10;

                // Update game logic in each frame
                updateParticles(); //particles
                e.emit(new Point3D(playerReference.getCurrPosition().getX(),playerReference.getCurrPosition()

                //collision
                if (playerReference.hasCollided(lane))
                {
                    System.out.println("Collision Detected!");
                }

                // movement
                if (movingZ) {
                    moveShip();

                }
                else
                {
                    stopShip();
                }
                playerReference.updateLanePillarsPosition(lane);

                c.bindToCamera(pause.screen, UI_Offset);
                c.bindToCamera(hud.screen, hud.pos);
```

1. Method Signature:

- "public void startGameLoop(Lane lane, Point3D UI_Offset, GameCamera c, StaticEntity obstacle, StaticEntity stars)": This method starts the game loop and takes several parameters, including the racing lane ("lane"), user interface offset ("UI_Offset"), game camera ("c"), and static entities representing obstacles and stars ("obstacle" and "stars").

2. Animation Timer:

- Inside the method, an "AnimationTimer" named "gameLoop" is created. This timer is responsible for continuously updating the game logic and rendering frames.

3. Game State Check:

- The "switch" statement checks the current game state ("Main.gameState") to determine the actions to perform during each frame. In this case, it checks if the game is in the "RUNNING" state.

4. Particle Emission:

- "updateParticles()": This method updates the particles emitted by the ship.

- "e.emit(...)": This line emits particles from the ship's engines. The method "emit" is called on the emitter "e", passing the emission point, number of particles, and velocity parameters.

5. Collision Detection:

- "playerReference.hasCollided(lane)": This line checks if the player's ship has collided with any obstacles on the racing lane ("lane"). If a collision is detected, a message is printed indicating the collision.

6. Ship Movement:

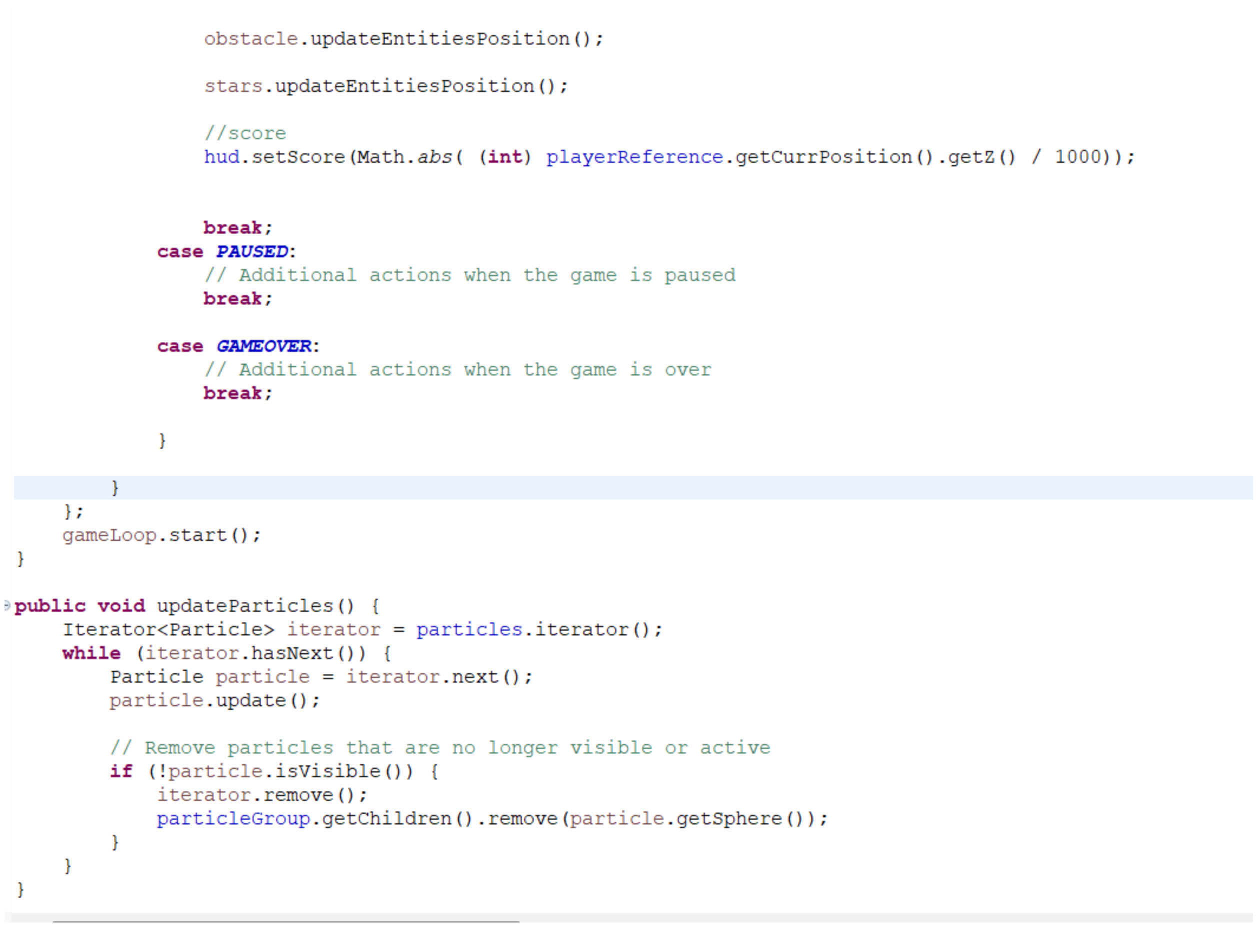
- "moveShip()" and "stopShip()": These methods handle the movement of the player's ship based on the "movingZ" flag. If the ship is set to move ("movingZ" is "true"), "moveShip()" is called to move the ship forward. Otherwise, "stopShip()" is called to stop the ship's movement.

- "playerReference.updateLanePillarsPosition(lane)": This method updates the positions of the lane pillars based on the movement of the player's ship.

7. Camera Binding:

- "c.bindToCamera(...)": These lines bind the game camera ("c") to the HUD and pause screen, ensuring that they remain fixed in the viewport regardless of the camera's movement.

6.1.7 Update particles



```
                obstacle.updateEntitiesPosition();

                stars.updateEntitiesPosition();

                //score
                hud.setScore(Math.abs( (int) playerReference.getCurrPosition().getZ() / 1000));


                break;
            case PAUSED:
                // Additional actions when the game is paused
                break;

            case GAMEOVER:
                // Additional actions when the game is over
                break;

            }

        }
    };
    gameLoop.start();
}

public void updateParticles() {
    Iterator<Particle> iterator = particles.iterator();
    while (iterator.hasNext()) {
        Particle particle = iterator.next();
        particle.update();

        // Remove particles that are no longer visible or active
        if (!particle.isVisible()) {
            iterator.remove();
            particleGroup.getChildren().remove(particle.getSphere());
        }
    }
}
```

1. Updating Entity Positions:

- "obstacle.updateEntitiesPosition()": This method call updates the positions of the obstacles in the game world. Presumably, this method updates the positions of static obstacles to simulate movement or adjust their positions based on the player's progress.

- "stars.updateEntitiesPosition()": Similarly, this method call updates the positions of the star entities in the game world. These stars may represent background elements or decorative objects that enhance the visual experience of the game.

2.Scoring:

- "hud.setScore(Math.abs((int) playerReference.getCurrPosition().getZ() / 1000))": This line updates the score displayed in the HUD. It calculates the score based on the player's current position along the Z-axis (likely representing distance traveled) and divides it by 1000. The "Math.abs()" function ensures that the score is always positive.

3. Particle Update ("updateParticles" method):

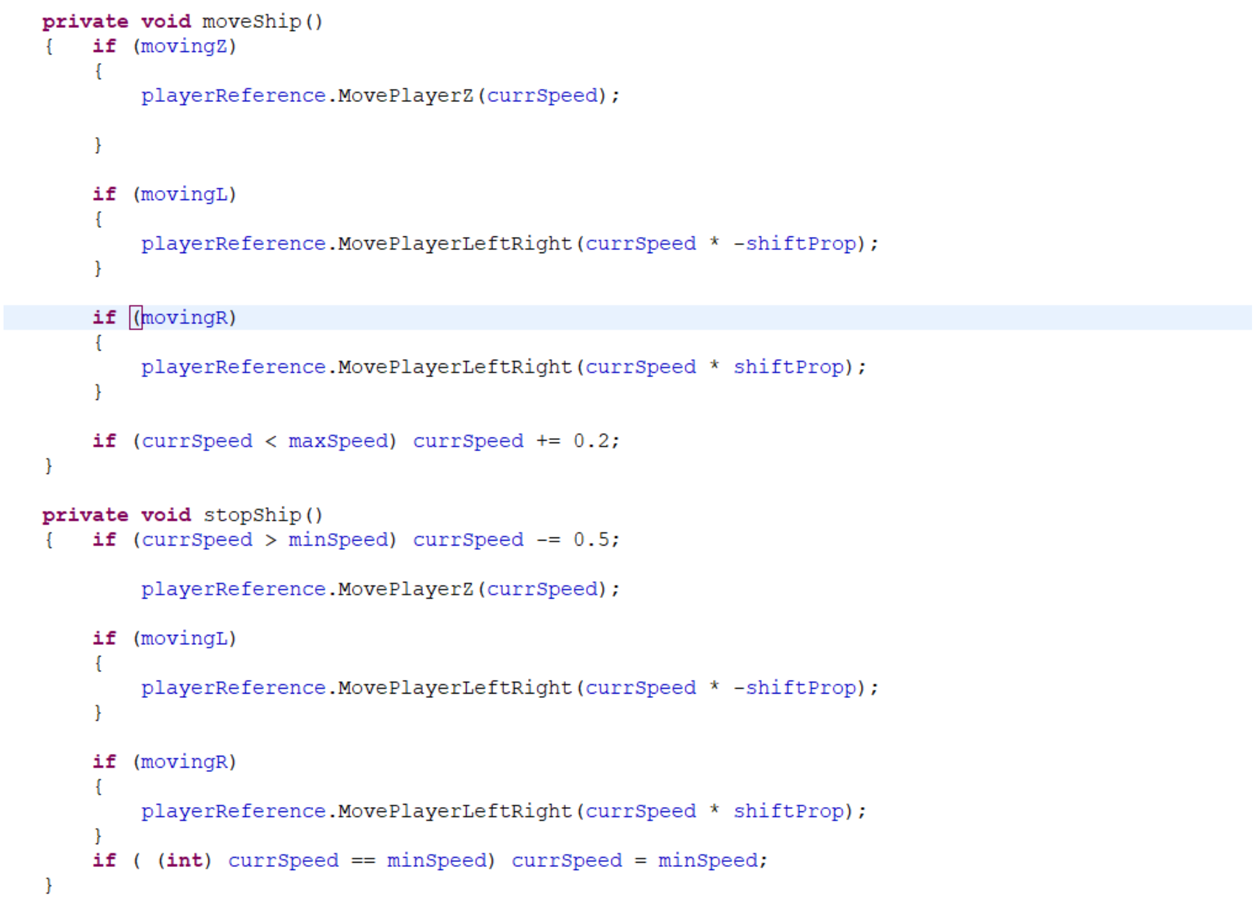
- The "updateParticles" method iterates through each particle in the "particles" list and updates its position and other properties.

- For each particle, the "update" method is called to advance its state (e.g., position, velocity, lifespan).

- Particles that are no longer visible or active are removed from the scene by removing them from the "particleGroup" and the "particles" list.

- This method ensures that the particle effects remain synchronized with the game's state and are properly managed to prevent performance issues.

6.1.8 Move ship/stop ship functionality



```
private void moveShip()
{   if (movingZ)
    {
        playerReference.MovePlayerZ(currSpeed);

    }

    if (movingL)
    {
        playerReference.MovePlayerLeftRight(currSpeed * -shiftProp);
    }

    if (movingR)
    {
        playerReference.MovePlayerLeftRight(currSpeed * shiftProp);
    }

    if (currSpeed < maxSpeed) currSpeed += 0.2;
}

private void stopShip()
{   if (currSpeed > minSpeed) currSpeed -= 0.5;

        playerReference.MovePlayerZ(currSpeed);

    if (movingL)
    {
        playerReference.MovePlayerLeftRight(currSpeed * -shiftProp);
    }

    if (movingR)
    {
        playerReference.MovePlayerLeftRight(currSpeed * shiftProp);
    }
    if ( (int) currSpeed == minSpeed) currSpeed = minSpeed;
}
```

1. "moveShip" Method:

- This method is responsible for moving the player's ship based on the current movement flags.

- Inside the method, there are conditional checks for each movement direction ("movingZ", "movingL", "movingR") to determine the action to take.

- If "movingZ" is "true", indicating forward/backward movement, the "MovePlayerZ" method of the "playerReference" object (presumably representing the player's ship) is called with the current speed ("currSpeed") as the parameter. This likely moves the ship along the Z-axis (forward/backward).

- If "movingL" is "true", indicating leftward movement, the "MovePlayerLeftRight" method of the "playerReference" object is called with a modified speed ("currSpeed * -shiftProp"). This likely moves the ship to the left.

- If "movingR" is "true", indicating rightward movement, the "MovePlayerLeftRight" method of the "playerReference" object is called with a modified speed ("currSpeed * shiftProp"). This likely moves the ship to the right.

- Additionally, if the current speed is less than the maximum speed ("maxSpeed"), it is gradually increased by 0.2 units.

2. "stopShip" Method:

- This method is responsible for gradually slowing down the player's ship when no movement keys are pressed.

- Similar to "moveShip", there are conditional checks for each movement direction to determine the action to take.

- If the current speed is greater than the minimum speed ("minSpeed"), it is gradually decreased by 0.5 units.

- Regardless of whether the ship is moving or not, the ship's forward/backward movement is adjusted using the "MovePlayerZ" method with the current speed.

- If the ship is also moving left or right ("movingL" or "movingR"), the ship's lateral movement is adjusted using the "MovePlayerLeftRight" method.

- Finally, if the current speed reaches the minimum speed, it is set to exactly match the minimum speed to avoid any discrepancies.

6.1.9 Handlekey

```
    private void handleKeyPress(KeyCode code) {
        switch (code) {
            case W:
                this.movingZ = true;
                break;
            case A:
                this.movingL = true;
                break;
            case D:
                this.movingR = true;
                break;
            case ESCAPE:
                pause.togglePause();
            default:
                break;
        }
    }

    private void handleKeyRelease(KeyCode code) {
        switch (code) {
        case W:
            this.movingZ = false;
            break;
        case A:
            this.movingL = false;
            break;
        case D:
            this.movingR = false;
            break;
        default:
            break;
    }
    }

}
```

1. Method Signature:

- "private void handleKeyRelease(KeyCode code)": This method takes a "KeyCode" parameter, which represents the key that was released.

2. Switch Statement:

- The method uses a "switch" statement to determine the action to take based on the released key.

- It evaluates the value of the "code" parameter to determine which key was released.

3. Key Release Handling:

- For each case in the "switch" statement, the method updates the corresponding movement flag to indicate that the associated movement action should cease.

- If the 'W' key is released ("case W"), it sets "movingZ" to "false", indicating that the ship should stop moving along the Z-axis (forward/backward).

- If the 'A' key is released ("case A"), it sets "movingL" to "false", indicating that the ship should stop moving left.

- If the 'D' key is released ("case D"), it sets "movingR" to "false", indicating that the ship should stop moving right.

- If none of the above cases match (default), the method takes no action.

4. Movement Flags:

- These movement flags ("movingZ", "movingL", "movingR") are boolean variables that control the ship's movement in different directions.

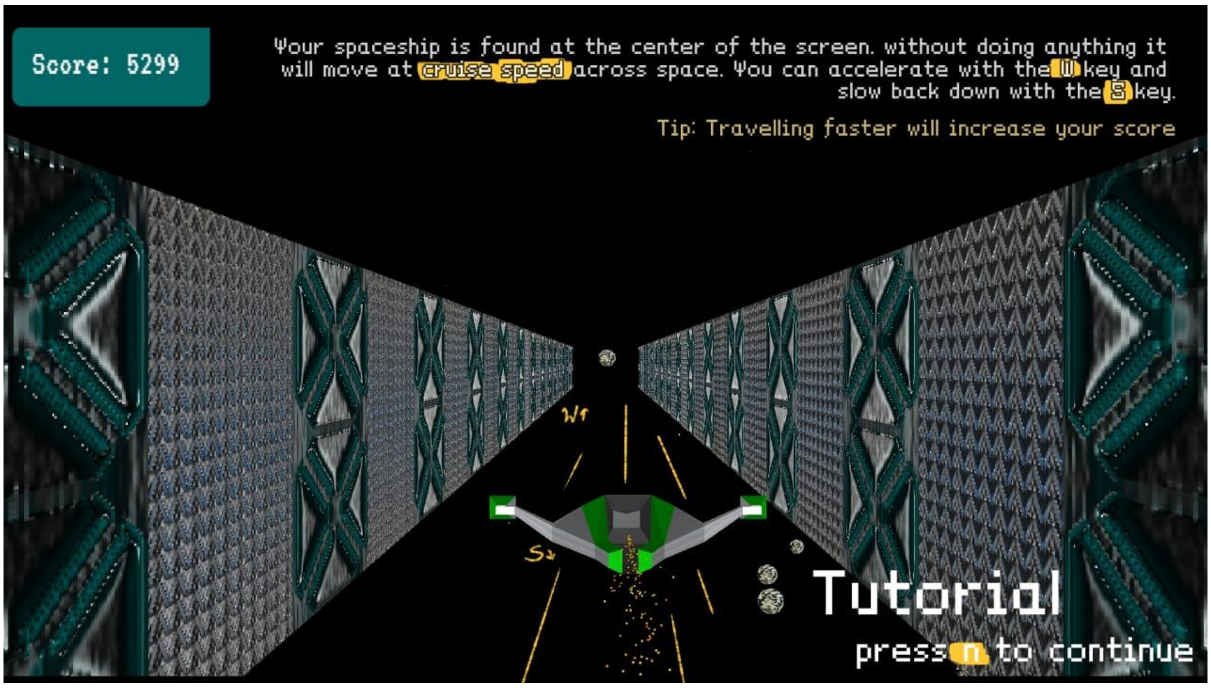
- When a key is released, the corresponding movement flag is set to "false", indicating that the ship should stop moving in that direction.

5. Default Case:

- The "default" case in the "switch" statement is a fallback for handling keys other than 'W', 'A', or 'D'.

- If the released key is not one of these, the method takes no action.

6.2 Demo

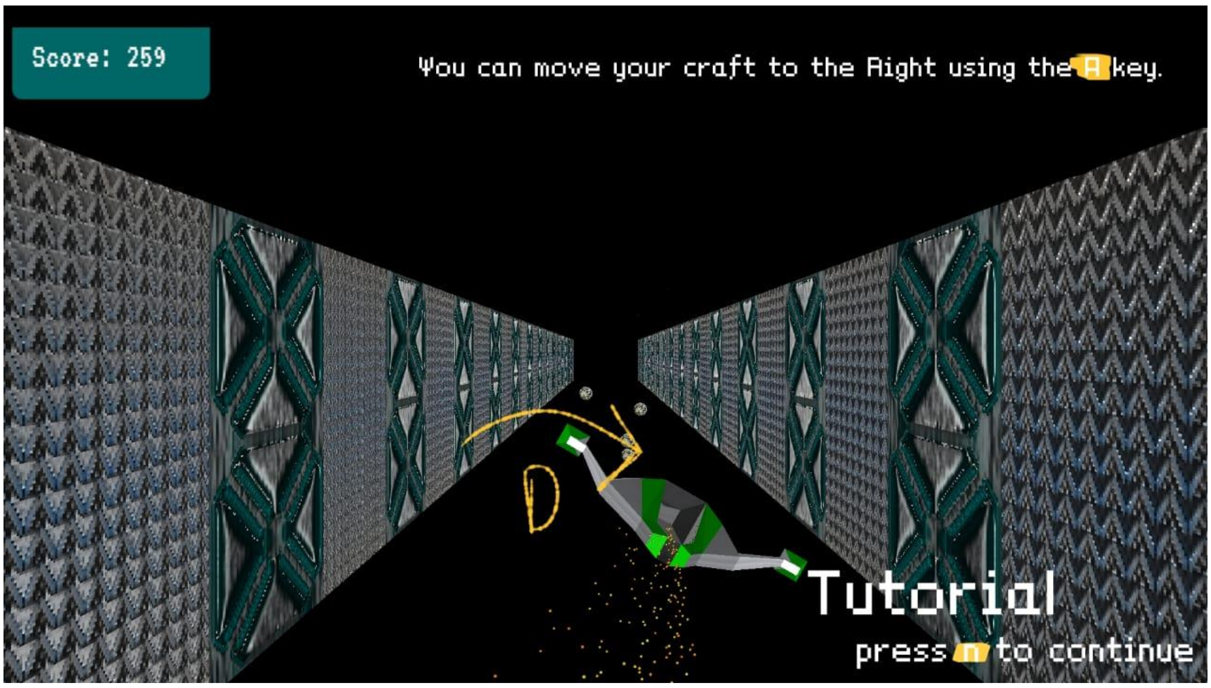

Score: 5299

Your spaceship is found at the center of the screen. without doing anything it will move at cruise speed across space. You can accelerate with the W key and slow back down with the S key.

Tip: Travelling faster will increase your score

Tutorial

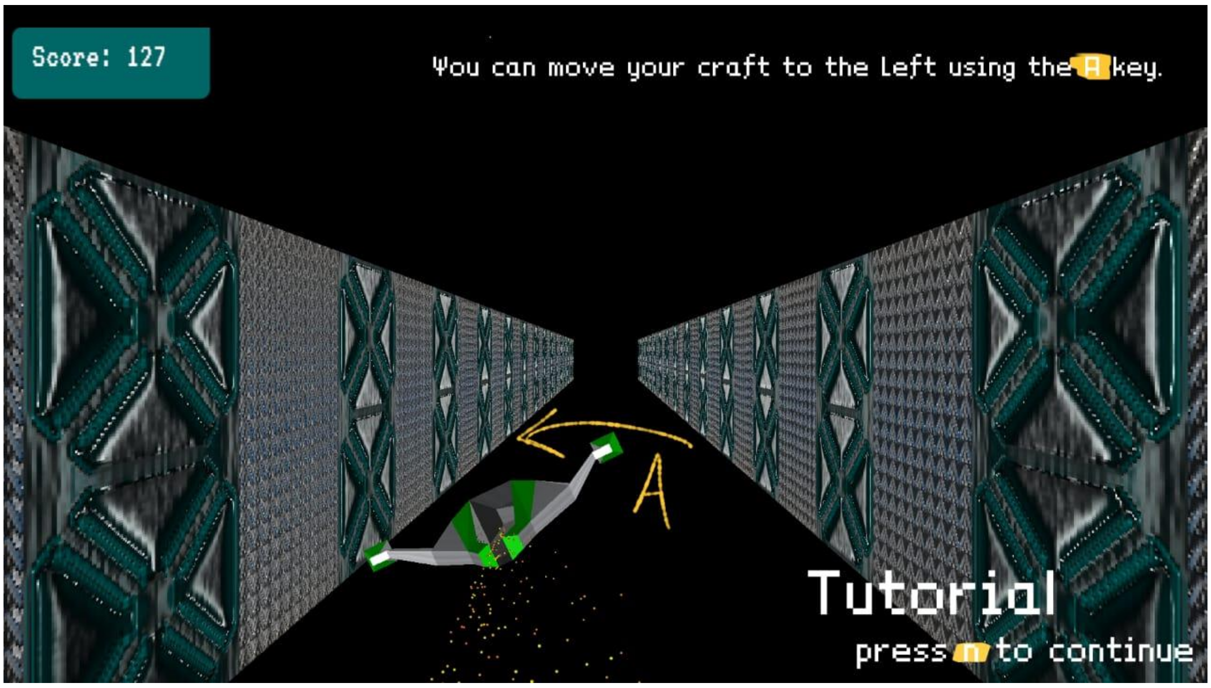
press n to continue


Score: 259

You can move your craft to the Right using the A key.

Tutorial

press n to continue

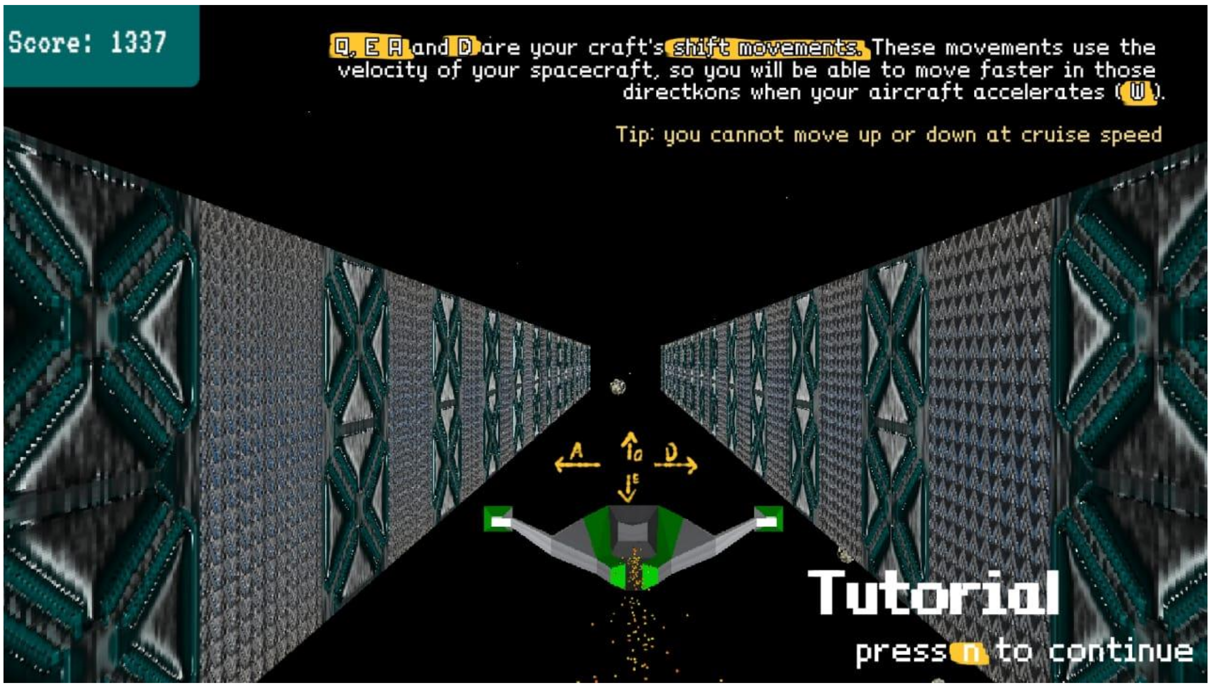


Score: 127

You can move your craft to the Left using the A key.

Tutorial

press n to continue



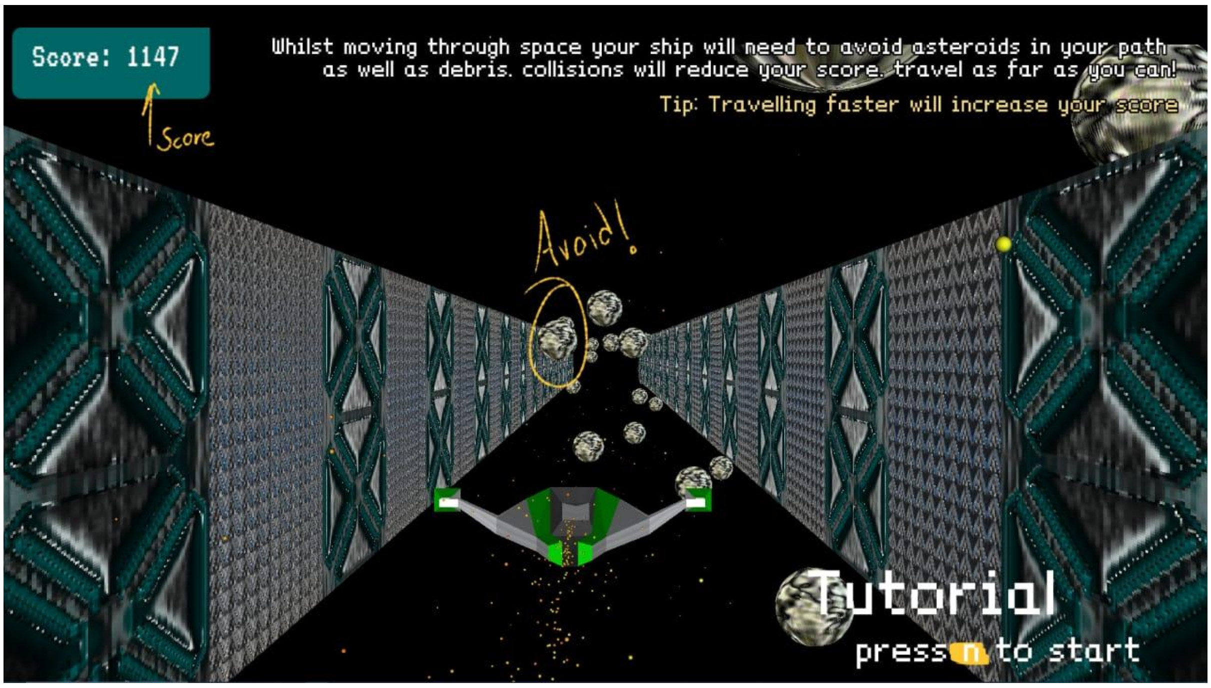
Score: 1337

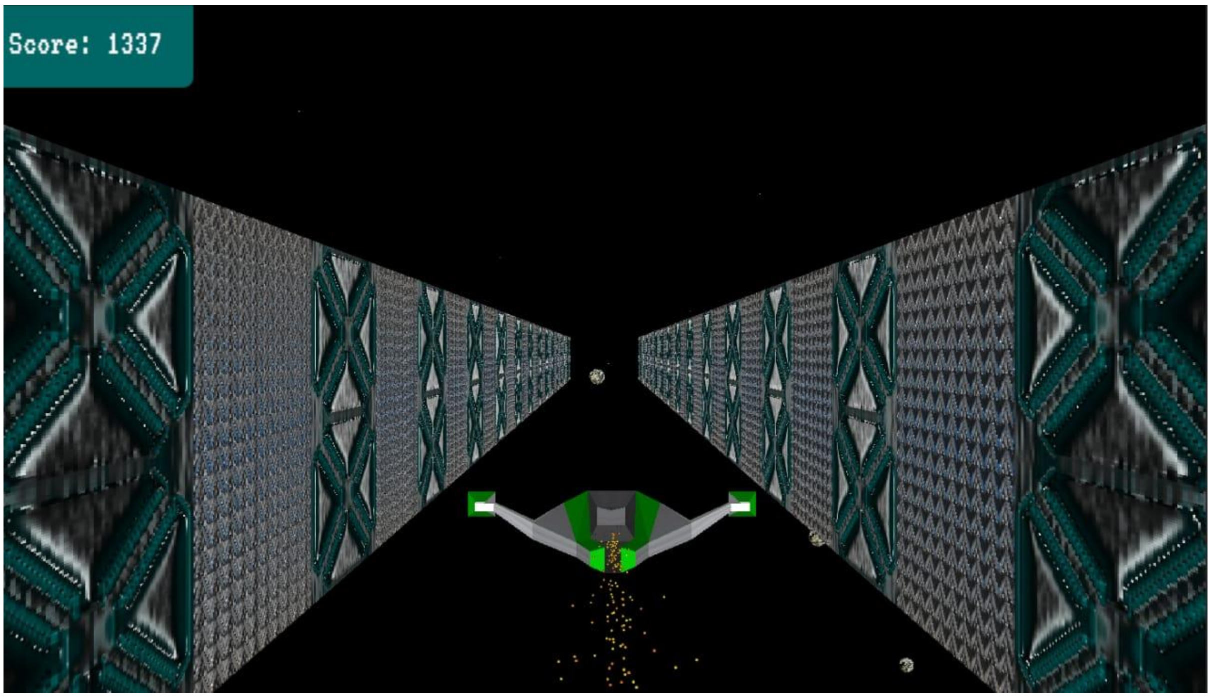
Q, E A and D are your craft's shift movements. These movements use the velocity of your spacecraft, so you will be able to move faster in those directkons when your aircraft accelerates (W).

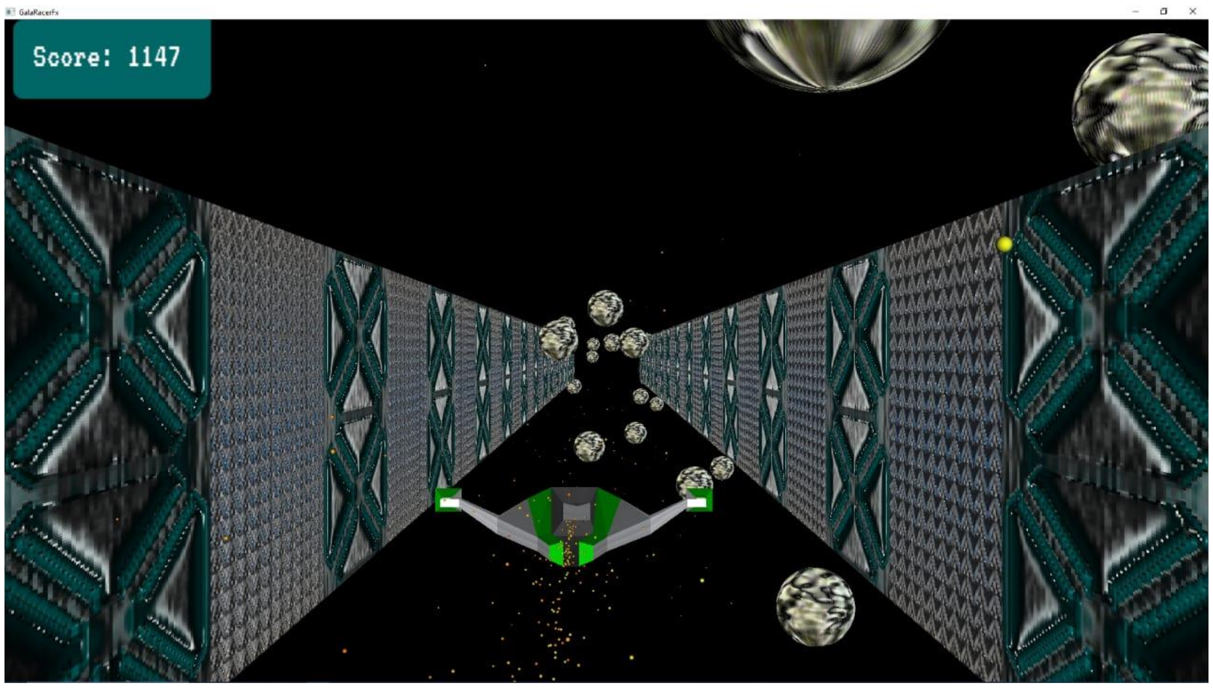
Tip: you cannot move up or down at cruise speed

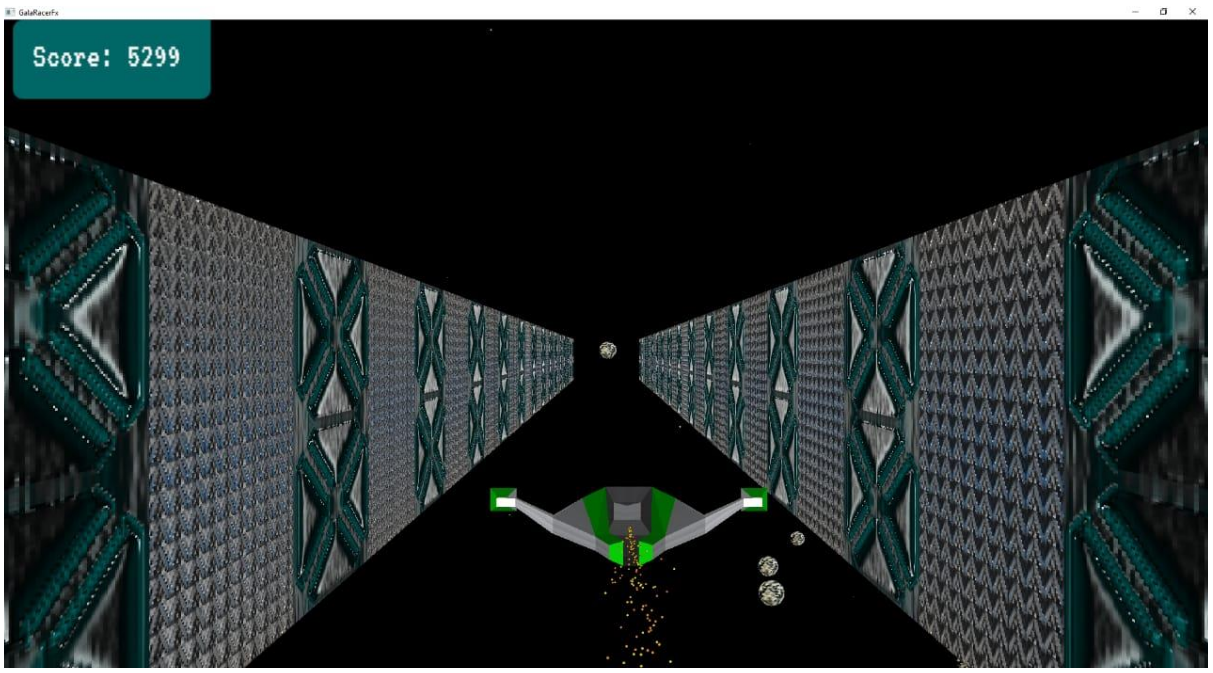
Tutorial

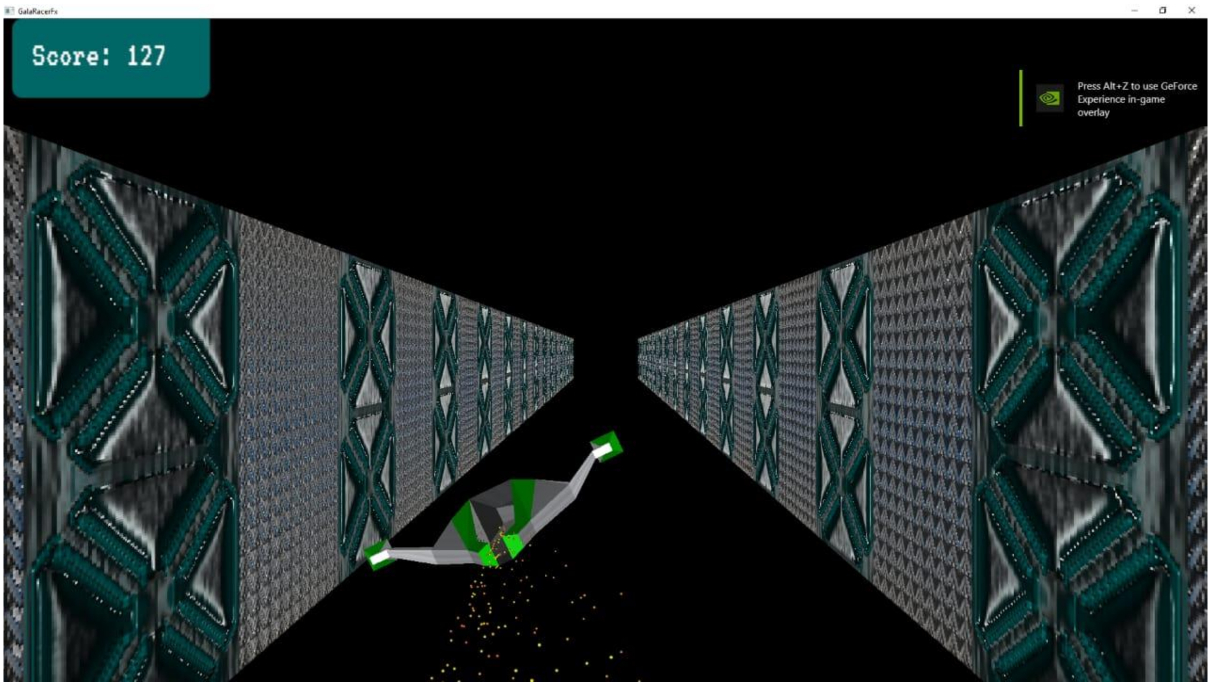
press n to continue

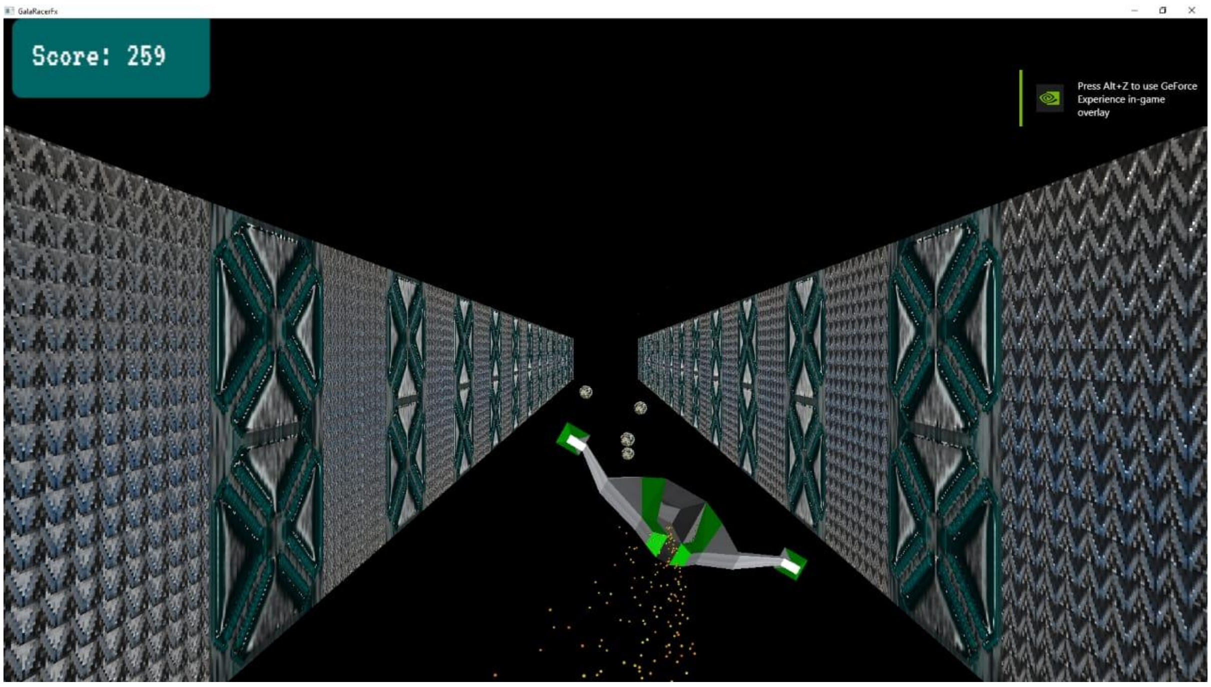

Score: 1147
Score
Whilst moving through space your ship will need to avoid asteroids in your path as well as debris. collisions will reduce your score. travel as far as you can!
Tip: Travelling faster will increase your score
Avoid!
Tutorial
press n to start


Score: 1337


GalaRacerFx
Score: 1147


GalaRacerFx
Score: 5299


GalaRacerFx
Score: 127
Press Alt+Z to use GeForce Experience in-game overlay


GalaRacerFx
Score: 259
Press Alt+Z to use GeForce Experience in-game overlay

6.3 Work attribution

Names	Task Allocation
1. **Jameel Edoo**	• UI Logic • Refactoring • Lighting • GameOver Screen • Report
2. **Kritika Bissessur**	• Static Interface • Pause Screen • Title Screen • Camera • Powerpoint • Report
3. **Yassir Hoossan Buksh**	• Entities • State • Main • Game • Powerpoint • Report