```java
// src/main/java/org/thunderfighter/core/abstractor/AbstractAircraft.java

package org.thunderfighter.core.abstractor;

import java.util.List;
import javafx.geometry.BoundingBox;
import javafx.geometry.Bounds;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import org.thunderfighter.core.entity.Aircraft;

/**
 * Abstract Aircraft Class
 *
 * <p>Common parent class for all aircraft entities (player aircraft, enemy aircraft, etc.),
 * Provides basic functionalities such as health points, drawing, movement updates, and collision
 * boundaries.
 *
 * <p>When inheriting from this class, the `{@link #move()}` method must be implemented, and `{@link
 * #onUpdate(List)}` and `{@link #onDie()}` can be overridden as needed.
 */
public abstract class AbstractAircraft extends AbstractEntity implements Aircraft {

  /** current health */
  protected int hp;

  /** Airplane texture */
  protected Image sprite; // aircraft texture

  /**
   * Update the aircraft's collision boundaries
   *
   * <p>Based on the current coordinates (x, y) and size, Create a new {@link BoundingBox} as the
   * collision area.
   */
  protected void updateCollisionBounds() {
    if (size != null) {
      collisionBounds = new BoundingBox(x, y, size.getWidth(), size.getHeight());
    }
  }

  /**
   * Get current health
   *
   * @return current HP
   */
  @Override
  public int getHp() {
    return hp;
  }

  /**
   * Takes damage
   *
   * <p>Reduces health. When health is less than or equal to 0:
   *
   * <ul>
   *   <li>Sets the survival flag to false
   *   <li>Triggers the death callback {@link #onDie()}
   * </ul>
   *
   * * * @param damage The amount of damage taken
   */
  @Override
  public void takeDamage(int damage) {
    hp -= damage;
    if (hp <= 0) {
      aliveFlag = false;
      onDie();
    }
  }

  /**
   * Get the aircraft's collision boundaries
   *
   * @return the current {@link Bounds}
   */
  @Override
  public Bounds getCollisionBounds() {
    return collisionBounds;
  }

  /**
   * Update logic for each frame (template method)
   *
   * <p>This method is declared as final to ensure consistent update process:
   *
   * <ol>
   *   <li>Check if alive
   *   <li>Execute movement logic {@link #move()}
   *   <li>Update collision boundaries
   *   <li>Execute subclass extension logic {@link #onUpdate(List)}
   * </ol>
   *
   * @param worldEntities All entities in the current world
   */
  @Override
  public final void update(List<AbstractEntity> worldEntities) {
    if (!aliveFlag) return;
    move();
    updateCollisionBounds();
    onUpdate(worldEntities);
  }

  /**
   * Draw the aircraft
   *
   * <p>Draw only while the aircraft is alive, Render using sprite at current coordinates and size.
   *
   * @param gc JavaFX drawing context
   */
  @Override
  public void draw(GraphicsContext gc) {
    if (!aliveFlag) return;
    gc.drawImage(sprite, x, y, size.getWidth(), size.getHeight());
  }

  /**
   * Callback method when the plane dies
   *
   * <p>Subclasses can override this method to implement explosion effects, dropped items, scoring
   * logic, etc.
   */
  protected void onDie() {} // aircraft death callback

  /**
   * Aircraft movement logic (must be implemented by a subclass)
   *
   * <p>For example:
   *
   * <ul>
   *   <li>Player aircraft: Moves according to keyboard input
   *   <li>Enemy aircraft: Moves according to AI or a preset path
   * </ul>
   */
  protected abstract void move(); // aircraft movement logic

  /**
   * Additional update logic per frame
   *
   * <p>Subclasses can implement this here:
   *
   * <ul>
   *   <li>Shooting
   *   <li>AI Behavior
   *   <li>Skill Cooldown
   * </ul>
   *
   * @param worldEntities All entities in the current world
   */
  protected void onUpdate(
      List<AbstractEntity> worldEntities) {} // extra logic per frame, like shooting, AI, etc.
}
```

```java
// src/main/java/org/thunderfighter/core/abstractor/AbstractBullet.java

package org.thunderfighter.core.abstractor;

import java.util.List;
import javafx.geometry.BoundingBox;
import javafx.geometry.Bounds;
import org.thunderfighter.core.entity.Aircraft;
import org.thunderfighter.core.entity.Bullet;
import org.thunderfighter.core.entity.Trajectory;

/**
 * AbstractBullet
 *
 * <p>Base class for all bullets (including items). Canvas is inherited from AbstractEntity.
 */
public abstract class AbstractBullet extends AbstractEntity implements Bullet {

  protected double dx;
  protected double dy;

  protected boolean fromPlayer;
  protected Trajectory trajectory;

  /** Lifetime in ticks; -1 means infinite until out-of-bounds. */
  protected int lifeTicks = -1;

  /** Spawn origin (semantic info for trajectories). */
  protected double originX;

  protected double originY;

  /** Canvas bounds (must be set by bullet constructor). */
  protected double canvasW;

  protected double canvasH;

  @Override
  public Bounds getCollisionBounds() {
    // size must be initialized in each concrete bullet constructor
    return new BoundingBox(x, y, size.getWidth(), size.getHeight());
  }

  @Override
  public final boolean isFromPlayer() {
    return fromPlayer;
  }

  // ---------------------------
  // Trajectory-required accessors
  // ---------------------------

  /** Position X (delegated to AbstractEntity state). */
  public double getX() {
    return x;
  }

  /** Position Y (delegated to AbstractEntity state). */
  public double getY() {
    return y;
  }

  /** Set position X. */
  public void setX(double x) {
    this.x = x;
  }

  /** Set position Y. */
  public void setY(double y) {
    this.y = y;
  }

  /** Velocity X per tick. */
  public double getDx() {
    return dx;
  }

  /** Velocity Y per tick. */
  public double getDy() {
    return dy;
  }

  /** Set velocity X per tick. */
  public void setDx(double dx) {
    this.dx = dx;
  }

  /** Set velocity Y per tick. */
  public void setDy(double dy) {
    this.dy = dy;
  }

  /** Allow runtime trajectory swapping (optional but useful). */
  public void setTrajectory(Trajectory trajectory) {
    this.trajectory = trajectory;
  }

  public final void setOrigin(double ox, double oy) {
    this.originX = ox;
    this.originY = oy;
  }

  public final void setLifeTicks(int ticks) {
    this.lifeTicks = ticks;
  }

  /** Decrements lifetime if enabled. When lifeTicks reaches 0, bullet dies. */
  protected final void tickLife() {
    if (lifeTicks > 0) {
      lifeTicks--;
      if (lifeTicks <= 0) aliveFlag = false;
    }
  }

  /** Kill bullet when leaving canvas. Canvas comes from AbstractEntity. */
  protected final void killIfOutOfBounds() {
    if (canvas == null) return;

    double w = canvas.getWidth();
    double h = canvas.getHeight();

    if (x + size.getWidth() < 0 || x > w || y + size.getHeight() < 0 || y > h) {
      aliveFlag = false;
    }
  }

  /** One movement step by the current trajectory (if any). */
  protected final void moveOnce() {
    if (trajectory != null) {
      trajectory.update(this);
    }
  }

  public double getOriginY() {
    return originY;
  }

  @Override
  public abstract void update(List<AbstractEntity> worldEntities);

  @Override
  public abstract void onHit(Aircraft target);
}
```

```java
// src/main/java/org/thunderfighter/core/abstractor/AbstractEnemyAircraft.java

package org.thunderfighter.core.abstractor;

import java.util.List;
import org.thunderfighter.core.manager.ScoreManager;

/**
 * Abstract Enemy Aircraft Class
 *
 * <p>Inherited from {@link AbstractAircraft}, serving as the common parent class for all enemy
 * aircraft. Provides common shooting controls, cooldown logic, and kill score calculation for all
 * enemy aircraft. *
 *
 * <p>Subclasses only need to focus on the specific movement methods and shooting implementation
 * ({@link #doShoot(List)}).
 */
public abstract class AbstractEnemyAircraft extends AbstractAircraft {

  /** can shoot or not */
  protected boolean canShoot;

  /** firing interval per frame */
  protected int shootInterval;

  /** shooting cooldown */
  protected int shootCooldown;

  /** destruction score */
  protected int score;

  /**
   * Indicates whether the aircraft is a player aircraft
   *
   * @return always returns false, indicating an enemy unit
   */
  @Override
  public boolean isPlayer() {
    return false;
  }

  /**
   * Attempt to fire
   *
   * <p>Firing logic will only be executed if the enemy aircraft has the ability to fire and the
   * cooldown has ended.
   *
   * @param worldEntities All entities in the current world
   */
  @Override
  public void shoot(List<AbstractEntity> worldEntities) {
    if (!canShoot) return;
    if (shootCooldown <= 0) {
      doShoot(worldEntities);
      shootCooldown = shootInterval;
    }
  }

  /**
   * Check whether the enemy moving out of the map.
   *
   * <p>If it is out of the map, it is not alive.
   *
   * <p>Since the enemy is only generated within the width of {@code canvas}, so the {@code x} is
   * always inside the range.
   *
   * <p>However, the enemy is generated above the {@code canvas}, so the upper boundary can be
   * skipped to check.
   *
   * <p>In this case, this function only checks whether the {@code y} is greater than the height of
   * {@code canvas}.
   */
  private final void checkBoundary() {
    if (this.y >= canvas.getHeight()) {
      this.aliveFlag = false;
    }
  }

  /**
   * Per-frame update logic (enemy aircraft only)
   *
   * <p>Responsible for:
   *
   * <ul>
   *   <li>Decrease firing cooldown
   *   <li>Trigger firing attempt
   * </ul>
   *
   * @param worldEntities All entities in the current world
   */
  @Override
  protected void onUpdate(List<AbstractEntity> worldEntities) {
    if (shootCooldown > 0) shootCooldown--;
    shoot(worldEntities);
    checkBoundary();
  }

  /**
   * Callback for enemy aircraft death
   *
   * <p>When an enemy aircraft is destroyed, add the corresponding score to {@link ScoreManager}.
   */
  @Override
  protected void onDie() {
    ScoreManager.getInstance().addScore(this.score);
  }

  /**
   * Actual firing logic
   *
   * <p>Implemented by a specific enemy subclass, used to generate bullets, Set ballistics or
   * special attack behaviors.
   *
   * @param worldEntities All entities in the current world
   */
  protected abstract void doShoot(List<AbstractEntity> worldEntities);
}
```

```java
// src/main/java/org/thunderfighter/core/abstractor/AbstractEntity.java

package org.thunderfighter.core.abstractor;

import java.util.List;
import javafx.geometry.Bounds;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import org.thunderfighter.core.entity.Entity;

/**
 * Abstract base class for entities
 *
 * <p>The common parent class for all interactive objects in the game, such as airplanes, bullets,
 * and items. This class defines the basic attributes of entities, including position, speed, size,
 * survival status, and collision boundaries.
 *
 * <p>The specific update logic and drawing methods are implemented by subclasses.
 */
public abstract class AbstractEntity implements Entity {

  /** The x-coordinate of the entity on the canvas */
  protected double x;

  /** The y-coordinate of the entity on the canvas */
  protected double y;

  /** Entity's movement speed */
  protected double speed;

  /** Whether the entity is alive */
  protected boolean aliveFlag = true;

  /** Entity size (usually corresponds to texture size) */
  protected Dimension2D size;

  /** Current collision boundaries of the entity */
  protected Bounds collisionBounds;

  /** The canvas containing the entity */
  protected Canvas canvas;

  /**
   * Sets the canvas containing the entity
   *
   * @param canvas JavaFX Canvas
   */
  public void setCanvas(Canvas canvas) {
    this.canvas = canvas;
  }

  /**
   * Get the canvas containing the entity
   *
   * @return Current Canvas
   */
  public Canvas getCanvas() {
    return canvas;
  }

  /**
   * Get the collision boundaries of the entity
   *
   * @return current {@link Bounds}
   */
  @Override
  public Bounds getCollisionBounds() {
    return collisionBounds;
  }

  /**
   * Get entity dimensions
   *
   * @return entity dimensions {@link Dimension2D}
   */
  @Override
  public Dimension2D getSize() {
    return size;
  }

  /**
   * Get the entity's movement speed
   *
   * @return speed
   */
  @Override
  public double getSpeed() {
    return speed;
  }

  /**
   * Determine if the entity is still alive
   *
   * @return true if the entity is alive, false if the entity has been destroyed
   */
  @Override
  public boolean isAlive() {
    return aliveFlag;
  }

  /**
   * Get the current x-coordinate of the entity
   *
   * @return x-coordinate
   */
  public double getX() {
    return this.x;
  }

  /**
   * Get the current y-coordinate of the entity
   *
   * @return y-coordinate
   */
  public double getY() {
    return this.y;
  }

  /**
   * Update logic for each frame
   *
   * <p>Specific behaviors are implemented by subclasses, for example:
   *
   * <ul>
   *   <li>Position update
   *   <li>State change
   *   <li>Collision detection
   * </ul>
   *
   * @param worldEntities All entities in the current world
   */
  @Override
  public abstract void update(List<AbstractEntity> worldEntities);

  /**
   * Draw the entity
   *
   * <p>The specific drawing method is implemented by subclasses.
   *
   * @param gc JavaFX drawing context
   */
  @Override
  public abstract void draw(GraphicsContext gc);
}
```

```java
// src/main/java/org/thunderfighter/core/abstractor/AbstractPlayerAircraft.java

package org.thunderfighter.core.abstractor;

import java.util.List;
import javafx.scene.canvas.Canvas;
import org.thunderfighter.game.Game;

/**
 * Abstract player aircraft class
 *
 * <p>Inherited from {@link AbstractAircraft}, used for player-controlled aircraft. Provides player
 * input state management, shooting cooldown logic, and shooting behavior templates.
 *
 * <p>Specific shooting methods are implemented by subclasses using {@link #doShoot(List)}.
 */
public abstract class AbstractPlayerAircraft extends AbstractAircraft {

  /** Firing interval (in frames) */
  protected int shootInterval;

  /** Current firing cooldown time */
  protected int shootCooldown;

  /** Whether the player wants to shoot is controlled by input */
  protected boolean wantToShoot = false;

  /** Player direction input status */
  protected boolean up, down, left, right;

  /** The canvas where the player's plane is located (used for rendering or input detection) */
  protected Canvas canvas;

  /** The current game instance, which can be used to obtain game status or manage entities */
  protected Game game;

  /**
   * Sets the upward movement state
   *
   * @param up true indicates upward movement
   */
  public void setUp(boolean up) {
    this.up = up;
  }

  /**
   * Sets the downward movement state
   *
   * @param down true indicates downward movement
   */
  public void setDown(boolean down) {
    this.down = down;
  }

  /**
   * Sets the leftward movement state
   *
   * @param left true indicates leftward movement
   */
  public void setLeft(boolean left) {
    this.left = left;
  }

  /**
   * Sets the rightward movement state
   *
   * @param right true indicates rightward movement
   */
  public void setRight(boolean right) {
    this.right = right;
  }

  /**
   * Sets the player's shooting intention
   *
   * @param shooting true indicates that the player wants to shoot
   */
  public void setShooting(boolean shooting) {
    this.wantToShoot = shooting;
  }

  /**
   * Get whether the player wants to fire
   *
   * @return true indicates that the player currently requests to fire
   */
  public boolean wantToShoot() {
    return wantToShoot;
  } // shooting info

  /**
   * Determine if the aircraft is controlled by a player
   *
   * @return Always returns true
   */
  @Override
  public boolean isPlayer() {
    return true;
  }

  /**
   * Shooting logic
   *
   * <p>When the shooting cooldown ends, trigger the actual shot and reset the cooldown.
   *
   * @param worldEntities All entities in the current world
   */
  @Override
  public void shoot(List<AbstractEntity> worldEntities) {
    if (shootCooldown <= 0) {
      doShoot(worldEntities);
      shootCooldown = shootInterval;
    }
  }

  /**
   * Per-frame update logic
   *
   * <p>Responsible for basic update logic of the player's aircraft, such as firing cooldown
   * reduction. Specific movement logic is implemented by {@link #move()}.
   *
   * @param worldEntities All entities in the current world
   */
  @Override
  protected void onUpdate(List<AbstractEntity> worldEntities) {
    if (shootCooldown > 0) {
      shootCooldown--;
    }
  }

  /**
   * Actual shooting implementation
   *
   * <p>Implemented by a specific player aircraft subclass, used to generate bullets or perform
   * special attacks.
   *
   * @param worldEntities All entities in the current world
   */
  protected abstract void doShoot(List<AbstractEntity> worldEntities); // implemented by subclass
}
```

```java
// src/main/java/org/thunderfighter/core/abstractor/AbstractUiDialog.java

package org.thunderfighter.core.abstractor;

public abstract class AbstractUiDialog {
  public abstract void showDialog();
}
```

```java
// src/main/java/org/thunderfighter/core/abstractor/AbstractUiMenu.java

package org.thunderfighter.core.abstractor;

import javafx.scene.layout.VBox;

public abstract class AbstractUiMenu extends VBox {
  public abstract void showMenu();

  public abstract void hideMenu();
}
```

```java
// src/main/java/org/thunderfighter/core/collision/CollisionDetector.java

package org.thunderfighter.core.collision;

import java.util.List;
import javafx.geometry.Bounds;
import org.thunderfighter.core.abstractor.AbstractAircraft;
import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.abstractor.AbstractEnemyAircraft;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.core.entity.Bullet;
import org.thunderfighter.game.aircraft.enemy.BossEnemy;
import org.thunderfighter.game.aircraft.player.PlayerAircraft;

/** Collision Detector */
public class CollisionDetector {

  /**
   * Detect collisions among a list of {@code AbstractEntity}.
   *
   * @param entities List of entities.
   */
  public static void detectCollision(final List<AbstractEntity> entities) {
    final int n = entities.size();
    for (int i = 0; i < n; i++) {
      final AbstractEntity a = entities.get(i);
      if (!a.isAlive()) {
        continue;
      }

      for (int j = i + 1; j < n; j++) {
        final AbstractEntity b = entities.get(j);
        if (!b.isAlive()) {
          continue;
        }
        if (isCollision(a, b)) {
          handleCollision(a, b);
        }
      }
    }
  }

  /**
   * Determine whether collision happens between two entities.
   *
   * <p>Specially, collisions won't happen between bullets or between enemies.
   *
   * @param a An entity
   * @param b Another entity
   * @return Return `true` if collision happens. Otherwise, it will return `false`.
   */
  private static boolean isCollision(final AbstractEntity a, final AbstractEntity b) {
    // Collision won't happen between bullets and enemies.
    if ((a instanceof Bullet && b instanceof Bullet)
        || (a instanceof AbstractEnemyAircraft && b instanceof AbstractEnemyAircraft)) {
      return false;
    }

    // Avoid `null` value raising exceptions.
    final Bounds b1 = a.getCollisionBounds();
    final Bounds b2 = b.getCollisionBounds();
    if (b1 != null && b2 != null) {
      return b1.intersects(b2);
    }

    return false;
  }

  /**
   * Actions to be taken when collision happens.
   *
   * <p>This method should be used after `isCollision()`.
   *
   * <p>As `isCollision()` is called before this method, situations to be handled include collisions
   * between player and enemy, and between aircraft and bullet.
   *
   * @param a An entity.
   * @param b Another entity.
   */
  private static void handleCollision(final AbstractEntity a, final AbstractEntity b) {
    if (a instanceof AbstractAircraft && b instanceof AbstractAircraft) {
      handleAircraftCollision((AbstractAircraft) a, (AbstractAircraft) b);
    } else {
      handleAircraftBulletCollision(a, b);
    }
  }

  /**
   * Handle collisions between aircrafts.
   *
   * <p>If both aircrafts are not Boss, then, both of them will get one point damage.
   *
   * <p>Otherwise, there will one player and one Boss, where the user will be killed immediately.
   *
   * @param a An {@link AbstractAircraft}.
   * @param b Another {@link AbstractAircraft}.
   */
  private static void handleAircraftCollision(final AbstractAircraft a, final AbstractAircraft b) {
    if (!(a instanceof BossEnemy || b instanceof BossEnemy)) {
      a.takeDamage(1);
      b.takeDamage(1);
      return;
    }

    // One of the aircraft is boss.
    if (a instanceof PlayerAircraft) {
      a.takeDamage(a.getHp());
    } else {
      b.takeDamage(b.getHp());
    }
  }

  /**
   * Handle collisions between {@link AbstractAircraft} and {@link AbstractBullet}.
   *
   * <p>This method automatically determines which object passed is {@code Aircraft} or {@code
   * Bullet}. And it prevents both player and enemy from suicide.
   *
   * @param a An {@link AbstractEntity} which can be either {@link AbstractAircraft} or {@link
   *     AbstractBullet}.
   * @param b Another {@link AbstractEntity} which can be either {@link AbstractAircraft} or {@link
   *     AbstractBullet}.
   */
  private static void handleAircraftBulletCollision(
      final AbstractEntity a, final AbstractEntity b) {
    AbstractAircraft aircraft;
    AbstractBullet bullet;
    if (a instanceof AbstractAircraft) {
      aircraft = (AbstractAircraft) a;
      bullet = (AbstractBullet) b;
    } else {
      aircraft = (AbstractAircraft) b;
      bullet = (AbstractBullet) a;
    }

    // Prevent player and enemy from suicide.
    if (aircraft.isPlayer() == bullet.isFromPlayer()) {
      return;
    }
    bullet.onHit(aircraft);
  }
}
```

```java
// src/main/java/org/thunderfighter/core/entity/Aircraft.java

package org.thunderfighter.core.entity;

import java.util.List;
import org.thunderfighter.core.abstractor.AbstractEntity;

public interface Aircraft extends Entity {
  int getHp(); // current health

  void takeDamage(int damage);

  boolean isPlayer(); // is player aircraft or not

  void shoot(List<AbstractEntity> worldEntities);

  // fire bullets, but only declares the "behavior", does not create the bullets
  // all bullet are managed by worldEntity list
}
```

```java
// src/main/java/org/thunderfighter/core/entity/BulletEmitter.java

package org.thunderfighter.core.entity;

public interface BulletEmitter {
  void emit(Bullet bullet);
}
// A simple callback interface to decouple Aircraft.shoot() from the world bullet list.
// Aircraft can call emitter.emit(...) without directly accessing collections in the game world.
```

```java
// src/main/java/org/thunderfighter/core/entity/Bullet.java

package org.thunderfighter.core.entity;

public interface Bullet extends Entity {

  void onHit(Aircraft target); // Called when this bullet collides with an aircraft.

  boolean isFromPlayer(); // Indicates whether this bullet is fired by the player.
}
```

```java
// src/main/java/org/thunderfighter/core/entity/Entity.java

package org.thunderfighter.core.entity;

import java.util.List;
import javafx.geometry.Bounds;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.GraphicsContext;
import org.thunderfighter.core.abstractor.AbstractEntity;

public interface Entity {
  void update(List<AbstractEntity> worldEntities); // update object status

  void draw(GraphicsContext gc); // used to draw objects

  Dimension2D getSize(); // get the object texture size, for drawing and collision

  double getSpeed(); // the object's base movement speed

  boolean isAlive(); // is alive or not, it will be removed when it is false

  Bounds getCollisionBounds();
}
```

```java
// src/main/java/org/thunderfighter/core/entity/Trajectory.java

package org.thunderfighter.core.entity;

import org.thunderfighter.core.abstractor.AbstractBullet;

public interface Trajectory {
  void update(AbstractBullet bullet);
}
```

```java
// src/main/java/org/thunderfighter/core/manager/ScoreManager.java

package org.thunderfighter.core.manager;

public class ScoreManager {
  private static ScoreManager instance = new ScoreManager();
  private static int score = 0;

  private ScoreManager() {}

  public static ScoreManager getInstance() { // call method
    return instance;
  }

  public int getScore() { // get from private
    return score;
  }

  public void addScore(int s) { // accumulator
    score += s;
  }

  public void reset() { // clear the score
    score = 0;
  }
}
```

// src/main/java/org/thunderfighter/game/aircraft/enemy/BossEnemy.java

```java
package org.thunderfighter.game.aircraft.enemy;

import java.util.List;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.Canvas;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractEnemyAircraft;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.game.Game;
import org.thunderfighter.game.bulletfactory.BulletFactory;
import org.thunderfighter.game.trajectory.HomingTrajectory;
import org.thunderfighter.utils.Constant;

/**
 * Boss Enemy Boss-level enemy Has multi-stage behavior; its attack patterns and movement speed
 * change based on its health.
 */
public class BossEnemy extends AbstractEnemyAircraft {

  /** Boss dimensions (width 200, height 150) */
  public static final Dimension2D SIZE = new Dimension2D(200, 150);

  /** Move to the right now or not */
  private boolean movingRight = true;

  /** Screen width, used to limit the Boss's horizontal movement range */
  private static final double SCREEN_WIDTH = 800;

  /** The previous stage, used to detect if the stage has changed */
  private Stage lastStage = null;

  /** The main game object, used to obtain the player's plane and modify the game state */
  private Game game;

  /**
   * Boss's three phases Stage 1: High health Stage 2: Medium health Stage 3: Low health (Enrage)
   */
  private enum Stage {
    stage1,
    stage2,
    stage3
  }

  /** Current stage of the Boss */
  private Stage stage = Stage.stage1;

  /**
   * Constructor
   *
   * @param x Initial X coordinate
   * @param y Initial Y coordinate
   * @param game Game object
   */
  public BossEnemy(double x, double y, Game game) {
    this.x = x;
    this.y = y; // birth coordinates

    this.hp = 30;
    this.speed = 1;
    this.score = 3000; // kill reward

    this.canShoot = true;
    this.shootInterval = 180; // the number of frames are needed between two shots
    this.shootCooldown =
        shootInterval; // the number of frames are left before the next shot can be fired

    this.size = SIZE;

    this.sprite = new Image(getClass().getResourceAsStream("/images/Aircraft/BossEnemy.png"));
    this.game = game;
  }

  /** Boss's movement logic Moves back and forth between the center of the screen */
  @Override
  protected void move() {
    double LEFT_BOUND = SCREEN_WIDTH * 0.25;
    double RIGHT_BOUND = SCREEN_WIDTH * 0.75 - size.getWidth();

    if (movingRight) {
      x += speed;
      if (x >= RIGHT_BOUND) movingRight = false;
    } else {
      x -= speed;
      if (x <= LEFT_BOUND) movingRight = true;
    }
  }

  /** Triggered when the Boss dies Game state switches to victory */
  @Override
  protected void onDie() {
    super.onDie();
    game.setGameState(Constant.GAME_STATE.SUCCESS);
  }

  /**
   * Update Boss status every frame Switch phases based on health, and adjust speed and firing rate
   * accordingly
   */
  @Override
  protected void onUpdate(List<AbstractEntity> worldEntities) {
    super.onUpdate(worldEntities);

    if (hp >= 20) {
      stage = Stage.stage1;
    } else if (hp >= 10) {
      stage = Stage.stage2;
      speed = 1.2;
    } else if (hp > 0) {
      stage = Stage.stage3;
      speed = 1.8;
    }

    if (stage != lastStage) {
      switch (stage) {
        case stage1 -> shootInterval = 90;
        case stage2 -> shootInterval = 150;
        case stage3 -> shootInterval = 120;
      }

      shootCooldown = shootInterval;
      lastStage = stage;
    }
  }

  /** Boss Shooting Entry Activate different shooting modes based on the current stage */
  @Override
  protected void doShoot(List<AbstractEntity> worldEntities) {
    switch (stage) {
      case stage1:
        shootStage1(worldEntities);
        break;
      case stage2:
        shootStage2(worldEntities);
        break;
      case stage3:
        shootStage3(worldEntities);
        break;
    }
  }

  /**
   * First stage firing method - Three straight-line bullets in a scattering pattern - Two curved
   * bullets
   */
  private void shootStage1(List<AbstractEntity> worldEntities) {
    Canvas c = getCanvas();
    if (c == null) return;

    double cx = x + size.getWidth() / 2; // @params
    double by = y + size.getHeight();

    for (int i = -1; i <= 1; i++) {
      worldEntities.add(
          BulletFactory.createEnemyBullet(c, cx + i * 20, by + 4, i * 0.6, 3.5, false));
    }

    worldEntities.add(BulletFactory.createCurvedEnemyBullet(c, cx - 40, by, -1.2, 3.0, 0.05));
    worldEntities.add(BulletFactory.createCurvedEnemyBullet(c, cx - 40, by, -1.2, 3.0, -0.05));
  }

  /** Second stage firing method - 1 homing bullet - 7 fan-shaped straight bullets */
  private void shootStage2(List<AbstractEntity> worldEntities) {
    Canvas c = getCanvas();
    if (c == null) return;

    double cx = x + size.getWidth() / 2;
    double by = y + size.getHeight();

    worldEntities.add(
            BulletFactory.createHomingBullet(
                c,
                cx,
                by,
                0,
                2.5,
                150, // Tracking bullet (locks onto player for 2.5 seconds)
                new HomingTrajectory.TargetProvider() {
                  @Override
                  public double getTargetX() {
                    return game.getPlayerAircraft() != null ? game.getPlayerAircraft().getX() : cx;
                  }

                  @Override
                  public double getTargetY() {
                    return game.getPlayerAircraft() != null
                        ? game.getPlayerAircraft().getY()
                        : by + 200;
                  }
                }));

    for (int i = -3; i <= 3; i++) {
      worldEntities.add(BulletFactory.createEnemyBullet(c, cx, by + 4, i * 0.8, 3.8, i == 0));
    }
  }

  /**
   * Third stage firing mode (Berserk) - Laser - Homing projectile - High-speed linear projectile
   */
  private void shootStage3(List<AbstractEntity> worldEntities) {
    Canvas c = getCanvas();
    if (c == null) return;

    double cx = x + size.getWidth() / 2;
    double by = y + size.getHeight();

    worldEntities.add(
        BulletFactory.createLaserBullet(c, cx - 30, by + 4, 0, 0, 60, 20)); // create Laser bullet

    worldEntities.add(
        BulletFactory.createHomingBullet(
            c,
            cx + 40,
            by + 4,
            0,
            2.5,
            120, // tracking 2 seconds
            new HomingTrajectory.TargetProvider() {
              @Override
              public double getTargetX() {
                return game.getPlayerAircraft() != null ? game.getPlayerAircraft().getX() : cx;
              }

              @Override
              public double getTargetY() {
                return game.getPlayerAircraft() != null
                    ? game.getPlayerAircraft().getY()
                    : by + 200;
              }
            }));

    worldEntities.add(
        BulletFactory.createEnemyBullet(c, cx, by + 4, 0, 4.5, true)); // create straight bullet
  }
}
```

```java
// src/main/java/org/thunderfighter/game/aircraft/enemy/EliteEnemy.java

package org.thunderfighter.game.aircraft.enemy;

import java.util.List;
import javafx.geometry.Dimension2D;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractEnemyAircraft;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.game.bullet.NormalEnemyBullet;

/**
 * EliteEnemy Elite Enemy Plane Characteristics: - Low Health - Moves vertically downwards in a
 * straight line - Periodically fires single, linear bullets
 */
public class EliteEnemy extends AbstractEnemyAircraft {

  /** Elite enemy aircraft dimensions (width 60, height 80) */
  public static final Dimension2D SIZE = new Dimension2D(60, 80); // @params

  /**
   * Constructor
   *
   * @param x The X coordinate at birth (Y coordinate is fixed at the top of the screen)
   */
  public EliteEnemy(double x) {
    this.size = SIZE;

    this.x = x;
    this.y = -size.getHeight();

    this.hp = 3;
    this.speed = 2;
    this.score = 50;
    this.canShoot = true;
    this.shootInterval = 120;

    this.sprite = new Image(getClass().getResourceAsStream("/images/Aircraft/EliteEnemy.png"));
  }

  /** Update logic per frame Currently only calling parent class logic */
  @Override
  protected void onUpdate(List<AbstractEntity> worldEntities) {
    super.onUpdate(worldEntities);
  }

  /** Movement logic Elite enemy aircraft only move vertically downwards */
  @Override
  protected void move() {
    y += speed;
  }

  /** Shooting logic Fire a standard enemy bullet directly downwards */
  @Override
  protected void doShoot(List<AbstractEntity> worldEntities) {
    NormalEnemyBullet bullet =
        new NormalEnemyBullet(
            x + size.getWidth() / 2 - 4, y + size.getHeight() + 4, 0, 4, false); // @params
    worldEntities.add(bullet);
  }
}
```

```java
// src/main/java/org/thunderfighter/game/aircraft/enemy/NormalEnemy.java

package org.thunderfighter.game.aircraft.enemy;

import java.util.List;
import javafx.geometry.Dimension2D;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractEnemyAircraft;
import org.thunderfighter.core.abstractor.AbstractEntity;

/** Normal Enemy Characteristics: - Lowest Health - Only moves vertically downwards - Cannot fire */
public class NormalEnemy extends AbstractEnemyAircraft {

  /** Typical enemy aircraft dimensions (width 50, height 60) */
  public static final Dimension2D SIZE = new Dimension2D(50, 60); // @params

  /**
   * Constructor
   *
   * @param x The X coordinate at birth (Y coordinate is fixed at the top of the screen)
   */
  public NormalEnemy(double x) {
    this.size = SIZE;

    this.x = x;
    this.y = -size.getHeight();

    this.hp = 1;
    this.speed = 2;
    this.score = 10;
    this.canShoot = false;

    this.sprite = new Image(getClass().getResourceAsStream("/images/Aircraft/NormalEnemy.png"));
  }

  /** Update logic per frame Currently only performing basic updates for the parent class */
  @Override
  protected void onUpdate(List<AbstractEntity> worldEntities) {
    super.onUpdate(worldEntities);
  }

  /** Movement logic Normal enemy aircraft always move vertically downwards */
  @Override
  protected void move() {
    y += speed;
  }

  /** Firing logic Ordinary enemy aircraft do not have firing capabilities */
  @Override
  protected void doShoot(List<AbstractEntity> worldEntities) {
    // Normal enemy does not shoot
  }
}
```

```java
// src/main/java/org/thunderfighter/game/aircraft/player/PlayerAircraft.java

package org.thunderfighter.game.aircraft.player;

import java.util.List;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.Canvas;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.core.abstractor.AbstractPlayerAircraft;
import org.thunderfighter.game.Game;
import org.thunderfighter.game.bullet.PlayerBullet;
import org.thunderfighter.utils.Constant.GAME_STATE;

/**
 * Player-controlled aircraft Features: - Supports movement up, down, left, and right - Diagonal
 * movement undergoes vector normalization to ensure consistent speed - Can fire at player targets -
 * Has damage cooldown
 */
public class PlayerAircraft extends AbstractPlayerAircraft {

  /** Player aircraft dimensions (width 60, height 80) */
  public static final Dimension2D SIZE = new Dimension2D(60, 80); // @params

  /** Timestamp of the last time damage was taken (milliseconds) */
  private long lastDamageTime = 0; // last damage

  /** Injury cooldown (milliseconds), used to prevent continuous health loss */
  private final long damageCooldown = 1000; // damage cd

  /**
   * Constructor
   *
   * @param x Initial X coordinate
   * @param y Initial Y coordinate
   * @param hp Initial health
   * @param speed Movement speed
   * @param shootInterval Shooting interval (frames)
   * @param canvas Game canvas
   * @param game Main game object
   */
  public PlayerAircraft(
      double x, double y, int hp, double speed, int shootInterval, Canvas canvas, Game game) {
    this.x = x;
    this.y = y;
    this.hp = hp;
    this.speed = speed;
    this.shootInterval = shootInterval;
    this.size = SIZE;

    this.sprite = new Image(getClass().getResourceAsStream("/images/Aircraft/PlayerAircraft.png"));
    this.canvas = canvas;
    this.game = game;
  }

  /**
   * Player movement logic - Calculate the movement vector based on the arrow key state - Normalize
   * the vector for diagonal movement to prevent it from being faster - Detect boundaries to prevent
   * players from flying off-screen
   */
  @Override
  protected void move() {
    double dx = 0;
    double dy = 0;
    double tempX = this.x;
    double tempY = this.y;

    if (up) {
      dy -= 1;
    }
    if (down) {
      dy += 1;
    }
    if (left) {
      dx -= 1;
    }
    if (right) {
      dx += 1;
    }

    if (dx != 0 || dy != 0) {
      double length = Math.sqrt(dx * dx + dy * dy);
      dx /= length;
      dy /= length;
    }

    tempX += dx * speed;
    tempY += dy * speed;

    if (tempX <= 0
        || tempX + this.size.getWidth() >= canvas.getWidth()
        || tempY <= 0
        || tempY + this.size.getHeight() >= canvas.getHeight()) {
      return;
    }

    this.x = tempX;
    this.y = tempY;
  } // to prevent the aircraft from traveling faster at an angle than straight.

  /** Player shooting logic Fire a player bullet directly above the plane */
  @Override
  protected void doShoot(List<AbstractEntity> worldEntities) {
    PlayerBullet bullet = new PlayerBullet(x + size.getWidth() / 2 - 4, y - 10); // @params
    bullet.setCanvas(canvas);
    worldEntities.add(bullet); // by manager instance to manage it.
  }

  /**
   * Player damage logic - There is a cooldown period for taking damage (invincibility frames) -
   * Game ends when health reaches zero
   */
  @Override
  public void takeDamage(int amount) {
    long currentTime = System.currentTimeMillis();
    if (currentTime - lastDamageTime >= damageCooldown) {
      this.hp -= amount;
      lastDamageTime = currentTime; // update damage cd time
    }
    if (hp <= 0) {
      aliveFlag = false;
      game.setGameState(GAME_STATE.FAIL);
    }
  }
}
```

```java
// src/main/java/org/thunderfighter/game/bullet/BulletType.java

package org.thunderfighter.game.bullet;

/**
 * BulletType
 *
 * <p>Useful for boss/elite pattern selection and readable configs. This enum does not create
 * bullets; BulletFactory does.
 */
public enum BulletType {
  PLAYER_NORMAL,
  ENEMY_NORMAL,
  ENEMY_CURVE,
  ENEMY_HOMING,
  ENEMY_LASER
}
```

```java
// src/main/java/org/thunderfighter/game/bullet/Clearable.java

package org.thunderfighter.game.bullet;

public interface Clearable {
  void clearImmediately();
}
```

```java
// src/main/java/org/thunderfighter/game/bullet/CurveEnemyBullet.java

package org.thunderfighter.game.bullet;

import java.util.List;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.core.entity.Aircraft;
import org.thunderfighter.game.trajectory.CurveTrajectory;

/**
 * CurveEnemyBullet
 *
 * <p>Enemy bullet that follows a curved trajectory.
 *
 * <p>This type of bullet is typically used for area control, forcing the player to reposition
 * rather than dodge straight lines. The curvature behavior is determined by the curveFactor.
 */
public class CurveEnemyBullet extends AbstractBullet {

  private static final int DAMAGE = 1;

  private static final Image SPRITE =
      new Image(CurveEnemyBullet.class.getResourceAsStream("/images/Bullet/enemy_bullet.png"));

  /**
   * Constructs a curved enemy bullet with an initial velocity and curvature.
   *
   * @param startX starting x position
   * @param startY starting y position
   * @param dx initial horizontal velocity
   * @param dy initial vertical velocity
   * @param curveFactor curvature strength for the trajectory
   */
  public CurveEnemyBullet(double startX, double startY, double dx, double dy, double curveFactor) {

    // Initial position
    this.x = startX;
    this.y = startY;
    this.originX = startX;
    this.originY = startY;

    // Bullet dimensions
    this.size = new Dimension2D(6, 12);

    // Initial velocity and speed
    this.dx = dx;
    this.dy = dy;
    this.speed = Math.hypot(dx, dy);

    // Enemy bullet flag
    this.fromPlayer = false;

    // Assign curved trajectory
    this.trajectory = new CurveTrajectory(curveFactor);

    // Infinite lifetime unless destroyed or out of bounds
    this.lifeTicks = -1;
  }

  /**
   * Updates bullet state for one game tick.
   *
   * <p>Handles movement, lifetime ticking, and boundary checks.
   */
  @Override
  public void update(List<AbstractEntity> worldEntities) {
    if (!aliveFlag) return;

    moveOnce();
    tickLife();
    killIfOutOfBounds();
  }

  /**
   * Called when this bullet collides with an aircraft.
   *
   * <p>Applies damage and destroys the bullet.
   */
  @Override
  public void onHit(Aircraft target) {
    target.takeDamage(DAMAGE);
    aliveFlag = false;
  }

  /** Renders the bullet sprite to the canvas. */
  @Override
  public void draw(GraphicsContext gc) {
    if (!aliveFlag) return;
    gc.drawImage(SPRITE, x, y, size.getWidth(), size.getHeight());
  }
}
```

```java
// src/main/java/org/thunderfighter/game/bulletfactory/BulletFactory.java

package org.thunderfighter.game.bulletfactory;

import javafx.scene.canvas.Canvas;
import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.game.bullet.CurveEnemyBullet;
import org.thunderfighter.game.bullet.HomingEnemyBullet;
import org.thunderfighter.game.bullet.LaserBullet;
import org.thunderfighter.game.bullet.NormalEnemyBullet;
import org.thunderfighter.game.bullet.PlayerBullet;
import org.thunderfighter.game.trajectory.CurveTrajectory;
import org.thunderfighter.game.trajectory.HomingTrajectory;
import org.thunderfighter.game.trajectory.StraightTrajectory;

/**
 * BulletFactory
 *
 * <p>Centralized factory responsible for creating all bullet instances.
 *
 * <p>Responsibilities: - Encapsulates bullet construction logic - Assigns appropriate trajectories
 * - Injects Canvas dependency into bullets
 *
 * <p>This class belongs to the game layer and prevents bullet creation logic from being scattered
 * across entities.
 */
public final class BulletFactory {

  /** Utility class: prevent instantiation */
  private BulletFactory() {}

  // ------------------------------------------------
  // Internal helper
  // ------------------------------------------------

  /**
   * Injects Canvas into a bullet instance if available.
   *
   * <p>This avoids forcing every bullet constructor to depend on Canvas and keeps rendering
   * concerns centralized.
   */
  private static <T extends AbstractBullet> T injectCanvas(Canvas canvas, T bullet) {
    if (canvas != null) {
      bullet.setCanvas(canvas);
    }
    return bullet;
  }

  // ------------------------------------------------
  // Player Bullets
  // ------------------------------------------------

  /** Creates a standard straight-moving player bullet. */
  public static PlayerBullet createPlayerBullet(Canvas canvas, double x, double y) {
    PlayerBullet bullet = new PlayerBullet(x, y);
    injectCanvas(canvas, bullet);
    bullet.setTrajectory(new StraightTrajectory());
    return bullet;
  }

  // ------------------------------------------------
  // Enemy Bullets
  // ------------------------------------------------

  /**
   * Creates a normal enemy bullet with fixed initial velocity.
   *
   * @param large whether to use large bullet visuals
   */
  public static NormalEnemyBullet createEnemyBullet(
      Canvas canvas, double x, double y, double dx, double dy, boolean large) {

    NormalEnemyBullet bullet = new NormalEnemyBullet(x, y, dx, dy, large);
    injectCanvas(canvas, bullet);
    bullet.setTrajectory(new StraightTrajectory());
    return bullet;
  }

  /** Creates an enemy bullet that follows a curved trajectory. */
  public static CurveEnemyBullet createCurvedEnemyBullet(
      Canvas canvas, double x, double y, double dx, double dy, double curveFactor) {

    CurveEnemyBullet bullet = new CurveEnemyBullet(x, y, dx, dy, curveFactor);
    injectCanvas(canvas, bullet);
    bullet.setTrajectory(new CurveTrajectory(curveFactor));
    return bullet;
  }

  /**
   * Creates a homing enemy bullet.
   *
   * <p>The bullet gradually turns toward the target for a limited number of ticks before expiring.
   */
  public static HomingEnemyBullet createHomingBullet(
      Canvas canvas,
      double x,
      double y,
      double dx,
      double dy,
      int trackingTicks,
      HomingTrajectory.TargetProvider provider) {

    HomingEnemyBullet bullet = new HomingEnemyBullet(x, y, dx, dy, trackingTicks, provider);

    injectCanvas(canvas, bullet);
    bullet.setTrajectory(new HomingTrajectory(provider, 0.12));
    return bullet;
  }

  // ------------------------------------------------
  // Laser Bullets (Boss)
  // ------------------------------------------------

  /**
   * Creates a laser-type bullet, typically used by bosses.
   *
   * <p>Laser bullets usually manage their own lifetime and rendering and therefore may not require
   * a trajectory.
   */
  public static LaserBullet createLaserBullet(
      Canvas canvas,
      double x,
      double y,
      double dx,
      double dy,
      int durationTicks,
      double thickness) {

    LaserBullet bullet = new LaserBullet(x, y, dx, dy, durationTicks, thickness);

    injectCanvas(canvas, bullet);
    // bullet.setTrajectory(new StraightTrajectory());
    return bullet;
  }
}
```

```java
// src/main/java/org/thunderfighter/game/bullet/HomingEnemyBullet.java

package org.thunderfighter.game.bullet;

import java.util.List;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.core.entity.Aircraft;
import org.thunderfighter.game.trajectory.HomingTrajectory;

/**
 * HomingEnemyBullet
 *
 * <p>Enemy bullet that tracks the player for a limited duration.
 *
 * <p>The bullet gradually adjusts its direction using a homing trajectory, making it slower and
 * more predictable than instant-lock bullets. This design provides pressure without being unfair to
 * the player.
 */
public class HomingEnemyBullet extends AbstractBullet {

  private static final int DAMAGE = 1;

  private static final Image SPRITE =
      new Image(HomingEnemyBullet.class.getResourceAsStream("/images/Bullet/enemy_bullet.png"));

  private int trackingTicks;

  /**
   * Constructs a homing enemy bullet.
   *
   * @param startX initial x position
   * @param startY initial y position
   * @param initDx initial horizontal velocity
   * @param initDy initial vertical velocity
   * @param trackingTicks number of ticks the bullet will track the target
   * @param provider supplies real-time target position
   */
  public HomingEnemyBullet(
      double startX,
      double startY,
      double initDx,
      double initDy,
      int trackingTicks,
      HomingTrajectory.TargetProvider provider) {

    // Initial position
    this.x = startX;
    this.y = startY;
    this.originX = startX;
    this.originY = startY;

    // Bullet dimensions
    this.size = new Dimension2D(12, 24);

    // Initial velocity and speed
    this.dx = initDx;
    this.dy = initDy;
    this.speed = Math.hypot(initDx, initDy);

    // Enemy bullet flag
    this.fromPlayer = false;

    // Assign homing trajectory
    this.trajectory = new HomingTrajectory(provider, 0.10);

    // Ensure tracking duration is at least one tick
    this.trackingTicks = Math.max(1, trackingTicks);

    // Infinite lifetime unless expired or destroyed
    this.lifeTicks = -1;
  }

  /**
   * Updates bullet state for one game tick.
   *
   * <p>Decreases remaining tracking time and removes the bullet once the tracking duration expires.
   */
  @Override
  public void update(List<AbstractEntity> worldEntities) {
    if (!aliveFlag) return;

    trackingTicks--;
    if (trackingTicks ≤ 0) {
      aliveFlag = false;
      return;
    }

    moveOnce();
    tickLife();
    killIfOutOfBounds();
  }

  /**
   * Called when this bullet collides with an aircraft.
   *
   * <p>Applies damage and destroys the bullet.
   */
  @Override
  public void onHit(Aircraft target) {
    target.takeDamage(DAMAGE);
    aliveFlag = false;
  }

  /** Renders the bullet sprite to the canvas. */
  @Override
  public void draw(GraphicsContext gc) {
    if (!aliveFlag) return;
    gc.drawImage(SPRITE, x, y, size.getWidth(), size.getHeight());
  }
}
```

// src/main/java/org/thunderfighter/game/bullet/LaserBullet.java

```java
package org.thunderfighter.game.bullet;

import java.util.List;
import javafx.geometry.BoundingBox;
import javafx.geometry.Bounds;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.core.entity.Aircraft;

/**
 * LaserBullet
 *
 * <p>Design: - Sprite default direction is DOWN (+Y). - Laser is a beam (not a moving projectile).
 * - It grows to the canvas boundary within growTicks. - Stays for durationTicks, can be cleared
 * immediately by CLEAR item. - Execute mechanic: takeDamage(Integer.MAX_VALUE).
 */
public class LaserBullet extends AbstractBullet implements Clearable {

  // ------------------------------------------------------------
  // Sprite (default direction: DOWN)
  // ------------------------------------------------------------
  private static final Image LASER_SPRITE =
      new Image(LaserBullet.class.getResourceAsStream("/images/Bullet/laser.jpg"));

  // ------------------------------------------------------------
  // Parameters
  // ------------------------------------------------------------
  private final double thickness;
  private int remainTicks;

  /** How many ticks for the beam to fully reach the boundary (0.5~0.7s @60TPS -> 30~42). */
  private final int growTicks;

  /** Current age in ticks (used for growing). */
  private int ageTicks = 0;

  /** Normalized direction (unit vector). */
  private final double dirX;

  private final double dirY;

  /** Beam start anchor (muzzle point). */
  private final double startX;

  private final double startY;

  /** Cached full beam length to boundary. */
  private double fullLength = -1; // lazy compute after canvas available

  /**
   * Recommended ctor: - dx/dy defines direction (laser points to that direction) - durationTicks:
   * how long laser exists - thickness: beam width (pixels) - growTicks: how long it takes to grow
   * to boundary (e.g., 36)
   */
  public LaserBullet(
      double startX,
      double startY,
      double dx,
      double dy,
      int durationTicks,
      double thickness,
      int growTicks) {

    this.startX = startX;
    this.startY = startY;

    // Keep AbstractBullet fields consistent
    this.x = startX;
    this.y = startY;
    this.originX = startX;
    this.originY = startY;

    this.thickness = thickness;
    this.size = new Dimension2D(thickness, thickness);

    this.fromPlayer = false;

    this.remainTicks = Math.max(1, durationTicks);
    this.growTicks = Math.max(1, growTicks);

    // Normalize direction
    double len = Math.hypot(dx, dy);
    if (len < 1e-6) {
      this.dirX = 0.0;
      this.dirY = 1.0; // default DOWN
    } else {
      this.dirX = dx / len;
      this.dirY = dy / len;
    }

    // semantic only
    this.dx = this.dirX;
    this.dy = this.dirY;
    this.speed = 0; // beam does not move
    this.lifeTicks = -1; // we use remainTicks
  }

  /** Convenience ctor with default growTicks = 36 (~0.6s @60TPS). */
  public LaserBullet(
      double startX, double startY, double dx, double dy, int durationTicks, double thickness) {
    this(startX, startY, dx, dy, durationTicks, thickness, 36);
  }

  /** Beam thickness in pixels. */
  public double getThickness() {
    return thickness;
  }

  /** Current beam length (growing). */
  public double getCurrentLength() {
    ensureFullLengthComputed();
    if (fullLength <= 0) return 0;

    double t = Math.min(1.0, ageTicks / (double) growTicks);
    return fullLength * t;
  }

  @Override
  public void update(List<AbstractEntity> worldEntities) {
    if (!aliveFlag) return;

    ageTicks++;
    remainTicks--;
    if (remainTicks <= 0) aliveFlag = false;
  }

  @Override
  public void onHit(Aircraft target) {
    target.takeDamage(target.getHp()); // execute
    // piercing by default
  }

  @Override
  public void clearImmediately() {
    aliveFlag = false;
  }

  /**
   * Laser collision: Use an axis-aligned bounding box covering the current beam segment (AABB
   * approximation).
   */
  @Override
  public Bounds getCollisionBounds() {
    double len = getCurrentLength();
    double endX = startX + dirX * len;
    double endY = startY + dirY * len;

    double minX = Math.min(startX, endX) - thickness / 2.0;
    double minY = Math.min(startY, endY) - thickness / 2.0;
    double maxX = Math.max(startX, endX) + thickness / 2.0;
    double maxY = Math.max(startY, endY) + thickness / 2.0;

    return new BoundingBox(minX, minY, Math.max(1, maxX - minX), Math.max(1, maxY - minY));
  }

  @Override
  public void draw(GraphicsContext gc) {
    if (!aliveFlag) return;

    double len = getCurrentLength();
    if (len <= 1) return;

    // rotation: sprite default DOWN (+Y)
    // angleToDir is angle from +X; DOWN is +90deg => subtract 90deg
    double angleToDir = Math.atan2(dirY, dirX);
    double rotationDegrees = Math.toDegrees(angleToDir - Math.PI / 2.0);

    gc.save();
    gc.translate(startX, startY);
    gc.rotate(rotationDegrees);

    // draw stretched laser downward (y+)
    gc.drawImage(LASER_SPRITE, -thickness / 2.0, 0, thickness, len);

    gc.restore();
  }

  // ------------------------------------------------------------
  // Internal
  // ------------------------------------------------------------
  private void ensureFullLengthComputed() {
    if (fullLength > 0) return;

    // canvas comes from AbstractEntity; must be injected like aircraft
    if (this.canvas == null) {
      // no canvas yet, keep safe
      fullLength = 0;
      return;
    }

    double w = canvas.getWidth();
    double h = canvas.getHeight();
    if (w <= 1 || h <= 1) {
      fullLength = 0;
      return;
    }

    fullLength = computeRayLengthToBounds(startX, startY, dirX, dirY, w, h);
  }

  private static double computeRayLengthToBounds(
      double x, double y, double dx, double dy, double w, double h) {

    double tMin = Double.POSITIVE_INFINITY;

    if (Math.abs(dx) > 1e-9) {
      double tx1 = (0 - x) / dx;
      double tx2 = (w - x) / dx;
      if (tx1 > 0) tMin = Math.min(tMin, tx1);
      if (tx2 > 0) tMin = Math.min(tMin, tx2);
    }

    if (Math.abs(dy) > 1e-9) {
      double ty1 = (0 - y) / dy;
      double ty2 = (h - y) / dy;
      if (ty1 > 0) tMin = Math.min(tMin, ty1);
      if (ty2 > 0) tMin = Math.min(tMin, ty2);
    }

    if (!Double.isFinite(tMin) || tMin <= 0) {
      return Math.hypot(w, h);
    }

    return Math.min(Math.hypot(w, h), tMin);
  }
}
```

```java
// src/main/java/org/thunderfighter/game/bullet/NormalEnemyBullet.java

package org.thunderfighter.game.bullet;

import java.util.List;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.core.entity.Aircraft;
import org.thunderfighter.game.trajectory.StraightTrajectory;

/**
 * NormalEnemyBullet
 *
 * <p>Straight enemy bullet. - Damage: 1 - Dies when out of canvas bounds (uses injected canvas) -
 * Supports small / large size
 *
 * <p>Compatibility: - Provides a 2-parameter constructor to match existing Aircraft.doShoot(x, y) -
 * Also provides full constructor for advanced usage (boss / elite patterns)
 */
public class NormalEnemyBullet extends AbstractBullet {

  private static final int DAMAGE = 1;

  /** Default downward speed (per tick) for legacy constructor. */
  private static final double DEFAULT_DY = 8.0;

  /** Enemy bullet sprite (red dot). */
  private static final Image SPRITE =
      new Image(NormalEnemyBullet.class.getResourceAsStream("/images/Bullet/enemy_bullet.png"));

  /** Legacy constructor: straight downward, small bullet. */
  public NormalEnemyBullet(double startX, double startY) {
    this(startX, startY, 0.0, DEFAULT_DY, false);
  }

  /**
   * Full constructor (recommended when spawning patterns).
   *
   * <p>NOTE: - canvasW/canvasH removed - Out-of-bounds uses injected canvas from AbstractEntity
   */
  public NormalEnemyBullet(double startX, double startY, double dx, double dy, boolean large) {

    this.x = startX;
    this.y = startY;
    this.originX = startX;
    this.originY = startY;

    // Size: round dot style
    this.size = large ? new Dimension2D(28, 28) : new Dimension2D(16, 16);

    // Velocity per tick
    this.dx = dx;
    this.dy = dy;

    // Semantic speed
    this.speed = Math.hypot(dx, dy);

    this.fromPlayer = false;
    this.trajectory = new StraightTrajectory();
    this.lifeTicks = -1;
  }

  @Override
  public void update(List<AbstractEntity> worldEntities) {
    if (!aliveFlag) return;
    moveOnce();
    tickLife();
    killIfOutOfBounds(); // now depends on injected canvas size
  }

  @Override
  public void onHit(Aircraft target) {
    target.takeDamage(DAMAGE);
    aliveFlag = false;
  }

  @Override
  public void draw(GraphicsContext gc) {
    if (!aliveFlag) return;
    gc.drawImage(SPRITE, x, y, size.getWidth(), size.getHeight());
  }
}
```

```java
// src/main/java/org/thunderfighter/game/bullet/PlayerBullet.java

package org.thunderfighter.game.bullet;

import java.util.List;
import javafx.geometry.Dimension2D;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.core.entity.Aircraft;
import org.thunderfighter.game.trajectory.StraightTrajectory;

/**
 * PlayerBullet
 *
 * <p>Straight bullet fired by the player. - Damage: 1 - Dies when out of canvas bounds
 *
 * <p>Canvas bounds are handled via AbstractEntity.setCanvas(...)
 */
public class PlayerBullet extends AbstractBullet {

  private static final int DAMAGE = 1;

  /** Default per-tick speed upward. */
  private static final double DEFAULT_DY = -12.0;

  /** Player bullet sprite. */
  private static final Image SPRITE =
      new Image(PlayerBullet.class.getResourceAsStream("/images/Bullet/player_bullet.png"));

  /** Constructor. Compatible with existing Aircraft.doShoot(x, y) */
  public PlayerBullet(double startX, double startY) {
    this.x = startX;
    this.y = startY;

    this.originX = startX;
    this.originY = startY;

    // Bullet size (adjust to sprite if needed)
    this.size = new Dimension2D(12, 24);

    // Straight upward velocity
    this.dx = 0.0;
    this.dy = DEFAULT_DY;

    // Semantic speed (optional, if AbstractEntity uses it)
    this.speed = Math.abs(DEFAULT_DY);

    this.fromPlayer = true;
    this.trajectory = new StraightTrajectory();

    // Infinite life; removed by out-of-bounds
    this.lifeTicks = -1;
  }

  @Override
  public void update(List<AbstractEntity> worldEntities) {
    if (!aliveFlag) return;

    moveOnce();
    tickLife();
    killIfOutOfBounds();
  }

  @Override
  public void onHit(Aircraft target) {
    target.takeDamage(DAMAGE);
    aliveFlag = false;
  }

  @Override
  public void draw(GraphicsContext gc) {
    if (!aliveFlag) return;

    gc.drawImage(SPRITE, x, y, size.getWidth(), size.getHeight());
  }
}
```

```java
// src/main/java/org/thunderfighter/game/Game.java

package org.thunderfighter.game;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javafx.animation.AnimationTimer;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import org.thunderfighter.core.abstractor.AbstractEnemyAircraft;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.core.collision.CollisionDetector;
import org.thunderfighter.core.manager.ScoreManager;
import org.thunderfighter.game.aircraft.player.PlayerAircraft;
import org.thunderfighter.game.spawn.EnemySpawner;
import org.thunderfighter.ui.KeyboardController;
import org.thunderfighter.ui.ScoreBoard;
import org.thunderfighter.ui.UiMenu;
import org.thunderfighter.ui.UiOverlay;
import org.thunderfighter.ui.UiScoreStorage;
import org.thunderfighter.utils.Constant;
import org.thunderfighter.utils.Constant.GAME_STATE;
import org.thunderfighter.utils.Constant.PHASE;

/** Control and manage the game. */
public class Game {

  // GUI related.
  private AnimationTimer animationTimer;
  private final Canvas canvas;
  private final GraphicsContext graphicsContext;
  private final Scene scene;
  private final StackPane root;
  private KeyboardController keyboardController;
  private final UiOverlay overlay;
  private final UiMenu menu;
  private ScoreBoard scoreBoard;
  private final Image backgroundImage =
      new Image(getClass().getResourceAsStream("/images/Background/bg.png"));

  private long lastTime = 0;

  // Player.
  private PlayerAircraft playerAircraft;

  // Manage all enetities.
  private final List<AbstractEntity> entities = new ArrayList<>();

  // Enemy related.
  private EnemySpawner enemySpawner;
  private int numberOfEnemy = 0;
  private Constant.PHASE enemyStage = Constant.PHASE.NORMAL;

  // Game related.
  GAME_STATE gameState;
  private boolean fromMenuStart = false;
  private boolean scoreStored = false;
  private boolean scoreRead = false;

  public Game(final Stage stage) {
    overlay = new UiOverlay(this);
    menu = new UiMenu(this, overlay);
    canvas = new Canvas(800, 900);
    graphicsContext = canvas.getGraphicsContext2D();
    root = new StackPane(canvas, overlay, menu);
    gameState = GAME_STATE.MENU;
    this.scene = new Scene(root);

    // Load CSS style.
    String css = getClass().getResource("/css/styles.css").toExternalForm();
    scene.getStylesheets().add(css);

    stage.setScene(this.scene);
    stage.show();

    initGame();
  }

  public void setGameState(final GAME_STATE gameState) {
    this.gameState = gameState;
  }

  public GAME_STATE getGameState() {
    return gameState;
  }

  /**
   * Toggle pause state.
   *
   * <p>If the current state is {@code GAME_STATE.RUNNING}, then the game state should be set to
   * {@code GAME_STATE.PAUSE}.
   *
   * <p>Otherwise, the game should be resume, whose game state should be {@code GAME_STATE.RUNNING}.
   */
  public void togglePause() {
    if (gameState == GAME_STATE.RUNNING) {
      gameState = GAME_STATE.PAUSE;
    } else {
      gameState = GAME_STATE.RUNNING;
    }
  }

  public void start() {
    if (animationTimer != null) {
      animationTimer.start();
    }
  }

  public void stop() {
    animationTimer.stop();
  }

  /**
   * Update frames of the game.
   *
   * <p>The order of updating is:
   *
   * <ol>
   *   <li>Generate enemy.
   *   <li>Determine whether the player want to shoot.
   *   <li>Detect collisions.
   *   <li>Update position and state of each {@link Entity}.
   *   <li>Remove dead objects and update {@code numberOfEnemy}.
   *   <li>Update the score board.
   * </ol>
   */
  public void update() {
    generateEnemy();
    if (playerAircraft.wantToShoot()) {
      playerAircraft.shoot(entities);
    }

    CollisionDetector.detectCollision(entities);

    final Iterator<AbstractEntity> it = entities.iterator();
    final List<AbstractEntity> tempList = new ArrayList<>();
    while (it.hasNext()) {
      final AbstractEntity entity = it.next();
      if (entity instanceof AbstractEnemyAircraft) {
        entity.update(tempList);
      } else {
        entity.update(entities);
      }
      if (!entity.isAlive()) {
        it.remove();
        if (entity instanceof AbstractEnemyAircraft) {
          numberOfEnemy--;
        }
      }
    }
    entities.addAll(tempList);
    scoreBoard.update();
  }

  /**
   * Draw each frame.
   *
   * <p>This method will draw the background at first, then it will draw each entity stored in
   * {@code entites}.
   */
  public void draw() {
    graphicsContext.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
    graphicsContext.drawImage(backgroundImage, 0, 0, canvas.getWidth(), canvas.getHeight());
    for (final AbstractEntity entity : entities) {
      entity.draw(graphicsContext);
    }
  }

  public PlayerAircraft getPlayerAircraft() {
    return playerAircraft;
  }

  /**
   * Core logic of running the game.
   *
   * <p>It offers different methods to run under different game state.
   */
  private void initAnimationTimer() {
    animationTimer =
        new AnimationTimer() {
          @Override
          public void handle(final long now) {
            // If this is the first run, initialize the timestamp.
            if (lastTime == 0) {
              lastTime = now;
              return;
            }

            // Calculate the time interval (nanoseconds) required for each frame.
            // 1seconds = 1,000,000,000 ns
            final long interval = 1_000_000_000 / Constant.TPS;

            // The logic is executed only when the difference
            // between the current time and the previous time is greater than or equal to the
            // interval.
            if (now - lastTime >= interval) {
              switch (gameState) {
                case MENU -> handleMenuState();
                case RUNNING -> handleRunningState();
                case PAUSE -> handlePauseState();
                case SUCCESS -> handleSuccessState();
                case FAIL -> handleFailState();
              }
              // Update last execution time
              lastTime = now;
            }
          }
        };
  }

  /** Initialize the game at first launch. */
  private void initGame() {
    initEntities();
    enemySpawner = new EnemySpawner(canvas, entities, this);
    scoreBoard = new ScoreBoard(root, playerAircraft);
    this.keyboardController = new KeyboardController(playerAircraft, this);
    keyboardController.operation(this.scene);
    ScoreManager.getInstance().reset();
    initAnimationTimer();
  }

  /** Initialize entities and register them into the {@code entites} list. */
  private void initEntities() {
    entities.clear();
    numberOfEnemy = 0;
    playerAircraft =
        new PlayerAircraft(
            canvas.getWidth() / 2,
            canvas.getHeight() - PlayerAircraft.SIZE.getHeight() - 10,
            3,
            10,
            20,
            canvas,
            this);
    entities.add(playerAircraft);
  }

  /** Restart the game when clicking the start button in the main menu. */
  private void restartGame() {
    ScoreManager.getInstance().reset();
    enemyStage = PHASE.NORMAL;
    initEntities();
    keyboardController.setPlayer(this.playerAircraft);
    scoreBoard.setPlayerAircraft(this.playerAircraft);
    enemySpawner.reset();
  }

  /** Actions to be taken when the user is in menu. */
  private void handleMenuState() {
    graphicsContext.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
    menu.setVisible(true);
    overlay.setVisible(false);
    scoreBoard.setVisible(false);
    fromMenuStart = true;
    scoreStored = false;
    if (!scoreRead) {
      UiScoreStorage.readFromFile();
      scoreRead = true;
    }
  }

  /** Actions to be taken during running the game. */
  private void handleRunningState() {
    // A new round of game.
    if (fromMenuStart) {
      restartGame();
      fromMenuStart = false;
    }
    menu.setVisible(false);
    overlay.setVisible(false);
    scoreBoard.setVisible(true);
    scoreRead = false;
    update();
    draw();
  }

  /** Actions to be done when the game is paused. */
  private void handlePauseState() {
    overlay.showPause();
  }

  /**
   * Utility function used to store scores to the data file.
   *
   * <p>It avoid adding scores repeatly by using a flag {@code scoreStored}.
   *
   * <p>If the flag is {@code true}, it will skip writing into file. Otherwise, it will do so.
   */
  private void storeScore() {
    if (!scoreStored) {
      UiScoreStorage.addScore(ScoreManager.getInstance().getScore());
      scoreStored = true;
      UiScoreStorage.writeToFile();
    }
  }

  /** Actions to be taken when the player is successed. */
  private void handleSuccessState() {
    overlay.showSuccess();
    storeScore();
  }

  /** Actions to be taken if the player failed in the game. */
  private void handleFailState() {
    overlay.showFail();
    storeScore();
  }

  /**
   * Generate enemies according to different phases and scores.
   *
   * <p>If the number of enemy reaches the limit, this method will do nothing.
   */
  private void generateEnemy() {
    if (numberOfEnemy >= Constant.ENEMY_NUMBER_LIMIT) {
      return;
    }

    // Update the phase before entering switch-case statements.
    final int currentScore = ScoreManager.getInstance().getScore();
    if (enemyStage == PHASE.NORMAL && currentScore >= Constant.GENERATE_ELITE_SCORE) {
      enemyStage = PHASE.ELITE;
    } else if (enemyStage == PHASE.ELITE && currentScore >= Constant.GENERATE_BOSS_SCORE) {
      enemyStage = PHASE.BOSS;
    }

    // | Phase  |     Enemy       |
    // | ------ | --------------- |
    // | NORMAL | Normal          |
    // | ELITE  | Elite + Normal  |
    // | BOSS   | Boss            |
    switch (enemyStage) {
      case NORMAL:
        if (enemySpawner.spawnNormal()) {
          numberOfEnemy++;
        }
        break;
      case ELITE:
        if (enemySpawner.spawnElite()) {
          numberOfEnemy++;
        }
        if (enemySpawner.spawnNormal()) {
          numberOfEnemy++;
        }
        break;
      case BOSS:
        if (enemySpawner.spawnBoss(this)) {
          numberOfEnemy++;
        }
        break;
    }
  }
}
```

```java
// src/main/java/org/thunderfighter/game/spawn/EnemySpawner.java

package org.thunderfighter.game.spawn;

import java.util.List;
import java.util.Random;
import javafx.scene.canvas.Canvas;
import org.thunderfighter.core.abstractor.AbstractEntity;
import org.thunderfighter.game.Game;
import org.thunderfighter.game.aircraft.enemy.*;
import org.thunderfighter.utils.Constant;

/**
 * Enemy Generator Responsible for: - Normal Enemy Generator (NormalEnemy) - Elite Enemy Generator
 * (EliteEnemy) - Boss Generator (BossEnemy) Uses cooldown (CD) to control the generation frequency
 */
public class EnemySpawner {

  /** Random number generator used to randomly generate the X coordinate of enemy aircraft */
  private final Random rng = new Random();

  /** Game canvas */
  private final Canvas canvas;

  /** World entity list, used to store generated enemy aircraft */
  private List<AbstractEntity> entities;

  /** Cooldown for normal enemy aircraft spawning (frames) */
  private int normalCd = 0;

  /** Cooldown for elite enemy aircraft spawning (frames) */
  private int eliteCd = 0;

  /** Current Boss instance (guaranteed that only one Boss exists at a time) */
  private BossEnemy boss = null;

  /**
   * Constructor
   *
   * @param canvas Game canvas
   * @param entities List of world entities
   * @param game Game object (used when the boss is generated)
   */
  public EnemySpawner(Canvas canvas, List<AbstractEntity> entities, Game game) {
    this.canvas = canvas;
    this.entities = entities;
  }

  /** Resets the generator state Usually called when restarting the game */
  public void reset() {
    normalCd = eliteCd = 0;
    boss = null;
  }

  /**
   * Generate a regular enemy aircraft
   *
   * @return Whether the generation was successful
   */
  public boolean spawnNormal() {
    if (normalCd-- > 0) {
      return false;
    }
    normalCd = Constant.TPS;

    NormalEnemy enemy = new NormalEnemy(randomX());
    enemy.setCanvas(canvas);
    entities.add(enemy);

    return true;
  }

  /**
   * Generate elite enemy aircraft
   *
   * @return Whether generation was successful
   */
  public boolean spawnElite() {
    if (eliteCd-- > 0) {
      return false;
    }
    eliteCd = 3 * Constant.TPS;

    EliteEnemy enemy = new EliteEnemy(randomX());
    enemy.setCanvas(canvas);
    entities.add(enemy);

    return true;
  }

  /**
   * Generate a Boss Only one Boss is allowed at a time
   *
   * @param game Game object (used for Boss state control)
   * @return Whether the Boss was successfully generated
   */
  public boolean spawnBoss(Game game) {
    if (boss != null && boss.isAlive()) {
      return false;
    }

    boss = new BossEnemy(canvas.getWidth() / 2 - 100, 0, game);
    boss.setCanvas(canvas);
    entities.add(boss);

    return true;
  }

  /**
   * Generate random X coordinates Ensure enemy aircraft do not exceed the right side of the canvas
   */
  private double randomX() {
    return rng.nextDouble() * (canvas.getWidth() - 60);
  }
}
```

```java
// src/main/java/org/thunderfighter/game/trajectory/CurveTrajectory.java

package org.thunderfighter.game.trajectory;

import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.entity.Trajectory;

/**
 * CurveTrajectory
 *
 * <p>Implements a curved (arc-like) bullet movement by applying a perpendicular acceleration and
 * gravity-like downward force.
 *
 * <p>This trajectory is commonly used for arcing bullets, falling projectiles, or stylized enemy
 * patterns.
 */
public class CurveTrajectory implements Trajectory {

  private final double factor;
  private final double gravity;
  private final boolean keepSpeed;
  private final double minDownVy;
  private final double maxSpeed;

  /**
   * Convenience constructor with default parameters.
   *
   * @param factor curvature strength
   */
  public CurveTrajectory(double factor) {
    this(factor, 0.65, true, 4.0, 18.0);
  }

  /**
   * Constructor allowing custom gravity while keeping other defaults.
   *
   * @param factor curvature strength
   * @param gravity downward acceleration
   */
  public CurveTrajectory(double factor, double gravity) {
    this(factor, gravity, true, 4.0, 18.0);
  }

  /**
   * Full constructor for advanced control over the trajectory behavior.
   *
   * @param factor curvature strength
   * @param gravity downward acceleration
   * @param keepSpeed whether to normalize speed each frame
   * @param minDownVy minimum downward velocity
   * @param maxSpeed maximum allowed speed
   */
  public CurveTrajectory(
      double factor, double gravity, boolean keepSpeed, double minDownVy, double maxSpeed) {
    this.factor = factor;
    this.gravity = gravity;
    this.keepSpeed = keepSpeed;
    this.minDownVy = Math.max(0.0, minDownVy);
    this.maxSpeed = maxSpeed;
  }

  /**
   * Updates bullet velocity and position for one frame.
   *
   * @param b the bullet to update
   */
  @Override
  public void update(AbstractBullet b) {
    // Current velocity components
    double vx = b.getDx();
    double vy = b.getDy();

    // Compute perpendicular acceleration to create curved motion
    // Rotates velocity slightly each frame
    double axCurve = -vy * factor;
    double ayCurve = vx * factor;

    // Apply curvature acceleration and gravity
    double newVx = vx + axCurve;
    double newVy = vy + ayCurve + gravity;

    // Optionally normalize velocity magnitude to keep speed stable
    // This prevents exponential acceleration and keeps the curve smooth
    if (keepSpeed) {
      double oldSpeed = Math.hypot(vx, vy);
      double newSpeed = Math.hypot(newVx, newVy);
      if (oldSpeed > 1e-6 && newSpeed > 1e-6) {
        double scale = oldSpeed / newSpeed;
        newVx *= scale;
        newVy *= scale;
      }
    }

    // Enforce a minimum downward velocity to ensure falling behavior
    if (minDownVy > 0.0 && newVy < minDownVy) {
      newVy = minDownVy;
    }

    // Clamp velocity magnitude to avoid unrealistic speeds
    if (maxSpeed > 0.0) {
      double s = Math.hypot(newVx, newVy);
      if (s > maxSpeed && s > 1e-6) {
        double scale = maxSpeed / s;
        newVx *= scale;
        newVy *= scale;
      }
    }

    // Apply updated velocity to bullet
    b.setDx(newVx);
    b.setDy(newVy);

    // Move bullet according to updated velocity
    b.setX(b.getX() + b.getDx());
    b.setY(b.getY() + b.getDy());
  }
}
```

```java
// src/main/java/org/thunderfighter/game/trajectory/HomingTrajectory.java

package org.thunderfighter.game.trajectory;

import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.entity.Trajectory;

/**
 * HomingTrajectory
 *
 * <p>Implements a homing movement that gradually rotates the bullet's velocity vector toward a
 * moving target.
 *
 * <p>This produces smooth tracking behavior instead of instant turning, which feels more natural
 * and fair in gameplay.
 *
 * <p>turnStrength recommended range: 0.05 ~ 0.20
 */
public class HomingTrajectory implements Trajectory {

  /**
   * TargetProvider
   *
   * <p>Supplies real-time target coordinates, typically the player. This abstraction avoids direct
   * dependency on player classes.
   */
  public interface TargetProvider {
    double getTargetX();

    double getTargetY();
  }

  private final TargetProvider provider;
  private final double turnStrength;

  /**
   * Constructs a homing trajectory.
   *
   * @param provider supplies target position
   * @param turnStrength interpolation factor for turning (clamped to [0,1])
   */
  public HomingTrajectory(TargetProvider provider, double turnStrength) {
    this.provider = provider;
    this.turnStrength = Math.max(0.0, Math.min(1.0, turnStrength));
  }

  /**
   * Updates bullet direction and position for one frame.
   *
   * @param b bullet instance to update
   */
  @Override
  public void update(AbstractBullet b) {
    // Current bullet position
    double bx = b.getX();
    double by = b.getY();

    // Vector from bullet to target
    double tx = provider.getTargetX() - bx;
    double ty = provider.getTargetY() - by;

    double targetLen = Math.hypot(tx, ty);
    if (targetLen < 1e-6) {
      // Target overlaps bullet; continue straight without turning
      b.setX(bx + b.getDx());
      b.setY(by + b.getDy());
      return;
    }

    // Normalized direction toward target
    double ux = tx / targetLen;
    double uy = ty / targetLen;

    // Current velocity and speed
    double vx = b.getDx();
    double vy = b.getDy();
    double speed = Math.hypot(vx, vy);
    if (speed < 1e-6) speed = 1.0;

    // Current velocity unit vector
    double vxUnit = vx / speed;
    double vyUnit = vy / speed;

    // Interpolate direction toward target (smooth turning)
    double nx = vxUnit + (ux - vxUnit) * turnStrength;
    double ny = vyUnit + (uy - vyUnit) * turnStrength;

    double nLen = Math.hypot(nx, ny);
    if (nLen < 1e-6) nLen = 1.0;

    // Restore original speed after direction change
    b.setDx((nx / nLen) * speed);
    b.setDy((ny / nLen) * speed);

    // Move bullet using updated velocity
    b.setX(bx + b.getDx());
    b.setY(by + b.getDy());
  }
}
```

```java
// src/main/java/org/thunderfighter/game/trajectory/StraightTrajectory.java

package org.thunderfighter.game.trajectory;

import org.thunderfighter.core.abstractor.AbstractBullet;
import org.thunderfighter.core.entity.Trajectory;

/**
 * StraightTrajectory
 *
 * <p>Each tick moves the bullet by its velocity vector (dx, dy).
 */
public class StraightTrajectory implements Trajectory {

  @Override
  public void update(AbstractBullet b) {
    b.setX(b.getX() + b.getDx());
    b.setY(b.getY() + b.getDy());
  }
}
```

```java
// src/main/java/org/thunderfighter/Main.java

package org.thunderfighter;

import javafx.application.Application;
import javafx.stage.Stage;
import org.thunderfighter.game.Game;

public class Main extends Application {
  public static void main(final String[] args) {
    launch();
  }

  @Override
  public void start(final Stage stage) {
    final Game game = new Game(stage);
    game.start();
  }
}
```

```java
// src/main/java/org/thunderfighter/ui/KeyboardController.java

package org.thunderfighter.ui;

import javafx.scene.Scene;
import javafx.scene.input.KeyCode;
import org.thunderfighter.core.abstractor.AbstractPlayerAircraft;
import org.thunderfighter.game.Game;

// Keyboard input controller for player aircraft, enable operations based on key presses
public class KeyboardController {

  private AbstractPlayerAircraft player;
  private final Game game;

  public KeyboardController(AbstractPlayerAircraft player, Game game) {
    this.player = player;
    this.game = game;
  } // create a keyboard controller for the player aircraft

  public void setPlayer(AbstractPlayerAircraft player) {
    this.player = player;
  }

  public void operation(Scene scene) {

    // Handle key pressed events
    scene.setOnKeyPressed(
        event -> {
          KeyCode code = event.getCode();
          switch (code) {
            case W, UP -> player.setUp(true);
            case S, DOWN -> player.setDown(true);
            case A, LEFT -> player.setLeft(true);
            case D, RIGHT -> player.setRight(true);
            case SPACE -> {
              player.setShooting(true);
              event.consume();
            }
            case P -> {
              // Pause/resume is driven by the global game state.
              game.togglePause();
              event.consume();
            }
            default -> {}
          }
        });
    // When users press the key board, operate the player aircraft
    // W, A, S, D or Arrow Keys to move the aircraft
    // Space to shoot

    scene.setOnKeyReleased(
        event -> {
          KeyCode code = event.getCode();
          switch (code) {
            case W, UP -> player.setUp(false);
            case S, DOWN -> player.setDown(false);
            case A, LEFT -> player.setLeft(false);
            case D, RIGHT -> player.setRight(false);
            case SPACE -> {
              player.setShooting(false);
              event.consume();
            }
            default -> {}
          }
        });
    // when users release the key board, stop the operation of the player aircraft
  }
  // Enable users to operate by the keyboard

  /* Notes:
  The reason why I write the same method twice is to realize two operations:
  Control the player aircraft when users press the key board
  Stop controlling the player aircraft when users release the key board
  */
}
```

```java
// src/main/java/org/thunderfighter/ui/ScoreBoard.java

package org.thunderfighter.ui;

import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import org.thunderfighter.core.manager.ScoreManager;
import org.thunderfighter.game.aircraft.player.PlayerAircraft;

/** Board to display score and lives. */
public class ScoreBoard {

  private VBox scoreBoardContainer;
  private Label scoreLabel;
  private Label livesLabel;
  private PlayerAircraft playerAircraft;

  /**
   * Constructor of {@code ScoreBoard}.
   *
   * @param root The root pane.
   * @param playerAircraft Player's aircraft to display the lives.
   */
  public ScoreBoard(final StackPane root, final PlayerAircraft playerAircraft) {
    this.playerAircraft = playerAircraft;
    initScoreBoardContainer();

    root.getChildren().add(scoreBoardContainer);
  }

  public void setPlayerAircraft(final PlayerAircraft playerAircraft) {
    this.playerAircraft = playerAircraft;
  }

  public void setVisible(final boolean visible) {
    scoreBoardContainer.setVisible(visible);
  }

  /** Update the score board. */
  public void update() {
    scoreLabel.setText("Score: " + ScoreManager.getInstance().getScore());
    livesLabel.setText("Lives: " + playerAircraft.getHp());
  }

  /** Initialize the score board and set its style. */
  private void initScoreBoardContainer() {
    scoreBoardContainer = new VBox(12);
    scoreBoardContainer.setAlignment(Pos.TOP_RIGHT);
    StackPane.setAlignment(scoreBoardContainer, Pos.TOP_RIGHT);
    StackPane.setMargin(scoreBoardContainer, new Insets(20, 20, 0, 0));
    scoreBoardContainer.setPrefWidth(200);
    scoreBoardContainer.setMaxWidth(200);
    scoreBoardContainer.setMaxHeight(100);
    final Color bgColor = Color.rgb(0, 0, 0, 0.6);
    final CornerRadii radii = new CornerRadii(15);
    scoreBoardContainer.setBackground(
        new Background(new BackgroundFill(bgColor, radii, Insets.EMPTY)));
    scoreBoardContainer.setPadding(new Insets(15));
    scoreBoardContainer.setMouseTransparent(true);
    final String labelStyle =
        "-fx-font-family: 'Arial Black'; -fx-font-size: 20px; -fx-text-fill: white; -fx-effect:"
            + " dropshadow(gaussian, black, 4, 0, 2, 2);";
    scoreLabel = new Label("Score: " + ScoreManager.getInstance().getScore());
    scoreLabel.setStyle(labelStyle);

    livesLabel = new Label("Lives: " + playerAircraft.getHp());
    scoreBoardContainer.getChildren().addAll(scoreLabel, livesLabel);
    livesLabel.setStyle(labelStyle);
  }
}
```

```java
// src/main/java/org/thunderfighter/ui/UiDialog.java

package org.thunderfighter.ui;

import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import org.thunderfighter.core.abstractor.AbstractUiDialog;

/*
 UiDialog class for displaying various dialogs in the Thunder Fighter game.
 This class will be used in UiScoreStorage, UiMenu and etc. to show dialogs.
 In this class, we define methods to show information dialogs, about dialog, and game rule dialog.
 Therefore, we can reuse this class to show different types of dialogs by using the methods showInfoDialog, showAboutDialog, and showRuleDialog.
 */
public class UiDialog extends AbstractUiDialog {
  private final Alert alert;

  // Create an information dialog with given title and message
  public UiDialog(String title, String message) {
    alert = new Alert(AlertType.INFORMATION);
    alert.setTitle(title);
    alert.setHeaderText(null);
    alert.setContentText(message);
  }

  @Override
  public void showDialog() {
    alert.showAndWait();
  } // Show the dialog and wait for user to close it

  public static void showInfoDialog(String title, String message) {
    Alert alert = new Alert(AlertType.INFORMATION);
    alert.setTitle(title);
    alert.setHeaderText(null);
    alert.setContentText(message);
    alert.showAndWait();
  } // show generic information of dialog

  public static void showAboutDialog() {
    showInfoDialog("About", "Thunder Fighter © 2025\nAll rights reserved.");
  } // show the copyright in dialog

  public static void showRuleDialog() {
    showInfoDialog(
      "Game Rule",
      "1. Use arrow keys or WASD to move the fighter.\n"
          + "2. Press space to shoot.\n"
          + "3. Avoid enemy fire and obstacles.\n"
          + "4. Score points by destroying enemies.\n"
          + "5. When the score comes to 100, you will meet elite enemy.\n"
          + "6. The Boss will appear when the score reaches 500.\n"
          + "7. Boss has three stages:\n"
          + "in first stage, it shoots less frequently;\n"
          + "in second stage, it shoots the homing bullet;\n"
          + "in third stage, it releases fatal laser.\n"
          + "8. Defeat the Boss to win the game!");
  } // show game rule dialog
}
```

```java
// src/main/java/org/thunderfighter/ui/UiMenu.java

package org.thunderfighter.ui;

import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import org.thunderfighter.core.abstractor.AbstractUiMenu;
import org.thunderfighter.game.Game;
import org.thunderfighter.utils.Constant;

/*
 * UiMenu class is responsible for the main menu of the Thunder Fighter game.
 * This class extends AbstractUiMenu to implement the main menu UI.
 * In this class, we define buttons for starting the game, viewing history scores, about information, and game rules.
 * Therefore, we can reuse this class to create and manage the main menu of the game
 */
public class UiMenu extends AbstractUiMenu {
  // The element of the main menu
  public final Label title = new Label("Thunder Fighter");
  public final Button startButton = new Button("Start Game");
  public final Button historyButton = new Button("History Score");
  public final Button aboutButton = new Button("About");
  public final Button ruleButton = new Button("Game Rule");

  // Initialize the main menu
  public UiMenu(Game game, UiOverlay overlay) {
    this.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE); // make the menu cover the whole window
    this.setPickOnBounds(true);

    this.setAlignment(Pos.CENTER);
    this.setSpacing(20); // set the spacing between elements in the menu

    String imagePath = getClass().getResource("/images/Background/log_in_bg.png").toExternalForm();
    this.setStyle(
        "-fx-background-image: url('"
            + imagePath
            + "');"
            + "-fx-background-size: cover;"
            + "-fx-background-position: center;"
            + "-fx-background-repeat: no-repeat;"); // add background image to the menu

    this.title.getStyleClass().add("title");

    createButton(startButton);
    createButton(historyButton);
    createButton(ruleButton);
    createButton(aboutButton);
    // Add all elements to the menu
    this.getChildren().addAll(title, startButton, historyButton, ruleButton, aboutButton);

    showMenu(); // show the menu

    // set the action for each button in the menu
    startButton.setOnAction(
        e -> {
          game.setGameState(Constant.GAME_STATE.RUNNING);
          game.start();
        }); // control the game start, using the start method in the Game class

    historyButton.setOnAction(e -> UiScoreStorage.showScoreDialog());
    ruleButton.setOnAction(e -> UiDialog.showRuleDialog());
    aboutButton.setOnAction(e -> UiDialog.showAboutDialog());
  }

  private void createButton(Button button) {
    button.setPrefWidth(200);
    button.setPrefHeight(40);
  } // the basic style for buttons

  @Override
  public void showMenu() {
    this.setVisible(true);
    this.setManaged(true);
  } // show the menu

  @Override
  public void hideMenu() {
    this.setVisible(false);
    this.setManaged(false);
  } // hide the menu
}
```

```java
// src/main/java/org/thunderfighter/ui/UiOverlay.java

package org.thunderfighter.ui;

import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import org.thunderfighter.core.abstractor.AbstractUiMenu;
import org.thunderfighter.game.Game;
import org.thunderfighter.utils.Constant;

/*
Create 3 modes:
PAUSE: resume game
SUCCESS: show victory and offer back to menu button
FAIL: show game over and offer back to menu button
We can use the showPause, showSuccess, showFail methods to show different overlays.
When users key in p, we can call showPause to show the pause overlay.
When users win the game, we can call showSuccess to show the success overlay.
When users lose the game, we can call showFail to show the fail overlay.
 */
public class UiOverlay extends AbstractUiMenu {

  /** Overlay modes, deciding title text and visible buttons. */
  private enum OverlayMode {
    PAUSE,
    SUCCESS,
    FAIL
  }

  private OverlayMode mode = OverlayMode.PAUSE;

  public final Label title = new Label("Paused");

  public final Button continueButton = new Button("Continue");
  public final Button restartButton = new Button("Back to Menu");
  public final Button historyButton = new Button("History Score");
  public final Button aboutButton = new Button("About");
  public final Button ruleButton = new Button("Game Rule");

  public UiOverlay(Game game) {
    // Make overlay cover the whole scene to prevent click-through.
    this.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
    this.setPickOnBounds(true);

    this.setAlignment(Pos.CENTER);
    this.setSpacing(20);

    this.title.getStyleClass().add("title");
    createButton(continueButton);
    createButton(historyButton);
    createButton(restartButton);
    createButton(ruleButton);
    createButton(aboutButton);

    applyMode(OverlayMode.PAUSE);
    hideMenu();

    // Continue: only valid in PAUSE mode
    continueButton.setOnAction(
        e -> {
          game.setGameState(Constant.GAME_STATE.RUNNING);
          hideMenu();
        });

    // Back to menu(valid in SUCCESS and FAIL modes)
    restartButton.setOnAction(
        e -> {
          game.setGameState(Constant.GAME_STATE.MENU);
          hideMenu();
        }); // come back to menu

    historyButton.setOnAction(e -> UiScoreStorage.showScoreDialog());
    ruleButton.setOnAction(e -> UiDialog.showRuleDialog());
    aboutButton.setOnAction(e -> UiDialog.showAboutDialog());
  }

  private void createButton(Button button) {
    button.setPrefWidth(220);
    button.setPrefHeight(40);
    button.setFocusTraversable(true);
  }

  // Show pause overlay.
  // Called by Game.handlePauseState().
  public void showPause() {
    applyMode(OverlayMode.PAUSE);
    showMenu();
  }

  // Show success overlay.
  // Called by Game.handleSuccessState().
  public void showSuccess() {
    applyMode(OverlayMode.SUCCESS);
    showMenu();
  }

  // Show fail overlay.
  // Called by Game.handleFailState().
  public void showFail() {
    applyMode(OverlayMode.FAIL);
    showMenu();
  }

  // Logic to apply the mode changes to the overlay UI.
  private void applyMode(OverlayMode newMode) {
    this.mode = newMode;

    switch (mode) {
      case PAUSE -> {
        title.setText("Paused");
        continueButton.setVisible(true);
        continueButton.setManaged(true);

        restartButton.setVisible(false);
        restartButton.setManaged(false);
      }
      case SUCCESS -> {
        title.setText("Victory!");
        continueButton.setVisible(false);
        continueButton.setManaged(false);

        restartButton.setText("Back to Menu");
        restartButton.setVisible(true);
        restartButton.setManaged(true);
      }
      case FAIL -> {
        title.setText("Game Over");
        continueButton.setVisible(false);
        continueButton.setManaged(false);

        restartButton.setText("Back to Menu");
        restartButton.setVisible(true);
        restartButton.setManaged(true);
      }
      default -> {}
    } // For different game status, show different overlay

    this.getChildren()
        .setAll(title, continueButton, restartButton, historyButton, ruleButton, aboutButton);
  }

  @Override
  public void showMenu() {
    setVisible(true);
    setMouseTransparent(false);
    toFront();
  } // Show the menu overlay

  @Override
  public void hideMenu() {
    setVisible(false);
    setMouseTransparent(true);
  } // Hide the menu overlay
}
```

```java
package org.thunderfighter.ui;

import javafx.scene.Scene;
import javafx.scene.layout.StackPane;

/*
Manages the main scene, including menu and overlay
1. UiSceneManager class is responsible for managing the scene of the Thunder Fighter game.
2. This class contains a StackPane as the root layout, which holds both the main menu and the overlay.
3. In this class, we define methods to toggle the overlay visibility, show and hide the menu and overlay.
4. Therefore, we can reuse this class to manage the main scene of the game by using the methods provided.
*/

// Manages the main scene, including menu and overlay
public class UiSceneManager {
  private final StackPane root = new StackPane();
  private final Scene scene;

  private final UiMenu menu;
  private final UiOverlay overlay;

  public UiSceneManager(UiMenu menu, UiOverlay overlay) {
    this.menu = menu;
    this.overlay = overlay;
    root.getChildren().addAll(menu, overlay);
    scene = new Scene(root, 800, 600);
    overlay.hideMenu();
  } // Initialize the scene manager with menu and overlay

  public void togglePause() {
    if (overlay.isVisible()) {
      hideOverlay();
    } else {
      showOverlay();
    }
  } // Toggle the overlay visibility

  public Scene getScene() {
    return scene;
  } // Get the main scene

  public void showOverlay() {
    overlay.showMenu();
  }

  public void hideOverlay() {
    overlay.hideMenu();
  }

  public void showMenu() {
    menu.showMenu();
  }

  public void hideMenu() {
    menu.hideMenu();
  }
}
```

```java
// src/main/java/org/thunderfighter/ui/UiScoreStorage.java

package org.thunderfighter.ui;

import java.io.IOException;
import java.io.PrintWriter;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import org.thunderfighter.utils.AppDataDirectory;

/*
UiScoreStorage class is responsible for storing and showing history scores in Thunder Fighter.
This class provides methods to add new scores, retrieve formatted scores, and show score dialogs.
In this class, we define methods readFromFile to read scores from a file, and writeToFile to write scores to a file.
Therefore, we can reuse this class to handle score storage and show history scores in the game.
*/
public class UiScoreStorage {
  private static final List<Integer> scores = new ArrayList<>();
  private static final String fileName = "score.txt";
  private static final Path dataFile =
      Paths.get(AppDataDirectory.getAppDataDirectory().toString(), fileName);

  public static void addScore(int score) {
    scores.add(score);
  } // The method to add a new score to the storage

  public static void sortScoreDescending() {
    scores.sort(
        (a, b) -> {
          return b - a;
        });
  }

  public static List<Integer> getScores() {
    return new ArrayList<>(scores);
  } // The method to get a copy of the score list

  public static String getFormattedScores() {
    if (scores.isEmpty()) {
      return "No scores yet!";
    }

    sortScoreDescending();
    StringBuilder string_builder = new StringBuilder();
    int i = 1;

    for (int score : scores) {
      string_builder.append(i + ". " + score + "\n");
      i++;
    } // Format scores as a numbered list

    return string_builder.toString();
  }

  public static void showScoreDialog() {
    UiDialog.showInfoDialog("History Score", getFormattedScores());
  } // Show the score dialog with formatted scores

  public static void readFromFile() {
    if (!Files.exists(dataFile)) {
      return;
    } // if the file does not exist, return

    try {
      Scanner scanner = new Scanner(dataFile);
      int score = 0;
      scores.clear();
      while (scanner.hasNextInt()) {
        score = scanner.nextInt();
        scores.add(score);
      }
      scanner.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  } // Read scores from the file and store them in the score list

  public static void writeToFile() {
    if (!Files.exists(dataFile)) {
      AppDataDirectory.createFile(fileName);
    } // if the file does not exist, create it

    try {
      PrintWriter printWriter = new PrintWriter(dataFile.toString());
      sortScoreDescending();
      for (int score : scores) {
        printWriter.println(score);
      }
      printWriter.close();
    } catch (IOException e) {
      e.printStackTrace();
    } // Write scores from the score list to the file
  }
}
```

```java
// src/main/java/org/thunderfighter/utils/AppDataDirectory.java

package org.thunderfighter.utils;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

/** Operation of app data directory. */
public class AppDataDirectory {

  private static final String APP_NAME = "thunder-fighter";

  /**
   * Get app data directory according to different operating systems.
   *
   * @return App data directory on different operating systems.
   */
  public static Path getAppDataDirectory() {
    final String os = System.getProperty("os.name").toLowerCase();
    Path appDir;

    if (os.contains("win")) {
      appDir = Paths.get(System.getenv("APPDATA"), APP_NAME);
    } else if (os.contains("mac")) {
      appDir =
          Paths.get(System.getProperty("user.home"), "Library", "Application Support", APP_NAME);
    } else {
      String configHome = System.getenv("XDG_CONFIG_HOME");
      if (configHome == null || configHome.isEmpty()) {
        configHome = System.getProperty("user.home") + "/.config";
      }
      appDir = Paths.get(configHome, APP_NAME);
    }

    return appDir;
  }

  /**
   * Ensure the data directory of game exists under the app data directory.
   *
   * @return The path of data directory.
   */
  public static Path ensureAppDataDirectory() {
    final Path dir = getAppDataDirectory();
    try {
      Files.createDirectory(dir);
    } catch (final IOException e) {
      e.printStackTrace();
    }
    return dir;
  }

  /**
   * Create data file under the data file directory.
   *
   * @param fileName The name of data file.
   */
  public static void createFile(final String fileName) {
    final Path dir = ensureAppDataDirectory();
    final Path file = dir.resolve(fileName);
    try {
      Files.writeString(file, "");
    } catch (final IOException e) {
      e.printStackTrace();
    }
  }
}
```

```java
// src/main/java/org/thunderfighter/utils/Constant.java

package org.thunderfighter.utils;

/** Constant */
public class Constant {

  public static final int ENEMY_NUMBER_LIMIT = 16;

  public static final int GENERATE_ELITE_SCORE = 100;
  public static final int GENERATE_BOSS_SCORE = 500;

  public static final int TPS = 120;

  public static enum PHASE {
    NORMAL,
    ELITE,
    BOSS
  }

  public static enum GAME_STATE {
    MENU,
    RUNNING,
    PAUSE,
    SUCCESS,
    FAIL,
  }
}
```