Java 2 Alien Marauders

Hordur Joensen

December 2025

Alien Marauders – Project Report

Introduction

This project is a multi-stage development of a 2D space shooter game written in JavaFX. Across several iterations, the system evolved from a simple player movement demo into a complete game with menus, concurrency, enemy waves, formations, animations, and an extensible architecture.

The primary goals of the project were:

- to structure the game using the MVC pattern,
- to use Java's concurrency features for game loops and background logic,
- to implement clean separation of menus, game states, and controllers,
- to refactor animation logic into its own graphics subsystem.

This report summarises the complete project structure, design choices, and technical features built over the course of development.

Overall Architecture

The project is structured around the **Model-View-Controller (MVC)** pattern:

- **Model**: Contains all game state, entities, enemy waves, score tracking, and animation containers.
- **View**: Responsible for rendering the game world, menus, and UI elements using JavaFX canvases and nodes.
- **Controller**: Handles input (keyboard and buttons), manages game flow, and switches between views.

Additional supporting packages include:

- `entities`: Player, Enemy, Shot, and shared entity features.
- `movement`: Enemy movement strategies.
- `formations`: Classes that spawn enemies in different shapes.
- `menu`: Controllers and views for menus, settings, login, and chat.
- `graphics`: Animation system introduced in the final phase.

This modular structure makes the project clean, extensible, and easy to maintain.

Gameplay Model

The core gameplay model manages:

- player position, movement, and shooting,
- enemy spawning through formation classes,
- collision detection,

- scoring and wave progression,
- a flashing red damage overlay when the player is hit,
- maintaining a list of active shots and enemies.

Game updates run on a JavaFX-friendly loop using timestamps and `AnimationTimer`. This allows smooth motion and consistent gameplay speed.

Enemy Formations and Strategy Pattern

Two design patterns play a major role in enemy behaviour:

Movement Strategies

A MovementStrategy interface defines:

```
void moveEnemy(Enemy enemy, double dt);
void setSpeedMultiplier(double m);
double getSpeedMultiplier();
```

Concrete strategies include:

- **NoMoveStrategy**
- **MoveDownStrategy**
- **ZigZagStrategy** using a sine function

This allows enemies to switch behaviour without modifying their class, following the Open/Closed principle.

Formation System

Each wave can spawn enemies using a chosen formation class:

- **GridFormation**
- **VFormation**
- **ArcFormation**

Formations are responsible only for positioning and bundling enemies. The model then handles updates and collisions.

Menus, Settings, Login, and Chat (UI System)

A significant part of the project involved building a complete UI system:

- Main Menu
- Settings Menu (background selection, difficulty)
- Login Menu (text fields, input validation)
- Chat Menu (multiplayer-style message feed)

A custom **SwitchViewBuilder** holds references to all screens and allows controllers to switch between them. This avoids reloading new stages and improves performance and consistency.

The chat interface uses a JavaFX `TextFlow` to support per-user colors, and usernames are ensured unique through model-side validation.

Concurrency and Threads

Because JavaFX requires all UI changes to happen on the **JavaFX Application Thread**, the project separates real-time gameplay updates from potentially slow operations (e.g., validation, chat operations) using Java concurrency.

FX Thread: Real-time game loop

The main gameplay loop runs on the JavaFX thread using `AnimationTimer`. Each frame uses timestamps to compute `dt` (delta time), then updates entities (player, shots, enemies), handles collisions, and finally renders the scene. Keeping updates + rendering on the FX thread avoids unsafe cross-thread access to JavaFX nodes and canvases.

Worker Threads: Non-blocking background work

Tasks that could block the UI (or cause frame drops) are executed off the FX thread using background threads (e.g., `Thread`, `ExecutorService`, or JavaFX `Task`). Examples include:

- chat message handling and message formatting,
- login / username validation (including uniqueness checks),
- preparation or pre-processing of collision data, with final application performed on the FX thread.

Thread-safety rules used

To keep the system correct and responsive, the following rules are enforced:

- **Never** update JavaFX UI components from worker threads. Worker threads communicate results back using `Platform.runLater(...)`.
- Shared mutable game state is either confined to the FX thread (preferred), or accessed via thread-safe handoff (e.g., queues or synchronized blocks) so that rendering and updates do not race.
- Collections that are iterated during rendering (shots, enemies, animations) are modified in a controlled way (e.g., mark-for-removal and cleanup step) to avoid concurrent modification during iteration.

In practice, most mutable gameplay state is confined to the JavaFX Application Thread. This thread-confinement approach minimizes synchronization complexity and avoids race conditions. Worker threads are used only to prepare data or perform validation, with all authoritative state updates and rendering executed safely on the FX thread.

Why this matters

This design prevents the UI from freezing during slow operations and avoids common concurrency bugs such as: *frame drops, race conditions, and illegal JavaFX thread access.* Overall, concurrency is used to improve responsiveness while preserving the rule that JavaFX rendering remains single-threaded and predictable.

Graphics and Animation Subsystem

The final evolution of the project involved a major refactor: all animation logic was moved into a new `graphics` package.

This subsystem contains:

Animatable Interface

Defines the contract for any short-lived visual effect:

```
void renderAnimation(GraphicsContext gc);
boolean isActive();
```

ImageStride

A reusable component that:

- slices horizontal sprite sheets into frames,
- tracks elapsed time to switch frames,
- supports looping and non-looping animations,
- renders the correct frame each update.

This class centralises all sprite animation timing and eliminates code duplication.

AnimationContainer T extends Animatable

A generic container that manages temporary animations:

- stores any effects implementing `Animatable`,
- updates and renders all animations each frame,
- automatically removes animations that report inactive.

The type parameter `T extends Animatable` enforces type safety while allowing flexibility.

Implemented Effects

Three animations were implemented using the subsystem:

- **ExplosionAnimation**: plays when an enemy dies.
- **HitSparkAnimation**: plays when a shot strikes an enemy.
- **ShotFlashAnimation**: muzzle flash when the player fires.

Each of these uses `ImageStride` to animate sprite sheets and inherits the automatic removal behaviour via the container.

Rendering Flow

Each frame, the rendering order is:

1. Clear canvas and draw background.
2. Draw player, shots, and enemies.
3. Draw wave banner or damage overlay if active.
4. Draw temporary animations via:

   ```
   model.getAnimations().renderAnimations(gc);
   ```

5. Draw score and UI overlays.

This layering ensures visual effects appear on top of gameplay elements.

Design Patterns Used

The project uses several classical software engineering patterns:

- **Model-View-Controller (MVC)**
- **Strategy Pattern** for enemy movement behaviours
- **Factory-like Formation classes** for spawning enemy waves
- **Facade-like View Switching** using SwitchViewBuilder
- **Observer-like Input Handling** for keyboard events
- **Generic container pattern** for managing visual effects

These patterns keep the code modular, testable, and easier to extend.

Conclusion

Throughout the project, the system evolved from a minimal JavaFX game framework into a fully-featured 2D shooter with structured architecture, enemy formations, dynamic backgrounds, animations, UI menus, and high-level gameplay features.

The final graphics subsystem provides a highly extensible animation framework using interfaces, generics, and a dedicated container. This significantly improves maintainability and allows new effects to be added with minimal effort.

The completed project demonstrates:

- solid object-oriented design,
- practical use of modern Java features,
- clean separation of concerns,
- a modular architecture suitable for future extensions.

Overall, the development process successfully integrates multiple software engineering concepts into a cohesive and polished game.