PROJECT REPORT: GALAXY WARS

Course: Object-Oriented Programming

University: FAST , Islamabad Campus

Group Members:

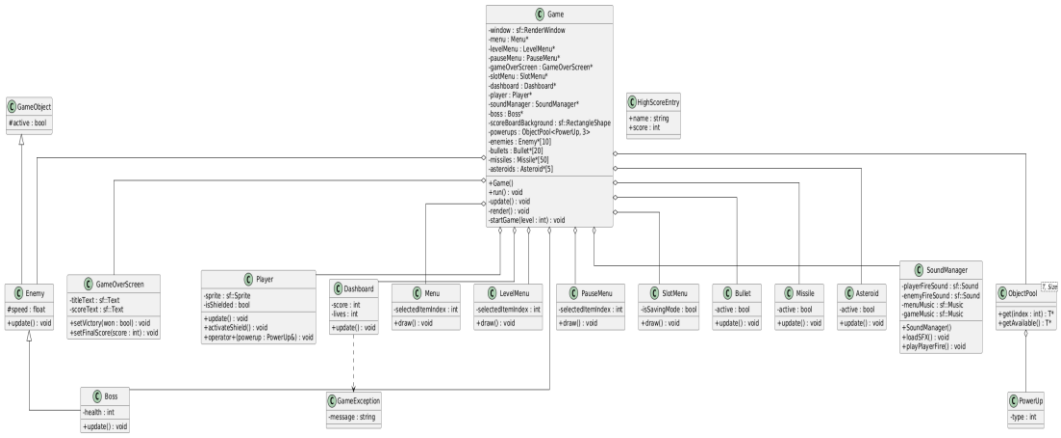
- **Danyal Aziz** (24i-0815)
- **Maaz Hussain** (24i-0708)

Submission Date: December 10, 2025

1. Overview and Motivation

Galaxy Wars is a 2D space survival strategy game designed in C++ using the SFML library. The main drive for this project was to apply advanced Object-Oriented Programming methodology to the development of a complex, interactive software system.
During the game, a player is placed in the cockpit of a starship that needs to protect the galaxy against waves of alien invaders and asteroids. The goal of this game is pretty simple but challenging: to survive as long as possible, to destroy enemies for a high score, and to finally defeat the final boss.
From a technical viewpoint, this project is an all-encompassing testbed for concepts like polymorphism, memory management, file handling, and generic programming through the use of templates. In actually constructing a game engine from scratch, we obtained deep insight into how objects interoperate in a real-time simulation loop.

2. Class Diagram



3. Mapping OOP Concepts to Code

This project strictly adheres to OOP principles. Below is a mapping of the required concepts to their specific implementation in our source code:

OOP Concept	Implementation Details	Source File
Encapsulation	All classes (`Player`, `Enemy`, `Dashboard`) use `private` attributes for data (health, speed, score) and `public` methods for access.	All Headers
Abstraction	The `GameObject` class is an abstract base class containing pure virtual functions (`update()`, `draw()`).	`GameObject.h`
Inheritance (Single)	`Player` inherits from `GameObject`.	`Player.h`
Inheritance (Multilevel)	`Boss` inherits from `Enemy`, which inherits from `GameObject`.	`Boss.h`
Polymorphism	`update()` and `draw()` are virtual functions overridden in `Player`, `Enemy`, `Asteroid`, and `Boss` to provide specific behaviors.	`GameObject.h`
Templates	The `ObjectPool<T, MaxSize>` class uses templates to efficiently manage memory for repeated objects like PowerUps.	`ObjectPool.h`
Friend Classes	The `Player` class declares `friend class Game` to allow the engine to access private timers (e.g., shield duration).	`Player.h`

OOP Concept	Implementation Details	Source File
Operator Overloading	The `+` operator is overloaded in `Player` to handle logic when a player interacts with a powerup (`player + powerup`).	`Player.h`
Static Members	`Enemy` contains a static variable `totalEnemiesSpawned` to track global spawn counts across all instances.	`Enemy.h`
Exception Handling	A custom `GameException` class is used to handle critical errors (e.g., missing assets or file write failures).	`GameException.h`
File Handling	**Text:** Used for High Score leaderboard (`scores.txt`). **Binary:** Used for Save/Load functionality (`save_x.dat`).	`Game.h`
Composition	The `Game` class comprises instances of `Menu`, `Player`, `Dashboard`, and `SoundManager`.	`Game.h`

4. Game Mechanics and Features

Gameplay

The player navigates a spaceship using keyboard controls. The game features three distinct levels of difficulty:

1. **Level 1 (Easy):** Basic enemy spawning and asteroids.
2. **Level 2 (Medium):** Increased enemy speed and density.
3. **Level 3 (Hard):** Introduction of the **Boss Enemy** which utilizes zigzag movement patterns and targeted firing.

Power-Ups

We implemented an `ObjectPool` to manage power-ups that drop randomly:

- **Shield:** Grants temporary invulnerability
- **Life:** Adds an extra life to the player.

- **Score Star:** Grants bonus points.
- **Freeze:** Temporarily freezes all enemies and projectiles on screen.

Audio System

A dedicated `SoundManager` class handles spatial sound effects (lasers, explosions, collisions) and background music. It switches tracks dynamically between the Menu (Chill theme) and Gameplay (War theme).

5. File Handling and Persistence

High Score Board - Text File

We used ofstream and ifstream to handle a leaderboard.

• Logic: When the game terminates, the player's name, score, and time played are appended to scores.txt.

• Sorting: Upon displaying the scores, the data is read into a struct array, sorted through a bubble sort algorithm in descending order, and then displayed on the GUI.

Save/Load System (Binary File)

We implemented binary serialization to fulfill the bonus requirement.

• Saving: Serializes game state - Player Name, Level, Score, Lives in raw binary format to .dat files, such as save_1.dat. This prevents users from easily editing their save files via a text editor.

• Loading: It reads the binary data directly into memory variables to restore the exact state of the session.

6. Exception Handling

Robustness was a key focus. We defined a custom exception class:

C++
```
class GameException : public std::exception { ... };
```

Usage:

1. **Asset Loading:** If fonts or textures (e.g., `Arial.ttf`) are missing, a `GameException` is thrown during initialization.
2. **File I/O:** If the game cannot open `scores.txt` for writing, the exception is caught, and an error is logged to the console without crashing the game completely.

7. Challenges and Solutions

Challenge 1: Memory Management for Projectiles
Problem: At first, the program experienced performance lag whenever the player fired because of continuous creation and deletion of bullets through dynamic memory allocation.
Solution: We implemented an array-based approach, whereby objects are instantiated once at startup, including ObjectPool for powerups. We maintain an active boolean flag that toggles them on/off, reusing memory rather than re-allocating it.

Challenge 2: Boss Movement Logic
Problem: Making Boss movement feel distinct from regular enemies.
Solution: We overridden the update() method in the Boss class. We utilized std::sin() for zigzag vertical movement and a state-machine approach (Idle vs. Moving) to make the boss behavior unpredictable.

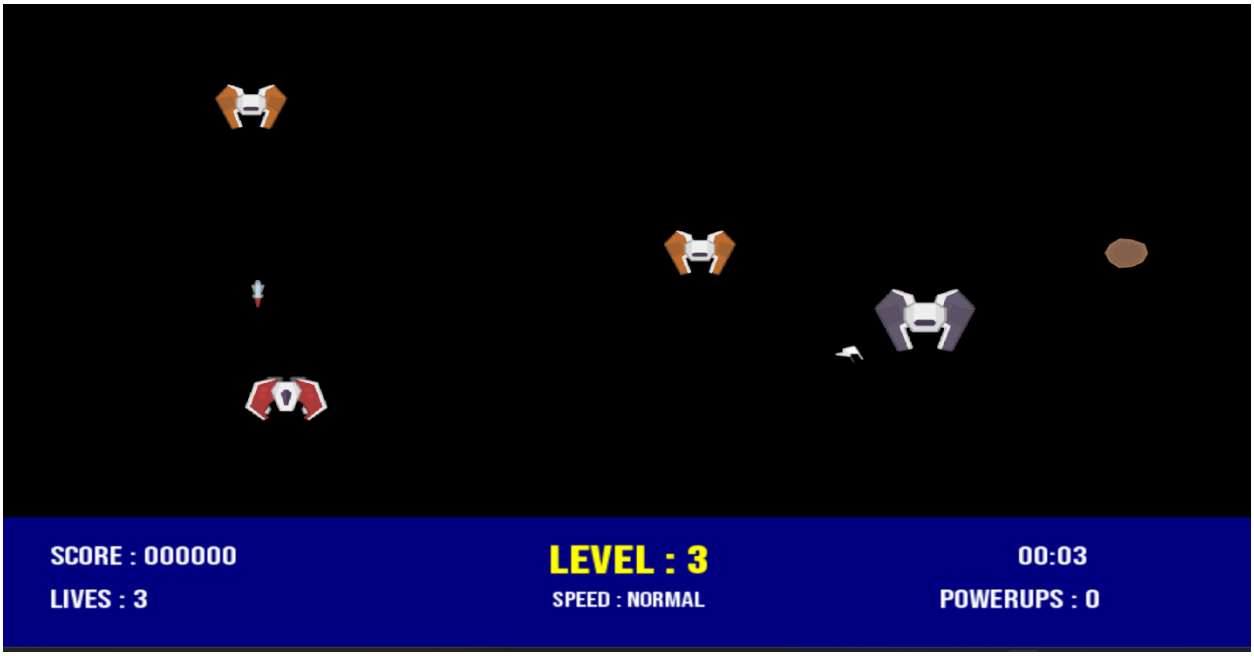
Challenge 3: Collision Precision
Problem: Rectangular collision boxes were registering hits when sprites didn't visually touch.
Solution: We tuned the getBounds() logic and sprite scaling/origins so hitboxes closely corresponded to the visible textures.

8. Game Screenshots

Main Menu

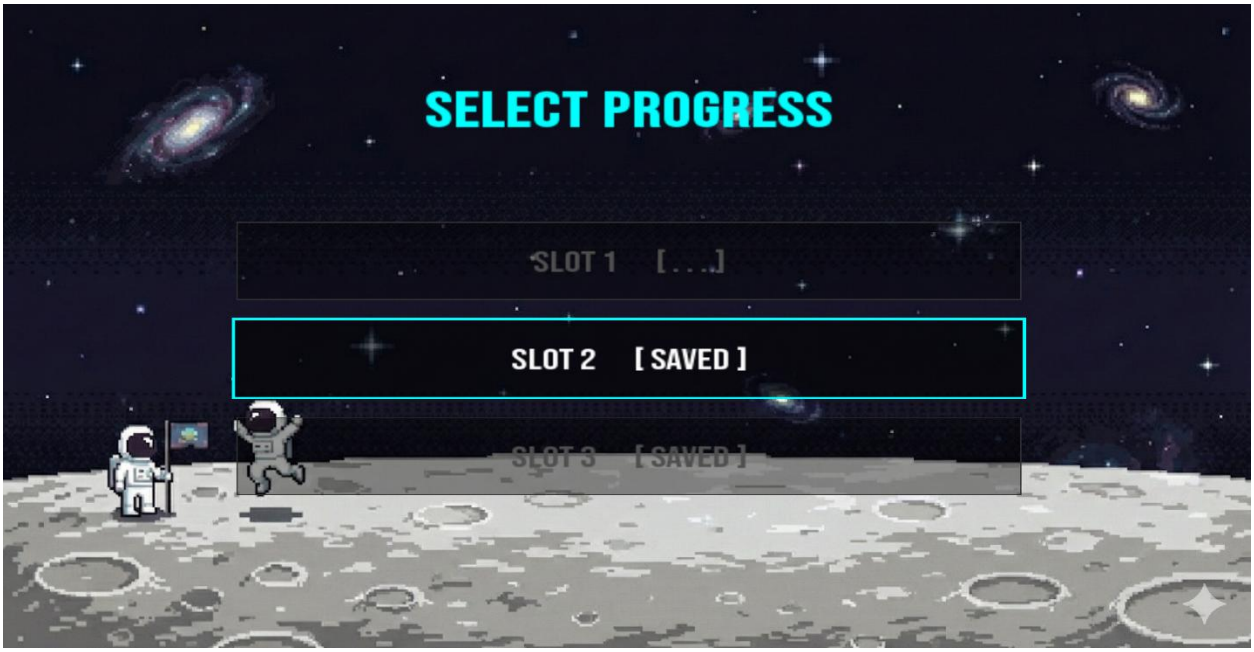
Gameplay Action (Level 2)

The Boss Battle



Scoreboard

Save/Load Slots



9. Conclusion

Galaxy Wars effectively combines all the necessary OOP concepts into one coherent and playable game. This project shows not only the syntax of C++ but also the architectural advantages of ObjectOriented design-specifically how the use of inheritance and polymorphism leads to extendable game entities and how exception handling makes for a robust user experience. Adding binary file handling and templates further optimized the performance and functionality of the game.