High Level Design for Tank Battle Game

Assignment 1

Design Decisions and Alternatives

In this project, the `GameManager` is responsible for all movement and updates on the board. It handles the full game logic: querying each tank's `Algorithm` for its next action, validating and applying the action if legal, moving shells across the board, and detecting collisions. Algorithms only suggest actions based on the current board state; they cannot directly modify the game. This strict separation keeps the code modular and aligns with the assignment requirements.

The `GameManager` manages all tanks using a `vector<Tank>`. Although the first assignment only involves one tank per player, we chose to use a vector to allow for future scalability if multiple tanks per player will be required. An alternative would have been to store each tank independently (e.g., `Tank tank1`, `Tank tank2`), but this would have made the code repetitive and inflexible when generalizing to multiple tanks later.

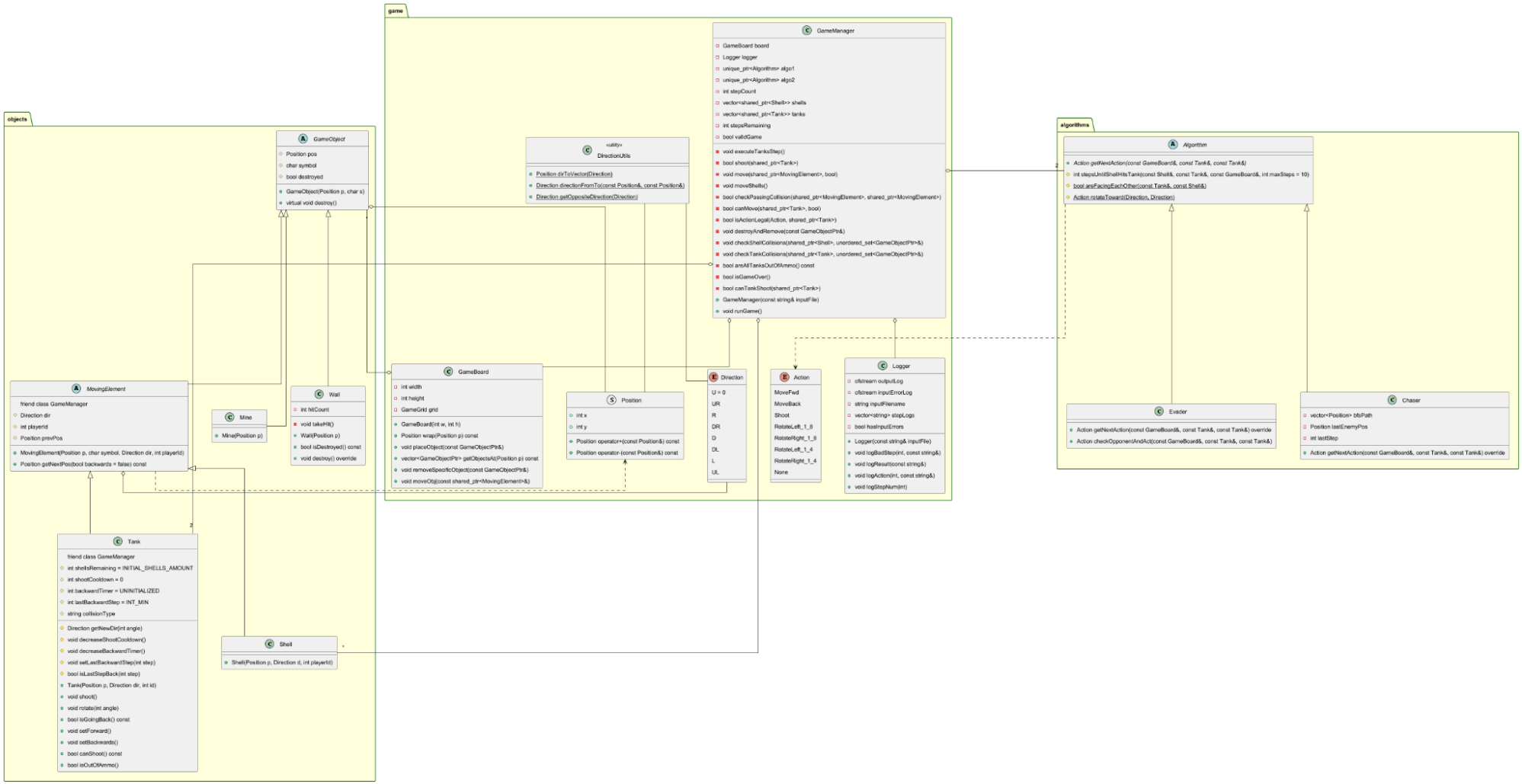
The main game loop represents half of a game step: each iteration moves shells by one block, and only after two iterations are tank actions applied. This was necessary because shells move twice as fast as tanks, and managing the loop at half-step resolution allows collisions to be detected properly without skipping over intermediate positions. Alternatively, the game loop could have stayed at full-step resolution by moving each shell twice inside the same loop and checking for collisions after each move. While technically correct, this would complicate shell movement by requiring two separate collision checks and would not align well with the intuition of a continuous world where objects move smoothly rather than in abrupt jumps.

To better organize the movement-related behavior of tanks and shells, we introduced a `MovingElement` class that inherits from `GameObject`, with both `Tank` and `Shell` inheriting from it. This allows shared movement logic to be handled consistently across different moving objects. An alternative would have been to let each class manage movement separately, but this would have led to code duplication and inconsistencies in behavior.

Testing Approach

We tested the project manually by running it on different input files and reviewing the outputs. Several scenarios were checked to make sure the program behaved as expected. Testing focused on verifying general correctness across a variety of cases.

Class UML Diagram



Sequence UML Diagram