Lior Pernik
Alin Loshevsky

HW1 – Tank Game

Game Overview:

The code represents a **tank battle game** with turn-based mechanics, shell collisions, and grid-based movement.

Map Structure (as given in example):

- #: **Wall** (2 HP, blocks movement/shots).
- @: **Mine** (instantly destroys tanks that step on them).
- 1/2: Starting positions for **Player 1** and **Player 2**.
- S: Shell moving on board (in bonus basic visual of game steps 'gameSteps_output
- _inputFileName.txt')

Turn Order:

Each turn consists of:

1. Shell movement first (2 cells per turn) with collision resolution.
2. Tank actions (movement/rotation/firing, based on the Algorithm's decision).

The manager validates moves, updates cooldowns, checks for all types of collisions, and logs all events. If a tank is hit by a shell or mine, it's destroyed.

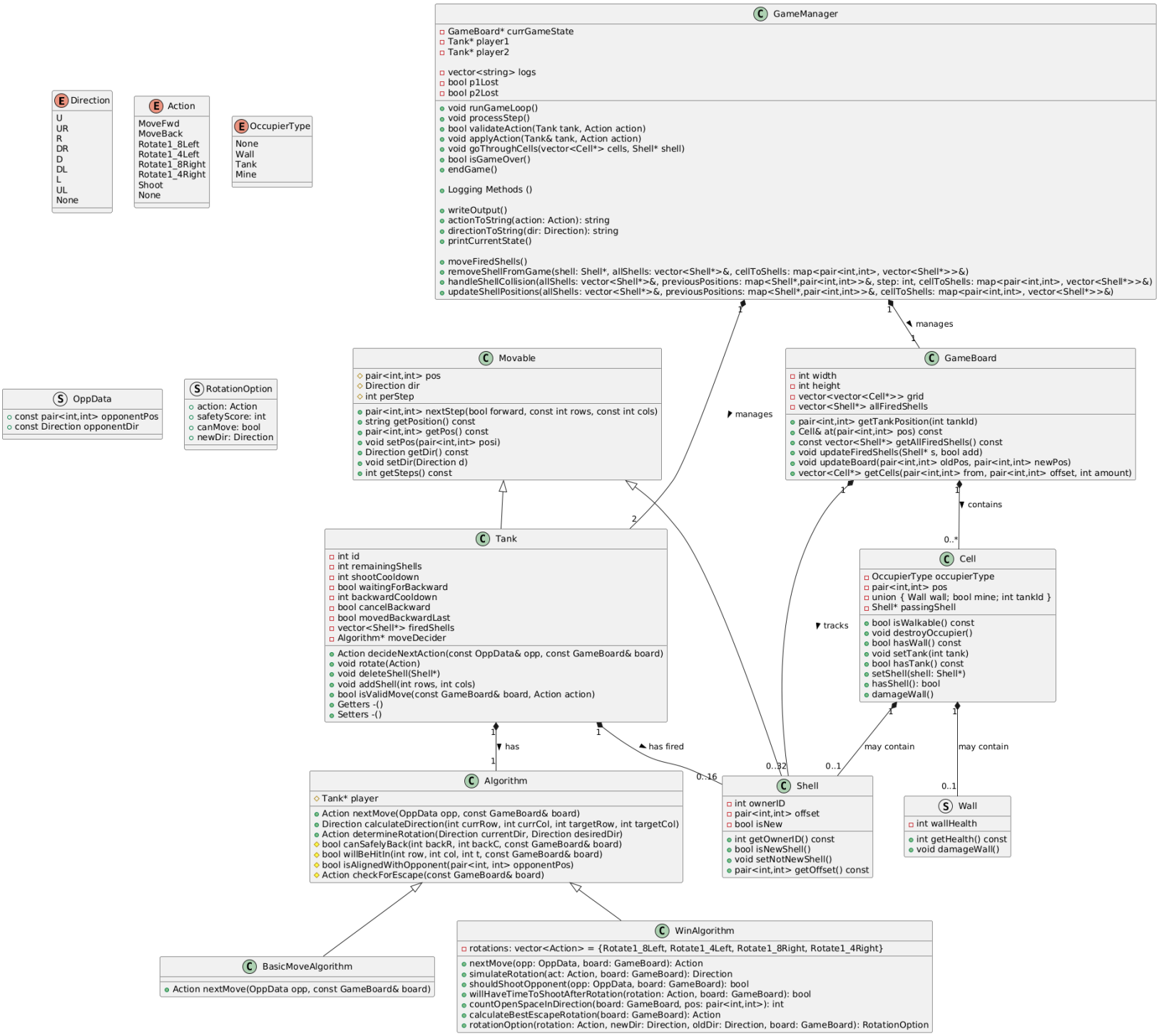
The game ends when one tank remains or after 40 turns where both tanks have no shells left.

Important Remarks:

- **Move Decision -** both tanks decide their next move and only after both have decided, it happens. So, the algorithm decides according to the last step data (Ensures players react to projectile threats before moving (Forces players to anticipate threats before acting).
- **Shell Collision** - If multiple shells land in the same cell, **all** are destroyed and if there's a wall\tank in that cell, they're destroyed as well.

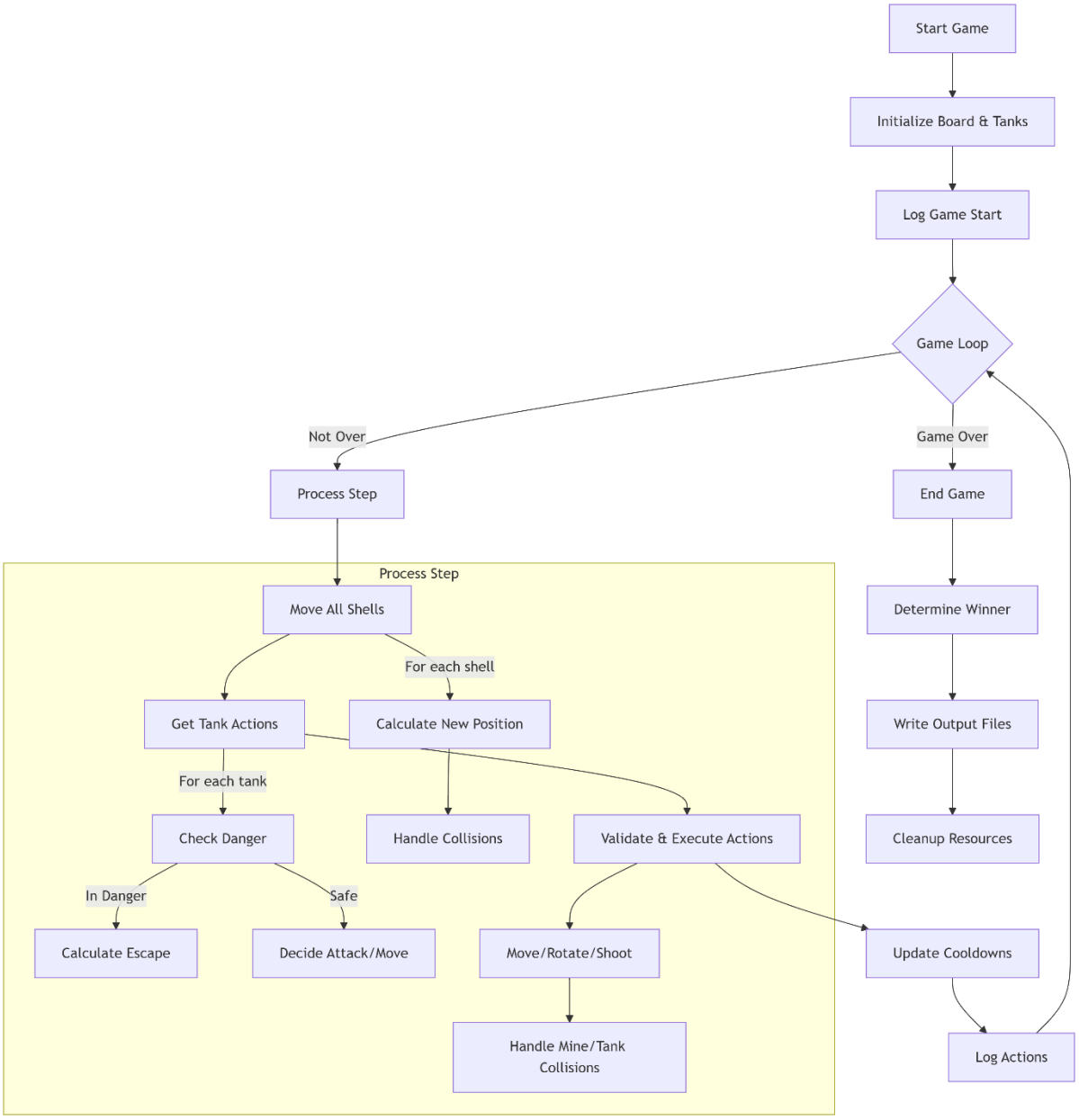
Lior Pernik
Alin Loshevsky

Class UML of the game:



Lior Pernik
Alin Loshevsky

Game flow chart



Lior Pernik
Alin Loshevsky

Design Considerations & Alternatives

Here are our key design choices and potential alternatives:

1. Inheritance vs. Composition

Current Design

- **Inheritance**:
  1. `Movable` is a base class for `Tank` and `Shell` as both are the only movable objects in the game.
  2. `Algorithm` is inherited by `BasicMoveAlgorithm` and `WinAlgorithm` (as they share the same base logic).
- **Composition**:
  1. `Tank` owns an `Algorithm*` for decision-making (decoupling behavior).
  2. `GameBoard` contains `Cell` objects (grid management).
  3. 'GameManager' contains all other classes to access and manage everything.

Alternatives

- **Strategy Pattern (Alternative to Inheritance)**:
  Replace `Algorithm` hierarchy with a `std::function` or interface for runtime strategy swapping (Recommended by ChatGPT)
  Pros: More flexible, avoids subclass explosion.
  Cons: Slightly harder to debug.
- **Entity-Component-System (ECS)**:
  Break `Tank`/`Shell` into components (e.g., `ShootingComponent`).
  Pros: Better for complex entity interactions.
  Cons: Overkill for this scale.

2. Movement & Collision Handling

Current Design

- **Grid-Based Movement**:
  1. `Movable::nextStep()` handles wraparound via `wrap()` for tunnels.
  2. `Cell` build for game map.
- **Shell Movement**:
  1. `GameManager` updates shell positions and resolves collisions in `moveFiredShells()`.

Lior Pernik
Alin Loshevsky

Collision Handling Overview

The code implements a **two-phase collision detection and resolution** system for shells in the tank battle game. Here's how it works:

1. Movement Phase (updateShellPositions**)**

- **Shell Position Updates**:
  - Stores each shell's previous position in previousPositions.
  - Checks for immediate collisions with tanks/walls at the current position.
  - Damages walls if a new shell hits them.
  - Calculates next position using getCells() based on shell direction.
- **Tracking**:
  - Maintains cellToShells map to group shells by their new positions.
  - Logs all shell movements via logShellMove().

2. Collision Detection (handleShellCollisions**)**

Shells move **2 cells per turn. Collisions are checked after each step to ensure realism (mid-movement collisions).**

<u>Key Design Considerations</u>

1. detect head-on collisions by comparing past/current positions.
2. group shells by destination cell, simplifying same-cell collision checks.
3. prioritize shell-shell collisions before environment interactions.
4. Detailed event logging (logShellsCollided, logWallDestroyed, …).

<u>Alternatives</u>

- **Continuous Coordinate**:
  Use floating-point positions for smoother movement.
  Pros: More realistic physics.
  Cons: Complex collision detection.
- QuadTrees for **Collision Detection**:
  Suggested by ChatGPT, claiming it optimizes collision checks for large grids.
  Pros: Faster for many shells.
  Cons: Added complexity.

Lior Pernik
Alin Loshevsky

3. Logging System

Current Design

- **Centralized Logging**:
  1. `GameManager` handles all logging (`logTankAction()`, `logGameOver()`, etc.). Outputs to 'gameLog.txt' file and/or to the console.

Alternatives

- **Observer Pattern**:
  1. Loggers subscribe to game events (e.g. `onTankHit`) \ Class of logger with all functions.
     Pros: Decouples logging from game logic.
     Cons: More repetition in multiple places with little to no variation.

4. Game State Management

Current Design

- **Turn-Based Loop**:
  1. `GameManager::runGameLoop()` processes turns sequentially.
  2. `Tank` actions validated via `validateAction()`.

5. Tank Decision-Making

Current Design

- **Algorithm Hierarchy**:
  1. `BasicMoveAlgorithm` (simple moves) vs. `WinAlgorithm` (advanced tactics).
  2. Uses `OppData` for opponent awareness.

6. Memory Management

Current Design

- **Raw Pointers**:
  1. `GameBoard` owns `Cell` grid; `Tank` owns `Shell` objects.
  2. Manual cleanup in destructors.

Alternatives

- **Smart Pointers**:
  1. Use `std::unique_ptr<Cell>` and `std::shared_ptr<Shell>`.
     Pros: Safer, automatic cleanup.
     Cons: Slight overhead.
- **Object Pooling**:
  1. Reuse `Shell` objects to avoid allocations.
     Pros: Better performance for frequent shoots.
     Cons: prone to bugs

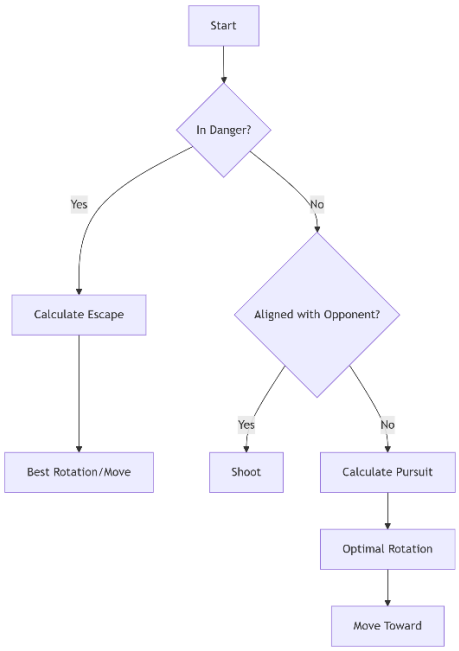
Lior Pernik
Alin Loshevsky

The current design is simple, prioritizing:

- Clear inheritance for shared mechanics.
- Centralized control in `GameManager`.
- Turn-based predictability.

Move algorithms short explanation:

WinAlgorithm:



This algorithm controls a tank with **three key behaviors**:

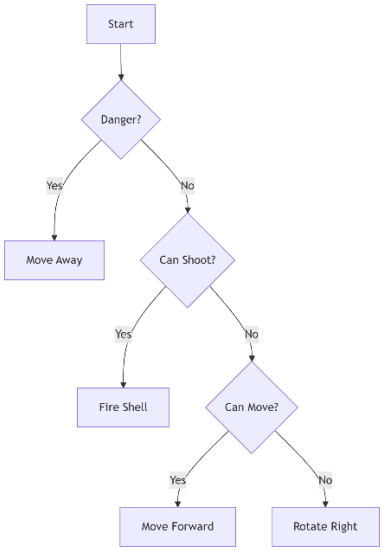
1. **Danger Avoidance**:
   - Uses willBeHitIn() to detect incoming shells.
   - Escapes via calculateBestEscapeRotation() (prioritizes safe rotations) or checkForEscape() (moves forward/backward).
2. **Targeting**:
   - Shoots if aligned with the opponent (shouldShootOpponent() predicts enemy movement based on their current position and direction).
   - Rotates toward the enemy (determineRotation()) when not aligned.
3. **Movement**:
   - Prefers moving forward to get closer to enemy if safe.

Lior Pernik
Alin Loshevsky

Key Considerations

1. **Predictive Targeting**:
   - shouldShootOpponent() predicts enemy positions 1 step ahead, but assumes linear movement.
2. **Safety Scoring**:
   - scores rotations based on:
     - Immediate safety (willBeHitIn() checks).
     - Open space surrounding a given position (countOpenSpaceInDirection()).

BasicMoveAlgorithm:



This algorithm is a simple version of the Win, and it does not chase the opponent, only avoids immediate threats and shoots when aligned.

<u>Why Win is better than the Basic:</u>

1. <u>Low Danger Avoidance:</u>
   - Basic only checks for immediate danger. (A shell 2 cells away will hit the tank next turn), while the win algorithm checks for both immediate and 2 steps ahead of danger to try to escape).
2. <u>Basic has no Strategic Rotation and No Escape Planning.</u>
3. <u>Basic has poor Opponent Targeting:</u>
   - Shoots only if *currently aligned* with the opponent.

Lior Pernik
Alin Loshevsky

BasicMoveAlgorithm Might Occasionally Win in open maps with no walls/mines (where its simplicity isn't penalized). Also due to lack of danger avoidance might shoot himself in long run.

Alternatives For Win:

- **Utility-Based**:
  1. Score actions (e.g., `shootScore = distanceToEnemy * dangerLevel`).
     Pros: Dynamic decision-making.
     Cons: requires more map awareness

Testing Approach:

We did a combination of unit tests, AI-generated scenarios(gameboards), and interactive debugging to validate the tank battle game logic.

- **Unit Testing (Core Logic)**
  Focus: Validate low-level mechanics in isolation.
  Test Cases: Movement & Collision, Wall Damage

- **Scenario Testing (AI-Generated)**
  Focus: Cover edge cases with multiple gameboard options (ChatGPT generated edge-case scenarios) and we tested them.
  We also manually verified those scenarios, debugged failed tests by replaying steps and fixed what we could spot.