**HW2 – 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**.
* \*: Shell moving on board (in bonus basic visual of game steps ‘gameSteps\_output
* \_inputFileName.txt’)

**Turn Sequence**

Shell Movement Phase:

1. All shells move 2 cells per turn
2. Collision detection happens after each movement step

Tank Action Phase

1. Each tank decides action based on its algorithm
2. Actions validated and executed sequentially

After each turn, Destroyed objects removed from board and the round is logged.

GameManager processes the rounds, updates its info about the tanks, and logs all changes, while the movement validation and application of all objects on the map and their collisions are done in BoardManager.

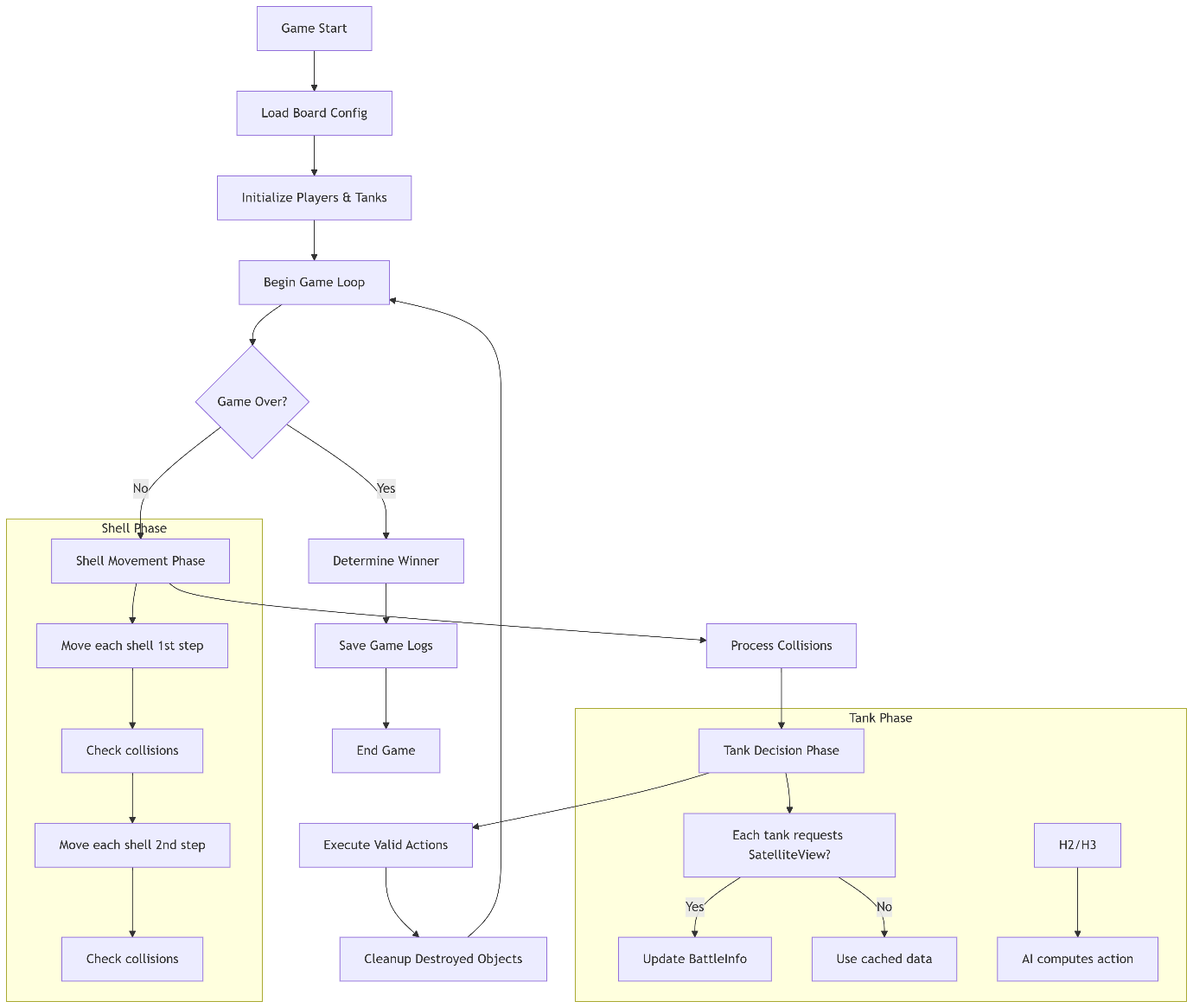
The game ends when only tanks of one player are still on the board, after 40 turns where all tanks have no shells left or after MaxSteps given is reached.

**Important Remarks:**

* **Move Decision -** all tanks decide their next move and only after both have decided, it happens. The actions decided are depending on the algo’s gathered information so far.
* **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.

**Class UML -**

**Game flow Chart –**



**Design Considerations & Alternatives**

Here are our key design choices and potential alternatives:

**1. Architecture:**

**Current Design –**

* GameManager` acts as the orchestrator, responsible for:  
   - Reading the board.  
   - Running the main game loop.  
   - Logging and result reporting.  
   - It owns all major components: (BoardManager, Players, TankAlgorithms,…).
* **Game Map Structure:**

- game\_map is a 3D `vector<vector<vector<unique\_ptr<GameObject>>>>`.- Each `Cell` contains a list of objects (Mine/Wall\Empty + Tank\Shell).

**Alternatives –**

* **Event-Driven Architecture:**1.event queues and listeners.  
  2.Actions like `TankFired`, `TankMoved`, `TankDestroyed` trigger subscribers.  
  **Pros:** More modular, suitable for extensibility.  
  **Cons:** Overhead and unnecessary complexity for current scale.
* **Micro-Service Style Modules:**  
  Separate modules for ShellManager, TankLogic, CollisionHandler.  
  **Pros:** Independent testing, scalability.  
  **Cons:** Far too heavy for this project.
* **Flat Object Registry + Spatial Hashing:**  
  Maintain a global object list + map from coordinates to indices.  
  **Pros:** Better for large maps and performance tuning.  
  **Cons:** Adds complexity without functional gain at small scale.
* **2D Grid with Type Masks:**  
  Use enums or bitmasks to represent presence of different object types.  
  **Pros:** Lightweight and cache-friendly.  
  **Cons:** Loses polymorphism and behavior encapsulation.

**2. Object Management**

**Current Design –**

* Uses `unique\_ptr<GameObject>` in `BoardManager::game\_map` for ownership.
* Movement via moveFireShells, applyMoves, extractObjectFromMap() and updateMap functions in BoardManager.

**Alternatives –**

* **Shared Ownership:**  
  Use shared\_ptr<GameObject> to allow access across multiple modules.  
  **Pros:** Flexible object lifetime.  
  **Cons:** Risks of dangling references, harder to trace lifetime.
* **Object Pooling for Shells:**  
  Reuse `Shell` objects instead of frequent allocations.  
  **Pros:** Improved performance in shell-heavy rounds.  
  **Cons:** Harder to track state resets and ownership.

**3. Movement and Collision Handling**

**Current Design** –

* **Movement steps are grid-based:**  
  - `moveFiredShells()` handles double-step shell movement with collision detection.  
  - `applyMoves()` handles tanks movements per round with collision detection.

- `processCollision()` handles compound interactions in one cell.

* **Delayed Actions (Backward Movement, Shooting Cooldowns):**  
  - Backward move: 2-step cooldown implemented via `waitingForBackward`, `backwardCooldown`, `movedBackwardLast`.  
  - Shooting cooldowns: `isWaitingToShoot` flag and cooldown counter.

**Alternatives** –

* **Continuous Coordinate**:

Use floating-point positions for smoother movement.

**Pros**: More realistic physics.

**Cons**: Complex collision detection.

* **Task Scheduler for delayed actions:**  
  - Each tank has a queue of scheduled delayed actions.  
  **Pros:** Clean handling of complex delays.  
  **Cons:** Adds more complexity, less intuitive for fixed delays.

**4. Tank Decision-Making:**

**Current Design** –

* **Strategy Pattern:**  
  - `TankAlgorithm` is abstract, `MyTankAlgorithm`, `AdvancedTankAlgorithm`, and `BasicTankAlgorithm` implement it.  
  - Tank behavior depends on the algorithm pointer (`MyTankAlgorithmFactory` assigns Basic algo to tanks of Player 2 and Advanced algo to player 1).

**Alternatives –**

* **Behavior Trees or State Machines:**  
  Represent tank logic as transitions between states (like Dodge, Attack, Reload).  
  **Pros:** Visual, modular, expandable.  
  **Cons:** Requires more boilerplate, logic less centralized.
* **Neural Network or Reinforcement Learning:**  
  Use past rounds to learn optimal moves.  
  **Pros:** High adaptability.  
  **Cons:** Overkill for turn-based grid game.

**5. Logging and Output**

**Current Design –**

* GameManager logs all events and generates per-turn summaries.
* BoardManager::printBoard() captures board state snapshots and by the end of the game, we get a simple visual of the game.

**Alternatives –**

* **Observer Pattern:**  
  if there is a need of a more informational logging, have a `Logger` subscribe to events like `TankDestroyed` or `ShellFired` or something else we’d like to log.  
  **Pros:** Separation of concerns.  
  **Cons:** Too many callbacks, not beneficial unless logging logic grows.
* **Structured JSON Logging:**  
  Log moves and collisions in JSON for replay or analysis.  
  **Pros:** Great for visualization tools.  
  **Cons:** More parsing effort, not needed for simple logs.

**6. Memory Safety**

**Current Design –**

* unique\_ptr used extensively (in tanks, shells, etc.).
* No raw pointer ownership (except temporary in maps)

**Alternatives –**

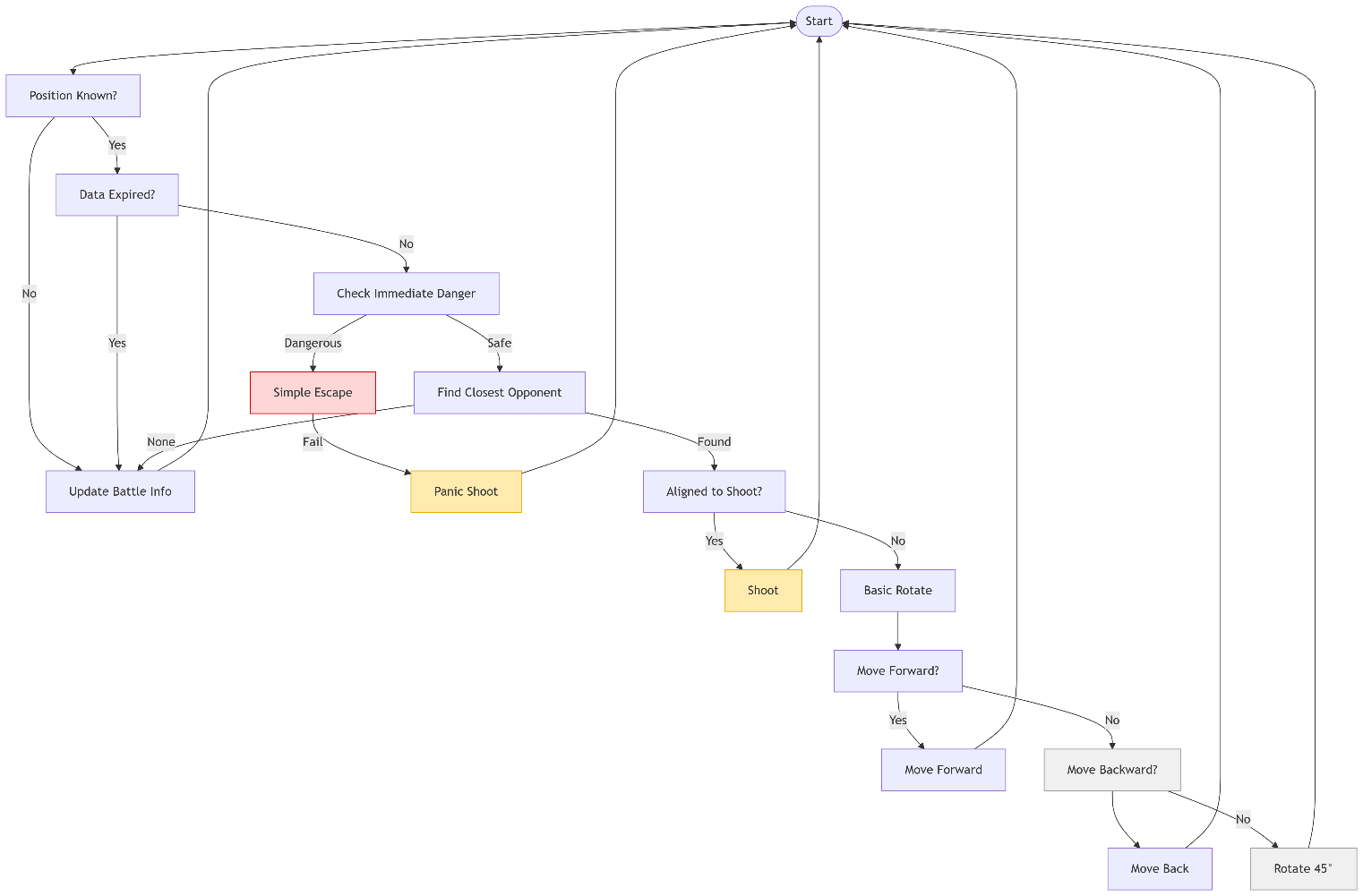
* **Smart Pointer Everywhere:**  
  Use shared\_ptr for shell tracking, for `Tank\*` in maps.  
  **Pros:** Uniform memory management.  
  **Cons:** Possible cyclic references or overhead.

**Algorithm explanation Basic vs Advanced**

**BasicTankAlgorithm –**

This algorithm has a simple reactive behavior:

* + Checks immediate danger (1 step ahead)
  + Shoots when aligned with opponent
  + Rotates toward opponent when not aligned
  + Moves forward when path is clear
  + Defaults to rotating right when stuck

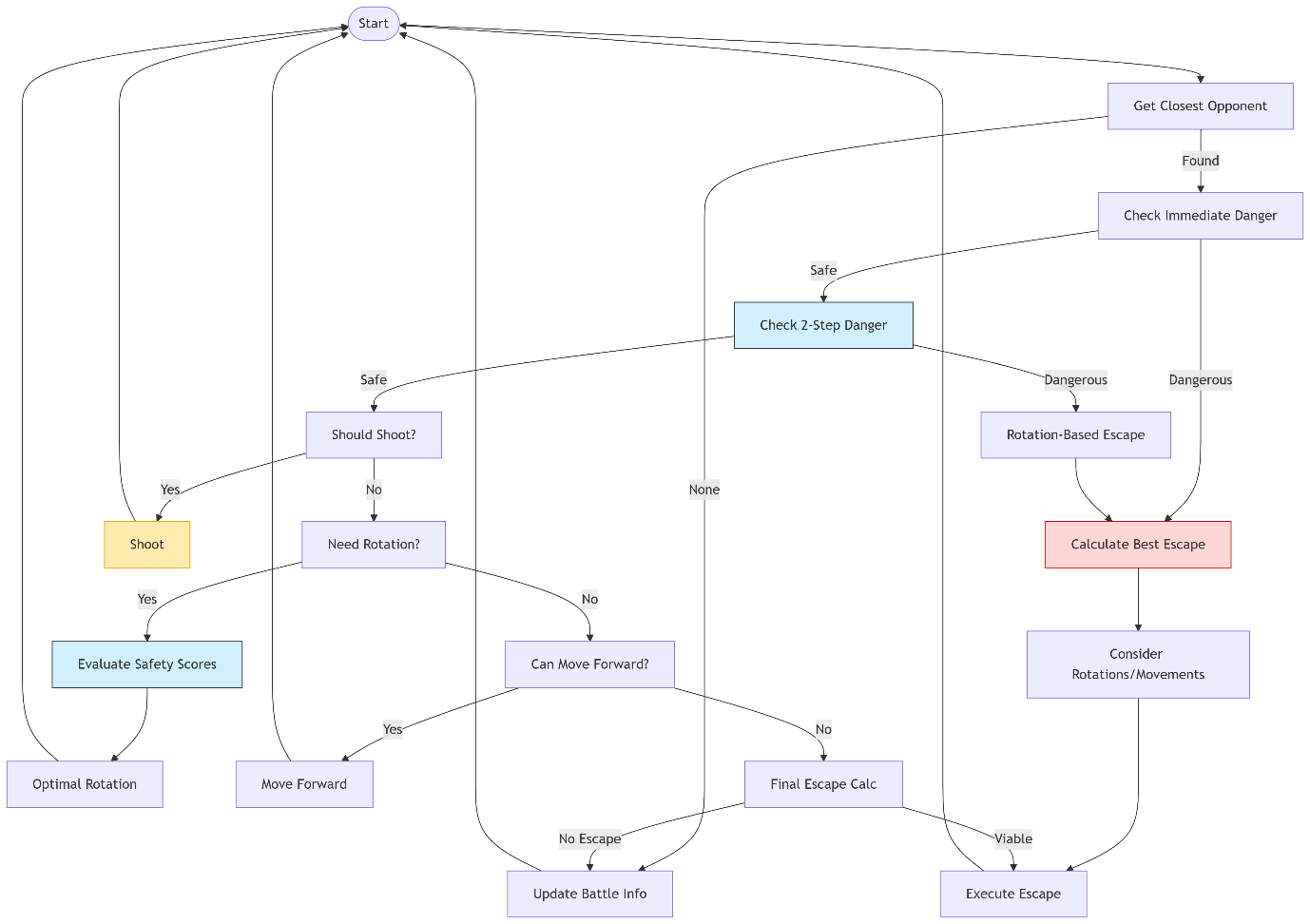
**Flow chart –**

**AdvancedTankAlgorithm –**

This algorithm has a more sophisticated behavior:

* + Multi-step danger assessment (1 and 2 steps ahead)
  + Predictive targeting ?
  + Safety scoring system for rotations
  + Active shell tracking and prediction
  + Space-aware movement decisions (checkForEscape() func)
  + Comprehensive battle info analysis

**Flow chart –**



**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.