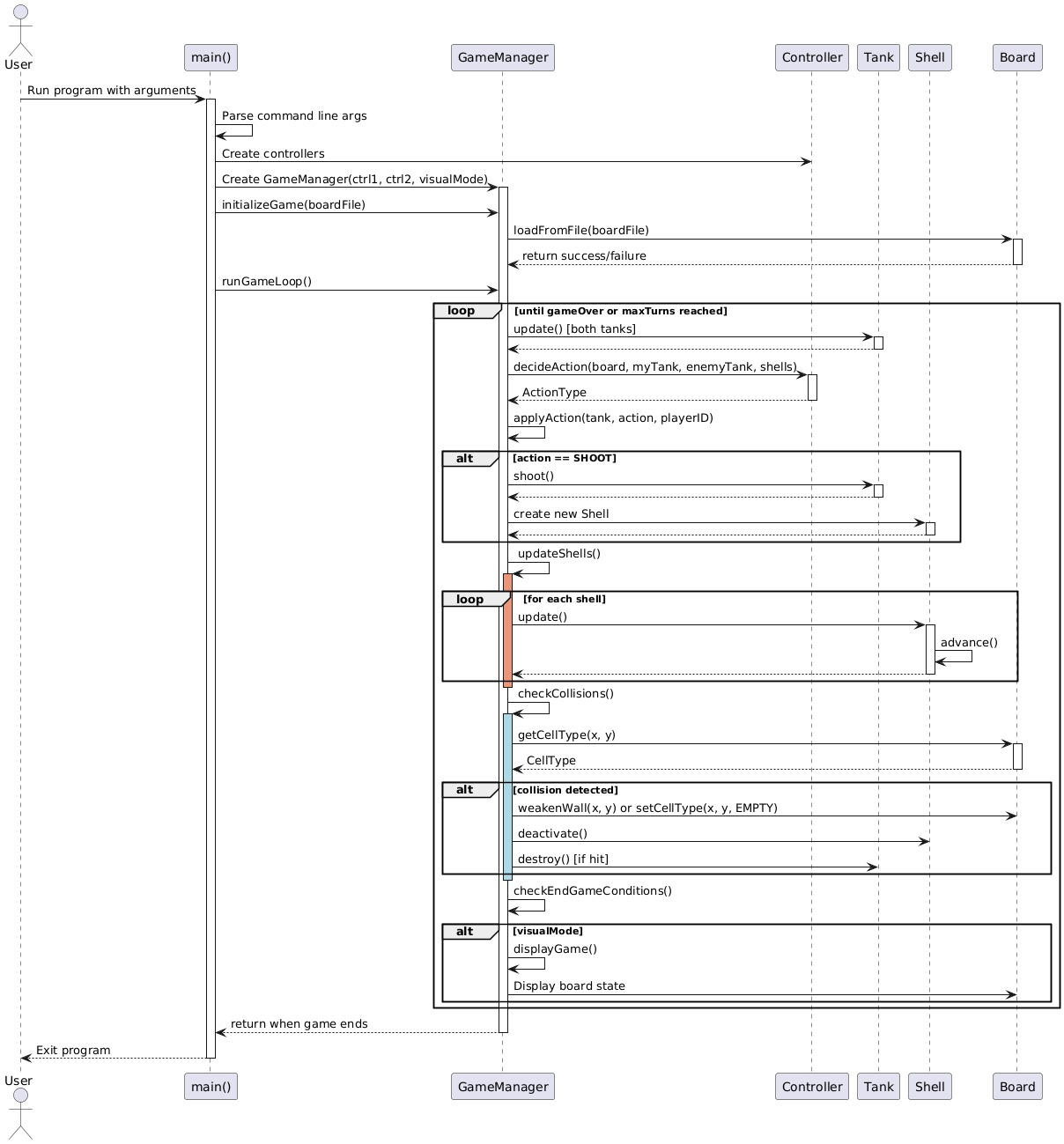
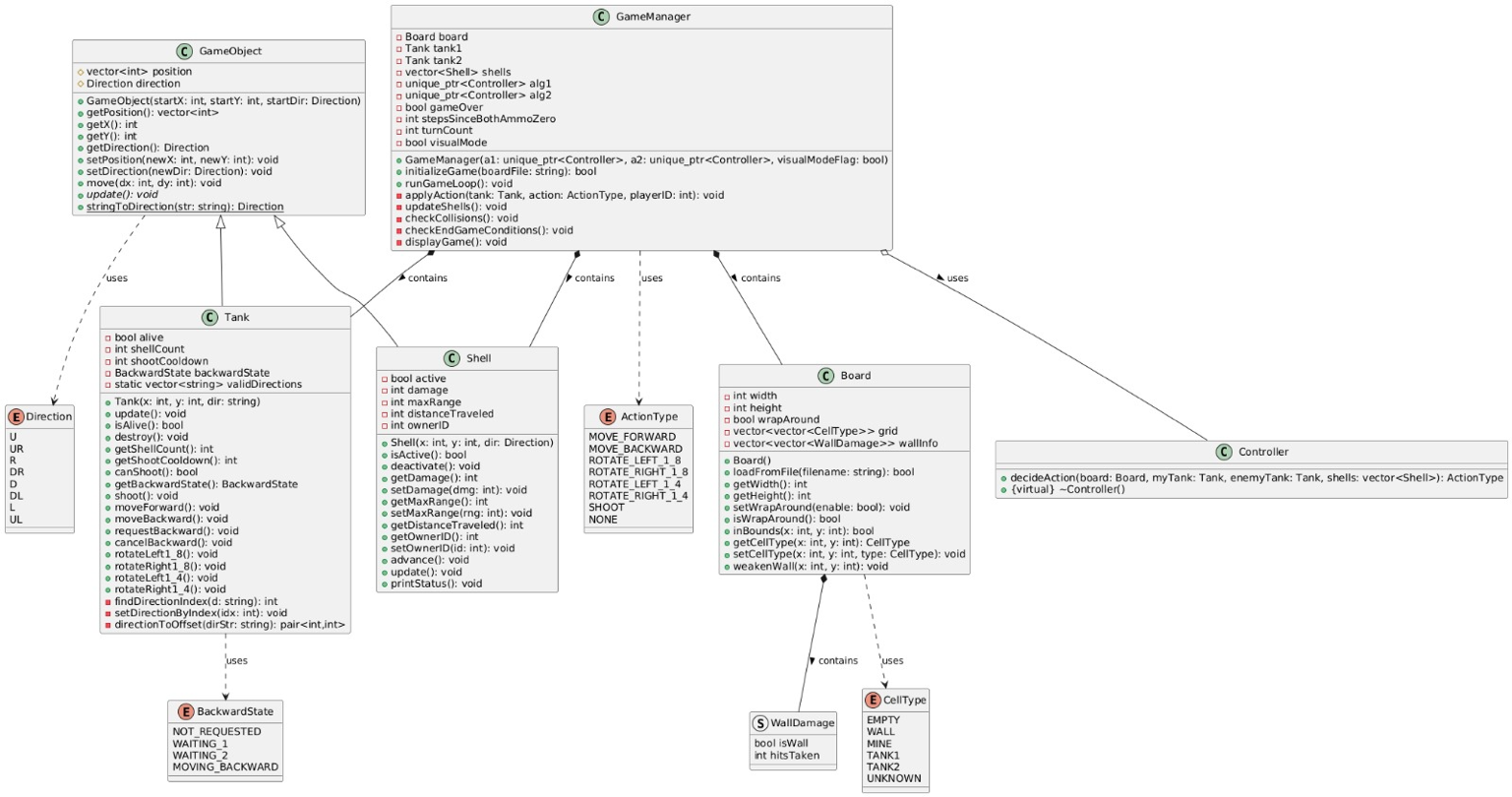
Design Documentation – Project 1

Dor Liberman & Shachar Gabbay





Object-Oriented Hierarchy

Current Design: A GameObject base class with Tank and Shell as derived classes. This establishes a clear hierarchy and allows for polymorphic handling of game entities.

Alternatives Considered:

- Component-Based System: Instead of inheritance, objects could be composed of components (movement, damage, rendering). This would provide more flexibility but increase implementation complexity.

- Entity-Component-System (ECS): A more modern approach that separates data (entities, components) from systems that operate on them. This would be more scalable but overkill for this project's scope.

- Interface-Based Design: Using interfaces (pure abstract classes) rather than concrete base classes. This would allow more flexible implementation but add complexity.

Rationale: The inheritance-based approach was chosen for its simplicity and direct mapping to the problem domain. The game has a clear hierarchy of objects with specialized behaviors, making inheritance appropriate. The shared functionality in GameObject provides a clean way to handle common attributes like position and direction.

Movement and Rotation System

Current Design: Direction-based movement with 8-way rotation (45-degree increments). Tanks can rotate by 1/8th or 1/4th, and move forward or backward.

Alternatives Considered:

- Vector-Based Movement: Using velocity vectors instead of discrete directions. This would allow for smoother movement but complicate the turn-based nature.

- Grid-Based Movement: Restricting movement to the four cardinal directions. Simpler, but less engaging gameplay.

- Continuous Rotation: Allow rotation by arbitrary angles. Would provide more precision but complicate the direction representation and user input.

- Physics-Based Movement: Including inertia, acceleration, and friction. Would create more realistic movement but significantly increase complexity.

Rationale: The 8-direction system balances gameplay complexity with implementation simplicity. It provides sufficient strategic depth while remaining approachable for players and straightforward to implement. The discrete rotation steps create interesting tactical decisions while keeping the action space manageable.

Collision Detection

Current Design: Cell-based collision detection. Objects in the same cell are considered colliding, and appropriate game logic handles the outcome (destruction, damage, etc.).

Alternatives Considered:

- Geometric Collision: Checking for actual intersection of object shapes. More accurate but unnecessary for this grid-based game.

- Predictive Collision: Detecting collisions before they occur to prevent them. Would require more complex movement planning.

- Ray Casting: For shell trajectory collision detection. Would allow for more realistic shell behavior but significantly increase complexity.

- Continuous Collision Detection: Checking for collisions at sub-step intervals. More accurate but computationally expensive.

Rationale: Cell-based collision is appropriate for a grid-based game. It's simple to implement and understand, and matches the game's turn-based nature. The system also naturally integrates with the board representation, making it efficient and easy to debug.

Game State Management

Current Design: A centralized GameManager controls the game loop and manages interactions between objects.

Alternatives Considered:

- Event-Based System: Objects communicate through events rather than direct method calls. More decoupled but harder to reason about.

- State Machine: Formalize game states (setup, player1Turn, player2Turn, gameOver) for clearer transitions. Useful for more complex games.

- Command Pattern: Encapsulate actions as command objects for easier undo/replay functionality.

- Model-View-Controller (MVC): Separate the game state, display, and control logic. Would improve maintainability but add structural complexity.

Rationale: The centralized manager provides clear control flow and makes the game loop explicit. For a relatively simple game like this, it offers a good balance of organization without over-engineering. The GameManager's responsibilities are well-defined, making the code easier to understand and modify.

Visual Representation System

Current Design: Optional visual mode using emoji characters to represent game elements in the terminal, with a displayMap function in utils.cpp.

Alternatives Considered:

- Graphical UI Library: Using a proper graphics library like SFML or SDL for rendering. Would provide better visuals but increase complexity and dependencies.

- Separate UI Layer: Completely separating game logic from visualization using the Observer pattern. More maintainable but overkill for the current scope.

- ASCII Art Only: Using only ASCII characters without emoji. More compatible across platforms but less visually appealing.

- Web-Based Interface: Creating a web frontend to visualize the game. Would enable remote play but significantly increase development effort.

Rationale: The emoji-based terminal visualization provides a good balance between visual clarity and implementation simplicity. The --visual flag makes it optional, allowing for both human-readable output and faster execution when not needed. This approach keeps dependencies minimal while still providing useful visual feedback during development and gameplay.

Controller Design

Current Design: A Controller interface with a decideAction method that can be implemented by different algorithms.

Alternatives Considered:

- Rule-Based AI: Hard-coded rules for specific situations. Simple but limited in adaptability.

- State-Based AI: Different behaviors for different game states. More complex but potentially more effective.

- Machine Learning Approaches: Using reinforcement learning or other ML techniques. Potentially more powerful but much more complex to implement and train.

- Human Input Controller: Allow human players to control tanks directly. Would enable multiplayer but require input handling code.

Rationale: The Controller interface provides a clean separation between the game mechanics and decision-making. This makes it easy to swap different implementations and compare their performance. The current dummy implementation can be gradually replaced with more sophisticated strategies without changing the core game logic.

Error Handling and Robustness

Current Design: Basic error checking with console output for failures and some defensive programming.

Alternatives Considered:

- Exception-Based Error Handling: Using C++ exceptions for error conditions. More structured but potentially affects performance.

- Result Objects: Returning result objects that contain success/failure information. More explicit but verbose.

- Assertion-Based Development: Using assertions for invariants and preconditions. Good for development but not suitable for production code.

- Logging Framework: Integrating a proper logging framework. More powerful but adds dependencies.

Rationale: The current approach balances simplicity with adequate error detection. For a project of this size, simple error messages to the console are sufficient for debugging and user feedback. The code includes validation for critical operations like board loading and bounds checking, which prevents most potential runtime errors.

Testing Approach

Unit Testing

- Board Logic: Test board loading, cell manipulation, and wall damage tracking.

- Tank Movement: Verify proper movement, rotation, and backward movement state transitions.

- Shell Behavior: Ensure shells follow proper trajectories and deactivate under appropriate conditions.

Integration Testing

- Collision Handling: Test various collision scenarios (tank-shell, shell-shell, tank-mine, etc.).

- Game Flow: Verify the entire game loop executes as expected with mock controllers.

System Testing

- End-to-End Scenarios: Run complete games with different board configurations and verify outcomes.

- Edge Cases: Test boundary conditions like board edges, simultaneous destructions, and full ammo depletion.

Manual Testing

- Visual Verification: Use the visual representation mode to manually verify gameplay behavior.

- Input Validation: Test different board files, including malformed ones, to verify robust error handling.

Testing Tools and Methods

- Custom Test Boards: Create specific board layouts to test particular scenarios.

- Automated Testing Framework: Implement simple test harnesses to execute and validate test cases.

- Logging and Instrumentation: Add logging to track key events and state changes during testing.

Specific Test Cases

Tank Movement Tests:

- Moving into empty spaces

- Attempting to move into walls

- Backward movement delay mechanism

- Rotation at board edges

Combat Tests:

- Direct hits between tanks

- Wall destruction after multiple hits

- Mine triggering

- Multiple simultaneous collisions

End Game Condition Tests:

- One tank destroyed

- Both tanks destroyed simultaneously

- Both tanks out of ammo (40-step countdown)

- Maximum turn limit reached

Visual Presentation Bonus Feature Implementation

The visual presentation mode was implemented as a bonus feature for the Tank Battle Game. This feature provides an intuitive visual representation of the game state using emojis, making it easier to understand the game's dynamics and test the implementation.

Implementation details:

1. Command-line Integration:

- Added "--visual" flag support in main.cpp

- The flag is parsed and passed to the GameManager constructor

- Completely optional: the game runs perfectly in text mode without visualization

2. Enhanced Display:

- Created a specialized display function that uses emojis to represent game elements

- Implemented in utils.cpp with the displayMap function

- Used throughout the GameManager displayGame method

3. Visual Elements:

- Tank 1: 🚗

- Tank 2: 🚙

- Wall: 🟩

- Mine/Shell: 💥

- Empty space: ⬜

4. Display Logic:

- The board state is converted to a 2D character grid

- Game objects (tanks, shells) are overlaid on this grid

- In visual mode, each turn's state is displayed with a small delay (300ms)

- The final game state is always displayed regardless of mode

5. Testing Benefits:

- Allowed real-time observation of game dynamics

- Made collision detection and resolution issues more obvious

- Provided intuitive feedback on movement and firing mechanics

- Helped verify win/loss conditions visually

6. Implementation Challenges:

- Ensuring proper terminal emoji support across different platforms

- Managing the display rate to balance between readability and game speed

- Coordinating the visual representation with the underlying game state

7. How to Check:

- Run the game with the --visual flag: ./tank\_game resources/exampleBoard.txt --visual

- Or use the provided Makefile target: make run\_visual ARGS=resources/exampleBoard.txt

- The game will display the board state using emojis after each turn

This visual mode significantly enhanced both the user experience and our ability to debug complex game interactions, while keeping the core game logic separate and maintainable.