OOP Implementation Guide: Tank Warfare

Project Overview

Tank Warfare leverages Object-Oriented Programming (OOP) concepts to manage game state, handle player data, and structure the backend logic. The system is divided into a JavaScript frontend (Client-side) and a PHP backend (Server-side), both utilizing OOP principles to ensure code modularity, reusability, and maintainability.

1. Encapsulation

Definition: Encapsulation is the bundling of data (attributes) and methods (functions) that operate on that data into a single unit, restricting direct access to some of an object's components.

Implementation in Tank Warfare:

- **Game State Management (JavaScript):** In `game.js`, the entire state of the application is encapsulated within the `gameState` object. This object acts as a container that holds specific entities like `players`, `bullets`, `walls`, and `scores`. Instead of having loose variables floating globally, every piece of data belongs to a specific parent object.
    - *Example:* The `player` objects encapsulate specific attributes such as coordinates (`x`, `y`), `ammo` count, `alive` status, and `direction`.
- **Data Handling (PHP):** In `backend.php`, the logic for file manipulation is encapsulated within specific functions like `saveGameResult()` and `updatePlayerStats()`. These functions protect the integrity of the `game_data.json` file by handling the read/write operations internally, ensuring that external parts of the application cannot corrupt the data structure directly.

2. Abstraction

Definition: Abstraction involves hiding complex implementation details and showing only the necessary features of an object or function to the outside world.

Implementation in Tank Warfare:

- **Maze Generation:** The complex logic of the Depth-First Search (DFS) algorithm used to create the map is hidden behind the `generateMazeWalls()` function. The main game

loop simply calls this function to receive a map, without needing to understand the mathematical complexity of cell visitation and wall removal.
- **Collision Detection:** The physics calculations for overlapping hitboxes are abstracted into the `checkCollision()` and `calculateCollisionNormal()` functions. The update loop simply asks "did these collide?" and receives a boolean result, keeping the high-level game logic clean and readable.
- **API Interface:** The PHP backend serves as an abstraction layer for the database (JSON file). The frontend simply requests actions like `get_records` or `save_game` without needing to know how the server stores, sorts, or limits the data to the last 10 entries.

3. Inheritance

Definition: Inheritance is the mechanism where a new class or object derives properties and characteristics from an existing parent class or object.

Implementation in Tank Warfare:

- **Bullet Instantiation (Prototypal Inheritance):** The game utilizes a "Base Bullet" template. When `shoot()` is called, new bullet instances are created that inherit the standard properties of a projectile (speed, size, direction).
- **Power-up Variations:** Special bullets (Split, Big, Penetrate) inherit the base properties of a standard bullet but extend functionality by modifying specific attributes (such as `size` or `penetrationCount`) or behaviors (such as the `SPLIT` type creating multiple projectiles). This allows the game to reuse the core movement logic for all bullet types while maintaining unique characteristics for "child" types.

4. Polymorphism

Definition: Polymorphism allows objects to be treated as instances of their parent class, enabling a single interface to control access to a general class of actions.

Implementation in Tank Warfare:

- **Dynamic Shooting Behavior:** The `shoot()` function exhibits polymorphism. Depending on the `powerup` state of the player object, the single "Shoot" command takes on different forms. It may fire a single projectile, a spread of three projectiles (Split), or a high-velocity projectile (Big). The input (pressing Spacebar) remains the same, but the output morphs based on the object's current state.
- **Entity Rendering:** The `renderGame()` loop treats various game entities polymorphically. Whether an object is a destructive bullet, a solid wall, or a pickup item, the render loop processes them within their respective arrays, drawing them to the canvas based on their internal properties (color, coordinates) without needing separate render loops for every single variation of an item.