Neon Space Shooter

Technical Design Document:

1. OOP Implementation Guide

The application logic is structured primarily within game.js, utilizing **Object-Oriented Programming (OOP)** to manage game state, entities, and behaviors. Below is a detailed breakdown of how the four pillars of OOP are applied.

A. Encapsulation

Definition: The bundling of data and methods that operate on that data within a single unit (Class), protecting internal state from interference.

Implementation Details: The **Player** class in game.js is the primary example of encapsulation. It holds all state variables specific to a player instance and exposes methods to manipulate them.

- **State Protection:** The player's position (x, y), health (lives), and status (isDead) are properties of the class instance. External functions do not manually adjust the player's coordinate variables. Instead, they call specific methods.

- **Controlled Access:**

Movement: The update(dt) method checks the keys object and modifies this.x and this.y. Crucially, it enforces boundaries (this.minX, this.maxX) internally. This prevents the main game loop from accidentally placing Player 1 into Player 2's lane.

Health Management: Damage logic is encapsulated in takeDamage(sourceType). This method handles shield checks, life deduction, explosion spawning, and respawn logic internally. The main loop simply notifies the player object that it has been hit; the object decides how to react.

B. Inheritance

Definition: A mechanism where a new class derives properties and behavior from an existing class.

Implementation Details: While the current version of game.js uses standalone classes for simplicity, the **Conceptual Inheritance** (or "Duck Typing") is evident in the entity structure.

Shared Interface: The Player, Enemy, Bullet, PowerUp, and Particle classes all implement a shared interface consisting of:

- constructor(): Initializes position and velocity.
- update(): Calculates the next frame's position.
- draw(): Renders the entity to the canvas context (ctx).

"Garbage" Inheritance: The Enemy class demonstrates behavioral inheritance via flags. Standard enemies and "Garbage" enemies share the same class structure, but the isGarbage flag modifies their HP, speed, and size in the constructor. This allows a "Garbage" enemy to inherit all movement and collision logic of a standard enemy while behaving like a "boss" variant.

C. Polymorphism

Definition: The ability of different objects to respond to the same method call in a way specific to their type.

Implementation Details: The Game Loop relies heavily on polymorphism to render the frame without needing to know the specifics of every object.

The Update Loop: In the `gameLoop()` function, the code iterates through different arrays:

Polymorphic Behavior:

- When `.update()` is called on a **Player**, it checks keyboard input (`keys['KeyA']`) and moves laterally.
- When `.update()` is called on a **Bullet**, it modifies `this.y` by `this.vy` (moving upward).
- When `.update()` is called on a **Particle**, it applies simple physics and reduces opacity (`this.life`). The main loop treats them all as "updatable entities," simplifying the code structure.

D. Abstraction

Definition: Hiding complex implementation details and showing only the necessary features of an object.

Implementation Details: Abstraction is used to separate the high-level game rules from low-level mathematics.

- **Collision Detection:** The function checkHit(rect1, rect2) abstracts the mathematical logic of AABB (Axis-Aligned Bounding Box) collision. The rest of the code simply asks "Did these hit?" without needing to know the formula r1x < rect2.x + rect2.width.
- **Spawning Logic:** The sendGarbage(targetIsP2) function abstracts the complexity of creating a new enemy. The caller doesn't need to know the enemy's spawn coordinates, speed, or color; it simply requests that garbage be sent.

2. Architecture Diagram & Flow

The application follows a **Loop-Based Architecture** typical of real-time simulations.

System Architecture Flow

1. **Initialization:** startGame() initializes Player objects, resets timers, and creates empty arrays for entity pooling.
2. **Input Handling:** Event listeners on the window object capture keydown and keyup events, storing the state in a keys object.
3. **The Game Loop (requestAnimationFrame):**

Delta Time Calculation: Calculates time passed since the last frame to ensure smooth movement.

Logic Update (The "Model"):

Player Input: Players update positions based on the keys object.

AI/Physics: Enemies move down; bullets move up.

Collision System: Checks overlap between Bullets/Enemies and Players/Enemies.

Game State: Checks win/loss conditions and updates the "Garbage Meter."

Render Pass (The "View"):

- Clears the canvas.
- Draws the background and lane divider.
- Iterates through all entity arrays calling .draw().

UI Update: Updates the HTML DOM elements (span tags) with current scores and lives.