ChatGPT

Refactored Architecture for Star Throne

To modernize **Star Throne** for modularity, maintainability, and multiplayer-readiness, we reorganize the code into clear subsystems with well-defined interfaces. The combat logic is centralized, AI decision-making is encapsulated in strategy classes, game state changes are emitted via events (decoupling logic from UI), and performance is tuned for low-end devices (e.g. Chromebooks). The following sections outline each architectural shift, proposed file/module names, interfaces, and key refactorings.

Combat System Modularization

- **Centralize combat logic in** `CombatSystem`. All battle resolution, army transfer, and throne-star rules move into a dedicated `CombatSystem.ts/js`. This class takes responsibility for validating attacks, calculating combat outcomes, handling throne-star captures, and updating player/territory state. For example, instead of each `Player` or `StarThrone` object performing its own battle math, they delegate to `game.combatSystem`. This follows the **Single Responsibility Principle**, isolating combat mechanics from player/AI logic.

- **Unified attack interface**. Introduce a single method, e.g. `combatSystem.attackTerritory(fromTerritory, toTerritory)`, used by both human and AI turns. In the player and AI code, replace inline battle calculations with a call to this method. For instance, in `Player.js` we remove the old attack math and use:

```
- // Battle calculation (old inline logic)
- const attackPower = attackingArmies * (0.8 + Math.random() * 0.4);
- const defensePower = defendingArmies * (1.0 + Math.random() * 0.2);
- if (attackPower > defensePower) { ... }
+ // Delegate to centralized CombatSystem
+ const game = gameMap.game;
+ if (game && game.combatSystem) {
+     game.combatSystem.attackTerritory(attackingTerritory,
defendingTerritory);
+ }
```

  This ensures consistent resolution for all attacks. Duplicate code in `StarThrone.js` (e.g. `attackTerritory` and `attackTerritoryWithAmount` methods) should similarly call `CombatSystem` instead of re-implementing combat logic.

- **Throne-star handling in** `CombatSystem`. The tricky throne-star capture logic (transferring entire empires, ending the game) is centralized in `CombatSystem.handleThroneStarCapture()`. The old approach of checking `oldOwner.id === 0` or assuming human ID 0 is removed. Instead, we use the `Player.type` property to identify human players (see Multiplayer section). The sample diff

below shows replacing ID-based checks:

```
// Old throne-detection in Player.js (legacy)
- const isHumanTarget = oldOwnerId === 0;  // assume human player ID = 0
- if (isHumanTarget) { /* end game */ }
+ // New: use player.type
+ if (oldOwner && oldOwner.type === 'human') {
+     // end the game via game event or UI state
+ }
```

- **New `CombatSystem.js` interface example**:

```
export class CombatSystem {
    constructor(game: StarThrone) { this.game = game; }
    // Resolves an attack from one territory to another.
    attackTerritory(from: Territory, to: Territory): boolean { ... }
    // (Optionally) specialized methods for AI attacks or custom modes.
}
```

By moving all battle math and effects into `CombatSystem`, the code is easier to test, tweak, and reuse. UI code simply triggers animations or sounds in response to events, but the outcomes (who wins, armies lost, territory ownership changes) are handled in this module.

AI Logic Refactor

- **Extract AI decision logic into `AIStrategyEngine`**. The `Player` class should no longer contain complex strategy code (e.g. `executeAggressiveStrategy`, `executeDefensiveStrategy`, etc.). Instead, create a new module (e.g. `AIStrategyEngine.ts`) that implements strategy **classes or functions** for each AI state. For example, define strategies like `EarlyGameExpansionStrategy`, `DefensivePostureStrategy`, or generic `AIStrategy` functions. Each strategy encapsulates one behavior pattern.

- **State/Strategy design**. We can apply the **State/Strategy pattern**: define an interface (or base class) for AI strategies, and implement each strategy in its own class. The `Player` holds a reference to its current strategy object. On each AI turn, the player calls `this.strategy.makeDecision(this, gameMap)`, letting the strategy object handle the details. This avoids large switch-statements in `Player.updateAI()`. As Game Programming Patterns suggests, “for each state, we define a class that implements the interface; its methods define the object’s behavior in that state” [1]. Then the main object (the Player) simply delegates to it [2]. For example:

```
interface AIStrategy {
  execute(player: Player, gameMap: GameMap): void;
}
```

```
class AggressiveStrategy implements AIStrategy { /* ... */ }
class DefensiveStrategy implements AIStrategy { /* ... */ }
// In Player.updateAI():
this.strategy.execute(this, gameMap);
```

Each strategy class can hold its own state/data (like attack counters) and update it, keeping AI logic modular and extensible.

- **AI engine usage**. In practice, `AIStrategyEngine` might export a factory or mapper, e.g.:

```
const strategies = {
    EARLY_GAME_EXPANSION: new EarlyGameExpansionStrategy(),
    DEFENSIVE_POSTURING: new DefensivePostureStrategy(),
    // ...
};
```

And `Player` simply picks one on creation or state change. This makes adding new behaviors or tuning existing ones straightforward without touching core `Player` code.

Multiplayer-Ready Architecture

- **Decouple game logic from UI rendering**. We follow MVC-like separation so the game state can run independently of any particular renderer. In practice, we organize code so that `StarThroneState` (or similar) handles all game rules, while **renderer/UI modules** handle display. As one experienced developer put it, "your game's model should be able to run completely independently of the renderer" [3]. In other words, rendering code (e.g. canvas drawing, React components) should *read* game state but not directly modify it. This allows swapping UIs or running a headless server with minimal changes. For example, input handlers would call into the game state (via commands or methods), and UI classes simply subscribe to state changes.

- **Event-driven updates (Observer pattern)**. Introduce a simple **event bus or observer system** for game-state changes. Key state changes (territory ownership changes, throne capture, player elimination, game over, etc.) are emitted as events. UI or other subsystems subscribe to these events to update the display or trigger animations. For instance, we can define a `GameEventBus` interface:

```
// Example in GameEventBus.ts
export type GameEvent =
  | { type: 'TERRITORY_CAPTURED'; from: Territory; to: Territory; by:
Player }
  | { type: 'THRONE_CAPTURED'; by: Player; of: Player }
  | { type: 'PLAYER_ELIMINATED'; player: Player }
  // ... other events
  ;
```

```ts
export interface GameEventBus {
  subscribe(listener: (event: GameEvent) => void): void;
  emit(event: GameEvent): void;
}
```

When, say, a throne star is captured, `CombatSystem` would emit a `{ type: 'THRONE_CAPTURED', by: attacker, of: defender }` event. The UI layer can listen for this to show messages or trigger animations. This **loosely couples** game logic to UI: the logic only broadcasts events, and any number of listeners (renderers, logs, network sync) can react.

- **Support multiple human players**. Replace hard-coded assumptions like "human player has ID 0" with a `player.type` check. For example:

```
- if (oldOwnerId === 0 || (oldOwner && oldOwner.type === 'human')) {
+ if (oldOwner && oldOwner.type === 'human') {
    // ... handle game end for a human player
}
```

This way, any player with `type === 'human'` is treated the same, allowing multiple humans in future multiplayer matches. The game flow (turns, inputs) can then distinguish human vs AI by checking `player.type` rather than a special ID.

- **Server-client separation**. (If extending to multiplayer) move toward a clear client-server architecture. The server holds authoritative `StarThroneState` and processes commands (e.g. an "attack" command), then emits updates to clients. The client's UI simply visualizes state. This is more of a structural goal beyond current refactoring, but designing with event-driven state sets the foundation.

Performance Optimization for Chromebooks

- **Throttled updates**. Avoid heavy per-frame computations. For example, instead of checking all supply routes or running a full AI decision scan every single frame, do these less frequently (e.g. once per second or on demand). Use timestamps or counters to schedule expensive tasks (AI thinking, route recalculation, etc.) at a slower interval. This drastically cuts CPU load on slower hardware.

- **Object pooling**. Reuse frequently created objects (like ship animations, probe objects, particle effects) from fixed pools. This minimizes garbage collection pressure and allocation overhead. The existing `shipAnimationPool` in `StarThrone` is a good start. More pooling can be applied to probes, floating text, or explosion effects. As *Game Programming Patterns* notes, an object pool "improves performance and memory use by reusing objects from a fixed pool" [4], which prevents memory fragmentation and GC spikes on devices with limited resources. (Chromebooks and mobile browsers are particularly sensitive to such GC hiccups.)

- **Lean data structures**. Review large in-memory structures (e.g. arrays of all territories, players) to ensure efficient iteration. For example, keep lists of **active** entities or visible territories, rather than scanning every territory each frame. Use simple arrays or typed arrays where possible, and avoid growing arrays each frame. Profile memory usage to eliminate leaks or redundant copies.

- **Minimize drawing workload**. Only redraw what's necessary. The renderer already tracks `visibleTerritories`; ensure occlusion culling and LOD so Chromebooks aren't overwhelmed by draw calls. Throttle FPS in config if needed.

Code Structure and Maintainability

- **Centralized constants (`gameConstants.ts`)**. All magic numbers and tunable parameters (attack ratios, spawn rates, map sizes, etc.) live in a shared `gameConstants.ts` or `.js`. This file (already present as shared code) holds immutable config and can be imported by all modules. It ensures balance tweaks are in one place.

- **TypeScript interfaces/types**. Use TypeScript (or JSDoc) to define clear interfaces for core entities, e.g.:

```
interface Player {
  id: number;
  name: string;
  color: string;
  type: 'human' | 'ai';
  territories: number[];
  // ... other stats fields
}
interface Territory {
  id: number;
  ownerId: number | null;
  armySize: number;
  neighbors: number[];
  isColonizable: boolean;
  isThronestar: boolean;
  // ... position, colors, etc.
}
interface Probe { /* ... */ }
```

  Placing these in a `types/` or shared module ensures consistency between frontend and backend code and aids maintainability.

- **Modular file layout**. Split large files into focused modules. For example:

- `StarThroneState.ts`: Contains core game state management (initialization, turn order, timers, applying commands, invoking CombatSystem, etc.). No rendering code here.

- `StarThroneRenderer.ts`: Deals solely with drawing the map, UI overlays, animations, and input-to-command translation. Subscribes to `GameEventBus` to update visuals when state changes.
- `GameEventBus.ts`: Implements the pub/sub event system. Modules can import a singleton event bus to emit or listen.
- `CombatSystem.ts`: As described, encapsulates all battle resolution logic. Independent of UI.
- `AIStrategyEngine.ts`: Contains AI logic classes. Potentially subdivided further (e.g. one file per strategy) if large.
- `GameMap.ts`, `Territory.ts`, `SupplySystem.ts`, **etc.** remain focused on map and logistics. Ensure they emit events on change (e.g. when a territory changes owner, emit an event).
- `gameConstants.ts` (shared): Central constants/config.
- `types/index.ts`: Shared TS types for cross-module consistency.

- **Example of splitting**: The original 3,400-line `StarThrone.js` might be refactored into several ~500-line modules. The `GameUI.js` or React components handle user input and visuals and call into `StarThroneState` methods rather than manipulating state directly.

- **Clean sample for an event emitter interface** (in `GameEventBus.ts`):

```
export type GameEvent =
  | { type: 'THRONE_CAPTURED'; by: Player; of: Player }
  | { type: 'PLAYER_ELIMINATED'; player: Player }
  // ... more event variants
  ;
export interface GameEventBus {
  subscribe(listener: (event: GameEvent) => void): void;
  emit(event: GameEvent): void;
}
```

Using this, any part of the code can `eventBus.emit({ type: 'THRONE_CAPTURED', by: attacker, of: defender })`, and other parts (UI, logs, sound manager) react accordingly.

Overall, this refactoring yields a **clean, modular codebase** where: - **Combat** is handled exclusively by `CombatSystem` (no duplicate code in players). - **AI** decisions are isolated in strategy classes (making behavior tunable and replaceable). - **Game logic** is separated from rendering/UI via events and decoupled modules [3]. - **Multiplayer** support is simpler because human players are identified by type, not hardcoded IDs, and state changes can be serialized and synced via the event bus. - **Performance** is improved by throttling and pooling (aligning with proven patterns [4]). - **Maintenance** is easier due to clear file structure, shared constants, and well-defined interfaces.

Each of these shifts makes Star Throne **extensible**: for example, adding a new AI strategy means adding a class in `AIStrategyEngine` without touching core game code; adding a new event type is a matter of defining it in `GameEventBus` and emitting it where needed. This prepares the game for future features like multiple human players, networked play, or even swapping out the rendering layer entirely, all while keeping the game logic robust and organized.

Sources: These design choices follow best practices in game architecture such as separating model and view logic [3], using the State/Strategy pattern for AI [1] [2], and applying object pools for performance [4]. This ensures the refactored code is both powerful and maintainable.

---

[1] [2] State · Design Patterns Revisited · Game Programming Patterns
https://gameprogrammingpatterns.com/state.html

[3] About separating game logic from rendering - Newbie & Debugging Questions - JVM Gaming
https://jvm-gaming.org/t/about-separating-game-logic-from-rendering/55355

[4] Object Pool · Optimization Patterns · Game Programming Patterns
https://gameprogrammingpatterns.com/object-pool.html