ChatGPT

General Code Hygiene Refactor – Detailed Changes

To improve consistency, clarity, and maintainability across the codebase, we will apply several refactoring changes. These include standardizing naming conventions, removing dead/debug code, and enhancing error handling/logging. Below is a detailed plan for each category, with examples and diff snippets illustrating the required changes in **all affected modules**.

1. Consistent Naming Conventions

Ensure that method and variable names follow a uniform **camelCase** or **PascalCase** style as appropriate (camelCase for functions/methods, PascalCase for classes, etc.). This improves readability and professionalism. We will fix any inconsistencies or typos in identifiers:

- **Method names:** Rename methods that use inconsistent casing. For example, in **StarThrone.js**, the method `updateVisibilityculling` should be corrected to `updateVisibilityCulling` (capitalize the "C" in *Culling* for proper camelCase). All references to this method should be updated accordingly:

```diff --- a/client/src/game/StarThrone.js +++ b/client/src/game/StarThrone.js @@ class StarThrone (method definition) - updateVisibilityculling() { + updateVisibilityCulling() { // ... method body ... }

@@ StarThrone game loop (method call) // Update visibility culling - this.updateVisibilityculling(); + this.updateVisibilityCulling(); // Update game timer... ```

- **Fix typos in identifiers:** Correct any misspelled names. For instance, in **CombatSystem.js**, rename the method `captureTerritoire` (French spelling) to `captureTerritory`. Update its definition and all calls. This ensures consistency and avoids confusion:

```
diff
  --- a/client/src/game/CombatSystem.js
  +++ b/client/src/game/CombatSystem.js
  @@ resolveBattle outcome (attacker wins branch)
-            this.captureTerritoire(battle, attackingFleets);
+   this.captureTerritory(battle, attackingFleets);
       } else {
           // Defender wins...
  @@ CombatSystem class (method definition)
-    captureTerritoire(battle, remainingFleets) {
+   captureTerritory(battle, remainingFleets) {
       const toTerritory = this.game.gameMap.territories[battle.toTerritoryId];
```
```

```
       const attacker = this.game.players.find(p => p.id === battle.attackerId);
       // ... method body ...
```

- **Uniform style:** Audit other functions and variables for consistent naming. For example, ensure acronyms are used consistently (e.g., use `AI` uniformly in names or all-lowercase when part of a longer identifier, as already done with variables like `aiPlayers`). Also confirm multi-word method names have each word capitalized (the above `updateVisibilityCulling` fix is a case in point). This step may involve scanning for any other irregular naming (like accidental uppercase letters or misspellings) and correcting them similarly across all modules.

By standardizing naming (and fixing typos like *Territoire* → *Territory*), the code becomes more intuitive. New contributors or modders can immediately recognize purpose (e.g. `updateVisibilityCulling` clearly relates to visibility optimization) without stumbling over inconsistent naming.

2. Removing Dead and Debug Code

Clean up any code that is commented out, obsolete, or meant only for temporary debugging. Removing such code eliminates confusion and potential misdirection when reading or modifying the code. Key actions in this category:

- **Strip out commented-out old implementations:** If any blocks of code from previous implementations are simply commented out and no longer needed, delete them. For example, if an older approach to tooltips or input handling was left in comments, it should be removed entirely to avoid clutter. (In a thorough scan of the code, no large deprecated code blocks were found, but this is a general mandate—ensure nothing irrelevant lingers in comments.)

- **Remove temporary debug logs:** Eliminate `console.log` statements that were used for debugging or are not part of the intended final functionality. These console prints can spam the console and degrade performance, and they often represent information that either isn't needed by end-users or should be conveyed via the UI or a controlled logging system instead.

Especially remove any debug logs related to the battle tooltip bug now that the issue is being resolved. For instance, if there are logs like `"No tooltip - hoveredTerritory: false"` (used to trace why the battle odds tooltip wasn't showing), those should be deleted. Once the tooltip is fixed, such logs serve no purpose. Similarly, any placeholder or test code around the tooltip feature (for example, dummy calls or stub elements intended to debug tooltip behavior) should be cleaned up.

- **Remove or silence unnecessary in-game logs:** Many `console.log` calls throughout the game logic are primarily for developer insight and are not needed in production. We will remove these or guard them behind a debug flag (see logging section). Below are examples of debug log removals in **InputHandler.js**, which had numerous logs for edge cases and user actions:

```
diff
  --- a/client/src/game/InputHandler.js
  +++ b/client/src/game/InputHandler.js
  @@ handleTerritoryRightClick (no selection case)
       if (!selectedTerritory) {
```

```
-          console.log('No territory selected for right-click action');
           return;
       }
  @@ handleFriendlyRightClick (same territory edge case)
       if (fromTerritory.id === toTerritory.id) {
-          console.log('Cannot transfer to same territory');
           return;
       }
  @@ handleColonizableRightClick (not enough fleets)
       if (fromTerritory.armySize < 10) {
-          console.log('Need at least 10 fleets to launch probe');
           return;
       }
  @@ handleEnemyRightClick (non-adjacent target)
       if (!fromTerritory.neighbors.includes(toTerritory.id)) {
-          console.log('Can only attack adjacent territories');
           return;
       }
  @@ handleEnemyRightClick (not enough fleets)
       if (fromTerritory.armySize < 2) {
-          console.log('Need at least 2 fleets to attack');
           return;
       }
  @@ transferFleets (not enough fleets)
       if (fromTerritory.armySize < 2) {
-          console.log('Need at least 2 fleets to transfer');
           return;
       }
  @@ transferFleets (after successful transfer)
       if (transferAmount > 0) {
           fromTerritory.armySize -= transferAmount;
           toTerritory.armySize += transferAmount;
           // ... create animation ...
-                      console.log(`Transferred ${transferAmount} fleets from ${fromTerritory.id} to ${toTerritory.id}`);
       }
  @@ attemptMultiHopTransfer (on transfer completion)
                       toTerritory.armySize += transferAmount;
-                      console.log(`Multi-hop transfer completed: ${transferAmount} fleets to ${toTerritory.id}`);
                   }, delay);
                   // ... create animation ...
  @@ attemptMultiHopTransfer (no path found case)
               } else {
-                  console.log('No path found for multi-hop transfer');
               }
  @@ handleDoubleClick (double-click detected feedback)
       handleDoubleClick(territory) {
-          console.log(`Double-click detected on territory ${territory.id}`);
           // Double-click on owned territory – potential supply route creation...
           if (territory.ownerId === this.game.humanPlayer?.id) {
               // ...
  @@ handleDoubleClick (supply route attempt feedback)
                          if (selectedTerritory && selectedTerritory.ownerId ===
this.game.humanPlayer?.id &&
                   selectedTerritory.id !== territory.id) {
                   // Create supply route between owned territories
                   this.game.createSupplyRoute(selectedTerritory, territory);
-                                 console.log(`Supply route creation attempted: $
{selectedTerritory.id} -> ${territory.id}`);
               }
           }
       }
```

In the above diff, we removed numerous `console.log` calls that were used to announce game actions or errors to the console (e.g. "No territory selected...", "Transferred X fleets...", etc.). These were likely added for debugging or initial feedback during development. They can now be removed to declutter the console. (If some of these messages are still desired for debugging in development builds, consider wrapping them behind a debug flag or using a logger – see below).

- **Trim combat logs to prevent spam:** The combat system currently logs detailed step-by-step battle information (e.g., each round's outcome). This can flood the console during large battles and is not suitable for normal gameplay. We will remove or comment out these per-round logs. For example, in **CombatSystem.js**, inside the battle resolution loop, remove the logs for each round outcome:

`diff`

```
  --- a/client/src/game/CombatSystem.js
  +++ b/client/src/game/CombatSystem.js
  @@ resolveBattle (combat round loop)
       if (roll < attackerChance) {
           // Attacker wins this round
           defendingFleets--;
-                     console.log(`Attacker wins round: ${attackingFleets} vs $
{defendingFleets}`);
           if (defendingFleets > 0) {
               toTerritory.flashColor = attacker.color;
               toTerritory.flashUntil = Date.now() + 100;
           }
       } else {
           // Defender wins this round
           attackingFleets--;
-                     console.log(`Defender wins round: ${attackingFleets} vs $
{defendingFleets}`);
```

```
            if (attackingFleets > 0) {
                toTerritory.flashColor = '#ff0000';
                toTerritory.flashUntil = Date.now() + 100;
            }
        }
```

Similarly, high-level combat logs such as `"Battle started: ... attacker chance X%"`, `"Battle won/lost..."` and other interim messages (e.g., in `startBattle`, `resolveBattle`, and `captureTerritory`) can be removed or silenced in production code. These messages were useful during development to verify combat mechanics, but in a clean codebase, they should either be removed or replaced with user-facing notifications (or moved to a debug logger if developers still need them). In fact, some events like throne star captures already have user notifications via `showMessage` calls, so the console logs duplicating that information (`console.log("  THRONE CAPTURE: ...")`, etc.) can be considered redundant and removed.

By removing dead code and debug prints, we **reduce noise and confusion**. The code becomes leaner – when reading it, one sees only the logic that actually runs, and important operations aren't obscured by commented-out blocks or printouts. This also preps the project for production (no irrelevant console spam) and slightly improves performance.

3. Improved Error Handling & Logging

Robust error handling and logging is planned (per the architecture refactoring notes), but we can already take steps to make the code safer and easier to debug, without overwhelming the console:

- **Meaningful error logs in catch blocks:** Ensure every `try/catch` captures exceptions and logs a clear, descriptive error message. For example, the AI update loop already catches errors when updating AI players. We will keep that, but route it through a centralized logger (so it can be toggled or formatted uniformly). Currently in **StarThrone.js**:

```
try {
    aiPlayers[i].update(deltaTime);
    this.updateAIPlayer(aiPlayers[i], deltaTime);
} catch (error) {
    console.error(`Error updating AI player ${aiPlayers[i].name}:`, error);
}
```

This prints a stack trace and identifies which AI caused an issue – which is good. We will maintain that level of detail, but use a unified logging utility to manage it.

- **Introduce a central Logger/flag for verbosity:** To prevent spamming the console and to easily toggle debug output, implement a simple logging utility or use an existing `GameUtils` module. For instance, we can add a `DEBUG_MODE` flag in our game constants and a helper in `GameUtils` for error logging. This way, in normal play (DEBUG_MODE = false), non-critical logs are suppressed, but during development or testing, setting DEBUG_MODE true will surface them.

Implementation steps:

1. **Add a debug flag constant** – e.g., in `common/gameConstants.js`:

```
--- a/common/gameConstants.js
+++ b/common/gameConstants.js
 export const GAME_CONSTANTS = {
     // ... other constants ...
+    DEBUG_MODE: false,  // Toggle verbose logging and debug features
 };
```

2. **Extend GameUtils for logging** – in `GameUtils.js`, add a static method for error logging (and import `GAME_CONSTANTS` at the top if not already):

```
--- a/client/src/game/GameUtils.js
+++ b/client/src/game/GameUtils.js
@@ import statements
 import { ... } from '...';
+import { GAME_CONSTANTS } from '../../../common/gameConstants.js';
 export class GameUtils {
     // ... existing utility methods ...
+    static logError(message, error) {
+        if (GAME_CONSTANTS.DEBUG_MODE) {
+            console.error(message, error);
+        }
+        // In non-debug mode, we could send these errors to a server or
store them if needed,
+        // but avoid spamming the console.
+    }
}
```

3. **Use the logger in catch blocks** – replace direct `console.error` calls with our new utility, so that they respect the debug flag. For example, in the AI update loop in **StarThrone.js**:

`diff`

```
 --- a/client/src/game/StarThrone.js
 +++ b/client/src/game/StarThrone.js
  for (let i = aiStartIndex; i < aiEndIndex; i++) {
      if (i < aiPlayers.length) {
          try {
              aiPlayers[i].update(deltaTime);
              this.updateAIPlayer(aiPlayers[i], deltaTime);
          } catch (error) {
-                 console.error(`Error updating AI player $
{aiPlayers[i].name}:`, error);
+                              GameUtils.logError(`Error updating AI player $
{aiPlayers[i].name}:`, error);
          }
      }
  }
```

Now, if an AI update throws an error, it will be logged only if `DEBUG_MODE` is true. This prevents potential repetitive errors from flooding the console during normal gameplay, while still capturing them during debugging sessions. (In a more advanced setup, `GameUtils.logError` could count occurrences or throttle repetitive messages to further reduce spam.)

4. **Apply similar patterns elsewhere:** We should audit if there are other places that could throw errors and consider adding `try/catch` with logging around them. For example, any asynchronous operations (`Promise` chains) that might reject could use a `.catch` to log errors via `GameUtils.logError`. One specific area to check is the pathfinding or multi-hop transfer code: ensure that if a promise is rejected or an unexpected condition occurs, it doesn't fail silently. Currently, most such cases are handled (e.g., logging "No path found" which we addressed by removing the direct log); in the future, those could be replaced with a `GameUtils.logError` or `logWarn` call if something truly exceptional happens.

5. **Avoid logging sensitive or excessive data:** We'll confirm that error logs do not dump overly verbose or sensitive info. In this project, most logs are game-related (no personal data). The main potential for spam was the per-frame/per-iteration logs, which we are removing or gating. The error logs should be concise and meaningful (like the AI player error example, which is good). We just need to be sure they won't print every frame unchecked. Using the `DEBUG_MODE` flag as shown means in release builds those errors won't appear at all (unless enabled), thus no spam. In development, if an error is continuously happening every frame, a developer would notice and fix it (and could temporarily add more fine-grained gating if needed).

By instituting a central logging approach, we **gain control over output verbosity**. Developers can turn on verbose logs when testing or diagnosing issues, and turn them off for performance or cleaner output. This makes debugging easier without sacrificing performance or user experience in production.

4. Other Cleanliness Improvements

(Beyond the specific points raised, the following general clean-up measures align with the project's goals of eliminating duplication and improving clarity:)

- **Use Named Constants for Magic Numbers:** Where applicable, replace “magic numbers” and unexplained literals with named constants (preferably defined in `GAME_CONSTANTS` or clearly at the top of modules). This helps developers understand the intent behind values. For example, the territory selection radius in `findTerritoryAt` is currently a hardcoded `25`. We can define a constant for this threshold:

```
--- a/common/gameConstants.js
+++ b/common/gameConstants.js
@@
     GAME_CONSTANTS = {
         // ... other constants ...
+        TERRITORY_SELECT_RADIUS: 25,
         ...
     }
```

And use it in **StarThrone.js** instead of the raw number:

```
diff
  @@ findTerritoryAt(x, y) in StarThrone.js
       for (const territory of Object.values(this.gameMap.territories)) {
           const distance = Math.sqrt((worldPos.x - territory.x) ** 2 + (worldPos.y - territory.y) ** 2);
-          if (distance < 25) {
+          if (distance < GAME_CONSTANTS.TERRITORY_SELECT_RADIUS) {
               return territory;
           }
       }
```

Similarly, other recurring values (e.g., the **100** pixel margin in visibility culling, the **200** minimum distance for throne star distribution, timing values like **1000ms** delays, etc.) should be considered for constant definitions. Many are already documented with comments (e.g., `// Minimum distance between throne stars`), which is good – turning them into named constants (like `THRONE_STAR_MIN_DISTANCE`) will make the code self-documenting and allow easy tweaking from a single place.

- **Eliminate duplicate logic via utility functions:** The documentation notes that some helper logic is duplicated across modules (for example, discovery bonuses or AI name generation). While not a focus of this specific cleanup request, it's worth noting as part of general hygiene. We should continue consolidating such code into **GameUtils or dedicated modules**, and then have all parts of the code call those single implementations. This avoids divergence in behavior and reduces maintenance effort. *(For instance, if AI name generation was copied in multiple files but we now have* `GameUtils.generateAIName`*, ensure all modules use that and remove their own duplicate code. The same goes for any discovery handling or other utilities that might be repeated.)*

By addressing these additional cleanliness issues, we make the code **more readable and safer to modify**. Developers (and advanced players who might be modding the game) can trust that:

- **Names are clear and uniform**, so a method or variable's purpose is easy to infer and search for.
- **There is no hidden or leftover code** that confuses the intended behavior – what you see is what's running.
- **Common operations are centralized**, meaning if you change a rule (like how a name is generated or how a discovery is processed), you do it in one place and it applies everywhere.

- **Magic numbers are minimized**, so the intent behind every threshold/duration is either obvious from a constant's name or documented, reducing guesswork.

All these refactoring steps support the project's stated goals of eliminating code duplication and fully utilizing utility modules. They also lay the groundwork for more robust features (like a comprehensive logging system or config-driven game balance) by ensuring the codebase is clean and consistent.

Overall, after applying these changes across **all relevant modules**, the code will be more professional in style and easier to work with, without altering the game's functionality. This cleanup will make future development (and debugging) significantly more efficient and less error-prone.