MANUALE TECNICO DI COLLABORAZIONE LLM PER THE SAFE PLACE (v0.7.22)

1. INTRODUZIONE E SCOPO DEL MANUALE

- **Scopo:** Questo documento serve come guida tecnica di riferimento per l'LLM (Cursor) durante lo sviluppo del videogioco "The Safe Place". Fornisce un contesto persistente sull'architettura del codice, le logiche implementate, le strutture dati chiave e lo stato attuale del progetto.
- **Utilizzo da Parte dell'LLM:**
    - Consultare questo manuale PRIMA di proporre o applicare modifiche al codice.
    - Usarlo come base per comprendere le interdipendenze tra i moduli.
    - Fare riferimento a specifiche sezioni quando vengono richieste modifiche a funzionalità esistenti.
    - Aiutare a mantenere la coerenza con le convenzioni e le logiche già implementate.

2. PRINCIPI GUIDA PER L'LLM NELLO SVILUPPO

- **Mantenere la Modularità:** Rispettare la suddivisione dei file JS esistente e le loro responsabilità. Se si aggiungono nuove funzionalità complesse, discutere con il designer la possibile creazione di nuovi moduli.
- **Priorità ai Bug:** Nella fase attuale, la risoluzione dei bug noti e la stabilizzazione delle funzionalità esistenti hanno la priorità sulla creazione di nuove meccaniche complesse.
- **Modifiche Esplicite:** Quando si applicano modifiche, essere il più specifici possibile: indicare chiaramente il file, la funzione e, se possibile, le righe di codice interessate.
- **Conferma Visuale:** Dopo aver applicato una modifica, mostrare sempre al designer la porzione di codice modificata per una verifica umana.
- **Log di Debug:** Utilizzare `if (typeof DEBUG_MODE !== 'undefined' && DEBUG_MODE) console.log(...);` per i log diagnostici temporanei. Rimuoverli o commentarli una volta che il problema è risolto, su indicazione del designer.
- **Convenzioni:** Seguire le convenzioni di denominazione (camelCase per variabili e funzioni, UPPER_CASE per costanti) e lo stile di scrittura del codice esistente.
- **Non Reinventare:** Prima di scrivere nuovo codice per una funzionalità apparentemente mancante, verificare se esiste già una logica simile o una funzione helper che può essere riutilizzata o adattata, consultando questo manuale o chiedendo al designer.

3. ARCHITETTURA GENERALE DEL CODICE

- **HTML (`index.html`):**
    - **Ruolo:** Definisce la struttura semantica dell'interfaccia utente (pannelli, overlay, popup, schermate di menu/gioco/fine). Carica tutti i file CSS e JavaScript.
    - **Ordine Caricamento JS (Critico):** `game_constants.js` -> `game_data.js` -> `game_utils.js` -> `dom_references.js` -> `ui.js` -> `player.js` -> `events.js` -> `map.js` -> `game_core.js`. Questo ordine DEVE essere mantenuto per rispettare le dipendenze.
- **CSS (Cartella `css/`):**
    - **Modularità:** 9 file CSS, ognuno con responsabilità specifiche (es. `base.css` per stili globali, `layout.css` per la struttura a 3 colonne, `panels.css` per i singoli pannelli, `map.css` per la mappa, `events.css` per i popup, ecc.).
- **JavaScript (Cartella `js/`):**
    - **Modularità:** 9 file JS, ognuno con responsabilità logiche distinte. La comunicazione avviene tramite variabili globali (definite in `game_constants.js`) e chiamate di funzione dirette.

4. DETTAGLIO DEI MODULI JAVASCRIPT

- **4.1. `js/game_constants.js`**
    - **Responsabilità Primaria:** Dichiara le variabili di stato globali del gioco (usando `let`) e definisce le costanti numeriche, stringhe e probabilistiche utilizzate da tutta la logica di gioco (usando `const`).
    - **Variabili Globali Chiave:**
        - `player`: Oggetto che rappresenta il personaggio giocante (stato, statistiche, inventario, ecc.).
        - `map`: Array 2D che rappresenta la mappa di gioco.
        - `messages`: Array per il log degli eventi testuali.
        - `gameActive`, `eventScreenActive`, `gamePaused`: Flag booleani per lo stato del gioco.
        - `isDay`, `dayMovesCounter`, `nightMovesCounter`, `daysSurvived`: Variabili per il ciclo giorno/notte.
        - `currentEventContext`, `currentEventChoices`: Variabili per la gestione degli eventi attivi.
    - **Costanti Fondamentali:** `GAME_VERSION`, `DEBUG_MODE`, `MAP_WIDTH`, `MAP_HEIGHT`, `MAX_MESSAGES`, `STARTING_FOOD/WATER`, `DAY_LENGTH_MOVES`, `*_COST`, `*_CHANCE`, `MAX_INVENTORY_SLOTS`, `PREDATOR_LOOT_WEIGHTS`, `RANDOM_REWARD_POOLS`,

`TIPO_ARMA_LABELS`, `ITEM_EFFECT_DESCRIPTIONS`, `SHELTER_TILES`, ecc.
    - **Punti di Attenzione per l'LLM:**
        - Le variabili globali qui definite sono il principale mezzo di condivisione dello stato tra i moduli.
        - Le costanti qui definite non devono essere modificate dinamicamente durante il gioco.
        - Quando si aggiungono nuove meccaniche che richiedono parametri globali o probabilità, considerarne l'inserimento qui.

- **4.2. `js/game_data.js`**

    - **Responsabilità Primaria:** Contiene dati statici e descrittivi del gioco, principalmente grandi oggetti o array che definiscono il contenuto (oggetti, tipi di tile, testi di eventi, ecc.).
    - **Costanti/Strutture Dati Fondamentali:**
        - `TILE_SYMBOLS`, `TILE_DESC`: Definizioni per i tipi di caselle della mappa.
        - `STATO_MESSAGGI`: Array di testi per i vari stati del giocatore.
        - `EVENT_CHANCE`: Probabilità base degli eventi per tipo di tile.
        - `EVENT_DATA`: Oggetto che mappa i tipi di tile a array di eventi specifici di quel tile.
        - `ITEM_DATA`: Oggetto principale che definisce tutti gli oggetti del gioco (risorse, cibo, armi, armature, ecc.) con le loro proprietà.
        - `CRAFTING_RECIPES`: Oggetto che definisce le ricette di crafting.
        - Vari array di testi: `loreFragments`, `mountainBlockMessages`, `tipiBestie`, `descrizioniIncontroBestie`, `descrizioniTracce`, `descrizioniIncontroPredoni`, `descrizioniOrroreIndicibile`, `esiti...`, `dilemmaEvents`.
    - **Punti di Attenzione per l'LLM:**
        - Questo file è il "database" del contenuto del gioco. Le modifiche qui hanno un impatto diretto su ciò che il giocatore può trovare, usare, craftare e sugli eventi che può incontrare.
        - Mantenere la coerenza nella struttura degli oggetti (specialmente in `ITEM_DATA` e `EVENT_DATA`).
        - Quando si aggiungono nuovi oggetti o eventi, assicurarsi che tutti i campi richiesti siano presenti.

- **4.3. `js/game_utils.js`**

    - **Responsabilità Primaria:** Contiene funzioni di utilità generiche, pure o quasi pure, riutilizzabili in diversi moduli. Non dovrebbero modificare direttamente lo stato globale del gioco, ma possono leggerlo.
    - **Funzioni Chiave:**
        - `getRandomInt(min, max)`: Genera intero casuale.

        - `getRandomText(textInput)`: Seleziona stringa casuale da array.
        - `addMessage(text, type, important)`: Aggiunge messaggio all'array `messages` (in `game_constants.js`) e chiama `renderMessages()` (da `ui.js`).
        - `performSkillCheck(statKey, difficulty)`: Esegue uno skill check basato su una statistica del `player` e una difficoltà, considerando stati negativi. Ritorna `{success, roll, bonus, total, finalDifficulty, text}`.
        - `getSkillCheckLikelihood(statKey, difficulty)`: Fornisce una stima testuale della probabilità di successo di uno skill check.
        - `isWalkable(tileSymbol)`: Controlla se un tile è attraversabile.
        - `getTipoArmaLabel(weaponType)`: Restituisce l'etichetta leggibile per un tipo di arma.
        - `getItemEffectsText(itemInfo)`: Genera testo descrittivo per gli effetti di un oggetto.
        - `chooseWeighted(weightedArray)`: Seleziona un elemento da un array pesato.
    - **Punti di Attenzione per l'LLM:**
        - Queste funzioni dovrebbero rimanere generiche. Se una nuova utilità è specifica per un solo modulo, considerare di definirla localmente in quel modulo.

- **4.4. `js/dom_references.js`**

    - **Responsabilità Primaria:** Centralizza la dichiarazione (`let DOM = {};`) e l'assegnazione dei riferimenti agli elementi del DOM.
    - **Funzioni Chiave:**
        - `assignAllDOMReferences()`: Seleziona tutti gli elementi HTML necessari tramite `document.getElementById` o `querySelector` e li assegna come proprietà dell'oggetto `DOM`. Viene chiamata automaticamente all'evento `DOMContentLoaded`.
    - **Punti di Attenzione per l'LLM:**
        - Qualsiasi nuovo elemento HTML a cui si deve accedere da JavaScript deve avere un ID e il riferimento deve essere aggiunto qui in `assignAllDOMReferences()` e all'oggetto `DOM`.
        - Tutto il codice JS che manipola il DOM dovrebbe usare `DOM.nomeElemento` invece di fare query dirette al documento.

- **4.5. `js/ui.js`**

    - **Responsabilità Primaria:** Gestisce tutto il rendering dell'interfaccia utente. Legge lo stato del gioco (da `game_constants.js`) e i dati statici (da `game_data.js`) e aggiorna il DOM (usando `dom_references.js`). **Non dovrebbe modificare attivamente lo stato logico del gioco.**

- **Funzioni Chiave:**
  - `renderStats()`: Aggiorna pannello statistiche, risorse, stato condizione, equipaggiamento, info gioco (posizione, ora). Applica classi CSS per feedback visivo (es. `.low-resource`, `.night-time-indicator`).
  - `renderMessages()`: Aggiorna il log eventi.
  - `renderInventory()`: Aggiorna la lista dell'inventario, gestisce i listener per tooltip.
  - `renderLegend()`: Aggiorna la legenda della mappa.
  - `renderMap()`: Disegna la mappa a griglia visibile, centrata sul giocatore, gestisce classi per tipi di tile, visitati, giocatore, fine. Aggiunge listener per tooltip mappa.
  - `showEventPopup(eventData)`: Mostra il popup modale per gli eventi, popolandolo con titolo, descrizione e scelte (o bottone "Continua").
  - `closeEventPopup()`: Nasconde il popup evento, riabilita i controlli. Gestisce logica speciale post-chiusura per rifugio notturno.
  - `buildAndShowComplexEventOutcome(...)`: Mostra un popup specifico per l'esito di eventi complessi.
  - `showItemTooltip(itemSlot, event)`, `hideItemTooltip()`: Gestiscono il tooltip degli oggetti inventario.
  - `getItemDetailsHTML(itemInstance)`: Genera l'HTML per i dettagli di un'istanza di oggetto (usato da tooltip e popup azioni oggetto), mostrando durabilità e statistiche.
  - `showMapTooltip(x, y, tileChar, event)`, `hideMapTooltip()`: Gestiscono il tooltip sulla mappa.
  - `disableControls()`, `enableControls()`: Abilitano/disabilitano i bottoni di movimento e impostano `gamePaused`.
  - `showCraftingPopup()`, `closeCraftingPopup()`, `populateCraftingRecipeList()`, `updateCraftingDetails()`: Gestiscono l'interfaccia del popup di crafting.
- **Punti di Attenzione per l'LLM:**
  - Le modifiche all'aspetto del gioco o a come le informazioni vengono presentate avvengono principalmente qui.
  - Assicurarsi che le funzioni di rendering siano chiamate ogni volta che lo stato rilevante del gioco cambia.
  - Evitare di inserire logica di gioco (calcolo danni, modifica inventario, ecc.) in questo file.

- **4.6. `js/player.js`**

  - **Responsabilità Primaria:** Gestisce tutta la logica relativa al personaggio giocante: generazione, statistiche, inventario (aggiunta, rimozione, uso, equipaggiamento oggetti), gestione durabilità, crafting.

- **Funzioni Chiave:**
  - `generateCharacter()`: Inizializza l'oggetto `player` con statistiche base, risorse, inventario ed equipaggiamento iniziale.
  - `addItemToInventory(itemId, quantity)`: Aggiunge oggetti all'inventario, gestendo la durabilità individuale per armi/armature (ogni istanza è uno slot separato) e la stackabilità per altri oggetti.
  - `removeItemFromInventory(itemId, quantityToRemove)`: Rimuove oggetti dall'inventario.
  - `useItem(itemId)`: Applica l'effetto di un oggetto usabile (da `ITEM_DATA[itemId].effects`). Gestisce vari tipi di effetto (cura, risorsa, status, riparazione, apprendimento ricetta, ecc.) e il consumo dell'oggetto.
  - `equipItem(itemId)`, `unequipItem(slotKey)`: Gestiscono l'equipaggiamento e la rimozione di armi/armature, trasferendo l'oggetto tra l'inventario e gli slot equipaggiati, preservando la `currentDurability`.
  - `dropItem(itemId, quantity)`: Lascia cadere un oggetto dall'inventario.
  - `applyWearToEquippedItem(slotKey, wearAmount)`: Riduce la `currentDurability` di un oggetto equipaggiato e logga messaggi se si danneggia o rompe.
  - `showRepairItemTypePopup(repairKitId, repairAmount, targetTypes, charges)`: Prepara e mostra il popup per selezionare quale oggetto riparare con un kit.
  - `applyRepair(repairKitId, instanceIdentifier, repairAmount, targetBaseItemId)`: Applica la riparazione a un'istanza specifica di oggetto e consuma il kit.
  - `checkAmmoAvailability()`, `consumeAmmo()`: Gestiscono la logica delle munizioni per armi a distanza.
  - `showItemActionPopup(itemId, source)`: Prepara e mostra il popup con le azioni disponibili per un oggetto (Usa, Equipaggia, Lascia, Rimuovi). Interagisce con `ui.js` per i dettagli HTML e la visualizzazione.
  - `closeItemActionPopup()`: Chiude il popup azioni oggetto.
  - `checkIngredients(ingredients)`, `attemptCraftItem(recipeKey)`: Gestiscono la logica del crafting (verifica e consumo ingredienti, aggiunta prodotto).
  - `checkAndLogStatusMessages()`: Logga messaggi relativi allo stato del giocatore (fame, sete, ferite, ecc.).
  - `showTemporaryMessage(message, duration)`, `createTemporaryMessagePopup()`: Per messaggi UI temporanei (es. effetto pillole).
- **Punti di Attenzione per l'LLM:**

        - La gestione dell'inventario e dell'equipaggiamento con durabilità individuale è complessa. Ogni oggetto equipaggiabile ha una sua `currentDurability`.
        - La funzione `useItem` è centrale per molti oggetti. Nuovi effetti di oggetti richiedono nuovi `case` nello `switch`.
        - Assicurarsi che le chiamate a `renderStats()` e `renderInventory()` siano fatte dopo modifiche a `player` o `player.inventory`.

- **4.7. `js/map.js`**

    - **Responsabilità Primaria:** Gestisce la generazione della mappa, il movimento del giocatore sulla mappa, il consumo di risorse legato al movimento, le transizioni giorno/notte e il trigger di eventi basati sulla posizione.
    - **Funzioni Chiave:**
        - `generateMap()`: Crea la mappa 2D (`map` array) con vari tipi di terreno (Pianure, Montagne, Foreste, Fiumi, Villaggi, Città, Rifugi) e posiziona Start ed End. Logica di generazione procedurale con cluster per POI.
        - `movePlayer(dx, dy)`: Funzione principale per il movimento. Controlla validità, aggiorna `player.x`, `player.y`, marca tile visitati. Chiama `consumeResourcesOnMove`, `applyPassiveStatusEffects`. Gestisce il contatore passi per le transizioni giorno/notte (`transitionToNight`, `transitionToDay`). Chiama `triggerTileEvent` e, se nessun evento tile si attiva, `triggerComplexEvent`. Gestisce la fine del gioco se si raggiunge 'E'.
        - `consumeResourcesOnMove()`: Riduce fame e sete del giocatore per ogni passo.
        - `applyPassiveStatusEffects()`: Applica danni passivi per stati negativi (fame, sete, ferite, ecc.).
        - `transitionToNight()`: Passa da giorno a notte. Applica costi/penalità se all'aperto. Se in rifugio, gestisce il riposo. Resetta `dayEventDone` flags.
        - `transitionToDay()`: Passa da notte a giorno. Incrementa `daysSurvived`.
        - `showRandomFlavorText(tileSymbol)`: Mostra testo ambientale casuale.
        - `checkForLoreFragment()`: Possibilità di trovare frammenti di lore.
    - **Punti di Attenzione per l'LLM:**
        - `movePlayer` è una funzione molto lunga e complessa con molte responsabilità interconnesse. Le modifiche qui devono essere molto attente.

        - La logica di transizione giorno/notte e la gestione dei rifugi sono cruciali.
        - La generazione della mappa è complessa; modifiche al bilanciamento dei tile devono essere fatte con cautela.

- **4.8. `js/events.js`**

    - **Responsabilità Primaria:** Gestisce il trigger e la risoluzione logica degli eventi di gioco (sia specifici del tile da `EVENT_DATA`, sia complessi/casuali come Predatori, Animali, Tracce, Pericolo Ambientale, Dilemmi, Orrore).
    - **Funzioni Chiave:**
        - `triggerTileEvent(tileSymbol)`: Controlla `EVENT_DATA` per il tile corrente e, se c'è un evento e la probabilità si verifica, lo mostra tramite `ui.showEventPopup`. Gestisce la logica speciale per l'evento diurno del Rifugio ('R').
        - `triggerComplexEvent(tileSymbol)`: Se nessun evento tile si è attivato, calcola se attivare un evento complesso generico (basato su `COMPLEX_EVENT_CHANCE`). Sceglie il tipo di evento (`PREDATOR`, `ANIMAL`, ecc.) usando `chooseWeighted` su `COMPLEX_EVENT_TYPE_WEIGHTS`. Adatta il tipo di evento a giorno/notte. Costruisce dinamicamente `eventData` (titolo, descrizione, scelte) e lo mostra con `ui.showEventPopup`.
        - `handleEventChoice(choiceIndex)`: **Funzione CRUCIALE.** Chiamata quando il giocatore clicca un'opzione in un popup evento.
            - Recupera la scelta e il contesto dell'evento.
            - Gestisce `choice.isSearchAction` (costo tempo).
            - Se la scelta ha un `outcome` o `effect` diretto, lo applica e mostra il risultato.
            - Se la scelta ha uno `skillCheck`, chiama `performSkillCheck`.
            - In base al successo/fallimento dello skill check e al tipo di evento (`currentEventContext.type`) e all'`actionKey` della scelta, applica `successReward` o `failurePenalty` (chiamando `applyChoiceReward` o `applyPenalty`), e determina `outcomeDescription`.
            - Chiama `ui.buildAndShowComplexEventOutcome` per mostrare l'esito.
            - Gestisce l'usura dell'arma se `choice.usesWeapon`.
        - `describeWeaponDamage(equippedWeaponId, currentTileSymbol)`: Restituisce una stringa che descrive il danno inflitto dal giocatore, tenendo conto dell'arma, della sua durabilità, delle munizioni e dei bonus situazionali.
        - `applyChoiceReward(rewardData)`: Aggiunge oggetti/risorse/status positivi al giocatore in base alla ricompensa.

Gestisce ricompense specifiche, array di item e tipi di ricompensa casuali (usando `handleRandomRewardType`). **DEVE loggare gli item specifici ottenuti.**
        - `handleRandomRewardType(rewardType, quantity)`: Sceglie un oggetto casuale da un `RANDOM_REWARD_POOLS` specifico e lo aggiunge all'inventario. **DEVE loggare l'item specifico ottenuto.**
        - `applyPenalty(penaltyObject)`: Applica penalità al giocatore (danno, perdita risorse, status negativi).
        - `performRestStopNightLootCheck()`: Logica per il loot passivo quando si riposa in un rifugio ('R') di notte.
    - **Punti di Attenzione per l'LLM:**
        - `handleEventChoice` è il cuore della logica degli eventi. Le modifiche qui sono molto delicate.
        - Assicurarsi che tutti i percorsi di `successReward` e `failurePenalty` in `EVENT_DATA` e nella logica di `handleEventChoice` siano gestiti correttamente e forniscano feedback chiaro.
        - La coerenza dei dati tra `EVENT_DATA` (in `game_data.js`) e la logica di gestione in `handleEventChoice` è fondamentale.
        - Il problema del danno `[object Object]0` risiede in `describeWeaponDamage`.
        - Il mancato log del loot specifico da rifugio diurno risiede in come `applyChoiceReward` gestisce il feedback per `successReward` di tipo `items` con `type` casuale.

- **4.9. `js/game_core.js`**

    - **Responsabilità Primaria:** Orchestratore principale. Inizializza il gioco, gestisce il flusso tra le schermate (menu, gioco, fine), cattura l'input globale dell'utente (tastiera, click delegati) e lo indirizza ai moduli/funzioni appropriati. Gestisce salvataggio e caricamento.
    - **Funzioni Chiave:**
        - `window.onload`: Entry point, chiama `initializeStartScreen` e `setupInputListeners`.
        - `showScreen(screenToShow)`: Mostra/nasconde i container principali dell'UI.
        - `initializeStartScreen()`: Imposta la schermata iniziale e i listener dei suoi bottoni.
        - `initializeGame()`: Resetta lo stato, chiama `generateCharacter`, `generateMap`, esegue i render iniziali, imposta `gameActive`.
        - `handleGlobalKeyPress(event)`: Listener principale per input da tastiera. Gestisce il movimento (chiamando `map.movePlayer`),

"Attendi" (spazio), e reindirizza l'input a `handleEventKeyPress` se un popup evento è attivo.
        - `handleEventKeyPress(event)`: Gestisce input da tastiera specifici per i popup evento (numeri per scelte, Enter/Esc per "Continua").
        - `handleChoiceContainerClick(event)`: Listener (delegato a `#event-choices`) per click sui bottoni scelta nei popup evento; chiama `events.handleEventChoice` o l'azione diretta del popup.
        - `setupInputListeners()`: Aggiunge tutti gli event listener globali e delegati necessari (tastiera, click su inventario, click su scelte evento, click su bottoni menu/restart, resize, ecc.).
        - `endGame(isVictory)`: Termina la partita, imposta `gameActive = false`, mostra la schermata di fine gioco.
        - `saveGame()`, `loadGame()`: Gestiscono il salvataggio e caricamento dello stato del gioco tramite `localStorage`.
        - `handleInventoryClick(event)` (anche se definita in `player.js`, il listener è qui): Gestisce i click sull'inventario per chiamare `player.showItemActionPopup`.
        - `handleInventoryPointerEnter/Leave`: Gestiscono i tooltip dell'inventario.
        - `handleRestartClick()`: Gestisce il click sul bottone "Torna al Menu" dalla schermata di fine.
        - `handleResize()`: Gestisce il ridimensionamento della finestra (attualmente solo per `renderMap`).
    - **Punti di Attenzione per l'LLM:**
        - Questo file orchestra l'intero gioco. Modifiche qui possono avere impatti ampi.
        - La gestione degli input e dei diversi stati dell'UI (menu, gioco, popup evento, popup azione oggetto) è complessa e deve essere mantenuta coerente.
        - Assicurarsi che i listener vengano aggiunti e rimossi correttamente per evitare comportamenti anomali o memory leak (specialmente per `handleEventKeyPress`).

5. STRUTTURE DATI FONDAMENTALI (DETTAGLIO)

- **5.1. Oggetto `player` (Variabile globale in `game_constants.js`, inizializzato in `player.js`)**
    - **Scopo:** Contiene tutte le informazioni dinamiche relative al personaggio giocante.
    - **Proprietà Chiave:**
        - `name`: (Stringa) Nome del giocatore (es. "Ultimo").

        - `vigore`, `potenza`, `agilita`, `tracce`, `influenza`, `presagio`, `adattamento`: (Numeri) Statistiche base del personaggio, usate per calcoli e skill check.
        - `acquisita`: (Numero) Punti esperienza/abilità da spendere (meccanica futura).
        - `hp`, `maxHp`: (Numeri) Punti vita attuali e massimi. `maxHp` è calcolato da `vigore`.
        - `food`, `water`: (Numeri) Livelli attuali di sazietà e idratazione (range tipico 0-10).
        - `x`, `y`: (Numeri) Coordinate attuali del giocatore sulla mappa.
        - `isInjured`, `isSick`, `isPoisoned`: (Booleani) Flag per stati negativi primari.
        - `poisonDuration`: (Numero) Turni/passi rimanenti di avvelenamento.
        - `currentLocationType`: (Stringa) Simbolo del tile su cui si trova il giocatore (aggiornato da `map.js`).
        - `knownRecipes`: (Array di stringhe) Lista delle `recipeKey` (da `CRAFTING_RECIPES`) che il giocatore ha imparato.
        - `hasBeenWarnedAboutNight`: (Booleano) Flag per evitare spam di messaggi.
        - `equippedWeapon`: (Oggetto o `null`)
            - Se equipaggiato: `{ itemId: "id_arma", currentDurability: numero }`
        - `equippedArmor`: (Oggetto o `null`)
            - Se equipaggiata: `{ itemId: "id_armatura", currentDurability: numero }`
        - `inventory`: (Array di oggetti) Rappresenta l'inventario. Ogni slot è un oggetto:
            - Per oggetti **senza durabilità individuale** (stackabili o no): `{ itemId: "id_oggetto", quantity: numero }`
            - Per oggetti **con durabilità individuale** (armi, armature): `{ itemId: "id_oggetto", quantity: 1, currentDurability: numero }` (ogni istanza occupa uno slot).
    - **Note per LLM:** La gestione di `equippedWeapon`, `equippedArmor` e `inventory` è cruciale per la meccanica della durabilità individuale. `currentDurability` si riferisce all'istanza specifica, mentre `maxDurability` è nel template in `ITEM_DATA`.

- **5.2. `ITEM_DATA` (Oggetto in `game_data.js`)**

    - **Scopo:** "Database" di tutti i template degli oggetti esistenti nel gioco. La chiave è l' `itemId` (stringa univoca).
    - **Struttura Oggetto Item (Esempio):**

'id_oggetto_unico': {

id: 'id_oggetto_unico', // Coerente con la chiave

name: "Nome Leggibile Oggetto",

nameShort: "Nome Breve", // Opzionale, per UI compatta

description: "Descrizione testuale dell'oggetto.",

type: 'food' | 'water' | 'medicine' | 'resource' | 'tool' | 'weapon' | 'armor' | 'ammo' | 'blueprint' | 'lore', // Categoria principale

usable: true | false, // Se l'oggetto può essere "Usato" dall'inventario

stackable: true | false, // (Default true se non specificato, tranne per armi/armature) Se più unità possono occupare uno stesso slot inventario. Armi/armature con durabilità NON sono stackabili.

weight: numero, // Peso dell'oggetto (per futura meccanica di ingombro)

value: numero, // Valore base (per futuro commercio o punteggio)

// Proprietà specifiche per 'weapon'

slot: 'weapon', // Obsoleto se type è 'weapon', ma può rimanere per chiarezza

weaponType: 'mischia' | 'bianca_lunga' | 'bianca_corta' | 'lancio' | 'fuoco' | 'balestra' | 'arco',

damage: numero | { min: numero, max: numero }, // Danno base o range di danno

maxDurability: numero, // Durabilità massima del template (currentDurability è sull'istanza)

ammoType: 'id_tipo_munizione', // (Solo per 'fuoco', 'balestra', 'arco') ID dell'oggetto munizione richiesto

ammoPerShot: numero, // (Solo per 'fuoco', 'balestra', 'arco') Munizioni consumate per colpo

magazineSize: numero, // (Solo per 'fuoco') Capienza caricatore

velocità: numero, // Statistica futura

raggio: numero, // Statistica futura

precisione: numero, // Statistica futura

rumore: numero, // Statistica futura

recuperabile: true | false, // (Solo per 'lancio' tipo frecce/dardi) Se può essere recuperato

// Proprietà specifiche per 'armor'

// slot: 'body' | 'head' | 'accessory', // Dove si equipaggia (già presente, ma specificare se type è armor)

armorValue: numero, // Valore di protezione

// maxDurability: numero, // Già comune

// Proprietà specifiche per 'ammo'

// ammoType: 'id_tipo_munizione', // Stesso ID del tipo munizione usato dall'arma

// Proprietà specifiche per 'blueprint'

// effects: [{ type: 'learn_recipe', recipeKey: 'chiave_ricetta' }] // L'effetto è imparare una ricetta

// Proprietà per oggetti usabili (food, water, medicine, tool, blueprint, lore)

effects: [// Array di oggetti effetto

{

type: 'add_resource', // Aumenta HP, cibo, acqua

resource_type: 'hp' | 'food' | 'water',

amount: numero | { min: numero, max: numero }

},

{

type: 'add_resource_poisonable', // Come add_resource ma con chance di avvelenamento

```
            resource_type: 'food' | 'water',
            amount: numero | { min: numero, max: numero },
            poison_chance: numero_tra_0_e_1
        },
        {
            type: 'add_resource_sickness', // Come add_resource ma con chance di malattia
            resource_type: 'food' | 'water',
            amount: numero | { min: numero, max: numero },
            sickness_chance: numero_tra_0_e_1
        },
        {
            type: 'cure_status',
            status_cured: 'isInjured' | 'isSick' | 'isPoisoned',
            chance: numero_tra_0_e_1, // (Default 1.0)
            heal_hp_on_success: numero // Opzionale
            success_message: "Testo successo", // Opzionale
            failure_message: "Testo fallimento" // Opzionale
        },
        {
            type: 'repair_item_type',
            item_type_target: ['weapon', 'armor'], // Array dei tipi di item che può riparare
            repair_amount: numero,
            charges: numero // Usi del kit
```

```
        },
        {
            type: 'reveal_map_area',
            radius: numero
        },
        {
            type: 'random_pill_effect',
            outcomes: [ { result: 'nome_esito_casuale', weight: numero }, ... ]
        },
        {
            type: 'learn_recipe',
            recipeKey: 'id_ricetta_da_CRAFTING_RECIPES'
        },
        {
            type: 'show_lore' // Mostra un frammento di lore casuale
        }
    ]
}
```

- **Note per LLM:** Quando si definisce un nuovo oggetto, è cruciale specificare l'`id`, `name`, `description`, `type`. Le altre proprietà dipendono dal `type`. Per armi/armature, `maxDurability` è fondamentale. Per oggetti usabili, l'array `effects` definisce il comportamento.

- **5.3. `EVENT_DATA` (Oggetto in `game_data.js`)**
  - **Scopo:** Mappa i tipi di tile (es. `PLAINS`, `FOREST`) a un array di possibili eventi specifici per quel tile.
  - **Struttura Oggetto Evento (interno all'array per un tipo di tile):**

```
{

    id: "id_evento_unico_per_tile",

    title: "Titolo dell'Evento",

    description: "Descrizione della situazione. Può contenere \n per andare a capo.",

    isUnique: true | false, // (Opzionale) Se l'evento può accadere solo una volta.

    choices: [

        {

            text: "Testo dell'opzione 1 (Statistica)",

            actionKey: "chiave_azione_1", // Identificatore univoco per la logica di esito

            skillCheck: { stat: 'agilita', difficulty: numero },

            successText: "Testo se lo skill check ha successo.",

            successReward: { itemId: "id_oggetto", quantity: 1 } /* o { items: [...] } o { type: 'random_...' } */,

            failureText: "Testo se lo skill check fallisce.",

            failurePenalty: { type: 'damage', amount: 5 } /* o { type: 'status', status: 'isInjured' } ecc. */,

            isSearchAction: true // (Opzionale) Se l'azione consuma tempo (passi)

            timeCost: numero // (Opzionale) Costo tempo personalizzato se isSearchAction è true

            usesWeapon: true // (Opzionale) Se l'azione implica l'uso dell'arma equipaggiata (per usura)

        },

        {

            text: "Testo dell'opzione 2 (Outcome Diretto)",
```

actionKey: "chiave_azione_2",

outcome: "Testo del risultato diretto di questa scelta."

// successReward può essere presente anche per outcome diretti

},

{

text: "Testo dell'opzione 3 (Effetto Diretto)",

actionKey: "chiave_azione_3",

effect: { type: 'add_resource', resource_type: 'hp', amount: 5, message: "Ti senti rinvigorito." }

// effect viene usato da handleEventChoice se non c'è skillCheck o outcome esplicito per determinare il messaggio e l'effetto meccanico.

}

]

}

- **Note per LLM:** Ogni evento ha un `id`, `title`, `description`. `choices` è un array. Ogni scelta deve avere `text` e un `actionKey`. Poi può avere `skillCheck` (con `successText`/`Reward`, `failureText`/`Penalty`), oppure `outcome` (per risultati diretti), oppure `effect` (per effetti meccanici diretti semplici).

- **5.4. `CRAFTING_RECIPES` (Oggetto in `game_data.js`)**
  - **Scopo:** Definisce tutte le ricette di crafting disponibili.
  - **Struttura Oggetto Ricetta:**

'chiave_ricetta_unica': { // Es. 'purify_water'

productName: "Nome Leggibile del Prodotto", // Per UI

productId: 'id_oggetto_prodotto_da_ITEM_DATA',

productQuantity: numero,

ingredients: [

{ itemId: 'id_ingrediente_1', quantity: numero },

{ itemId: 'id_ingrediente_2', quantity: numero }

],

description: "Breve descrizione della ricetta o dell'oggetto prodotto.", // Opzionale

successMessage: "Messaggio mostrato al successo del crafting." // Opzionale

// requirements: ['needs_fire'] // Futuro: array di requisiti speciali

}

- **Note per LLM:** `productName` è importante per l'UI. `productId` deve corrispondere a un `itemId` in `ITEM_DATA`. `ingredients` è un array di oggetti con `itemId` e `quantity`.

- **5.5. Array di Testo (in `game_data.js`)**
  - `STATO_MESSAGGI`, `loreFragments`, `mountainBlockMessages`, `tipiBestie`, `descrizioni...`, `esiti...`, `dilemmaEvents`.
  - **Scopo:** Fornire varietà testuale per descrizioni, log, esiti di eventi.
  - **Utilizzo:** Generalmente selezionati casualmente tramite `game_utils.getRandomText()`.
  - **Note per LLM:** Quando si aggiungono nuovi testi, assicurarsi che l'array esista e che i testi siano appropriati per il contesto. `dilemmaEvents` ha una struttura più complessa, simile a quella degli eventi in `EVENT_DATA`.

6. FLUSSI LOGICI CHIAVE DEL GIOCO

- **6.1. Inizializzazione (`game_core.js`):**
  1. `window.onload` -> `assignAllDOMReferences` (da `dom_references.js`, automatico) -> `initializeStartScreen` -> `setupInputListeners`.
  2. `initializeStartScreen`: Mostra menu principale.
  3. Click "Nuova Partita" -> `initializeGame`:
     - Resetta variabili stato globali (`gameActive = true`, ecc.).
     - `generateCharacter()` (da `player.js`): Crea `player` object, inventario iniziale.
     - `generateMap()` (da `map.js`): Crea `map` array, posiziona `player`.
     - Chiama le varie `render...()` (da `ui.js`).

        - Logga messaggio inizio avventura.

- **6.2. Movimento Giocatore (`map.js` -> `movePlayer(dx, dy)`):**

    1. Chiamata da `handleGlobalKeyPress` in `game_core.js`.
    2. Controlla se gioco attivo/pausa, validità nuove coordinate, se tile attraversabile.
    3. `disableControls()`.
    4. Aggiorna `player.x`, `player.y`, marca tile `visited`.
    5. Logga movimento e descrizione luogo.
    6. **Gestione Rifugio 'R' di Notte:** Se si entra in 'R' di notte, mostra popup informativo (`showEventPopup` con contesto `REST_STOP_NIGHT_LOOT_CHECK`). Il flusso si interrompe qui; `closeEventPopup` gestirà il loot check, consumo risorse e `transitionToDay`.
    7. Se non è caso rifugio notturno:
        - `consumeResourcesOnMove()`.
        - `applyPassiveStatusEffects()`.
        - Applica danno per movimento notturno all'aperto (se `!isDay` e non in rifugio).
        - **Controlla Morte:** Se `player.hp <= 0`, chiama `endGame(false)`.
        - **Gestione Contatori Passi e Transizioni Giorno/Notte:**
            - Incrementa `dayMovesCounter` o `nightMovesCounter`.
            - Se `dayMovesCounter >= DAY_LENGTH_MOVES`, chiama `transitionToNight()`.
            - Se `nightMovesCounter >= NIGHT_LENGTH_MOVES` (e non in rifugio), chiama `transitionToDay()`.
            - Le funzioni di transizione gestiscono il cambio di `isDay`, reset contatori, `daysSurvived`, messaggi, e chiamano `renderStats`/`renderMap`. `transitionToNight` gestisce anche penalità/costi se all'aperto, o transizione diretta a giorno se si finisce il giorno in rifugio.
        - **Trigger Eventi (se gioco attivo):**
            - Chiama `triggerTileEvent(targetTile.type)`.
            - Se nessun popup evento da `triggerTileEvent` (`!eventScreenActive`), chiama `triggerComplexEvent(targetTile.type)`.
            - Se nessun popup da nessuno dei due, chiama `checkAndLogStatusMessages`, `showRandomFlavorText`, `checkForLoreFragment`.
        - **Controlla Vittoria:** Se `targetTile.type === TILE_SYMBOLS.END`, chiama `endGame(true)`.

8. **Aggiornamento UI Finale:** Se gioco attivo e nessun popup evento è stato attivato, chiama `renderMap`, `renderStats`, e `enableControls()`. Altrimenti, l'aggiornamento UI e `enableControls` sono gestiti da `closeEventPopup`.

- **6.3. Risoluzione Evento (`events.js` -> `handleEventChoice(choiceIndex)`):**

1. Chiamata da `handleChoiceContainerClick` in `game_core.js`.
2. Recupera `currentEventContext` e la `choice` selezionata.
3. **Costo Tempo:** Se `choice.isSearchAction` (o `choice.timeCost`), avanza `dayMovesCounter` e gestisce potenziale transizione a notte.
4. **Logica Scelta:**
    - Se `choice.outcome` o `choice.effect` (senza skill check): Applica effetto diretto (es. cura HP da `choice.effect`), assegna `successReward` (se presente), determina `outcomeDescription`.
    - Se `choice.skillCheck`: Chiama `performSkillCheck()`.
        - **Successo:** Imposta `outcomeTitle = "Successo!"`, `outcomeDescription = choice.successText`. Chiama `applyChoiceReward(choice.successReward)`. Esegue logica specifica per tipo di evento (PREDATOR, ANIMAL, TRACKS, ecc.) per determinare ulteriori `outcomeConsequencesText` (es. danno inflitto, loot aggiuntivo).
        - **Fallimento:** Imposta `outcomeTitle = "Fallimento"`, `outcomeDescription = choice.failureText`. Chiama `applyPenalty(choice.failurePenalty)`. Esegue logica specifica per tipo di evento per determinare `outcomeConsequencesText` (es. danno subito).
    - **Controlla Morte:** Se `player.hp <= 0` a seguito di penalità, `applyPenalty` dovrebbe aver già chiamato `endGame(false)`. `handleEventChoice` dovrebbe ritornare se `!gameActive`.
5. **Feedback:** Chiama `addMessage` con l'esito e poi `buildAndShowComplexEventOutcome` (da `ui.js`) per visualizzare il popup di risultato.
6. **Usura Arma:** Se `choice.usesWeapon`, chiama `applyWearToEquippedItem()`.
7. Chiama `renderStats()` e `renderMap()`. `closeEventPopup` (chiamata dal bottone "Continua" del popup di esito) riabiliterà i controlli.

- **6.4. Interazione Oggetti (`player.js`):**

    - **Click Inventario:** `handleInventoryClick` (in `game_core.js`, ma chiama player) -> `showItemActionPopup(itemId)`.

- `showItemActionPopup`: Mostra popup con azioni (Usa, Equipaggia, Lascia, Rimuovi). Usa `getItemDetailsHTML` per i dettagli.
- Click su azione -> chiama `useItem`, `equipItem`, `dropItem`, o `unequipItem`.
- Queste funzioni modificano `player.inventory` o `player.equippedWeapon/Armor`, applicano effetti, loggano messaggi e chiamano `renderInventory`/`renderStats`. Poi chiamano `closeItemActionPopup`.
- **`useItem` per Kit Riparazione:** Chiama `showRepairItemTypePopup`, che mostra un altro popup (usando `showEventPopup`). L'azione di questo secondo popup chiama `applyRepair`.

- **6.5. Salvataggio/Caricamento (`game_core.js`):**

  - `saveGame()`: Serializza `player`, `map`, `dayMovesCounter`, `isDay`, `gameDay`, e altri flag in una stringa JSON e la salva in `localStorage` sotto la chiave `SAVE_KEY`.
  - `loadGame()`: Legge la stringa JSON da `localStorage`, la parsa, e ripristina le variabili globali. Chiama tutte le funzioni `render...()` per aggiornare l'UI. Imposta `gameActive = true`.

7. SINCRONIZZAZIONE STATO LAVORI (RIFERIMENTO v0.7.22 e LOG del 15 MAGGIO)

- **Funzionalità Implementate e Confermante Operative (dopo correzioni sintassi e test del 15 Maggio):**

  - Struttura UI di base (pannelli, mappa, log, stats, legenda).
  - Generazione mappa procedurale (bilanciamento da rivedere).
  - Movimento giocatore e meccaniche di base del tile.
  - Ciclo giorno/notte e transizioni (durata da bilanciare).
  - Sistema di sopravvivenza (HP, Fame, Sete) con costi e stati negativi passivi.
  - Inventario con limite slot.
  - Sistema Multi-Stato (Ferito, Malato, Avvelenato, ecc.) con messaggi.
  - **Sistema Eventi (Generale):** La struttura per `triggerTileEvent`, `triggerComplexEvent` e `handleEventChoice` è presente e le chiamate avvengono. Molti eventi specifici e tipi di evento complesso sono stati testati e sembrano funzionare a livello di flusso (scelte -> esiti).
  - **Durabilità Individuale Oggetti:** Implementata e funzionante (inclusa visualizzazione).
  - **Usura Armi (per azioni specifiche):** Confermata funzionante per scelte con `usesWeapon:true` (es. "Forza il passaggio").
  - **Crafting:** Sistema di apprendimento ricette tramite blueprint, UI crafting, logica di `attemptCraftItem` e `checkIngredients` sono implementati. Ricette base definite.

- **Salvataggio e Caricamento:** Logica presente. La durabilità individuale sembra essere preservata correttamente.
- **Popup Azioni Oggetto:** Apertura, visualizzazione dettagli (inclusa durabilità), creazione bottoni azione e chiusura sembrano funzionare.
- **Tooltip Inventario:** Funzionante, risolto bug persistenza.
- **Rifugio 'R':** Logica diurno (evento specifico con `dayEventDone`) e notturno (loot passivo e transizione a giorno) implementata e testata.
- Molti array di testo per eventi (`esiti...`, `descrizioni...`) sono stati ripristinati/aggiunti.

- **Bug Noti / Problemi Attualmente in Discussione (basati sui tuoi ultimi feedback e analisi):**

1. **Indicatore "Notte" non cambia colore (VERDE):** Nonostante la regola CSS esista e la logica JS sia stata (presumibilmente) aggiornata da Cursor, i log di debug specifici non appaiono. **Sospetto principale: la modifica a `js/ui.js` non è stata applicata/salvata correttamente da Cursor, o cache.** (PRIORITÀ MASSIMA DI DEBUG)
2. **Messaggio danno in combattimento: "Infliggi [object Object]0 danni..." (ROSSO):** Chiaramente un bug di formattazione in `describeWeaponDamage()` quando il danno è un range. (PRIORITÀ ALTA DI CORREZIONE)
3. **Feedback Loot da Eventi (es. Rifugio Diurno) nel Log Eventi poco chiaro/mancante (GIALLO):** Il giocatore non vede quali oggetti specifici ha ricevuto nel log, solo un messaggio generico di successo. Il Task B delle istruzioni precedenti mirava a risolvere questo. (DA VERIFICARE SE MODIFICA TASK B È STATA APPLICATA CORRETTAMENTE)
4. **Mancanza Incontri con Predoni (GIALLO/ROSSO):** Nonostante la logica esista, non si riescono a triggerare. (DA INVESTIGARE CON FORZATURA EVENTO)
5. **Bilanciamento Fame/Sete e Disponibilità Acqua (GIALLO - Design):** Il giocatore muore troppo in fretta per mancanza di risorse. (DA AFFRONTARE DOPO STABILIZZAZIONE BUG)
6. **Usura Arma in *Combattimento Effettivo* (GIALLO):** Il Task C mirava ad aggiungerla. Da verificare se la modifica è stata applicata e se funziona una volta che i combattimenti sono testabili e il danno è visualizzato correttamente.
7. **Loot Predatori (GIALLO):** `PREDATOR_LOOT_WEIGHTS` è definito, ma la logica di assegnazione loot dopo aver sconfitto un predatore in `handleEventChoice` (nel `case 'PREDATOR'`, `actionKey: 'lotta'`, successo) deve essere verificata e testata.
8. **Log Console Eccessivi (`DEBUG_MODE`):** Da ripulire una volta stabilizzato il core.
9. **File Rossi in IDE Cursor (`events.js`, `player.js`):** Erano dovuti a errori di sintassi (if-commento e indentazione). Con le ultime correzioni, dovrebbero

essere tornati gialli o verdi. Se sono ancora rossi, ci sono altri errori di sintassi.

- **Roadmap Prossimi Passi (Tecnici e di Design):**

    1. **Debug Bloccanti (Immediato):**
        - Risolvere Indicatore "Notte" (Probabilmente richiede intervento manuale su `js/ui.js` per applicare la funzione `renderStats` corretta, poi test con hard refresh e controllo console per i log di debug).
        - Correggere visualizzazione danno `[object Object]0` in `describeWeaponDamage()`.
    2. **Verifica e Finalizzazione Feedback Loot (Immediato dopo Punto 1):**
        - Assicurarsi che il Task B sia stato applicato correttamente e che il log eventi mostri gli oggetti specifici ricevuti.
    3. **Test Funzionalità Evento Predoni (Dopo Punto 1 e 2):**
        - Forzare l'evento Predatore.
        - Testare le scelte (Fuggi, Combatti, Parla) e i loro esiti.
        - Verificare l'usura dell'arma in combattimento.
        - Verificare il loot dai predoni (se sconfitti).
        - Rimuovere la forzatura dell'evento.
    4. **Bilanciamento Iniziale (Breve Termine):**
        - Affrontare il problema di fame/sete e disponibilità acqua/cibo.
        - Rivedere costi risorse, durata giorno/notte, frequenza eventi.
    5. **Contenuti e Rifiniture (Breve-Medio Termine):**
        - Riempire testi placeholder rimanenti (es. `esitiOrroreIndicibileAffrontaOk`).
        - Implementare logica bonus armi situazionali in `describeWeaponDamage`.
        - Pulire i `console.log` di `DEBUG_MODE`.
    6. **Roadmap a Lungo Termine (Come da Documento Globale):** Combattimento dettagliato, progressione, crafting avanzato, quest, ecc.