```
/************************************************************
 * Global State & Constants
 ************************************************************/
const gameState = {
    // Game map and positioning
    map: [],
    playerPos: { x: 2, y: 2 },
    gameStarted: false,

    // Turn-based properties
    currentTurn: 1,
    movementPointsRemaining: 6,  // 6 moves = 30 ft
    actionPointsRemaining: 1,      // 1 action per turn
    hasDashed: false,

    // Stats and Skills
    stats: [
        { name: "Strength", points: 3, bgColor: "green", textColor: "black" },
        { name: "Intelligence", points: 3, bgColor: "yellow", textColor: "black" },
        { name: "Dexterity", points: 3, bgColor: "orange", textColor: "black" },
        { name: "Constitution", points: 3, bgColor: "red", textColor: "black" },
        { name: "Perception", points: 3, bgColor: "cyan", textColor: "black" },
        { name: "Willpower", points: 3, bgColor: "blue", textColor: "white" },
        { name: "Charisma", points: 3, bgColor: "darkred", textColor: "white" },
        { name: "Marksmanship", points: 3, bgColor: "magenta", textColor: "black" }
    ],
    skills: [
        { name: "Animal Handling", points: 0, bgColor: "darkred", textColor: "white"
},
        { name: "Electronics", points: 0, bgColor: "yellow", textColor: "black" },
        { name: "Explosives", points: 0, bgColor: "magenta", textColor: "black" },
        { name: "Guns", points: 0, bgColor: "magenta", textColor: "black" },
        { name: "Intimidation", points: 0, bgColor: "darkred", textColor: "white" },
        { name: "Investigation", points: 0, bgColor: "cyan", textColor: "black" },
        { name: "Lockpick", points: 0, bgColor: "orange", textColor: "black" },
        { name: "Medicine", points: 0, bgColor: "yellow", textColor: "black" },
        { name: "Melee Weapons", points: 0, bgColor: "green", textColor: "black" },
        { name: "Persuasion", points: 0, bgColor: "darkred", textColor: "white" },
        { name: "Repair", points: 0, bgColor: "yellow", textColor: "black" },
        { name: "Sleight of Hand", points: 0, bgColor: "orange", textColor: "black"
},
        { name: "Stealth", points: 0, bgColor: "orange", textColor: "black" },
        { name: "Survival", points: 0, bgColor: "red", textColor: "black" },
        { name: "Unarmed", points: 0, bgColor: "green", textColor: "black" }
    ],

    // Interactable items and selections
    interactableItems: [],
    selectedItemIndex: -1,
    selectedActionIndex: -1,
```

```
    isActionMenuActive: false,

    // Tile definitions
    impassableTiles: ['▒', '=', '║', '╔', '╗', '╚', '╝'],
    interactableTiles: ['─', '│', '…', '┆'],

    // Stat/skill limits
    MAX_SKILL_POINTS: 30,
    MAX_STAT_VALUE: 10,
    MIN_STAT_VALUE: 1
};

/************************************************************
 * Utility & Helper Functions
 ************************************************************/
// Console logging helper – also updates on-screen console
function logToConsole(message) {
    console.log(message);
    const consoleElement = document.getElementById("console");
    if (consoleElement) {
        const para = document.createElement("p");
        para.textContent = message;
        consoleElement.appendChild(para);
        consoleElement.scrollTop = consoleElement.scrollHeight;
    }
}

// Check if a tile is passable (non-blocking)
function isPassable(tile) {
    const impassable = ['▒', '=', '║', '╔', '╗', '╚', '╝', '─', '│'];
    return !impassable.includes(tile);
}

/************************************************************
 * Map & Rendering Functions
 ************************************************************/
// Test Map Definition
const testMap = [
    ['▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒',
'▒', '▒'],
    ['▒', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '.', '.', '.', '╔', '=', '=', '=', '=', '=', '╗', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '.', '.', '.', '│', '.', '.', '.', '.', '.', '│', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '.', '.', '.', '║', '.', '╔', '=', '╗', '.', '║', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '.', '.', '.', '║', '.', '║', '.', '║', '.', '║', '.', '.', '.', '.', '.',
'.', '▒'],
```

```
    ['▒', '.', '.', '.', '╚', '═', '╝', '.', '╚', '═', '╝', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.',
'.', '▒'],
    ['▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒', '▒',
'▒', '▒']
];

// Initialize map with the testMap
function generateInitialMap() {
    gameState.map = testMap;
    renderMap();
}

function getWorldContainerAt(x, y) {
    for (let key in gameState.worldContainers) {
        let container = gameState.worldContainers[key];
        if (container.position && container.position.x === x && container.position.y
=== y) {
            return container;
        }
    }
    return null;
}

function renderMap() {
    if (!gameState.gameStarted || gameState.map.length === 0 ||
gameState.map[0].length === 0) return;
    const mapContainer = document.getElementById('map');
    if (!mapContainer) return;

    const { map, playerPos } = gameState;
    const visibleWidth = 41, visibleHeight = 21;
    const startX = Math.max(0, playerPos.x - Math.floor(visibleWidth / 2));
    const startY = Math.max(0, playerPos.y - Math.floor(visibleHeight / 2));
    const endX = Math.min(map[0].length - 1, startX + visibleWidth);
    const endY = Math.min(map.length - 1, startY + visibleHeight);
    let mapHtml = '';

    for (let i = startY; i <= endY; i++) {
        for (let j = startX; j <= endX; j++) {
            // Check if a world container is at this tile
            const container = getWorldContainerAt(j, i);
            if (container) {
```

```
                const color = container.symbolColor || "blue";
                mapHtml += `<span class="tile" data-x="${j}" data-y="${i}"
style="color: ${color};">${container.symbol}</span>`;
            } else if (i === playerPos.y && j === playerPos.x) {
                mapHtml += `<span class="tile" data-x="${j}" data-y="${i}"
style="color: green;">@</span>`;
            } else if (['│', '─', '┄', '┆', '>'].includes(map[i][j])) {
                mapHtml += `<span class="tile" data-x="${j}" data-y="${i}"
style="color: black; background-color: white;">${map[i][j]}</span>`;
            } else {
                mapHtml += `<span class="tile" data-x="${j}"
data-y="${i}">${map[i][j]}</span>`;
            }
        }
        mapHtml += '<br>';
    }
    mapContainer.innerHTML = mapHtml;
    updateMapHighlight();
}



// Highlight currently selected interactable items on the map
function updateMapHighlight() {
    const tiles = document.querySelectorAll('.tile');
    tiles.forEach(tile => tile.classList.remove('flashing'));
    if (gameState.selectedItemIndex !== -1 &&
gameState.interactableItems[gameState.selectedItemIndex]) {
        const selectedItem =
gameState.interactableItems[gameState.selectedItemIndex];
        const tileElement =
document.querySelector(`.tile[data-x="${selectedItem.x}"][data-y="${selectedItem.y}"
]`);
        if (tileElement) {
            tileElement.classList.add('flashing');
        }
    }
}

/************************************************************
 * Turn-Based & Movement Functions
 ************************************************************/
// Update the UI with current movement and action points
function updateTurnUI() {
    const movementUI = document.getElementById("movementPointsUI");
    const actionUI = document.getElementById("actionPointsUI");
    if (movementUI) movementUI.textContent = "Moves Left: " +
gameState.movementPointsRemaining;
    if (actionUI) actionUI.textContent = "Actions Left: " +
gameState.actionPointsRemaining;
```

```
}

// Start a new turn by resetting movement and action points
function startTurn() {
    gameState.movementPointsRemaining = 6;
    gameState.actionPointsRemaining = 1;
    gameState.hasDashed = false;
    logToConsole(`Turn ${gameState.currentTurn} started. Moves:
${gameState.movementPointsRemaining}, Actions: ${gameState.actionPointsRemaining}`);
    updateTurnUI();
}

// Allow the player to dash (double movement) if conditions are met
function dash() {
    if (!gameState.hasDashed && gameState.actionPointsRemaining > 0) {
        gameState.movementPointsRemaining = 12;
        gameState.hasDashed = true;
        gameState.actionPointsRemaining--;
        logToConsole(`Dashing activated. Moves now:
${gameState.movementPointsRemaining}, Actions left:
${gameState.actionPointsRemaining}`);
        updateTurnUI();
    } else {
        logToConsole("Already dashed this turn or no actions left.");
    }
}

// End the turn, update health crises, and prepare for the next turn
function endTurn() {
    logToConsole(`Turn ${gameState.currentTurn} ended.`);
    updateHealthCrisis();
    gameState.currentTurn++;
    startTurn();
    renderMap();
    updateTurnUI();
}

// Move the player based on direction input if movement points allow
function move(direction) {
    if (gameState.isActionMenuActive) return;
    if (gameState.movementPointsRemaining <= 0) {
        logToConsole("No movement points remaining. End your turn (press 't').");
        return;
    }
    const { playerPos, map } = gameState;
    const originalPos = { ...playerPos };
    const newPos = { ...playerPos };

    switch (direction) {
        case 'up':
```

```
        case 'w':
        case 'ArrowUp':
            if (newPos.y > 0 && isPassable(map[newPos.y - 1][newPos.x])) newPos.y--;
            break;
        case 'down':
        case 's':
        case 'ArrowDown':
            if (newPos.y < map.length - 1 && isPassable(map[newPos.y +
1][newPos.x])) newPos.y++;
            break;
        case 'left':
        case 'a':
        case 'ArrowLeft':
            if (newPos.x > 0 && isPassable(map[newPos.y][newPos.x - 1])) newPos.x--;
            break;
        case 'right':
        case 'd':
        case 'ArrowRight':
            if (newPos.x < map[0].length - 1 && isPassable(map[newPos.y][newPos.x +
1])) newPos.x++;
            break;
        default:
            return;
    }

    if (newPos.x === originalPos.x && newPos.y === originalPos.y) {
        logToConsole("Can't move that way.");
        return;
    }

    gameState.playerPos = newPos;
    gameState.movementPointsRemaining--;
    logToConsole(`Moved to (${newPos.x}, ${newPos.y}). Moves left:
${gameState.movementPointsRemaining}`);
    updateTurnUI();
    renderMap();
    detectInteractableItems();
    showInteractableItems();
}

/************************************************************
 * Interaction & Action Functions
 ************************************************************/
// Detect interactable items around the player
function detectInteractableItems() {
    gameState.interactableItems = [];
    const { playerPos, interactableTiles, map } = gameState;
    const radius = 2;
    for (let y = playerPos.y - radius; y <= playerPos.y + radius; y++) {
        for (let x = playerPos.x - radius; x <= playerPos.x + radius; x++) {
```

```
            if (map[y] && map[y][x] && interactableTiles.includes(map[y][x])) {
                gameState.interactableItems.push({ x, y, type: map[y][x] });
            }
        }
    }
}

// Display interactable items in the UI
function showInteractableItems() {
    const itemList = document.getElementById('itemList');
    if (!itemList) return;
    itemList.innerHTML = '';
    gameState.interactableItems.forEach((item, index) => {
        const itemElement = document.createElement('div');
        itemElement.textContent = `${index + 1}. ${item.type}`;
        itemElement.dataset.index = index;
        itemElement.classList.add('interactable-item');
        if (index === gameState.selectedItemIndex) {
            itemElement.classList.add('selected');
        }
        itemList.appendChild(itemElement);
    });
    updateMapHighlight();
}

// Get a list of possible actions based on the interactable item type
function getActionsForItem(item) {
    switch (item.type) {
        case '—':
        case '│':
            return ['Cancel', 'Open', 'Break Down'];
        case '┅':
        case '┆':
            return ['Cancel', 'Close'];
        default:
            return ['Inspect'];
    }
}

// Select an interactable item by its number/index
function selectItem(number) {
    if (!gameState.isActionMenuActive && number > 0 && number <=
gameState.interactableItems.length) {
        gameState.selectedItemIndex = number - 1;
        showInteractableItems();
    }
}

// Select an action from the displayed action list
function selectAction(number) {
```

```
    const actionList = document.getElementById('actionList');
    if (!actionList) return;
    const actions = actionList.children;
    if (number >= 0 && number < actions.length) {
        gameState.selectedActionIndex = number;
        Array.from(actions).forEach((action, index) => {
            action.classList.toggle('selected', index ===
gameState.selectedActionIndex);
        });
    }
}

// Show the available actions for the selected item
function interact() {
    if (gameState.selectedItemIndex === -1 || gameState.selectedItemIndex >=
gameState.interactableItems.length) return;
    const item = gameState.interactableItems[gameState.selectedItemIndex];
    const actions = getActionsForItem(item);
    const actionList = document.getElementById('actionList');
    if (!actionList) return;
    actionList.innerHTML = '';
    gameState.selectedActionIndex = -1;
    gameState.isActionMenuActive = true;
    actions.forEach((action, index) => {
        const actionElement = document.createElement('div');
        actionElement.textContent = `${index + 1}. ${action}`;
        actionElement.dataset.index = index;
        actionElement.classList.add('action-item');
        actionList.appendChild(actionElement);
    });
}

// Perform the action selected by the player
function performSelectedAction() {
    if (gameState.selectedActionIndex === -1) return;
    const actionList = document.getElementById('actionList');
    if (!actionList) return;
    const selectedActionElement =
actionList.children[gameState.selectedActionIndex];
    if (!selectedActionElement) return;
    const action = selectedActionElement.textContent.split('. ')[1];
    const item = gameState.interactableItems[gameState.selectedItemIndex];
    logToConsole(`Performing action: ${action} on ${item.type} at (${item.x},
${item.y})`);

    if (gameState.actionPointsRemaining > 0) {
        gameState.actionPointsRemaining--;
        updateTurnUI();
        performAction(action, item);
    } else {
```

```
        logToConsole("No actions left for this turn.");
    }

    cancelActionSelection();
}

// Carry out the specific action logic
function performAction(action, item) {
    if (action === "Open") {
        if (item.type === '─') {
            gameState.map[item.y][item.x] = '┄';
        } else if (item.type === '│') {
            gameState.map[item.y][item.x] = '┆';
        }
        logToConsole(`Opened door at (${item.x}, ${item.y}).`);
    } else if (action === "Close") {
        if (item.type === '┄') {
            gameState.map[item.y][item.x] = '─';
        } else if (item.type === '┆') {
            gameState.map[item.y][item.x] = '│';
        }
        logToConsole(`Closed door at (${item.x}, ${item.y}).`);
    } else if (action === "Break Down") {
        const strengthStat = gameState.stats.find(stat => stat.name === "Strength");
        const modifier = Math.floor(strengthStat.points / 2) - 1;
        const roll = Math.floor(Math.random() * 20) + 1;
        const total = roll + modifier;
        logToConsole(`Rolling to break door: d20(${roll}) + modifier(${modifier}) =
${total}`);
        if (total >= 13) {
            gameState.map[item.y][item.x] = '>';
            logToConsole(`Broke down door at (${item.x}, ${item.y}).`);
        } else {
            logToConsole(`Failed to break door at (${item.x}, ${item.y}).`);
        }
    } else if (action === "Cancel") {
        logToConsole("Action canceled.");
    } else if (action === "Inspect") {
        logToConsole(`Inspecting object at (${item.x}, ${item.y}).`);
    }
    renderMap();
    detectInteractableItems();
    showInteractableItems();
    updateMapHighlight();
}

// Cancel the current action selection
function cancelActionSelection() {
    gameState.isActionMenuActive = false;
    const actionList = document.getElementById('actionList');
```

```
    if (actionList) actionList.innerHTML = '';
    updateMapHighlight();
}

/************************************************************
 * Character Creation & Stats Functions
 ************************************************************/
// Update skill points from character creation
function updateSkill(name, value) {
    const index = gameState.skills.findIndex(skill => skill.name === name);
    if (index === -1) return;
    const newValue = parseInt(value) || 0;
    if (newValue < 0 || newValue > 100) {
        alert('Skill points must be between 0 and 100!');
        return;
    }
    const skills = gameState.skills;
    const currentTotal = skills.reduce((sum, skill) => sum + skill.points, 0);
    const updatedTotal = currentTotal - skills[index].points + newValue;
    if (updatedTotal > gameState.MAX_SKILL_POINTS) {
        alert('Not enough skill points remaining!');
        return;
    }
    skills[index].points = newValue;
    const skillPointsElement = document.getElementById('skillPoints');
    if (skillPointsElement) {
        skillPointsElement.textContent = gameState.MAX_SKILL_POINTS - updatedTotal;
    }
}

// Update stat values for the character
function updateStat(name, value) {
    const index = gameState.stats.findIndex(stat => stat.name === name);
    if (index === -1) return;
    const newValue = parseInt(value) || gameState.MIN_STAT_VALUE;
    if (newValue < gameState.MIN_STAT_VALUE || newValue > gameState.MAX_STAT_VALUE)
{
        alert(`Stat points must be between ${gameState.MIN_STAT_VALUE} and
${gameState.MAX_STAT_VALUE}!`);
        return;
    }
    gameState.stats[index].points = newValue;
    renderCharacterInfo();
}

// Render the tables for stats and skills on the character creator
function renderTables() {
    const statsBody = document.getElementById('statsBody');
    const skillsBody = document.getElementById('skillsBody');
    if (!statsBody || !skillsBody) return;
```

```
    const statsHtml = gameState.stats.map(stat => `
        <div class="stat" style="background-color: ${stat.bgColor}; color:
${stat.textColor};">
            <span>${stat.name}:</span>
            <input type="number" value="${stat.points}"
min="${gameState.MIN_STAT_VALUE}"
                   max="${gameState.MAX_STAT_VALUE}"
                   onchange="updateStat('${stat.name}', this.value)">
        </div>`).join('');
    const skillsHtml = gameState.skills.map(skill => `
        <div class="skill" style="background-color: ${skill.bgColor}; color:
${skill.textColor};">
            <span>${skill.name}:</span>
            <input type="number" value="${skill.points}" min="0" max="100"
                   onchange="updateSkill('${skill.name}', this.value)">
        </div>`).join('');
    statsBody.innerHTML = statsHtml;
    skillsBody.innerHTML = skillsHtml;
}

// Render character information for display in the game
function renderCharacterInfo() {
    const characterInfo = document.getElementById('characterInfo');
    if (!characterInfo) return;
    const nameInput = document.getElementById("charName");
    const levelSpan = document.getElementById("level");
    const xpSpan = document.getElementById("xp");
    if (!nameInput || !levelSpan || !xpSpan) return;
    const name = nameInput.value;
    const level = levelSpan.textContent;
    const xp = xpSpan.textContent;
    const statsHtml = gameState.stats.map(stat => `
        <div class="stats" style="background-color: ${stat.bgColor}; color:
${stat.textColor};">
            <span>${stat.name}:</span>
            <span>${stat.points}</span>
        </div>`).join('');
    const skillsHtml = gameState.skills.map(skill => `
        <div class="skills" style="background-color: ${skill.bgColor}; color:
${skill.textColor};">
            <span>${skill.name}:</span>
            <span>${skill.points}</span>
        </div>`).join('');
    characterInfo.innerHTML = `
        <div>Name: ${name}</div>
        <div>Level: ${level}</div>
        <div>XP: ${xp}</div>
        <h3>Stats</h3>
        ${statsHtml}
        <h3>Skills</h3>
```

```
        ${skillsHtml}
    `;
}

// Start the game by hiding the character creator, initializing systems, and
starting the turn
function startGame() {
    const characterCreator = document.getElementById('character-creator');
    const characterInfoPanel = document.getElementById('character-info-panel');
    const gameControls = document.getElementById('game-controls');

    if (characterCreator) characterCreator.classList.add('hidden');
    if (characterInfoPanel) characterInfoPanel.classList.remove('hidden');
    if (gameControls) gameControls.classList.remove('hidden');

    renderCharacterInfo();
    gameState.gameStarted = true;
    startTurn();
    generateInitialMap();
    detectInteractableItems();
    showInteractableItems();
    initializeHealth();
    initializeWorldContainers();
}

/**************************************************************
 * Inventory System Functions
 **************************************************************/
// Define container sizes
const InventorySizes = {
    XS: 3,
    S: 6,
    M: 12,
    L: 18,
    XL: 24,
    XXL: 36
};

// Inventory container constructor (e.g., for clothing)
function InventoryContainer(sizeLabel) {
    this.sizeLabel = sizeLabel;
    this.maxSlots = InventorySizes[sizeLabel];
    this.items = [];
}

// Define player inventory with default container and hand slots
gameState.inventory = {
    container: new InventoryContainer("M"),
    handSlots: [null, null] // left hand, right hand
};
```

```
// Item constructor
function Item(name, description, size, type, canEquip = false) {
    this.name = name;
    this.description = description;
    this.size = size;
    this.type = type;
    this.equipped = false;
}

// Check if an item can be added to the inventory container
function canAddItem(item) {
    const usedSlots = gameState.inventory.container.items.reduce((sum, itm) => sum +
itm.size, 0);
    return (usedSlots + item.size) <= gameState.inventory.container.maxSlots;
}

// Add an item to the inventory container
function addItem(item) {
    if (canAddItem(item)) {
        gameState.inventory.container.items.push(item);
        logToConsole(`Added ${item.name} to inventory.`);
        updateInventoryUI();
        return true;
    } else {
        logToConsole(`Not enough inventory space for ${item.name}.`);
        return false;
    }
}

// Remove an item by name from the container
function removeItem(itemName) {
    const inv = gameState.inventory.container.items;
    const index = inv.findIndex(item => item.name === itemName);
    if (index !== -1) {
        const removed = inv.splice(index, 1)[0];
        logToConsole(`Removed ${removed.name} from inventory.`);
        updateInventoryUI();
        return removed;
    } else {
        logToConsole(`${itemName} not found in inventory.`);
        return null;
    }
}

// Equip an item from inventory to a hand slot (0: left, 1: right)
function equipItem(itemName, handIndex) {
    const inv = gameState.inventory.container.items;
    const index = inv.findIndex(item => item.name === itemName && item.canEquip);
    if (index === -1) {
```

```
        logToConsole(`${itemName} is not available or cannot be equipped.`);
        return;
    }
    if (gameState.inventory.handSlots[handIndex] !== null) {
        logToConsole(`Hand slot ${handIndex + 1} is already occupied.`);
        return;
    }
    const item = inv.splice(index, 1)[0];
    item.equipped = true;
    gameState.inventory.handSlots[handIndex] = item;
    logToConsole(`Equipped ${item.name} in hand slot ${handIndex + 1}.`);
    updateInventoryUI();
}

// Unequip an item from a hand slot back into the container
function unequipItem(handIndex) {
    const item = gameState.inventory.handSlots[handIndex];
    if (!item) {
        logToConsole(`No item in hand slot ${handIndex + 1} to unequip.`);
        return;
    }
    if (canAddItem(item)) {
        item.equipped = false;
        gameState.inventory.handSlots[handIndex] = null;
        gameState.inventory.container.items.push(item);
        logToConsole(`Unequipped ${item.name} from hand slot ${handIndex + 1}.`);
        updateInventoryUI();
    } else {
        logToConsole(`Not enough space in inventory to unequip ${item.name}.`);
    }
}

// Update the UI for both inventory container items and hand slots
function updateInventoryUI() {
    const invList = document.getElementById("inventoryList");
    if (invList) {
        invList.innerHTML = "";
        gameState.inventory.container.items.forEach((item, idx) => {
            const div = document.createElement("div");
            div.textContent = `${idx + 1}. ${item.name} (${item.size}
slot${item.size > 1 ? "s" : ""})`;
            invList.appendChild(div);
        });
    }
    const handSlotsDiv = document.getElementById("handSlots");
    if (handSlotsDiv) {
        handSlotsDiv.innerHTML = "";
        gameState.inventory.handSlots.forEach((item, idx) => {
            const div = document.createElement("div");
            div.textContent = item
```

```
                ? `Hand Slot ${idx + 1}: ${item.name} (Equipped)`
                : `Hand Slot ${idx + 1}: Empty`;
            handSlotsDiv.appendChild(div);
        });
    }
}

/************************************************************
 * World Containers
 ************************************************************/
// Create world containers (these are not the player's inventory)
gameState.worldContainers = {
    drawer: {
        name: "Drawer",
        container: new InventoryContainer("S")
    },
    gunCase: {
        name: "Gun Case",
        container: new InventoryContainer("S")
    },
    trashCan: {
        name: "Trash Can",
        container: new InventoryContainer("XS")
    },
    safe: {
        name: "Safe",
        container: new InventoryContainer("M")
    }
};

// Initialize the containers with items, positions, symbols, and colors
function initializeWorldContainers() {
    // Drawer: add a Knife and Bandages.
    const knife = new Item("Knife", "A sharp knife that deals 1d4 damage.", 1,
"weapon", true);
    knife.damageDice = "1d4";
    const bandages = new Item("Bandages", "Used to heal wounds.", 1, "healing");

    // Gun Case: add a Baretta 92f.
    const baretta92f = new Item("Baretta 92f", "A handgun with 1d4 damage, a 15
round magazine, and fires 9mm rounds.", 2, "weapon", true);
    baretta92f.damageDice = "1d4";
    baretta92f.magazineCapacity = 15;
    baretta92f.ammoType = "9mm";

    // Trash Can: add a junk item (e.g., Scrap Metal).
    const scrapMetal = new Item("Scrap Metal", "Some useless scrap metal.", 1,
"junk");

    // Safe: add a valuable item (e.g., Cash Bundle).
```

```
    const cashBundle = new Item("Cash Bundle", "A bundle of cash.", 1, "valuable");

    // Populate each container with the items.
    gameState.worldContainers.drawer.container.items.push(knife, bandages);
    gameState.worldContainers.gunCase.container.items.push(baretta92f);
    gameState.worldContainers.trashCan.container.items.push(scrapMetal);
    gameState.worldContainers.safe.container.items.push(cashBundle);

    // Set positions, symbols, and symbol colors for each container.
    // (Adjust coordinates as needed to fit within your map.)
    gameState.worldContainers.drawer.position = { x: 5, y: 3 };
    gameState.worldContainers.drawer.symbol = "π";
    gameState.worldContainers.drawer.symbolColor = "brown";

    gameState.worldContainers.gunCase.position = { x: 10, y: 4 };
    gameState.worldContainers.gunCase.symbol = "Æ";
    gameState.worldContainers.gunCase.symbolColor = "olive";  // greenish gray tone

    gameState.worldContainers.trashCan.position = { x: 2, y: 7 };
    gameState.worldContainers.trashCan.symbol = "u";
    gameState.worldContainers.trashCan.symbolColor = "gray";

    gameState.worldContainers.safe.position = { x: 15, y: 6 };
    gameState.worldContainers.safe.symbol = "#";
    gameState.worldContainers.safe.symbolColor = "gray";
}

/************************************************************
 * Health System Functions
 ************************************************************/
// Initialize health for various body parts
function initializeHealth() {
    gameState.health = {
        head: { max: 5, current: 5, armor: 0, crisisTimer: 0 },
        torso: { max: 8, current: 8, armor: 0, crisisTimer: 0 },
        leftArm: { max: 7, current: 7, armor: 0, crisisTimer: 0 },
        rightArm: { max: 7, current: 7, armor: 0, crisisTimer: 0 },
        leftLeg: { max: 7, current: 7, armor: 0, crisisTimer: 0 },
        rightLeg: { max: 7, current: 7, armor: 0, crisisTimer: 0 },
    };
    renderHealthTable();
}

// Apply damage to a specified body part
function applyDamage(bodyPart, damage) {
    if (!gameState.health || !gameState.health[bodyPart]) return;
    let part = gameState.health[bodyPart];
    part.current = Math.max(part.current - damage, 0);
    logToConsole(`${bodyPart} took ${damage} damage. HP:
${part.current}/${part.max}`);
```

```
    if (part.current === 0 && part.crisisTimer === 0) {
        part.crisisTimer = 3;
        logToConsole(`${bodyPart} is in crisis! Treat within 3 turns or die.`);
    }
    renderHealthTable();
}

// Update crisis timers for body parts at the end of each turn
function updateHealthCrisis() {
    for (let partName in gameState.health) {
        let part = gameState.health[partName];
        if (part.current === 0 && part.crisisTimer > 0) {
            part.crisisTimer--;
            logToConsole(`${partName} crisis timer: ${part.crisisTimer} turn(s)
remaining.`);
            if (part.crisisTimer === 0) {
                logToConsole(`Health crisis in ${partName} was not treated. You have
died.`);
                gameOver();
                return;
            }
        }
    }
}

// Apply treatment to a damaged body part
function applyTreatment(bodyPart, treatmentType, restType, medicineBonus) {
    if (!gameState.health || !gameState.health[bodyPart]) return;
    let part = gameState.health[bodyPart];
    let dc, healing;

    if (treatmentType === "Well Tended") {
        dc = 18;
        healing = (restType === "short") ? 2 : part.max;
    } else if (treatmentType === "Standard Treatment") {
        dc = 15;
        healing = (restType === "short") ? 1 : 3;
    } else if (treatmentType === "Poorly Tended") {
        dc = 10;
        healing = (restType === "long") ? 1 : 0;
    } else {
        logToConsole("Invalid treatment type.");
        return;
    }

    const roll = Math.floor(Math.random() * 20) + 1;
    const total = roll + medicineBonus;
    logToConsole(`Medicine check on ${bodyPart} (${treatmentType}, ${restType}):
d20(${roll}) + bonus(${medicineBonus}) = ${total} (DC ${dc})`);
```

```
    if (total >= dc) {
        let oldHP = part.current;
        part.current = Math.min(part.current + healing, part.max);
        logToConsole(`Treatment successful on ${bodyPart}: HP increased from
${oldHP} to ${part.current}/${part.max}`);
        if (part.current > 0) {
            part.crisisTimer = 0;
            logToConsole(`Health crisis in ${bodyPart} resolved.`);
        }
    } else {
        logToConsole(`Treatment failed on ${bodyPart}.`);
    }
    renderHealthTable();
}

// Render the health table UI
function renderHealthTable() {
    const healthTableBody = document.querySelector("#healthTable tbody");
    healthTableBody.innerHTML = "";
    for (let part in gameState.health) {
        let { current, max, armor, crisisTimer } = gameState.health[part];
        let row = document.createElement("tr");
        row.innerHTML = `
            <td>${formatBodyPartName(part)}</td>
            <td>${current}/${max}</td>
            <td>${armor}</td>
            <td>${crisisTimer > 0 ? crisisTimer : "—"}</td>
        `;
        if (current === 0) {
            row.style.backgroundColor = "#ff4444";
        } else if (crisisTimer > 0) {
            row.style.backgroundColor = "#ffcc00";
        }
        healthTableBody.appendChild(row);
    }
}

// Format body part names for display
function formatBodyPartName(part) {
    const nameMap = {
        head: "Head",
        torso: "Torso",
        leftArm: "L Arm",
        leftHand: "L Hand",
        rightArm: "R Arm",
        rightHand: "R Hand",
        leftLeg: "L Leg",
        leftFoot: "L Foot",
        rightLeg: "R Leg",
        rightFoot: "R Foot"
```

```
    };
    return nameMap[part] || part;
}

// Game over logic placeholder
function gameOver() {
    logToConsole("GAME OVER.");
    // Further game-over logic here
}

/************************************************************
 * Event Handlers & Initialization
 ************************************************************/
// Keydown event handler for movement and actions
function handleKeyDown(event) {
    if (!gameState.isActionMenuActive) {
        switch (event.key) {
            case 'ArrowUp':
            case 'w':
            case 'ArrowDown':
            case 's':
            case 'ArrowLeft':
            case 'a':
            case 'ArrowRight':
            case 'd':
                move(event.key);
                event.preventDefault();
                break;
            default:
                if (event.key >= '1' && event.key <= '9') {
                    selectItem(parseInt(event.key));
                }
        }
        if (event.key === 'x') {
            dash();
            event.preventDefault();
        }
        if (event.key === 't') {
            endTurn();
            event.preventDefault();
        }
    }
    switch (event.key) {
        case 'f':
            if (gameState.isActionMenuActive) {
                performSelectedAction();
            } else if (gameState.selectedItemIndex !== -1) {
                interact();
            }
            event.preventDefault();
```

```
            break;
        case 'Escape':
            cancelActionSelection();
            event.preventDefault();
            break;
        case '1': case '2': case '3': case '4': case '5': case '6': case '7': case
'8': case '9':
            if (gameState.isActionMenuActive) {
                selectAction(parseInt(event.key) - 1);
                event.preventDefault();
            }
            break;
    }
}

// Initial setup on DOM content load
function initialize() {
    renderTables();
    generateInitialMap();
    document.addEventListener('keydown', handleKeyDown);
}

document.addEventListener('DOMContentLoaded', initialize);
```