ChatGPT

Copilot System Prompt for a Top-Down Stealth Game

Project Overview: Build a 2D top-down stealth exploration game set in a historical city, where the player is an assassin interacting with AI-driven NPCs. NPCs use an LLM (e.g. Google Gemini or OpenAI GPT) to generate dynamic dialogue, remember past interactions, and react to the player's reputation. The world is dynamic: NPCs have motivations/relationships, rumors spread, and missions are created on-the-fly based on player actions. Guards patrol with basic AI and line-of-sight detection, raising alarm if the player is seen or behaves suspiciously. Player and NPC states (memories, choices, reputation) persist between sessions via local storage or a cloud database [1] [2]. The game uses **React** (with Canvas or PixiJS) for rendering, **Zustand** or Redux for state management, an LLM API for dialogue/mission generation, and **Firebase** or Supabase for persistence [3] [4].

Project Structure

- **Folder layout:**

```
my-stealth-game/
  src/
    components/        # React components (Player, NPC, Guard, Dialogue,
Mission UI, etc.)
    game/              # Core game logic (map, movement, collision,
pathfinding, etc.)
    ai/                # AI modules (NPCAgent.js, MissionGenerator.js)
    state/             # Zustand or Redux store setup (gameState,
NPCMemory, reputation)
    services/          # External services (firebase.js, llmApi.js for
LLM calls)
    utils/             # Helper functions (pathfinding, line-of-sight
checks, etc.)
    App.js             # Main React app
    index.js           # Entry point
  public/              # Static assets (images, sounds)
  package.json
  README.md
```

- **Technology stack:**
- **Rendering:** React with Canvas or PixiJS using React-Pixi for declarative rendering of sprites and shapes [3] [5].
- **AI Integration:** An LLM (e.g. Gemini/ChatGPT) accessed via a REST API or WebSocket. This handles NPC dialogue, memory recall, and mission text generation [6] [7].
- **State Management:** Zustand (lightweight hook store) or Redux to hold game state (player stats, NPC data, global flags). Zustand is simple to set up, e.g. `const useStore = create(set => ({ counter:0, inc:()=>set(s=>({counter:s.counter+1})) }))` [8].

- **Persistence:** Firebase (Firestore or Realtime DB) or Supabase for saving persistent state (NPC memories, mission progress, player reputation) [4]. LocalStorage can be used for browser-only persistence.

Rendering (React + PixiJS Canvas)

Use React-Pixi for the 2D view. For example, in a `GameCanvas` component import `Stage` and `Sprite` from `@inlet/react-pixi` to create a PixiJS renderer [3] [5]:

```
import React from 'react';
import { Stage, Sprite } from '@inlet/react-pixi';
import { useStore } from '../state/store';

function GameCanvas() {
  const { playerPos, npcPositions } = useStore(state => ({
    playerPos: state.player.position,
    npcPositions: state.npcs.positions
  }));
  return (
    <Stage width={800} height={600} options={{ backgroundColor: 0x999999 }}>
      {/* Player sprite */}
      <Sprite image="assets/player.png" x={playerPos.x} y={playerPos.y}
anchor={0.5} />
      {/* NPC sprites */}
      {npcPositions.map((pos, idx) => (
        <Sprite
          key={idx}
          image="assets/npc.png"
          x={pos.x}
          y={pos.y}
          anchor={0.5}
        />
      ))}
    </Stage>
  );
}
```

Explanation: The `<Stage>` component creates a PixiJS canvas renderer [3]. Inside it, `<Sprite>` components render images at given `(x,y)` coordinates [5]. The `useStore` hook (from Zustand) provides dynamic positions from game state. This pattern lets Copilot scaffold movement and rendering code clearly.

NPC Dialogue & Memory (LLM-Driven)

Each NPC is backed by a chat agent. When the player speaks (or triggers dialogue), form a prompt including the NPC's profile, memory, and the player's latest message. For example:

```
// ai/NPCAgent.js
import { chatCompletion } from './llmApi'; // abstracts Gemini/GPT API

async function talkToNPC(npcId, playerMessage) {
  // Load NPC memory and context
  const memory = await loadNPCMemory(npcId);
  const profile = getNPCProfile(npcId); // personality, goals
  const prompt = `You are ${profile.name}, ${profile.description}. Memory: $
{memory}. ` +
                 `The player says: "${playerMessage}". Respond as you
would.`;
  // Call LLM for response
  const response = await chatCompletion(prompt);
  // Update memory with this interaction
  await saveNPCMemory(npcId, memory + ' Player said: ' + playerMessage + '
NPC answered: ' + response.text);
  return response.text;
}
```

Here, `chatCompletion` sends a prompt to the LLM and returns its response text. NPCs **remember** past interactions by storing conversation snippets [1] [6]. Their motivations and allegiances (loaded via `getNPCProfile`) inform the prompt. For example, an NPC might recall "the player helped me before" or "the player is known as a thief" to influence dialogue and decision-making [1] [6]. The Copilot prompt should emphasize including `npcId`-specific memory and context in each API call.

Dynamic Mission Generation (AI-Driven)

Generate new missions using the LLM based on game context (player reputation, city state, NPC requests). For example:

```
// ai/MissionGenerator.js
import { chatCompletion } from './llmApi';

async function generateMission(playerState, cityState) {
  const { reputation, pastMissions } = playerState;
  const prompt = `Create a new stealth mission for an assassin in a
Renaissance city. ` +
                 `Player reputation: ${reputation}. City situation: $
{JSON.stringify(cityState)}. ` +
                 `Must involve intel gathering or covert action. Return
mission details in JSON.`;
  const response = await chatCompletion(prompt);
  const mission = JSON.parse(response.text);
  return mission;
}
```

This uses an LLM to craft mission objectives (e.g. "steal a document from the guild hall" or "eavesdrop on guards") based on current state. Modern AI quest systems analyze player behavior and world state

to tailor missions [2]. Copilot should generate code that sends a structured prompt to the LLM and parses its JSON output. The mission object can then be added to state (e.g. in a `missions` array).

Guards: Patrol & Detection Logic

Implement basic guard AI with patrol paths and vision-based detection. Guards have a set of waypoints or a pathfinding routine (A*) to follow. In their update loop, check if the player is in sight and within a vision cone:

```
// utils/sight.js
function canSeePlayer(guard, player, obstacles) {
  const dx = player.x - guard.x;
  const dy = player.y - guard.y;
  const angleToPlayer = Math.atan2(dy, dx) * (180/Math.PI);
  const diff = Math.abs(normalizeAngle(angleToPlayer - guard.direction));
  // 60° cone of vision (±30°)
  if (diff > 30) return false;
  // Raycast to check line-of-sight (no walls blocking)
  if (raycast(guard.x, guard.y, player.x, player.y, obstacles)) return false;
  return true;
}

// game/Guard.js (simplified update loop)
function updateGuard(guard, player, obstacles) {
  if (canSeePlayer(guard, player, obstacles)) {
    guard.state = 'alert';
    guard.suspicion += 1;
    // Pursue player: compute path
    guard.path = findPath(guard.position, player.position, obstacles);
  } else if (guard.state === 'patrol') {
    // Continue patrol: if reached target, pick next waypoint
    if (guard.atTarget()) {
      guard.target = guard.nextPatrolPoint();
      guard.path = findPath(guard.position, guard.target, obstacles);
    }
  }
  guard.followPath();
}
```

Explanation: To determine line-of-sight, compute the angle between the guard's facing direction and the vector to the player, ensuring it falls within a cone (e.g., ±30°) [9]. Then raycast or check collisions with obstacles to ensure nothing blocks view [10]. If the player is spotted, switch the guard to an "alert" state and set a pursuit path (using an A* or similar pathfinding function [11]). Guards increment a suspicion level or alarm counter when sighting or hearing the player. Copilot should output code that checks both angle and obstruction, as in the example above.

Reputation & Suspicion System

Maintain a global reputation/suspicion metric (e.g. in state). Actions like committing visible crimes or being seen by guards should increase suspicion:

```
// state/store.js (Zustand example)
const useStore = create((set) => ({
  reputation: 0,
  increaseReputation: (amount) => set(state => ({ reputation:
state.reputation + amount })),
  // ... other state (player, NPCs, missions) ...
}));

// When an event occurs:
function onPlayerDetected() {
  useStore.getState().increaseReputation(5);
  if (useStore.getState().reputation > 100) {
    alertGuards(); // broader search or increased alertness
  }
}
```

NPCs and guards react to the `reputation` score. For example, if reputation is high (player notorious), NPC dialogue or guard behavior changes. This flow is inspired by how NPCs evolve based on player actions [1] (e.g., "remembering you lied three quests ago" or "turning on you if you ignore them"). In the prompt to Copilot, emphasize updating and checking a stored reputation value.

State Management (Zustand/Redux)

Use a store to keep game state. For simplicity, Zustand can be configured as follows [8]:

```
// state/store.js
import create from 'zustand';

export const useStore = create((set) => ({
  player: { x: 0, y: 0, reputation: 0 },
  npcs: { data: {}, positions: [] },
  missions: [],
  // Actions:
  movePlayer: (dx, dy) => set(state => ({
    player: { ...state.player, x: state.player.x + dx, y: state.player.y +
dy }
  })),
  addMission: (mission) => set(state => ({
    missions: [...state.missions, mission]
  })),
  updateNPCMemory: (id, mem) => set(state => ({
    npcs: { ...state.npcs, data: { ...state.npcs.data, [id]: mem } }
  })),
```

```
  // ... other actions ...
}));
```

This shows a basic pattern: state fields and updater functions (actions). Copilot should generate similar store setup code. Zustand's simple `create` store avoids boilerplate [8]. Components can then use `useStore` to read or update state, as in the rendering and event code examples above.

Persistence (Firebase/LocalStorage)

Persist key data so it survives page reloads. For example, use Firebase Firestore:

```
// services/firebase.js
import { initializeApp } from "firebase/app";
import { getFirestore, doc, setDoc, getDoc } from "firebase/firestore";

const firebaseConfig = { /* config */ };
const app = initializeApp(firebaseConfig);
const db = getFirestore(app);

export async function saveNPCMemory(npcId, memory) {
  await setDoc(doc(db, 'npc_memories', npcId), { memory });
}

export async function loadNPCMemory(npcId) {
  const docRef = doc(db, 'npc_memories', npcId);
  const docSnap = await getDoc(docRef);
  return docSnap.exists() ? docSnap.data().memory : "";
}
```

Explanation: Firebase's real-time database (or Firestore) is suited for storing game state as JSON [4]. Here we save each NPC's memory under a document key. Similarly, you can store player stats, completed missions, and reputation. Copilot should scaffold similar async save/load functions using the Firebase SDK.

As a simpler alternative, you can use browser `localStorage`:

```
function saveGameState(state) {
  localStorage.setItem('gameState', JSON.stringify(state));
}
function loadGameState() {
  const data = localStorage.getItem('gameState');
  return data ? JSON.parse(data) : {};
}
```

This lets the game persist between sessions without a backend (suitable for single-player local games).

Sample Code Snippets

- **React Canvas Rendering (Player & NPC):**
  Use `<Stage>` as the PixiJS canvas and `<Sprite>` for entities [3] [5].

```
<Stage width={800} height={600}>
  <Sprite image="assets/player.png" x={playerX} y={playerY} />
  {npcs.map(npc => (
    <Sprite key={npc.id} image="assets/npc.png" x={npc.x} y={npc.y} />
  ))}
</Stage>
```

- **Gemini/GPT Dialogue Call:**
  Incorporate NPC identity and memory into the prompt; update memory after response [1].

```
const prompt = `You are ${npc.name}, a ${npc.role}. Remember: $
{npc.memory}. ` +
               `Player asks: "${playerQuestion}"`;
const reply = await chatCompletion(prompt);
updateMemory(npc.id, playerQuestion, reply.text);
```

- **Guard Patrol Logic:**
  Check angle and raycast for line-of-sight [9], use A* for movement.

```
if (canSeePlayer(guard, player, walls)) {
  guard.state = 'alert';
  guard.path = aStar(guard.pos, player.pos, mapGrid);
}
guard.followPath();
```

- **Mission Generation (AI):**
  Prompt LLM with context and parse JSON output [2].

```
const missionPrompt = `Generate a stealth mission for a city
setting...`;
const missionJson = await chatCompletion(missionPrompt);
const mission = JSON.parse(missionJson.text);
addMission(mission);
```

- **State Persistence (Firebase):**
  Use Firestore `setDoc` and `getDoc` to save/load objects [4].

```
await setDoc(doc(db, 'players', playerId), playerState);
const snapshot = await getDoc(doc(db, 'players', playerId));
const loadedState = snapshot.exists() ? snapshot.data() : defaultState;
```

Each code snippet is annotated with comments and clear variable names to guide Copilot. Emphasize modular functions (e.g. `generateMission`, `talkToNPC`, `canSeePlayer`) to keep logic reusable. With these guidelines and examples, Copilot can iteratively build out the components: rendering loop, AI calls, guard AI, state store, and persistence layer.

Sources: Concepts for dynamic NPCs and dialogue are inspired by modern AI-driven game designs [1] [6]. Rendering approach follows React-Pixi best practices [3] [5]. State management patterns are informed by Zustand tutorials [8]. Guard vision logic uses standard line-of-sight algorithms [9]. Firebase usage is based on its real-time JSON storage model [4]. These references guide the system prompt's structure and code examples.

---

[1] How AI Is Revolutionizing NPCs: Dynamic, Memory-Driven Game Worlds | by Vishnu Sharma | Medium
https://thecodekaizen.medium.com/how-ai-is-revolutionizing-npcs-dynamic-memory-driven-game-worlds-21cd2b463a0e

[2] [6] AI-powered procedural quest generation is reshaping the landscape of modern game design by enabling dynamic narratives, adaptive mission creation, and deeply personalized gameplay. This article explores how AI and machine learning tools automate quest design, improve player engagement, reduce production costs, and push gaming toward more immersive, replayable open-world environments.
https://www.daydreamsoft.com/blog/ai-powered-procedural-quest-generation-transforming-narrative-depth-in-modern-games

[3] [5] Getting started with PixiJS and React: Create a canvas - LogRocket Blog
https://blog.logrocket.com/getting-started-pixijs-react-create-canvas/

[4] Storing game state in Firebase. Note: Since I wrote this in 2016, there... | by Victor Hom | Medium
https://medium.com/@heyvictor/storing-game-state-in-firebase-f510d4d00809

[7] Gaming with GPTs. Building a GPT Driven Game | by James Wanderer | Medium
https://medium.com/@jmwanderer/gaming-with-gpts-1287039ac702

[8] Zustand Made Simple: A Practical React State Guide for Busy Engineers | by ImranMSA | Medium
https://medium.com/@imranmsa93/zustand-made-simple-a-practical-react-state-guide-for-busy-engineers-203483d8dd67

[9] [10] flash - line of sight 2d - Stack Overflow
https://stackoverflow.com/questions/5493085/line-of-sight-2d

[11] path finding - A* mouse movement javascript top-down game - Game Development Stack Exchange
https://gamedev.stackexchange.com/questions/46899/a-mouse-movement-javascript-top-down-game