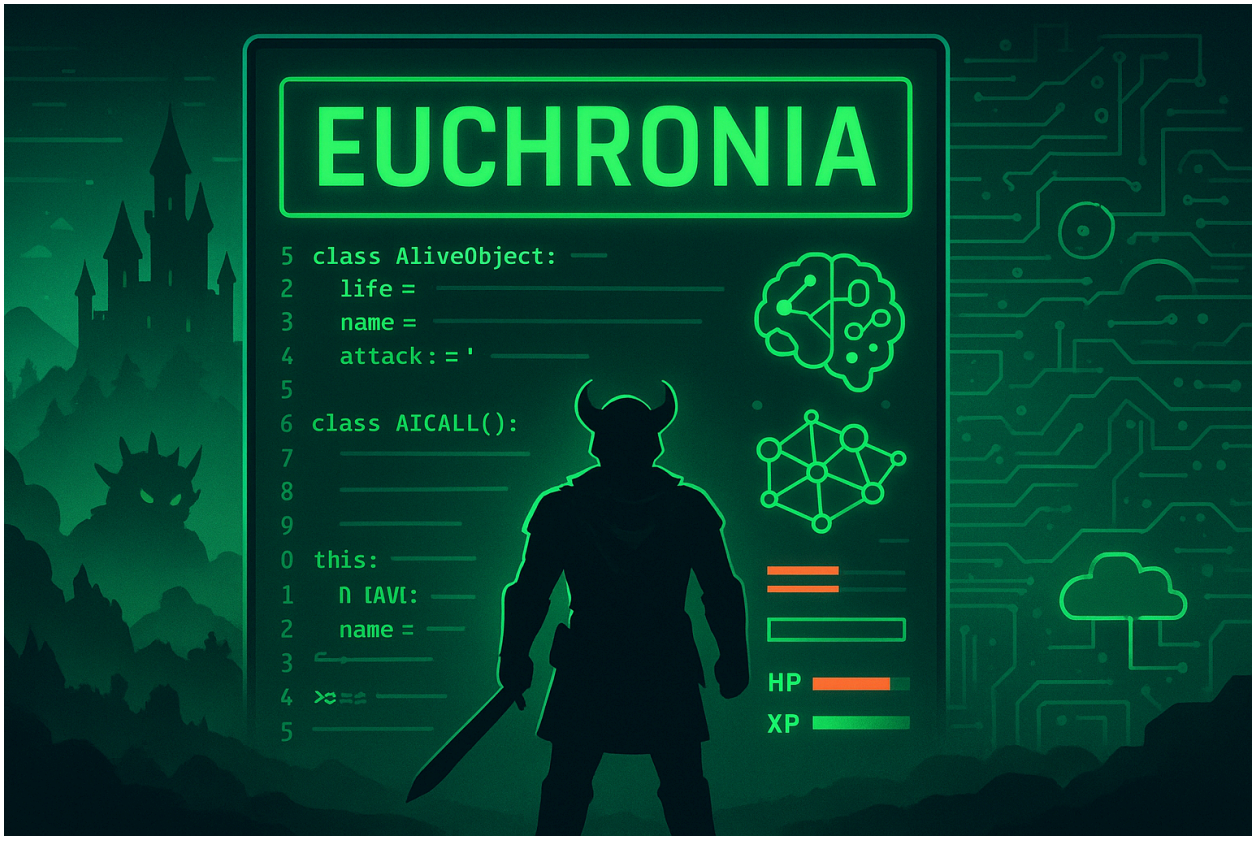

RPG - EUCHRONIA

POO + OpenAI

26/09/2025

Diogo Teodoro Dias Lamas

1. Visão Geral

EUCHRONIA é um RPG de terminal projetado para ser uma experiência de jogo auto-adaptável. O núcleo do projeto utiliza a API da OpenAI para gerar proceduralmente diálogos, inimigos, itens e uma narrativa contínua, moldada pelas ações do jogador. O objetivo principal é explorar a aplicação prática de IAs em projetos de software, criando uma relação simbiótica entre a lógica do jogo e a criatividade de um Large Language Model (LLM).

2. Objetivos do Projeto

1. **Fundação em POO:** Construir a base do sistema utilizando os princípios da Programação Orientada a Objetos para garantir um código limpo e escalável.
2. **Combate Estratégico:** Desenvolver módulos de combate robustos, com um sistema de Skills e Classes bem definido.
3. **Mundo Dinâmico (IA-Driven):** A história, os inimigos e os eventos do mundo serão gerados e controlados dinamicamente pela IA.
4. **Imersão Visual:** Utilizar bibliotecas (ex: `Colorama`, `Time`) para personalizar o terminal e criar uma experiência de usuário imersiva.
5. **Loot Inteligente:** Itens e consumíveis serão gerados pela IA dentro de parâmetros específicos de contexto e balanceamento.
6. **Persistência de Dados:** Salvar o estado do mundo (inimigos, lore, itens) em um sistema de banco de dados simples utilizando JSON.
7. **Sistema de Jogadores:** Implementar um sistema de Login e Senha para persistência do progresso individual.
8. **Personalização:** Oferecer ao jogador opções de configuração, como nível de dificuldade e gerenciamento de custos de API (tokens).
9. **Código Modular:** Estruturar o projeto com uma alta modularização, organizando o código em pastas e arquivos com responsabilidades claras.

3. Especificações Técnicas e de Arquitetura

O sistema será construído em **Python**, com a API da **OpenAI (modelo GPT-4o mini)** servindo como o cérebro criativo do jogo.

3.1. Arquitetura do Software

Modularização: O projeto será dividido em uma estrutura de pastas clara:

- `main.py`: Ponto de entrada e orquestrador principal.
- `AICALL.py`: Módulo centralizado para todas as chamadas à API, responsável pela engenharia de prompt.
- `Scripts/`: Módulos com a lógica do jogo (ex: `combate.py`, `exploracao.py`).
- `Data/`: Bancos de dados em JSON e arquivos de lore (ex: `itens.json`, `Lore.txt`).
- `Save/`: Dados de progresso de cada jogador.

Programação Orientada a Objetos (POO):

- Uma classe pai `AliveObject` definirá atributos comuns (Vida, Nome, Ataque, etc.).
- Classes filhas como `EnemyObject` e `HeroObject` herdarão de `AliveObject`, especializando seus comportamentos.

3.2. Integração com a IA

O módulo `AICALL.py` será o coração do sistema. Ele conterá funções projetadas para retornar dados formatados (ex: um objeto inimigo completo para o combate) a partir de prompt bem definidos. Para manter a consistência narrativa, a história gerada pela IA será salva em um `Lore.txt`, que será usado como contexto recursivo em chamadas futuras.

3.3. Persistência de Dados

Itens, armas e inimigos gerados serão salvos em arquivos `.json` para garantir a recorrência e a consistência do mundo. O sistema terá uma lógica de "cache": se a IA tentar gerar um objeto que já existe no nosso banco de dados, o objeto existente será reutilizado para economizar chamadas de API.

4. Arquitetura de Classes (Models)

Todas as classes de objetos do jogo serão definidas em um arquivo `Model.py`, seguindo uma estrutura de herança para maximizar a reutilização de código.

4.1. `ModelEntity` (Classe Pai)

É a classe base para todas as entidades vivas (Jogadores, Inimigos, NPCs). Seu objetivo é fornecer os atributos universais compartilhados por todos.

Atributos: `name`, `life`, `maxlife`, `base_attack`, `defense`, `speed`, `xp`, `level`.

```
class ModelEntity():

    def __init__(self, name, life, maxlife, base_attack, defense,
speed, xp, level):

        self.name = name

        self.life = life

        self.maxlife = maxlife

        self.base_attack = base_attack

        self.defense = defense

        self.speed = speed

        self.xp = xp

        self.level = level
```

(Exemplo de um ModelEntity)

4.2. `ModelPlayer`

Herda de `ModelEntity` e adiciona atributos e métodos exclusivos do jogador. Seus atributos iniciais serão definidos a partir de um arquivo JSON de classes, garantindo flexibilidade.

Atributos Adicionais: `gold`, `inventory`, `p_class`, `p_skills`.

Métodos Principais: `gold_add`, `gold_sub`, `xp_add`, `level_up`, `heal`, etc.

```
class ModelPlayer(ModelEntity):

    def __init__(self, effect, p_class, p_skills):

        super.__init__(

            life = p_class['maxlife'],

            maxlife = p_class['maxlife'],

            base_attack = p_class['base_attack'],

            defense = p_class['defense'],

            speed = p_class['speed']

        )

     self.next_level = p_class[‘next_level_atribute’]

         self.effect = {}

         self.gold = 0

         self.inventory = {}

         self.p_class = p_class

         self.p_skills = p_skills[p_class]
```

(Exemplo de ModelPlayer)

4.3. `ModelItens` (Classe Pai de Itens)

É a classe base para todos os itens do jogo, contendo informações gerais.

Atributos: `nome`, `tipo`, `valor`, `descricao`.

```
class ModelItens:


    def __init__(self, nome, tipo, valor, descricao):
        self.nome = nome
        self.tipo = tipo # "Arma", "Armadura", "Pocao", "Amuleto",
etc.
        self.valor = valor # Valor em moedas de ouro
        self.descricao = descricao


    def __str__(self):
        return f"{self.nome} ({self.tipo})"
```

(Exemplo dos atributos da classe ModelItens)

5. Estrutura do Banco de Dados (JSON)

Os dados do jogo serão armazenados em arquivos JSON para fácil acesso e manipulação, servindo como um fallback e base de conhecimento para a IA.

5.1. `Itens.json`

Armazena o registro de todos os itens base, como armas, armaduras e poções, com seus respectivos atributos.

```
{
    "Adaga de Couro": {
    "tipo": "Arma",
    "valor": 10,
    "descricao": "Leve e rápida, ideal para ataques furtivos.",
    "bonus": { "ataque": 3, "velocidade": 2 }
  },
  "Poção de Cura Menor": {
    "tipo": "Pocao",
    "valor": 25,
    "descricao": "Um frasco com um líquido vermelho que recupera a
vitalidade.",
    "efeitos": { "cura": 50 }
  }
}
```

(Exemplo de Itens.json)

5.2. NPC.json

Registra NPCs e Comerciantes que ganham notoriedade na história, fornecendo contexto para a IA e persistência ao mundo.

```
{
  "Thorgrim, o Ferreiro": {
    "tipo": "Comerciante",
    "ultima_vez_visto": "Cidade de Oakhaven",
    "contexto_conhecimento": "O herói o encontrou em sua forja procurando por uma arma melhor.",
    "lembranca_para_ia": "Thorgrim é um anão orgulhoso e um mestre ferreiro.",
    "mercadoria": ["Espada Curta de Ferro", "Cota de Malha"]
  },
  "Elara, a Curandeira": {
    "tipo": "NPC",
    "ultima_vez_visto": "Vilarejo de Pinhal",
    "contexto_conhecimento": "O herói a salvou de um ataque de goblins.",
    "lembranca_para_ia": "Elara é uma curandeira gentil e grata."
  }
}
```

(Exemplo de NPC.json)

5.3. Enemy.json

Guarda os atributos de todos os inimigos base, servindo de referência para a IA criar variações e como um fallback.

```
{
  "Goblin Saqueador": {
    "tipo": "Humanoide",
    "vida_base": 30,
    "ataque_base": 8,
    "defesa_base": 5,
    "velocidade": 12,
    "xp_concedido": 25,
    "skills": {},
    "loot_possivel": { "Adaga de Couro": 0.3 }
  }
}
```

(Exemplo de Enemy.json)

5.4. Skills.json

Armazena todas as informações das skills base do jogo. O save do jogador conterá apenas os nomes das skills que ele conhece, e os detalhes serão buscados neste arquivo.

```
{
  "Tiro Certeiro": {
    "description": "Um disparo focado que busca uma fraqueza na defesa do alvo.",
    "type": "Ataque",
    "uses": 99,
    "effect": { "damage_multiplier": 1.1, "critical_chance": 0.4 }
  }}
```

(Exemplo de Skills.json)

5.5. Classes.json

Armazena todas as informações das Classes do jogo. A Classe que vai atribuir os atributos principais da classe ModelPlayer.

```
"Warrior": {
    "description": "Uma muralha inabalável, focado em resistir e
proteger.",
    "life": 20,
    "base_attack": 8,
    "defense": 12,
    "level_up": {
      "maxlife": 4,
      "base_attack": 1,
      "defense": 2,
      "speed": 0
    },
    "skills_iniciais": [
      "Grito de Batalha",
      "Golpe Pesado"
    ]
  },
```

(Exemplo de Classes.json)

6. Mecânicas Principais

6.1. Sistema de Combate

- **6.1.1 Ordem de Turno (Action Time com Variância):** Um sistema dinâmico onde cada personagem tem um "contador de ação". A cada "tick", o contador sobe com base no atributo **speed + um fator de sorte.** O primeiro a atingir o limiar age, permitindo que personagens mais rápidos tenham mais turnos.

  ➢ **CONCEITO CENTRAL**

  - Cada personagem possui um atributo chamado “Contador de Ação”, que começa em 0.
  - O combate avança em “Ticks”. A cada tick, o contador de cada personagem é incrementado por um valor baseado em seu speed
  - O primeiro personagem a atingir um limiar (ex: 1000 pontos) ganha a ação.

  ➢ **O PAPEL DA VELOCIDADE (“speed”)**

  - A velocidade é o fator principal que determina a frequência das ações. Um Ladino com **speed 20** encherá sua barra de ação muito mais rápido que um Golem com **speed 10**, garantindo mais turnos ao longo do tempo.

  ➢ **O FATOR TENSÃO (“lucky”)**

  - Para evitar que o combate seja puramente matemático e previsível, a cada tick o incremento não é fixo. A fórmula é: `Incremento = speed_total + Fator de Sorte`.
  - O "Fator de Sorte" é um pequeno número aleatório (ex: de -5 a +5), o que significa que um personagem mais lento pode,

ocasionalmente, ter uma "rolagem de sorte" e agir um pouco antes do esperado, mantendo a tensão a cada momento.

➢ **TURNOS MÚLTIPLOS E “OVERFLOW”**

- Após agir, o contador do personagem não é zerado, mas sim **subtraído** do limiar (ex: `contador -= 1000`).
- Isso cria a possibilidade de "overflow". Se um personagem muito rápido atinge 1200 pontos, após agir seu contador será 200, dando-lhe uma enorme vantagem na próxima "corrida" para os 1000. Isso permite que personagens extremamente rápidos ou sob efeito de buffs de velocidade possam, eventualmente, agir duas vezes antes de um oponente mais lento.

➢ **IMPLICAÇÕES ESTRATÉGICAS**

- Este sistema torna o atributo `speed` extremamente valioso e estratégico.
- Skills e itens que manipulam a velocidade (buffs como "Haste" ou debuffs como "Slow") se tornam ferramentas táticas poderosas, capazes de mudar a maré do combate ao alterar a ordem e a frequência dos turnos.

- **6.1.2 Processamento de Skills (Padrão "Despachante"):** A função principal `processar_skill` em `combat_logic.py` não contém a lógica de cada skill. Ela atua como um "despachante", identificando o `tipo` da skill (Ataque, Buff, etc.) e chamando uma função especialista para lidar com ela. Isso evita uma "função deus" e torna o código limpo e expansível.

➢ **O “DESPACHANTE” E OS “ESPECIALISTAS”**

- A função principal, `processar_skill` em `combat_logic.py`, terá uma única e simples responsabilidade: atuar como um **"despachante"**.

- Ela lerá o atributo `"tipo"` da skill que foi usada.
- Com base nesse tipo, ela **delegará** o trabalho para uma **função especialista** (ex: `_processar_tipo_ataque`, `_processar_tipo_buff`).

➢ **FUNÇÕES ESPECIALISTAS**

- Cada tipo de skill terá sua própria função "privada" (ex: `_processar_tipo_ataque`), que contém apenas a lógica necessária para aquele tipo. A função de ataque só sabe sobre dano, a de buff só sabe sobre aplicar efeitos positivos, e assim por diante.

➢ **VANTAGENS DA ARQUITETURA**

- **Código Limpo (Responsabilidade Única):** Cada função faz uma única coisa e a faz bem.
- **Manutenção Fácil:** Precisa ajustar como os debuffs funcionam? Você só precisa mexer na função `_processar_tipo_debuff`, sem risco de quebrar a lógica de ataques.
- **Escalabilidade Perfeita:** Quer adicionar um novo tipo de skill, como "Invocação"? Basta criar uma nova função especialista `_processar_tipo_invocacao` e adicionar uma linha ao "despachante". O resto do sistema permanece intocado.

➢ O Json vai conter o tipo de skill, efeito, nome, usos e uma leve descrição da skill utilizada.

SKILL EFFECT					
TYPE	Attack	Dam_mult	Def_Ignore	Crit_Chance	Burn_Chance
	Buff	Def Multi	Dam_incre	Precision	
	Controll	Freeze_cha	Precision		
	Debuff	Speed_red	Def_multi	Precision	

ATTACK SKILLs

- Damage Multiplier:
  - Nome da Chave: **"damage_multiplier"**
  - Tipo de Valor: **Float**
  - Descrição Mecânica: Multiplica o dano base do strenght do personagem. Um valor de **1.5** significa que a skill causa **150% do dano normal**. Essencial para skills como "Golpe Pesado". Se não presente o valor padrão é **1.0**.
  - Exemplo de Uso: **"damage_multiplier" : 2.0** (Dobra o dano do ataque)

- Defense Ignore:
  - Nome da Chave: **"defense_ignore"**
  - Tipo de Valor: **Float** (0.0 a 1.0)
  - Descrição Mecânica: A porcentagem da defesa do alvo que será ignorada no cálculo de dano. Um valor de **0.5** significa que **50% da defesa do inimigo é ineficaz contra este ataque**. Perfeito para skills de Ladino ou "Tiro Perfurante".
  - Exemplo de Uso: **"defense_ignore" : 0.25** (Ignora 25% da defesa do alvo)

- Critical Chance:
  - Nome da Chave: **"critical_chance"**
  - Tipo de Valor: **Float** (0.0 a 1.0)
  - Descrição Mecânica: Aumenta a chance de um ataque ser um acerto crítico. O valor desta chave é somado à chance de crítico base do personagem. Um acerto crítico normalmente dobra o dano final.
  - Exemplo de Uso: **"critical_chance" : 0.4** (Adicione 40% de chance de crítico)

BUFF SKILLs

- Defense Multiplier:
  - Nome da Chave: **"defense_multiplier"**
  - Tipo de Valor: **Float** (0.0 a 1.0)

- Descrição Mecânica: Multiplica a defesa total do usuário enquanto o buff estiver ativo. Um valor de `1.5` aumenta a defesa em 50%.
- Exemplo de Uso: `"defense_multiplier": 1.5, "turn_duration": 3`

➔ Damage Increase:
- Nome da Chave: "**damage_increase**"
- Tipo de Valor:**Float** (X > 1.0)
- Descrição Mecânica: Multiplica o dano de todos os ataques do usuário enquanto o buff estiver ativo.
- Exemplo de Uso:`"damage_increase": 1.25, "turn_duration": 2` (Aumenta o dano em 25% por 2 turnos)

CONTROL SKILLs

➔ Freeze Chance:
- Nome da Chave: **"freeze_chance"**
- Tipo de Valor: **Float** (0.0 a 1.0)
- Descrição Mecânica: A probabilidade de aplicar o status "Congelado" no alvo, fazendo com que ele perca seu próximo turno. A duração é definida por `duracao_turnos`.
- Exemplo de Uso: `"freeze_chance": 0.7, "turn_duration": 2` (Tem 70% de chance de congelar o inimigo por 2 turnos)

DEBUFF SKILLs

➔ Speed Reduce:
- Nome da Chave: **"speed_reduce"**
- Tipo de Valor: **Float** (0.0 a 1.0)
- Descrição Mecânica: Multiplica a velocidade total do alvo, tornando-o mais lento na ordem de turno do "Action Time". Um valor de `0.8` reduz a velocidade do alvo em **20%**.
- Exemplo de Uso: **"speed_reduce" : 0.8, "turn_duration": 3**

➔ Defense Multiplier:
- Nome da Chave: **"defense_multiplier"**
- Tipo de Valor: **Float** (0.0 a 1.0)

- Descrição Mecânica: Multiplica a defesa total do alvo, tornando-o mais vulnerável a ataques. Um valor de **`0.7`** reduz a defesa do alvo em **30%**.
- Exemplo de Uso: **“defense_multiplier” : 0.5 , “turn_duration”: 3** (Multiplica a defesa por 0.5 ou seja 50% do valor inteiro reduzindo a defesa pela metade)

➔ Burn Chance:

- Nome da Chave: “burn_chance”
- Tipo de Valor: **Float** (0.0 a 1.0)
- Descrição Mecânica: A probabilidade de aplicar o status "Queimando" no alvo. Se bem-sucedido, ele adiciona um efeito de `Debuff` ao alvo com `dano_por_turno`.
- Exemplo de Uso: **“burn_chance” : 0.3** (30% de chance de aplicar queimadura)

SKILL REGULATION

➔ Precision:

- Nome da Chave: **“precision”**
- Tipo de Valor: **Float** (0.0 a 1.0)
- Descrição Mecânica: A precisão base da skill. Um valor de 0.9 significa que a skill tem **90% de chance** de ser um **HIT** ou **SCRATCH.** É o valor principal usado pela função **_calcular_precisao** para determinar o resultado do ataque.
- Exemplo de Uso: **“precision” : 0.85** (85% de chance de acerto)

➔ Effect Duration:

- Nome da Chave: **“turn_duration”**
- Tipo de Valor: **int**
- Descrição Mecânica: Define por quantos turnos um Buff, Debuff ou efeito de Controle permanece ativo no alvo. A cada turno do personagem afetado, esse valor é decrementado em 1, quando chegar a zero o efeito é removido.
- Exemplo de Uso: **“turn_duration”: 3** (O efeito dura por 3 turnos)

- **6.1.3 Cálculo de Dano:** Usa uma fórmula de redução de dano percentual com retornos decrescentes (`Defesa / (Defesa + 100)`), garantindo que a defesa seja sempre útil, mas nunca torne o personagem imune.

➢ **FILOSOFIA DO SISTEMA**

- Em vez de uma simples subtração (`Dano - Defesa`), que pode ser facilmente desbalanceada, usamos um sistema de redução percentual. Isso significa que a defesa reduz uma porcentagem do dano recebido.

➢ **A FÓRMULA CENTRAL**

- 1. Percentual de Redução = Defesa _Total / (Defesa_Total + 100)
- 2. Dano Final = Dano_Bruto * (1 - Percentual de Redução)

➢ **O CONCEITO DE "RETORNOS DECRESCENTES"**

- Esta é a chave para o balanceamento. A fórmula garante que cada ponto de defesa adicional seja um pouco menos eficaz que o anterior.
  - Com **50 de defesa**, um personagem tem `50 / (50 + 100) = 33%` de redução de dano.
  - Com **100 de defesa**, ele tem `100 / (100 + 100) = 50%` de redução.
  - Com **200 de defesa**, ele tem `200 / (200 + 100) = 66%` de redução.
- **Conclusão:** Para ir de 33% para 50% de redução, ele precisou de 50 pontos de defesa. Para ganhar os próximos 16%, ele precisou de mais 100 pontos. Isso evita que um personagem com defesa altíssima se torne imune, mantendo o desafio.

➢ **A REGRA DO DANO MÍNIMO**

- Para evitar a sensação frustrante de um ataque causar "0" de dano, todo golpe bem-sucedido causará pelo menos **1 ponto de dano**, independentemente da defesa do alvo. Isso mantém o ritmo do combate e a sensação de impacto.

6.2. Progressão do Personagem

- **6.2.1 Level Up Personalizado:** O método `level_up` na `PlayerModel` lê os dados de `ganhos_por_nivel` do `classes.json` para aplicar os bônus de atributos específicos da classe do jogador.

- **6.2.2 Sistema de Equipamento Flexível:**
  - O jogador possui um dicionário `equipamento` onde cada chave (ex: "Arma", "Amuleto") é uma lista.
  - Um dicionário `slot_limites` define quantos itens cabem em cada lista (permitindo, por exemplo, 2 anéis).
  - Os atributos totais (`ataque_total`, `defesa_total`) são implementados como **propriedades (`@property`)** que calculam em tempo real a soma dos atributos base com os bônus de todos os itens equipados.

6.3. Mapa e Exploração

6.4. Persistência de Dados (Save/Load)

- **6.3.1 Padrão `to_dict` / `from_dict`:** A classe `PlayerModel` possui um método `to_dict()` para converter seu estado em um dicionário e um método de classe `@classmethod from_dict()` para reconstruir o objeto a partir de um dicionário carregado do JSON.

- **6.3.2 Sistema de Slots:** O jogo apresenta um menu para gerenciar até 4 slots de save, escaneando a pasta `saves/` para mostrar o status de cada um (vazio ou com informações do personagem).

7. Integração com a Inteligência Artificial

A IA atua como um "Mestre de Jogo" unificado, com sua lógica centralizada em `ai_services.py`.

7.1. O "Pacote de Ação"

A função principal da IA, `ai_action_packedge()`, retorna um JSON estruturado chamado "Pacote de Ação". Este pacote informa ao `game_logic.py` tudo o que aconteceu: a `narrativa`, se um `iniciar_combate` foi acionado, os dados do `inimigo` a ser criado, etc.

7.2. Contexto é Rei

A IA recebe um "pacote de contexto" rico a cada chamada, contendo:

- **7.2.1 O estado atual do herói.**
- **7.2.2 Um mapa em ASCII para dar consciência geográfica.**
- **7.2.3 Um resumo da `lore.txt` do jogador.**

7.3. Gerenciamento de Memória da IA

- **7.3.1 Resumo Automático da Lore:** Uma função com gatilho de contagem de tokens (usando `tiktoken`) chama a própria IA para resumir a `lore.txt` quando ela fica muito grande, otimizando custos e mantendo o contexto relevante.

- **7.3.2 NPCs Memoráveis:** Uma "IA Analista" pode ser chamada periodicamente para ler a `lore.txt`, identificar NPCs que se tornaram importantes e salvá-los em um `npcs.json` no save do jogador, criando uma memória de longo prazo.

7.4. Criação Dinâmica de Conteúdo

- **7.4.1 Skills (Epifania):** A IA pode gerar o JSON de uma nova skill com base nas ações recentes do jogador. O código então usa `.update()` para adicionar essa nova skill ao dicionário de skills do objeto `PlayerModel`.

8. Estrutura de Prompt

Esta é a função mais importante. Ela orquestra a narrativa e os eventos do jogo. O segredo aqui é forçar a IA a pensar como um Mestre de Jogo e a responder em um formato de dados rigoroso (o "Pacote de Ação").

8.1. GameMaster: ai_action_packedge()

8.1.1 System Prompt

Este prompt define a **persona** e as **regras inquebráveis** da IA.

```
Você é o Mestre do Jogo (MJ) para o RPG de terminal EUCHRONIA. Sua voz
é descritiva, atmosférica e um pouco misteriosa. Sua tarefa é
interpretar a ação do jogador e decidir o que acontece a seguir,
retornando um "Pacote de Ação" em JSON. Seja criativo, adapte os
```

```
eventos à história e ao local, e siga as regras de formatação
rigorosamente.

### REGRAS DO PACOTE DE AÇÃO (JSON) ###

Você deve retornar APENAS um objeto JSON válido. A estrutura deve ser a
seguinte:

1.  **"narrativa"**: (string) Descreva a cena e o resultado da ação do
jogador de forma imersiva (2-3 parágrafos no máximo).

2.  **"iniciar_combate"**: (boolean) `true` se um combate deve começar,
senão `false`.

3.  **"inimigo"**: (objeto JSON ou null) Se `iniciar_combate` for
`true`, preencha com os dados de um inimigo. Senão, deve ser `null`. A
estrutura do inimigo deve ser:

    * `"nome"`: string (criativo e temático)

    * `"vida"`: int (balanceado para o nível do herói)

    * `"ataque"`: int (balanceado para o nível do herói)

    * `"defesa"`: int (balanceado para o nível do herói)

    * `"velocidade"`: int (entre 5 e 20)

    * `"xp_concedido"`: int

    * `"tipo"`: string (ex: "Fera", "Humanoide", "Morto-vivo")

4.  **"item_encontrado"**: (objeto JSON ou null) Se o jogador encontrar
um item, preencha com seus dados (`"nome"`, `"tipo"`). Senão, `null`
```

8.1.2 User Prompt

Este é o prompt que seu código montará a cada chamada, fornecendo todo o contexto que a IA precisa para tomar uma decisão.

```
# Exemplo de como montar o user_prompt no seu código

user_prompt = f"""

--- CONTEXTO ATUAL ---

Herói: {heroi.nome}, um {heroi.classe_nome} de Nível {heroi.level}.
```

```
Localização no Mapa:
{mapa_com_posicao}

Resumo da História Recente:
{lore_resumo}
---
AÇÃO DO JOGADOR: "{acao_jogador}"
Agora, gere o "Pacote de Ação" resultante em formato JSON, seguindo as regras do sistema.
"""
```

8.2. Gerador de Skills: ai_skill_gen()

Esta função é acionada em momentos de "Epifania". O objetivo é criar uma skill única que pareça uma recompensa pelas ações do jogador.

8.2.1 System Prompt (As Regras do Ferreiro de Habilidades)

```
Você é um mestre de jogo criador de habilidades. Sua tarefa é criar uma nova e única skill para um jogador, baseada em seu estilo de jogo. A skill deve ser balanceada e seguir rigorosamente a estrutura de dados solicitada. Retorne APENAS um objeto JSON.

### REGRAS DO JSON DA SKILL ###
O nome da nova skill deve ser a chave principal do objeto JSON. A estrutura interna deve conter:

1.  **"descricao"**: string (descrição criativa e curta da skill).
```

```
2.  **"tipo"**: string (escolha entre "Ataque", "Buff", "Debuff",
"Controle", "Cura").
3.  **"precisao"**: float (um valor entre 0.7 e 1.0).
4.  **"usos_por_combate"**: int (um valor baixo, entre 1 e 3).
5.  **"efeitos"**: objeto JSON (com chaves que a lógica do jogo
entende, como "multiplicador_dano", "duracao_turnos", "ignora_defesa",
etc.).
```

8.2.2 User Prompt (O Contexto da Epifania)

```
user_prompt = f"""
O herói é da classe: {classe_heroi}.
Contexto de suas ações recentes: {contexto_acao}

Crie uma nova skill para ele, seguindo as regras de formatação JSON do
sistema.
"""
```

8.3. O Escriba da Lore: ai_lore_resume()

Esta é uma função utilitária para manter seu contexto enxuto e otimizado.

8.3.1 System Prompt (As Regras do Historiador)

```
Você é um escriba ancião e mestre contador de histórias. Sua tarefa é
ler a crônica de uma aventura e resumi-la nos seus pontos mais
cruciais.

### REGRAS DO RESUMO ###
1.  Resuma o texto em 5 a 7 pontos principais, em ordem cronológica,
usando o formato de lista numerada.
```

```
2.  **Mantenha** os nomes de personagens importantes, locais-chave e itens mágicos.
3.  O resumo deve preservar o tom épico da aventura e focar nas **decisões e conquistas** do herói.
4.  Seja conciso.
```

8.3.2 User Prompt (O Texto a ser Resumido)

```
user_prompt = f"""
Por favor, resuma a seguinte crônica:


{texto_completo_da_lore}
"""
```

9. Bibliotecas Recomendadas

- **9.1. Essenciais:** `openai`, `python-dotenv` (segurança), `colorama` (cores simples), `pytest` (testes).

- **9.2. Para Elevar o Projeto:**
    - **`9.2.1 rich`**: Para criar interfaces de terminal ricas e bonitas (tabelas, painéis, etc.).
    - **`9.2.2 pydantic`**: Para validar os dados retornados pela IA, tornando o jogo robusto e à prova de falhas.
    - **`9.2.3 tiktoken`**: Para contar tokens com precisão e otimizar os custos da API.