Universidade Federal do ABC
Especialização em Tecnologias e Sistemas de Informação

Kelvyn Bicudo Garcia

INTEGRAÇÃO DE MODELOS DE LINGUAGEM DE GRANDE ESCALA (LLMs) EM JOGOS DIGITAIS PARA A GERAÇÃO DINÂMICA DE NARRATIVAS EMERGENTES

Santo André – SP
2024

Kelvyn Bicudo Garcia

INTEGRAÇÃO DE MODELOS DE LINGUAGEM DE GRANDE ESCALA (LLMs) EM JOGOS DIGITAIS PARA A GERAÇÃO DINÂMICA DE NARRATIVAS EMERGENTES

Monografia apresentada ao Curso de Especialização em Tecnologias e Sistemas de Informação, para a obtenção do grau de Especialista em Tecnologias e Sistemas de Informação.

Orientador: Prof. Dr. Mario Alexandre Gazziro.

Assinatura do Orientador

Santo André – SP
2024

Ficha catalográfica

Garcia, Kelvyn Bicudo

Integração de Modelos de Linguagem de Grande Escala (LLMs) em Jogos Digitais para a Geração Dinâmica de Narrativas Emergentes – Santo André, SP: UFABC, 2024

DEDICATÓRIA

Este trabalho é dedicado a todos(as) que me apoiaram durante minha trajetória no curso TSI, em especial:

Ao Prof. Dr. Mario Alexandre Gazziro, pela orientação, por estar sempre me incentivando e por toda sua dedicação;

À toda equipe do Polo UAB Três Corações, que sempre acolheu todos os alunos do curso com muito carinho e respeito;

Aos meus pais pela paciência, compreensão, apoio e conselhos.

RESUMO

Modelos de Linguagem de Grande Escala, também conhecidos como *LLMs* (do Inglês *Large Language Model*), como os existentes no *ChatGPT* da *OpenAI*, no *Gemini* da *Google*, no Copilot da Microsoft e o *LLaMA* da *Meta AI* têm a capacidade de processar linguagem natural, resolver tarefas básicas e seguir conjuntos de instruções complexos.

A demanda por conteúdo gerado proceduralmente, como mapas, texturas, modelos, personagens, objetos e demais elementos gerados por meio de algoritmos existe desde que os jogos digitais foram inventados, tendo em vista o custo financeiro e o esforço humano necessários para criar manualmente estes recursos.

O presente trabalho tem como objetivo providenciar ampla visão acerca dos desafios, riscos, custos, vantagens e desvantagens decorrentes da integração e utilização de Modelos de Linguagem de Grande Escala nos jogos digitais de forma prática por meio da criação de um sistema que possibilite a comunicação entre o jogador e um personagem de videogame (*NPC*) cujas respostas são produzidas através de um *LLM*, bem como apresentar brevemente o cenário atual em que estas técnicas já estão sendo utilizadas e suas possíveis aplicações futuras.

Palavras-chave: LLM, ChatGPT, IA, Jogos Digitais

ABSTRACT

Large Language Models, also known as LLMs, such as those found in OpenAI's ChatGPT, Google's Gemini, Microsoft's Copilot, Meta AI's LLaMA, have the ability to process natural language, solve basic tasks and follow complex instruction sets.

The demand for procedurally generated content, such as maps, textures, models, characters, objects and other such elements created through algorithms has existed since video games were invented, given the financial cost and human effort required to manually create these assets.

This work aims to provide a broad overview of the challenges, risks, costs, advantages and disadvantages arising from the integration and use of Large Language Models in video games in a practical way by developing a system that enables communication between a player and a video game character (NPC) whose answers are generated through an LLM, as well as briefly present the current landscape in which these techniques are are being used and its potential future applications.

Palavras-chave: LLM, ChatGPT, AI, Video Games

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

AI, IA	–	*Artificial Intelligence*: Inteligência Artificial.
LLM	–	*Large Language Model*: Modelo de Linguagem de Grande Escala.
API	–	*Application Programming Interface*: Interface de Programação de Aplicações.
NPC	–	*Non-Player Character*: Personagem Não Jogável.
JSON	–	*Javascript Object Notation*: Formato de arquivo de padrão aberto para a troca de dados em formato simples, legível a humanos e amplamente utilizado.
UI	–	*User Interface*: Interface de Usuário.
URL	–	*Uniform Resource Locator* ou Localizador Uniforme de Recursos.
HTTP	–	*Hypertext Transfer Protocol:* Protocolo de Transferência de Hipertexto amplamente difundido para comunicação na web.
RPG	–	*Role Playing Game:* jogo de “interpretação” em que os jogadores assumem/interpretam papéis de personagens dentro de uma narrativa.

SUMÁRIO

1. Introdução

Grandes avanços no campo da Inteligência Artificial Generativa nos últimos anos – chamado de primavera da IA (Inteligência Artificial) (Bommasani, 2023) ou de *boom* (explosão) da IA generativa (Newman, 2023) – culminaram no desenvolvimento de ferramentas capazes de gerar conteúdo digital como texto, imagens e vídeo de forma que, até então, só a inteligência humana era capaz de produzir (Carraro, 2023).

Modelos de Linguagem de Grande Escala, também conhecidos como *LLMs* (do Inglês *Large Language Model*), como os existentes no *ChatGPT* da *OpenAI*, no *Gemini* da *Google,* no Copilot da Microsoft e o *LLaMA* da *Meta AI* têm a capacidade de processar linguagem natural, resolver tarefas básicas e seguir conjuntos de instruções complexos (Chang *et al.*, 2024).

Um fator que tem grande influência no resultado obtido na interação do usuário com os *chatbots* e aplicações que integram estes modelos são os *prompts* utilizados, que são instruções ou comandos que o usuário fornece ao modelo, em linguagem natural, para obter o resultado desejado (Interaction Design Foundation, 2023), como por exemplo: “Mostre-me uma receita de bolo de chocolate com nozes”.

Com um conjunto de instruções detalhadas e, providenciando um contexto apropriado, é possível utilizar estes modelos para gerar os mais variados personagens e histórias, possibilitando a construção de narrativas interativas que envolvem o usuário ou jogador de forma muito mais imersiva, pois o enredo pode ser adaptado de forma dinâmica às suas escolhas e interações (Gursesli *et al.*, 2023).

Neste trabalho serão analisados os principais pontos pertinentes à implementação e utilização de *LLMs*, de forma remota através *APIs* (*Application Programming Interface*) ou local (executado diretamente na máquina do usuário). Também será fornecida uma visão geral acerca do emprego atual dessas tecnologias no contexto dos jogos digitais e suas perspectivas para o futuro.

1.1. Justificativa

Uma técnica amplamente utilizada na indústria de jogos digitais há décadas é a geração de conteúdo procedural ou PCG (do inglês Procedural Content Generation), que pode ser definida como “a criação algorítmica de conteúdo de jogos com limitada ou indireta interação do usuário”(Togelius *et al.*, 2011, tradução nossa)[1].

A demanda por conteúdo gerado proceduralmente, isto é, elementos como mapas, texturas, modelos, personagens, objetos etc. gerados por meio de algoritmos surgiu desde que os jogos digitais foram inventados, argumentam (Shaker; Togelius; Nelson, 2016, p. 3), tendo em vista o custo financeiro e o esforço humano necessários para criar manualmente estes recursos.

O gênero *Roguelike* e seus subgêneros baseiam-se em grande parte neste aspecto. Alguns exemplares de demais gêneros como Aventura, Ação, RPG, Estratégia etc. também fazem uso extensivo de geração procedural (Shaker; Togelius; Nelson, 2016, p. 4–5).

Nesse sentido, os Modelos de Linguagem de Larga Escala apresentam potencial quase ilimitado para a criação e o desenvolvimento de narrativas interativas (Yong; Mitchell, 2023), combinando a entrada do jogador (diálogo, interações), o contexto do universo onde os atores estão inseridos (cenário, época, ambientação), e um conjunto de regras bem definidas para garantir que o modelo restrinja suas saídas ao esperado para o contexto em que está inserido.

No entanto, há diversos fatores que dificultam o uso de *LLMs* nos jogos. *LLMs* locais, que rodam na máquina do usuário, requerem placas de vídeo com poder de processamento e armazenamento maior que os jogadores e até mesmo os estúdios de desenvolvimento de jogos têm acesso (Värtinen; Hämäläinen; Guckelsberger, 2024). No caso de *LLMs* remotas, que podem ser acessadas através de APIs, a latência, a necessidade de conexão constante, os custos da API e as restrições impostas aos modelos pelas empresas que os administram são fatores importantes a serem considerados.

[1] [...] *the algorithmical creation of game content with limited or indirect user input.*

Tudo isso, em conjunto com o alto crescimento do interesse do público em geral sobre a Inteligência Artificial Generativa nos últimos anos e seus potenciais impactos sociais e econômicos, evidencia a necessidade de recursos que possibilitem e facilitem o uso adequado dessas ferramentas por profissionais do ramo de desenvolvimento de jogos e seu entendimento pelo público que consome mídia desta natureza.

2. Objetivos

Providenciar ampla visão acerca dos desafios, riscos, custos, vantagens e desvantagens decorrentes da integração e utilização de Modelos de Linguagem de Grande Escala nos jogos digitais de forma prática por meio da criação de um sistema que possibilite a comunicação entre o jogador e um personagem de *videogame* cujas respostas são produzidas através de um *LLM*, bem como apresentar o cenário atual em que estas técnicas já estão sendo utilizadas e suas possíveis aplicações futuras.

3. Metodologia

3.1. Etapas

Este trabalho foi dividido em três etapas:

1) A primeira etapa consiste na revisão da literatura existente a respeito do tema, com o intuito de reunir informações a respeito do tema para embasar e fundamentar o desenvolvimento do trabalho, inclusive sobre a utilização de técnicas e métodos já consolidados com propósitos similares aos que são expostos aqui.

2) Na segunda etapa é apresentado e detalhado o processo de desenvolvimento de um protótipo ou *tech demo* que servirá como demonstração prática do uso da tecnologia no contexto dos jogos digitais. O foco desta demonstração está na interação entre o jogador e um *LLM* através da interface do jogo, simulando uma conversação natural entre o jogador e um personagem ou *NPC* (*Non-Player Character* ou Personagem Não Jogável).

3) Por fim, a terceira e última etapa é composta por uma breve apresentação do cenário atual do uso desta tecnologia, incluindo um exemplo de jogo que já a utiliza e como ela poderá vir a ser utilizada no futuro.

3.2. Desenvolvimento do *tech demo*

A princípio, foi criado um enredo e dois *NPCs* com *backstories* (história de fundo) simples e concisas para facilitar a exposição do potencial dos *LLMs* como ferramenta de criação de narrativas dinâmicas.

Para que o modelo de linguagem interprete corretamente o *NPC* e gere respostas da maneira que é esperada para tal personagem, são necessários ao menos três elementos:

1. Uma descrição do personagem e do universo em que o jogador e o personagem estão inseridos, que pode ser breve ou detalhada, a depender de fatores como:
    - O tipo de *NPC* e sua relevância para a construção da narrativa desejada (protagonista, antagonista, personagem secundário etc.);
    - O nível de detalhamento do universo (*lore*), que está também relacionado ao gênero do jogo (jogos de corrida, por exemplo, costumam ter maior foco em seus elementos interativos em detrimento dos narrativos);
    - O nível de conhecimento do próprio personagem acerca de seu ambiente;
    - Variedade da personalidade e das características físicas do personagem.

2. Um conjunto de instruções bem definidas que servirão de base para as respostas elaboradas pelo modelo. É fundamental que estas instruções descrevam de forma precisa o comportamento esperado do personagem, incluindo o que ele deve ou não dizer e como deve ou não deve se comportar, pois irão guiar a forma como as respostas serão geradas pelo modelo de linguagem. Isso é estabelecido na forma de um *prompt* de sistema – *system prompt*, em Inglês – inserido no início da conversa.

3. O histórico de mensagens trocadas entre o jogador e o personagem. Isto serve para que o personagem tenha uma “memória” da conversa que está tendo com o jogador até então. Por padrão, *LLMs* são tidas como *“stateless”*, o que significa que cada mensagem trocada entre o usuário e o modelo são processadas como uma nova interação. Portanto, para garantir o processamento adequado de informações, é imprescindível gerenciar e enviar toda a conversa junto com cada nova mensagem do usuário.

   Como consequência direta, a quantidade de tokens utilizados aumenta substancialmente com cada nova requisição. Alguns serviços oferecem formas de contornar este problema com soluções como *context caching*, disponível para os modelos *Gemini*, do *Google*.

   Esta funcionalidade permite armazenar temporariamente o contexto da conversa nos servidores da empresa para evitar ter que reenviá-los com muita frequência, podendo reduzir drasticamente a quantidade total de tokens necessários.

   Contudo, o custo de armazenamento (contabilizado por milhão de tokens armazenados por hora) pode ser maior que o de processamento dos tokens adicionais em alguns casos.

Estes três elementos são então combinados em um único *prompt* que é enviado ao modelo de linguagem através da *API* da OpenAI, e a resposta obtida é decodificada e apresentada ao jogador pela interface do jogo, conforme diagrama a seguir:

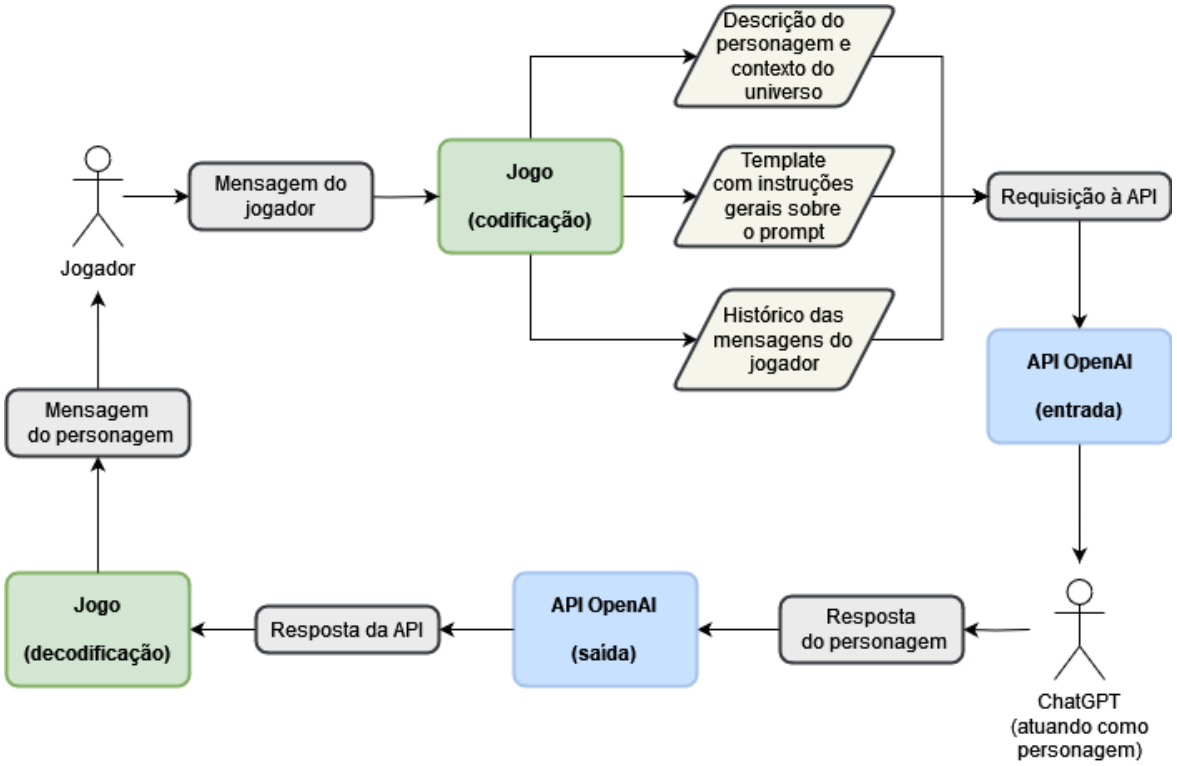

Figura 1: Fluxograma do processo de troca de mensagens

3.3. Tecnologias e ferramentas utilizadas

Para o desenvolvimento deste projeto, foram utilizadas diversas ferramentas e recursos tecnológicos, a saber:

- ***Godot***: *game engine* (ou motor de jogo) de código aberto multiplataforma;
- ***Visual Studio Code***: Editor de código-fonte de código aberto;
- **Humble Fonts**: Pacote de fontes. Autor: *somepx*.
- **DALL-E 3**: Ferramenta de IA para geração de imagens a partir de *prompts*. Usada para gerar os retratos dos personagens e imagens de fundo.
- ***ChatGPT* / *API* da OpenAI**: serviço de *chatbot* online que permite que o usuário interaja facilmente com um modelo de linguagem de grande escala. Este serviço foi escolhido pela facilidade de uso de sua *API*, seu baixo custo de utilização e pelo seu alto grau de popularidade entre o público em geral.
- **Audacity**: Software livre para edição de áudio. Usado para criação e edição de efeitos sonoros.

O código fonte e demais arquivos do projeto estão disponíveis em repositório público acessível por qualquer pessoa (Garcia, 2024)

4. Resultados esperados

Este trabalho almeja reunir informações e apresentar uma visão abrangente sobre os desafios, riscos e benefícios do emprego de ferramentas de Inteligência Artificial Generativa na forma de *LLMs* que possam servir de referência para desenvolvedores de jogos digitais independentes que tenham interesse em integrar um sistema similar em seus projetos.

O *tech demo* desenvolvido servirá como exemplo de abordagem do emprego da tecnologia e, por meio de sua análise, serão elencados os desafios inerentes à sua implementação no contexto dos jogos digitais. Para este fim, serão utilizadas ferramentas apropriadas e bastante disseminadas na indústria para o desenvolvimento de jogos.

Por fim, espera-se que os resultados obtidos demonstrem como esta tecnologia tem o potencial de impactar significativamente e positivamente a indústria moderna de jogos, como já acontece em várias outras (Ooi *et al.*, 2023).

5. Resultados obtidos

Utilizando o motor de jogos gratuito e de código aberto *Godot Engine,* foi desenvolvido um *tech demo* (demonstração prática da tecnologia) para servir de exemplo de aplicação prática do uso de *LLMs* como ferramenta para a criação de narrativas dinâmicas assim como uma forma de explorar as dificuldades inerentes à integração desta tecnologia no contexto dos jogos digitais.

O projeto consiste em uma interface de *chat* que foca exclusivamente na interação entre o jogador e um *NPC* cujo diálogo é controlado inteiramente pelo modelo de linguagem escolhido, que pode ser qualquer um dentre os disponibilizados pela *OpenAI* em sua *API*.

A escolha da *OpenAI* como provedor de serviço para esta finalidade, dentre os muitos que estão disponíveis atualmente, se dá por conta de três fatores. O primeiro fator é a facilidade de uso da ferramenta e sua API. O segundo é sua estrutura de preços acessível para os modelos mais simples: em julho de 2024, o modelo GPT-4o-mini custava apenas $0,15 por milhão de tokens de entrada (enviados) e $0,60 por milhão de tokens de saída (recebidos) (OpenAI, [*s. d.*]). E por último, a popularidade do serviço, que ultrapassou a marca de 200 milhões de usuários mensais em junho de 2023 (Moore, 2023).

Para este *tech demo*, foi criada uma narrativa curta em volta de uma pequena vila fictícia típica de *RPGs* estilo *D&D* (*Dungeons and Dragons*) chamada “Serraluz”, onde o jogador, um viajante, chega para descansar após uma longa e árdua viagem na estrada. Também foram criados dois personagens com perfil típico de narrativas deste gênero para atuarem como NPCs com quem o jogador pode interagir através do *chat*:

- Lirian “Chama Branda”, um homem perspicaz e com uma longa história que decidiu abandonar sua vida de aventureiro e abrir uma taverna após um evento traumático que transformou sua vida;
- Graldor Pedratorvo, um anão de barba comprida que trabalha como o ferreiro da pequena vila e adora ouvir e contar histórias de batalhas e lendas antigas.

Através do uso da classe *Resource* (recurso) da Godot Engine, foi possível elaborar uma subclasse *Character* que simplifica consideravelmente a criação de

personagens para uso no projeto: basta criar um novo recurso com esta subclasse e definir suas propriedades (nome, ícone, descrição, instruções etc.).

Através da única função desta subclasse, get_prompt(), suas propriedades nome, descrição e instruções são concatenadas e formatadas em um único *prompt* usado para instruir o modelo de linguagem escolhido sobre como interpretar tal personagem:

```
**NOME DO PERSONAGEM**
Graldor Pedratorvo

**DESCRICÃO DO PERSONAGEM**
Graldor Pedratorvo é um anão robusto e de barba longa,
sempre vestindo um avental de couro surrado e com um olhar
um tanto compenetrado. Ele é o ferreiro da pequena vila de
Serraluz, conhecido por suas habilidades excepcionais na
forja de armas e armaduras. Apesar de seu aspecto sério
quando está trabalhando, Graldor é amigável e gosta de uma
boa conversa, especialmente se envolver histórias de
batalhas e lendas antigas. Ele está sempre disposto a
ajudar os viajantes, oferecendo conselhos sábios e suas
habilidades como ferreiro exímio.

**INSTRUÇÕES**
Você assumirá o papel de Graldor Pedratorvo, um personagem
de um RPG estilo D&D. O jogador irá interagir com Graldor e
espera que suas respotas sejam coerentes com o perfil do
personagem e o universo em que estão inseridos. É
impreterível que suas respostas sejam restritas a este
contexto para que o jogador sinta-se imerso na história.

Serviços oferecidos por Graldor Pedratorvo:
  Afiar uma arma: 5 moedas de ouro
  Reparar uma peça de armadura danificada: 10 moedas de
ouro
  Forjar uma arma simples (espada, machado, martelo): 20
moedas de ouro
  Forjar uma peça de armadura leve (cota de malha, elmo):
30 moedas de ouro

Além as instruções acima, atente-se para estas observações
em especial:
1. Não negociar preços nem fazer fiado, se aplicável;
2. Não abandonar seu local nem acompanhar nenhum viajante;
3. Oferecer conselhos quando solicitado;
4. Não falar sobre assuntos que fogem do contexto do
universo da história;
5. Prezar pera originalidade, evitando repetições.
```

Figura 2: Exemplo de *prompt* que é usado para definir o comportamento do modelo

Ao iniciar o demo, é apresentado ao jogador uma tela com um texto introdutório para contextualização. Em seguida, é preciso fazer uma escolha: ir até a taverna, iniciando um *chat* com o taverneiro; ou ir até a forja, iniciando um *chat* com o ferreiro da vila.

Figura 3: Tela inicial do demo com o texto de introdução e botões de ação

Figura 4: Exemplo de conversa com o taverneiro

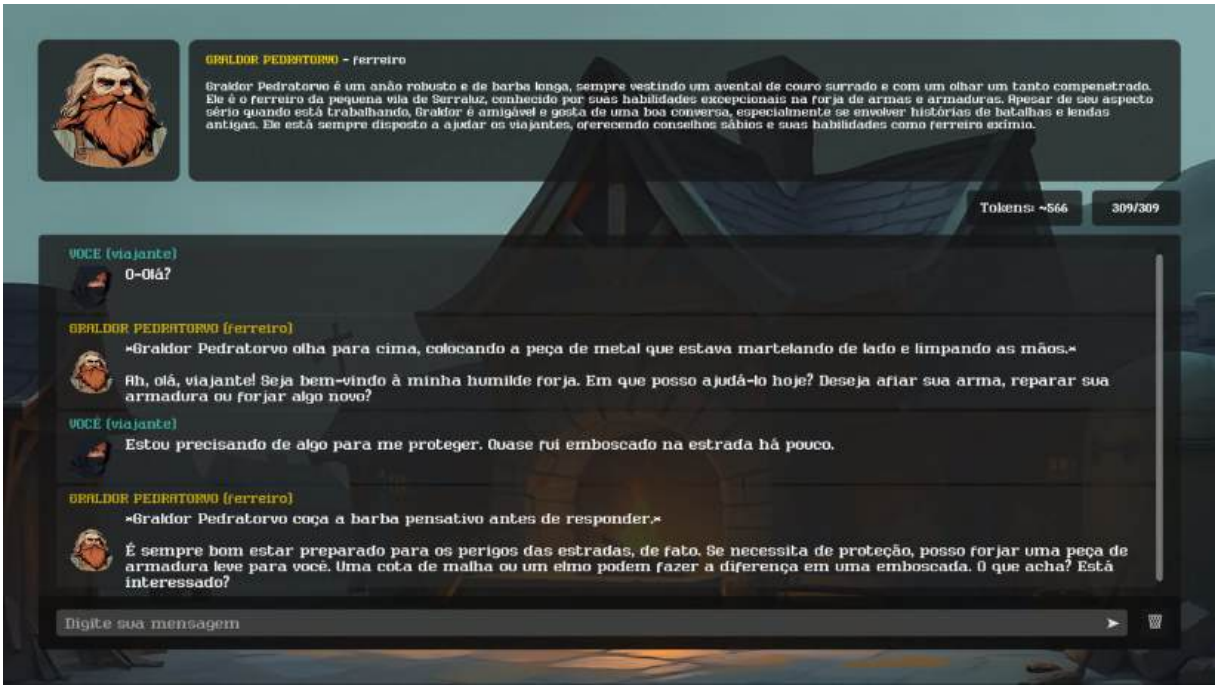

Figura 5: Exemplo de conversa com o ferreiro

5.1. Estrutura da Interface

A interface do *chat* foi construída inteiramente no editor da *Godot Engine*, que possibilita a criação de cenas de um jogo a partir de blocos fundamentais chamados de nós (*nodes*), que são combinados e organizados em uma hierarquia com outros nós dos mais variados tipos e com funcionalidades distintas para compor a estrutura da aplicação (Ranaweera; Mahmoud, 2024).

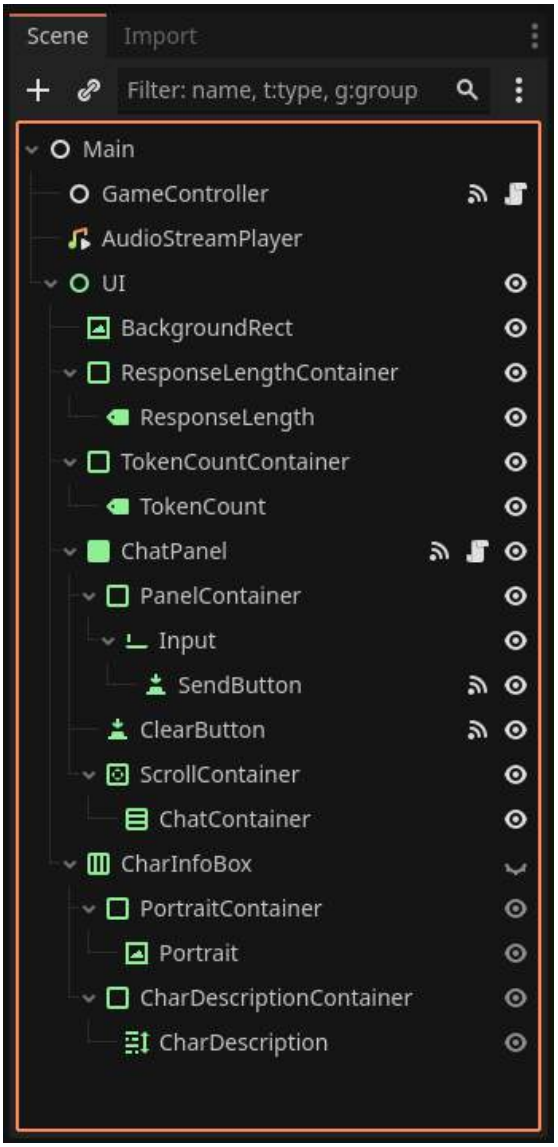

Figura 6: Estrutura dos nós contendo os elementos gráficos que compõem o painel do chat da interface

Esta arquitetura característica da filosofia de design adotada pela *Godot Engine* - chamada de composição - permite que os nós sejam compactados em objetos chamados de "cenas" que podem ser reutilizados em outras cenas quantas vezes forem necessárias, contribuindo para manutenibilidade dos projetos.

Neste projeto, essa arquitetura foi amplamente explorada por se prestar especialmente bem para o propósito almejado, visto que um *chat* é geralmente composto por um grande número de mensagens cujo conteúdo varia, porém sua forma mantém-se a mesma.

Dentro da cena principal *"*main.tscn*"*, com exceção dos nós *GameController* e *AudioStreamPlayer* - usados para controlar a lógica geral do jogo e para emitir efeitos sonoros, respectivamente - todos os nós são do tipo *Control*, indicados pela sua cor verde claro, usados para gerir os elementos de interface conforme especificado abaixo (com seu respectivo tipo indicado entre parênteses) na forma em que aparecem na hierarquia:

- **UI** (Control): Nó raiz para toda a interface do usuário (*UI*);
    - **BackgroundRect** (TextureRect): Imagem de fundo que é substituída pela imagem de fundo adequada ao ambiente de cada personagem;
    - **ResponseLengthContainer** (PanelContainer): Assim como muitos outros elementos do tipo "*Container*", a função principal deste nó é conceder certas características especialmente relacionadas à posição, escala e rotação de seus nós filhos. O PanelContainer também possibilita a customização de seu plano de fundo, ideal para criar um efeito sombreado de maior destaque;
        - **ResponseLength** (Label): As labels são rótulos que contêm textos simples, com opções de formatação limitadas. Esta label é responsável por conter o texto que indica a relação entre a quantidade de caracteres atualmente exibidos e a quantidade de caracteres recebidos como resposta da API na última mensagem;
    - **TokenCountContainer** (PanelContainer): Container para a label TokenCount;
        - **TokenCount** (Label): Indica a quantidade aproximada de tokens utilizados (enviados + recebidos);
    - **ChatPanel** (Panel): Nó raiz da interface de chat. Contém o script ChatController.gd que é responsável por gerenciar toda a conversa do chat, incluindo diversas propriedades para definir a velocidade da

animação do texto, o som da tecla que é tocado junto com a animação, o texto de introdução que é apresentado no início do jogo etc;

- **PanelContainer** (PanelContainer): Container para Input e SendButton
    - **Input** (LineEdit): Caixa de texto onde o jogador digita suas mensagens;
    - **SendButton** (Button): Botão para o jogador enviar sua mensagem;
- **ClearButton** (Button): Botão para limpar o chat e iniciar nova conversa com o personagem;
- **ScrollContainer** (ScrollContainer): Container para habilitar a funcionalidade de rolagem do conteúdo do chat;
    - **ChatContainer** (VBoxContainer): Container onde são adicionadas as mensagens. Este tipo de nó automaticamente dimensiona seus nós filhos em uma lista vertical ao serem adicionados à hierarquia, ideal para uma interface de chat;
- **CharInfoBox** (HBoxContainer): Container para o retrato e a descrição do personagem do chat;
    - **PortraitContainer** (PanelContainer): Container para o retrato do personagem;
        - **Portrait** (TextureRect): Imagem de retrato do personagem;
    - **CharDescriptionContainer** (PanelContainer): Container para a descrição do personagem;
        - **CharDescription** (RichTextLabel): rótulo ou label de texto enriquecido que permite opções de formatação mais avançadas. Neste caso, é usado para dar destaque ao nome e papel do personagem em conjunto com sua descrição sem a necessidade de adicionar mais nós à hierarquia.

Outras duas cenas foram criadas, com propósitos similares: “system_message_container.tscn” e “message_container.tscn”. Ambas servem para definir a aparência e o formato (*template*) das mensagens que serão inseridas no chat. A principal diferença entre elas é que “message_container” é usada como o template das mensagens dos personagens e do jogador e portanto inclui um ícone e o nome do personagem/jogador, elementos ausentes no outro template, que é usado para mensagens que não advém de um personagem imerso no universo da história, como um narrador, por exemplo.

5.2. Estrutura do código-fonte

Para a edição do código-fonte do projeto foi usado o *Visual Studio Code* (*VS Code*) que é um editor de código fonte de código aberto desenvolvido pela *Microsoft*. A linguagem de programação empregada é a *GDScript* que é uma linguagem de alto nível de abstração, imperativa e orientada a objetos, desenvolvida e mantida pelos próprios desenvolvedores da *Godot Engine*.

O código-fonte deste *tech demo* tem dois componentes principais: o *GameController* e o *ChatController*. O *GameController* é um componente comum em muitos jogos, e embora sua nomenclatura possa variar de projeto para projeto, ele é geralmente responsável por gerenciar de forma abrangente os aspectos gerais de um jogo. Aqui, ele gerencia as requisições à *API* da *OpenAI* e o envio das mensagens do *NPC* para a interface, gerenciada pelo *ChatController*. Por sua vez, o *ChatController* é encarregado de gerenciar o envio das mensagens do jogador ao *GameController* e a exibição delas na interface gráfica.

Um mecanismo da *Godot Engine* bastante explorado neste projeto foram os sinais, que permitem a comunicação entre objetos baseada em eventos: um objeto emite o sinal e todos os objetos conectados a este sinal recebem este evento e podem reagir apropriadamente.

Devido ao tamanho do código-fonte e à quantidade de arquivos necessários ao projeto, apenas trechos relevantes serão expostos aqui. O código-fonte completo e demais arquivos associados ao projeto podem ser encontrados no repositório público na plataforma github (Garcia, 2024)

5.2.1. Character.gd

Este componente do tipo *Resource* tem a função de armazenar diversos tipos de dados referentes a um personagem, como seu nome, ícone, papel (viajante, ferreiro, taverneiro etc.), descrição, plano de fundo e instruções para o modelo de linguagem. Também inclui um método simples chamado “get_prompt()“ que retorna o nome do personagem, sua descrição e seu conjunto de instruções, formatados numa única string pronta para ser passada a um modelo.

5.2.2. ChatController.gd

A classe ChatController reúne todas as propriedades e os métodos relacionados à exibição de mensagens na interface do chat. Algumas de suas principais propriedades e variáveis (e seus respectivos tipos, em parênteses) são:

- **player** (Character): Objeto que contém os dados do jogador;
- **characters** (Array[Character]): Lista de personagens com os quais o jogador pode iniciar um chat;
- **chars_per_second** (int): Determina a quantidade de caracteres que são exibidos por segundo e, por conseguinte, a velocidade da animação do texto de resposta dos personagens;
- **default_placeholder_text** (String): Texto que é mostrado por padrão na caixa de texto onde o jogador digita suas mensagens, como “Digite sua mensagem”.
- **awaiting_response_placeholder_text** (String): Texto que é mostrado na caixa de texto onde o jogador digita suas mensagens quando o NPC está escrevendo sua resposta, como “Aguardando resposta”.
- **intro_message** (String): Texto introdutório exibido ao iniciar o jogo;
- **link_template** (String): Trecho de *BBCode* usado para criar links nas mensagens que são usados para iniciar uma conversa com um personagem;
- **messages** (Array): Vetor que contém todos as mensagens trocadas entre o jogador e o personagem, no formato aceito pela *API*;
- **text_stream_chat_msg** (Node): Mensagem do chat cujo texto será texto será atualizado via *text streaming*;
- **current_character** (Character): Referência ao personagem atual com quem o jogador está conversando;

O ChatController faz uso dos seguintes sinais:

- **player_message_submitted** (msg: String, to: Character): Sinal emitido quando o jogador envia uma mensagem. Envia como parâmetro a mensagem do jogador (msg) e para qual personagem (to) é direcionada;
- **typing_started**: Sinal emitido quando a animação de digitação do texto é iniciada;
- **typing_char_added**: Sinal emitido toda vez que um novo caractere é exibido na animação de digitação de texto;
- **typing_char_finished**: Sinal emitido quando a animação de digitação do texto é finalizada;

Entre as principais funções da classe, estão:

ChatController.gd

```
func new_message(role: String, msg: String) -> Dictionary:
  return {"role": role, "content": msg}
```

Figura 7: função new_message(): Retorna uma string no formato de mensagem usado pela API

ChatController.gd

```
func get_role(character: Character) -> String:
  if character == player:
    return "user"
  if characters.has(character):
    return "assistant"
  return "system"
```

Figura 8: função get_role(): Retorna o parâmetro “role” da API com base no personagem (jogador, NPC ou sistema)

Figura 9: função clear_chat(): Limpa o chat, permitindo iniciar uma nova conversa com o mesmo personagem

Figura 10: função add_chat_message(): Adiciona nova mensagem ao chat

ChatController.gd

```
func load_system_prompt(character: Character) -> void:
  messages.append(new_message("system", character.get_prompt()))
```

Figura 11: função load_system_prompt(): Carrega o prompt de sistema (que contém a descrição do personagem, do universo em que está inserido e seu conjunto específico de instruções) no vetor “messages” para posteriormente ser enviado à API junto com a mensagem do jogador

ChatController.gd

```
func animate_text(chat_msg: Node) -> void:
  toggle_input(false)
  typing_started.emit()

  var text_field: RichTextLabel =
chat_msg.get_node("VBoxContainer/HBoxContainer/
CharacterMessage")
  while text_field.visible_characters <
text_field.text.length():
    text_field.visible_characters += 1
    typing_char_added.emit()
    await get_tree().create_timer(1.0 /
chars_per_second).timeout

  typing_finished.emit()
```

Figura 12: função animate_text(): Cria uma animação para o texto, exibindo um caractere de cada vez e emitindo um som levemente distinto a cada “tecla”

ChatController.gd

```
func start_new_chat(character: Character) -> void:
  clear_chat()
  set_background_image(character.background)
  load_system_prompt(character)
  load_info_box(character)
  current_character = character

  toggle_input(true)
  print("Novo chat iniciado com '%s'. Carregando prompt." %
character.name)
```

Figura 13: função start_new_chat(): Inicia novo chat com um personagem

ChatController.gd

```
func load_info_box(character: Character) -> void:
  var portrait :=
char_info_box.get_node("PortraitContainer/Portrait")
  var description :=
char_info_box.get_node("CharDescriptionContainer/
CharDescription")

  if character == null:
    portrait.texture = null
    description.text = ""
    char_info_box.visible = false
    return

  char_info_box.visible = true
  var new_text := "[color=gold]%s[/color] - %s\n\n%s" %
[character.name.to_upper(), character.role,
character.description]

  portrait.texture = character.icon
  description.text = new_text
```

Figura 14: função load_info_box(): Carrega os dados do personagem (nome, papel, descrição e ícone) em um painel acima do chat

5.2.3. GameController.gd

O GameController tem como principal finalidade gerenciar as requisições à *API* da *OpenAI* e suas respostas, que são processadas e formatadas para serem exibidas corretamente na interface. A classe faz uso de apenas uma propriedade, quatro sinais e duas variáveis:

- **max_retries** (int): Esta propriedade define a quantidade de tentativas de reconexão ao host (*API* da *OpenAI*) em caso de falha ou erro;
- **endpoint** (Dictionary): Esta variável mantém referência ao *endpoint* atual, aqui entendido como o conjunto de informações necessárias para fazer requisições à *API*, como a *URL*, o modelo escolhido e seus parâmetros, e o cabeçalho da requisição que contém a chave de *API* a ser utilizada;
- **client** (HTTPClient): Classe da *Godot Engine* que providencia diversos métodos para trabalhar com requisições *HTTP*;
- **request_sent**(): Sinal emitido logo após uma requisição ser enviada à API;
- **text_stream_started**(): Sinal emitido ao iniciar a transmissão de texto – o *text streaming* é uma funcionalidade da *API* da *OpenAI* que permite que a resposta seja enviada, em *chunks*, enquanto ela ainda está sendo gerada;
- **text_stream_data_received**(): Sinal emitido ao receber uma parcela da resposta (*chunk*) via *text streaming*;
- **text_stream_finished**(): Sinal emitido ao final da transmissão de texto;

Algumas das principais funções da classe são:

GameController.gd

```
func setup_client(retry_count: int = 0) -> HTTPClient:
  var new_client := HTTPClient.new()
  var host: String = endpoint["base_url"]
  print("Estabelecendo conexão com host '%s'..." % host)
  var error_code := new_client.connect_to_host(host)

  if error_code != OK and retry_count < max_retries:
    return await check_connection(new_client, true,
retry_count)

  while (new_client.get_status() ==
HTTPClient.STATUS_CONNECTING
  or new_client.get_status() ==
HTTPClient.STATUS_RESOLVING):
    new_client.poll()
    await get_tree().process_frame

  if new_client.get_status() != HTTPClient.STATUS_CONNECTED
and retry_count < max_retries:
    return await check_connection(new_client, true,
retry_count)

  print("Conectado")
  return new_client
```

Figura 15: função setup_client(): Configura o cliente *HTTP* para estabelecer comunicação com o servidor

```
func check_connection(some_client: HTTPClient, retry: bool = true, retry_count:
int = 0) -> HTTPClient:
  some_client.poll()
  var status := some_client.get_status()
  print("Status da conexão: %s" % status)

  if status == HTTPClient.STATUS_CONNECTED:
    return some_client

  if !retry:
    print("Conexão não estabelecida (retry: false).")
    return some_client

  print("Tentando restabecer conexão...")
  return await setup_client(retry_count + 1)
```

Figura 16: função check_connection(): Verifica e restabelece a conexão com o servidor se necessário

GameController.gd

```
func send_request_stream(model: String, msg_history: Array
= []) -> void:
  client = await check_connection(client)
  endpoint["params"]["model"] = model
  endpoint["params"]["messages"] = msg_history
  endpoint["params"]["stream"] = true

  var approx_token_count := str(msg_history).length() / 4.0

  print("Enviando requisição. Contexto (~%d tokens):\n%s" %
[approx_token_count, "\n".join(msg_history)])

  var error_code := client.request(
    HTTPClient.METHOD_POST,
    endpoint["chat_endpoint"],
    endpoint["headers"],
    JSON.stringify(endpoint["params"]),
  )

  assert(error_code == OK)
  print("Requisição enviada. Aguardando resposta...")
  request_sent.emit()
  handle_server_response()
```

Figura 17: função send_request_stream(): Envia uma requisição à *API*

GameController.gd

```
func handle_server_response() -> void:
  while (client.get_status() ==
HTTPClient.STATUS_REQUESTING):
    client.poll()
    await get_tree().process_frame

  assert(client.get_status() == HTTPClient.STATUS_BODY
    or client.get_status() == HTTPClient.STATUS_CONNECTED)

  if !client.has_response():
    print("Não foi possível obter uma resposta.")
    return

  print("Resposta recebida. Aguardando dados...")
  stream_server_response()
```

Figura 18: função handle_server_response(): aguarda até o servidor fornecer uma resposta

GameController.gd

```gdscript
func stream_server_response() -> void:
  var read_buffer := PackedByteArray()
  var has_started := false
  var content := ""

  while client.get_status() == HTTPClient.STATUS_BODY:
    client.poll()
    var chunk := client.read_response_body_chunk()
    if chunk.size() == 0:
      await get_tree().process_frame
      continue

    if !has_started:
      has_started = true
      text_stream_started.emit()
      print("Iniciando streaming de dados...")

    read_buffer += chunk
    var chunk_text := chunk.get_string_from_utf8()
    content += parse_chunk(chunk_text)
    text_stream_data_received.emit(parse_chunk(chunk_text))

    # print(chunk_text)

  print("Conteúdo completo da resposta:\n%s" % content)
  print("Bytes recebidos: ", read_buffer.size())
  text_stream_finished.emit(content)
```

Figura 19: função stream_server_response(): Dá início e gerencia a transmissão de texto, emitindo os sinais apropriados para que os objetos conectados possam reagir adequadamente

GameController.gd

```gdscript
func parse_chunk(chunk_text: String) -> String:
  var output := ""
  var lines := chunk_text.split("\n")
  for line in lines:
    if !line.begins_with("data: "):
      continue

    line = line.replace("data: ", "")
    if line == "[DONE]":
      break

    var json: Dictionary = JSON.parse_string(line)
    if !json["choices"][0]["delta"].has("content"):
      continue

    output += json["choices"][0]["delta"]["content"]

  return output
```

Figura 20: função parse_chunk(): Processa o *chunk* da resposta e retorna o conteúdo da mensagem

5.3. Integração com demais elementos do jogo

Apesar de não ter sido o foco deste projeto, há diversos elementos que podem ser incorporados ao conjunto de *prompts* e trocas de mensagens entre o jogador e o personagem para enriquecer a narrativa a ser construída e tornar a integração do modelo de linguagem ao jogo ainda mais proeminente e atrativa, como, por exemplo:

- Inclusão/atualização do contexto do universo no prompt para refletir as ações e conquistas obtidas pelo jogador ao longo de sua jornada;
- Nível de afinidade ou disposição do personagem em relação ao jogador, especialmente após este realizar alguma ação contra ou a favor do personagem, de modo que o comportamento do *NPC* reflita seu nível de relacionamento com o jogador;
- Interação com o inventário do jogador: é possível fazer com que o *NPC* adicione ou remova itens do inventário do jogador através de um prompt que instrua o modelo a produzir, por exemplo, um arquivo *json* (formato aberto bastante utilizado para armazenamento e troca de dados entre aplicações) como o abaixo:

exemplo.json

```
{
  "tipo": "arma",
  "nome": "Espada de Prata",
  "dano": 10,
  "valor": 100
}
```

Figura 21: Exemplo de arquivo *json* com dados variados para um item de um jogo

No entanto, isso implicaria incluir em cada prompt um controle rígido sobre os itens que podem ser transferidos e uma série de dados do *NPC* e do jogador, como recursos monetários e espaço de armazenamento disponíveis para ambos, o que pode aumentar significativamente a quantidade de tokens trocados em cada requisição, tanto de entrada como saída.

Uma consequência clara disso, quando implementado por meio da utilização de um serviço online, é o aumento nos custos de utilização do serviço, que são geralmente precificados com base na quantidade de tokens utilizados. Mesmo no caso de uma abordagem local, onde o *LLM* é executado na própria máquina do jogador, há de se considerar o tamanho máximo do contexto do modelo, que consiste em um limite na quantidade de tokens que podem ser processados.

Ademais, seria ainda necessário processar e interpretar adequadamente estes dados (*parsing*) para que possam então ser apresentados na interface do jogo. Para isto, é fundamental que os dados estejam sempre no formato especificado e não contenham valores inválidos ou ausentes.

Dada a propensão dos modelos de linguagem a produzir alucinações - termo dado ao texto gerado que é incoerente, não-fiel à fonte e indesejado (Ooi *et al.*, 2023) - também faz-se necessário identificar e tratar estes casos, como, por exemplo, por meio de rejeição da saída gerada pelo modelo ou pela geração de novo conteúdo.

5.4. Aplicações presentes e perspectiva para o futuro

Existem no mercado alguns exemplos de jogos digitais que fazem o uso de *LLMs* para a geração de conteúdo dinâmico, porém sua adoção ainda é restrita a certos nichos, devido, possivelmente, a fatores como:

- Limitação de *hardware*: LLMs locais fazem uso extensivo das capacidades gráficas do sistema, gerando conflito ao disputar esses recursos com os demais elementos do jogo;
- Custos de operação: no caso de serviços online como o *ChatGPT*, por mais que o custo individual pela geração de conteúdo para cada requisição seja baixo, o contexto necessário que deve ser enviado junto com cada mensagem cresce rapidamente, o que representa grande obstáculo em termos de escalabilidade;
- Questões éticas/legais: o rápido avanço dessa tecnologia, seu potencial e seus riscos trouxeram à tona questionamentos sobre a devida utilização de ferramentas dessa natureza que ainda precisam ser discutidas a fundo.

Um exemplo de jogo cuja mecânica central é em torno do uso de um *LLM* para a geração de conteúdo é *Verbal Verdict*[2], em que o jogador assume o papel de um detetive que deve interrogar suspeitos e acumular provas até que consiga resolver os casos sob seus cuidados. Este processo se dá mediante diálogo entre o jogador e os personagens usando linguagem natural. Os personagens, através de um *LLM*, respondem adequadamente aos questionamentos de acordo com seu perfil, criando narrativas dinâmicas ao longo do jogo.

Figura 22: Captura de tela do trailer do jogo com o título

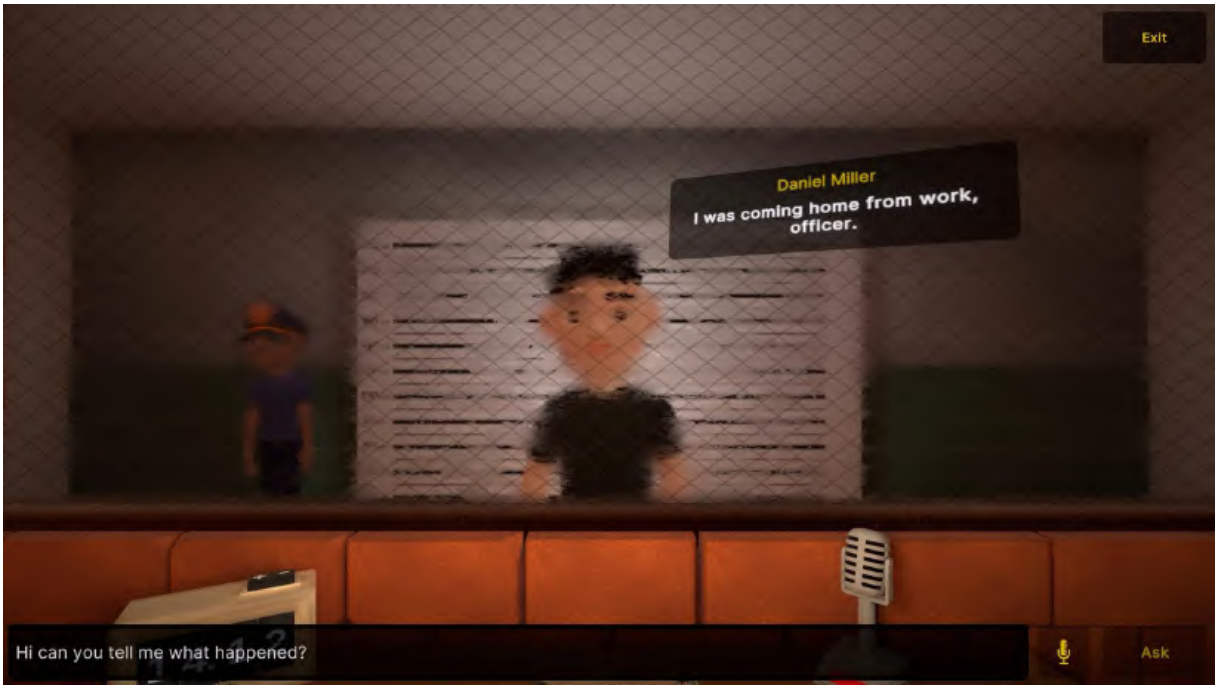

Figura 23: Captura de tela do trailer do jogo, mostrando o diálogo entre o jogador e um personagem

[2] *Verbal Verdict* é um jogo disponível para PCs que atualmente encontra-se disponível na plataforma steam: https://store.steampowered.com/app/2778780/Verbal_Verdict/

6. Conclusões

Através deste projeto desenvolvido como *tech demo,* foi possível comprovar a capacidade dos *LLMs* e seu potencial em termos de geração de narrativas emergentes, que são criadas e adaptadas a partir de poucas linhas de contexto prévio e da interação com o jogador, providenciando-lhe uma experiência imersiva com elevado grau de liberdade e dinamicidade.

Com apenas um parágrafo de um simples *background* (descrição ou histórico) de um personagem e algumas instruções para o sistema sobre como este deveria agir, o modelo de linguagem escolhido (*GPT-3.5-turbo*) foi capaz de produzir elementos narrativos que não haviam sido especificados anteriormente, como a invenção de uma localização supostamente repleta de perigos e recompensas, aptos a serem descobertos pelo jogador (vide figura 4).

É plausível que, com maiores avanços no campo da Inteligência Artificial Generativa, tais obstáculos sejam contornados ou minimizados e, consequentemente, a disseminação do uso da tecnologia aqui apresentada torne-se cada vez mais presente em jogos digitais dos mais variados gêneros e estilos.

7. Referências

BOMMASANI, Rishi. **AI Spring? Four Takeaways from Major Releases in Foundation Models**. [*S. l.*], 2023. Disponível em: https://hai.stanford.edu/news/ai-spring-four-takeaways-major-releases-foundation-models. Acesso em: 7 abr. 2024.

CARRARO, Fabrício. **Inteligência Artificial e ChatGPT: Da revolução dos modelos de IA generativa à Engenharia de Prompt**. [*S. l.*]: Casa do Código, 2023.

CHANG, Yupeng *et al.* A Survey on Evaluation of Large Language Models. **ACM Transactions on Intelligent Systems and Technology**, [*s. l.*], v. 15, n. 3, p. 39:1-39:45, 2024.

GARCIA, Kelvyn B. **A Chegada em Serraluz - Demo de RPG baseado em Texto com integração com ChatGPT**. [*S. l.: s. n.*], 2024 [2024]. Disponível em: https://github.com/kelbg/serraluz. Acesso em: 5 set. 2024.

GURSESLI, Mustafa Can *et al.* The Chronicles of ChatGPT: Generating and Evaluating Visual Novel Narratives on Climate Change Through ChatGPT. *In*: THE CHRONICLES OF CHATGPT, 2023, Cham. (Lissa Holloway-Attaway & John T. Murray, Org.)**Interactive Storytelling**. Cham: Springer Nature Switzerland, 2023. p. 181–194.

INTERACTION DESIGN FOUNDATION. **What Are AI Prompts?**. [*S. l.*], 2023. Disponível em: https://www.interaction-design.org/literature/topics/ai-prompts. Acesso em: 7 abr. 2024.

MOORE, Olivia. **How Are Consumers Using Generative AI?**. [*S. l.*], 2023. Disponível em: https://a16z.com/how-are-consumers-using-generative-ai/. Acesso em: 25 jul. 2024.

NEWMAN, Daniel. **Exploring The Ins And Outs Of The Generative AI Boom**. [*S. l.*], 2023. Disponível em: https://www.forbes.com/sites/danielnewman/2023/03/14/exploring-the-ins-and-outs-of-the-generative-ai-boom/. Acesso em: 7 abr. 2024.

OOI, Keng-Boon *et al.* The Potential of Generative Artificial Intelligence Across Disciplines: Perspectives and Future Directions. **Journal of Computer Information Systems**, [*s. l.*], v. 0, n. 0, p. 1–32, 2023.

OPENAI. **OpenAI API Pricing**. [*S. l.*], [*s. d.*]. Disponível em: https://openai.com/api/pricing/. Acesso em: 25 jul. 2024.

RANAWEERA, Mahesh; MAHMOUD, Qusay H. Deep Reinforcement Learning with Godot Game Engine. **Electronics**, [*s. l.*], v. 13, n. 5, p. 985, 2024.

SHAKER, Noor; TOGELIUS, Julian; NELSON, Mark J. **Procedural Content Generation in Games**. Cham: Springer International Publishing, 2016. (Computational Synthesis and Creative Systems). Disponível em: http://link.springer.com/10.1007/978-3-319-42716-4. Acesso em: 21 abr. 2024.

TOGELIUS, Julian *et al.* What is procedural content generation? Mario on the borderline. *In*: WHAT IS PROCEDURAL CONTENT GENERATION?, 2011, New York, NY, USA. **Proceedings of the 2nd International Workshop on Procedural Content Generation in Games**. New York, NY, USA: Association for Computing Machinery, 2011. p. 1–6. Disponível em: https://doi.org/10.1145/2000919.2000922. Acesso em: 21 abr. 2024.

VÄRTINEN, Susanna; HÄMÄLÄINEN, Perttu; GUCKELSBERGER, Christian. Generating Role-Playing Game Quests With GPT Language Models. **IEEE Transactions on Games**, [*s. l.*], v. 16, n. 1, p. 127–139, 2024.

YONG, Qing Ru; MITCHELL, Alex. From Playing the Story to Gaming the System: Repeat Experiences of a Large Language Model-Based Interactive Story. *In*: FROM PLAYING THE STORY TO GAMING THE SYSTEM, 2023, Cham. (Lissa Holloway-Attaway & John T. Murray, Org.)**Interactive Storytelling**. Cham: Springer Nature Switzerland, 2023. p. 395–409.