МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»

ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

**ПОЯСНЮВАЛЬНА ЗАПИСКА**

до кваліфікаційної роботи освітнього ступеня «магістр»

за спеціальністю 122 «Комп'ютерні науки»   
(освітня програма «Комп'ютерна графіка та розробка ігор»)

на тему:  
«Генерація ігрового контенту за допомогою штучного інтелекту: адаптивні сценарії та світи.»

|  |
| --- |
| Виконав студент групи КНм-23-1 ГАЛАС Тимур Тимурович |
|  |
| Керівник роботи:  ПЕТРОСЯН Руслан Валерікович |
|  |
| Рецензент:  ЛЕВКІВСЬКИЙ Віталій Леонідович |

Житомир – 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»

ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП’ЮТЕРНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП’ЮТЕРНИХ НАУК

ЗАТВЕРДЖУЮ

Завідувач кафедри комп’ютерних наук

Марина ГРАФ

«25» січня 2024 р.

**ЗАВДАННЯ**

на кваліфікаційну роботу освітнього ступеня «магістр»

за спеціальністю 122 «Комп'ютерні науки»

(освітня програма «Комп'ютерні науки»)

Здобувач вищої освіти: **ГАЛАС Тимур Тимурович**

Керівник роботи: **ПЕТРОСЯН Руслан Валерікович**

Тема роботи: **«Генерація ігрового контенту за допомогою штучного інтелекту: адаптивні сценарії та світи.»**,

затверджена наказом закладу вищої освіти від **«25» січня 2024 р., №22/с**

Термін здачі закінченої роботи: «05» червня 2024 р.

Вихідні дані роботи: об’єктом дослідження є гра, яка за допогою ШІ(Штучний інтелект) генерує історію на базі дій гравця. Предметом дослідження є двигун Unity, платформа розміщення публікацій, її побудова на сучасних технологіях розробки.

Консультанти випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Консультант | Дата | |
| Завдання видав | Завдання прийняв |
| 1 | Петросян Р.В |  |  |
| 2 | Петросян Р.В |  |  |
| 3 | Петросян Р.В |  |  |

**Календарний план**

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів випускної кваліфікаційної роботи | Термін виконання етапів роботи | Примітка |
| 1 | Постановка задачі. Пошук, огляд та аналіз аналогічних розробок. Формулювання технічного завдання. Опрацювання літературних джерел. |  | Виконано |
| 2 | Проектування структури системи |  | Виконано |
| 3 | Написання програмного коду |  | Виконано |
| 4 | Оформлення пояснювальної записки |  | Виконано |

Здобувач вищої освіти **Тимур ГАЛАС**

Керівник **Руслан ПЕТРОСЯН**

# **РЕФЕРАТ**

Пояснювальна записка до кваліфікаційної роботи на тему «Генерація ігрового контенту за допомогою штучного інтелекту: адаптивні сценарії та світи.» складається з переліку умовних скорочень, вступу, трьох розділів, висновків, списку використаної літератури та додатку.

Текстова частина викладена на **69** сторінках друкованого тексту.

Пояснювальна записка має **32** сторінки додатків. Список використаних джерел містить **18** найменування і займає **2** сторінки. В роботі наведено **10** рисунків. Загальний обсяг роботи – **112** сторінок.

У першому розділі були визначені основні вимоги та критерії до проекту, були оглянути альтернативні ігри та сервіси, було обрано стек технологій для розробки.

У другому розділі було проведено проектування гри та аналіз структури.

У третьому розділі реалізовано програмний код для механік гри, зокрема підрахунок токенів, збереження історії та відображення повідомлень, що було підтверджено тестуванням.

Висновок містить в собі результати виконаної роботи по розробці гри для генерації контенту за допомгою ШІ.

ABSTRACT

The explanatory note for the qualification work on the topic "Game Content Generation Using Artificial Intelligence: Adaptive Scenarios and Worlds" consists of a list of abbreviations, introduction, three chapters, conclusions, references, and appendices.

The main text is presented in 69 pages.

The explanatory note includes 32 pages of appendices. The list of references contains 18 entries and spans 2 pages. The work includes 10 figures. The total volume of the work is 112 pages.

The first chapter defines the main requirements and criteria for the project, reviews alternative games and services, and selects the technology stack for development.

The second chapter focuses on the game design and structure analysis.

The third chapter implements the game mechanics, including token counting, history saving, and message display, which have been confirmed through testing.

The conclusion summarizes the results of the development of a game for content generation using AI..

**ЗМІСТ**

[РЕФЕРАТ 3](#_Toc9420)

[ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 7](#_Toc8204)

[ВСТУП 8](#_Toc26658)

[РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ТА ПЕРЕГЛЯД ТЕХНОЛОГІЙ ДЛЯ 9](#_Toc14927)

[СТВОРЕННЯ ГРИ 9](#_Toc31911)

[1.1 Постановка задачі 9](#_Toc8818)

[1.2 Аналіз існуючих технологій реалізації ігрових додатків 11](#_Toc25673)

[1.3 Обґрунтування вибору інструментальних засобів розробки 16](#_Toc22541)

[1.4 Постановка задачі та технічне завдання 18](#_Toc6169)

[Висновки до першого розділу 20](#_Toc28765)

[РОЗДІЛ 2. ОПИС ПРИНЦИПУ РОБОТИ ГРИ ТА АРХІТЕКТУРИ ПРОЕКТУ 22](#_Toc7945)

[2.1 Принцип роботи гри 22](#_Toc10897)

[2.2 Архітектура гри 25](#_Toc18310)

[2.3 Опис процесів у проекті 27](#_Toc24433)

[Висновки до другого розділу 34](#_Toc768)

[РОЗДІЛ 3. РОЗРОБКА ТА ПРОГРАМУВАННЯ 36](#_Toc28493)

[3.1 Опис загальної структури коду 36](#_Toc6820)

[3.2 Опис скриптів та їх функціональності 38](#_Toc10328)

[3.3 Тестування та перевірка роботи гри 61](#_Toc28357)

[Висновки до третього розділу 75](#_Toc12499)

[ВИСНОВКИ 76](#_Toc16085)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 78](#_Toc7105)

[ДОДАТОК 80](#_Toc9687)

# **ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ**

AI – Artificial Intelligence (Штучний інтелект)

UI – User Interface (Користувацький інтерфейс)

JSON – JavaScript Object Notation (Об'єктно-орієнтований формат даних)

C# – C Sharp (Мова програмування C#)

IDE – Integrated Development Environment (Інтегроване середовище розробки)

SFX – Sound Effects (Звукові ефекти)

API – Application Programming Interface (Інтерфейс програмування додатків)

GUI – Graphical User Interface (Графічний інтерфейс користувача)

SDK – Software Development Kit (Набір інструментів для розробки програмного забезпечення)

DFD – Data Flow Diagram (Діаграма потоку даних)

RAM – Random Access Memory (Оперативна пам'ять)

# **ВСТУП**

Актуальність теми. Розвиток штучного інтелекту (ШІ) відкриває нові

горизонти в розробці інтерактивних додатків, зокрема в індустрії відеоігор.

Одним із найбільш перспективних напрямків є використання ШІ для створення адаптивних сценаріїв та динамічних ігрових світів. Такі системи дозволяють автоматично генерувати контент, який змінюється в залежності від дій гравця, тим самим підвищуючи інтерактивність та занурення в гру. Це дозволяє створювати унікальні ігрові досвіди, що варіюються в залежності від вибору кожного гравця, що має потенціал змінити традиційні підходи до розробки ігор. Впровадження ШІ в ігрову індустрію є важливим кроком у розвитку технологій, які сприяють створенню більш складних і багатих ігрових світів, що відповідають на дії користувача. Поширені технології, такі як нейронні мережі і мовні моделі (наприклад, GPT-3 та GPT-4), дозволяють генерувати не лише текстовий контент, але й адаптувати сценарії на основі контексту, створюючи живі ігрові світи, що динамічно реагують на зміну ситуацій.

Метою дослідження є розробка алгоритмів генерації адаптивних ігрових сценаріїв та реалізація системи, яка використовує ШІ для створення текстового контенту для інтерактивних ігор. Основним завданням є розробка програмного забезпечення для генерації унікальних історій і подій у залежності від вибору гравця, що дозволить зробити ігрові світи динамічними і адаптивними. Об'єкт дослідження — це процеси автоматичної генерації контенту в інтерактивних додатках за допомогою штучного інтелекту, зокрема в контексті ігор. Предметом дослідження є використання сучасних мовних моделей, зокрема GPT-4, для генерації ігрових сценаріїв та адаптивних світів.

# **РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ТА ПЕРЕГЛЯД ТЕХНОЛОГІЙ ДЛЯ**

# **СТВОРЕННЯ ГРИ**

## **Постановка задачі**

Результатом виконання проєкту має стати програмне забезпечення, яке забезпечує:

Динамічну генерацію сюжетів:

Програма повинна автоматично створювати текстові історії та події, що адаптуються до введення гравця. Це включає створення сюжетів різної складності, які відповідають вибраному жанру, світові та характеристикам персонажа. Інтерактивність ігрового процесу:

Розроблене програмне забезпечення дозволить гравцям впливати на розвиток сюжету через вибір дій, відповіді або введення тексту. Всі зміни в історії повинні бути логічно узгодженими та контекстуально доречними. Модульна архітектура:

Структура програми повинна бути побудована так, щоб легко інтегрувати нові функції, додавати додаткові модулі та розширювати можливості генерації. Зручний інтерфейс:

Інтерфейс взаємодії повинен бути інтуїтивно зрозумілим для користувачів, забезпечуючи простий спосіб введення параметрів (світ, персонаж, жанр) та отримання текстових результатів.

Інтеграція сучасних технологій ШІ:

Використання OpenAI API дозволить забезпечити якісну генерацію тексту з використанням сучасних мовних моделей (GPT-4). Програма має демонструвати точність, адаптивність і контекстуальність створених історій. Реалізацію функцій адаптивності: У залежності від жанру, світу та обраних характеристик персонажа, програма генеруватиме унікальні сценарії, які відповідають уподобанням гравця.

Тестування і оптимізація:

Програма повинна бути протестована на коректність роботи, продуктивність і здатність обробляти різні сценарії. Очікується, що користувачі отримуватимуть задоволення від гри без помітних збоїв або логічних невідповідностей.

Розширення можливостей для геймдизайнерів:

Завдяки автоматизованому створенню сюжетів, програмне забезпечення має стати інструментом для геймдизайнерів, які зможуть швидше розробляти ігрові сценарії та фокусуватися на творчих аспектах.

Вплив результатів:

Розробка такого програмного забезпечення дозволить забезпечити новий рівень адаптивності ігор, збільшити їхню інтерактивність і персоналізацію, а також спростити процес створення контенту для розробників. Це сприятиме подальшому розвитку ігрової індустрії та технологій штучного інтелекту.

## **Аналіз існуючих технологій реалізації ігрових додатків**

В аналізі ринку штучного інтелекту для створення інтерактивних наративів і текстових пригод важливим етапом є вивчення конкурентів, які активно використовують подібні технології. Далі представлено детальний огляд п’яти основних конкурентів у цій сфері: AI Dungeon, Latitude Voyager, AI Storytelling в Grammarly, Jasper, та NovelAI.

1.AI Dungeon

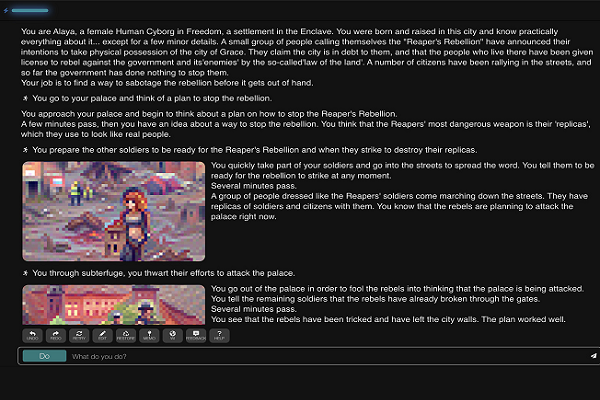


Рис 1.1. AI Dungeon

AI Dungeon — це один із найвідоміших інструментів для створення інтерактивних текстових пригод, який використовує GPT-3 від OpenAI для генерації наративу. Головною особливістю AI Dungeon є можливість створення повністю відкритого світу, де користувачі можуть вводити будь-які команди або дії, а система генерує відповіді на основі цих запитів.

Ключові особливості:

Непередбачуваність сюжету: AI Dungeon дозволяє користувачам впливати на хід історії через інтерактивні запити. Штучний інтелект генерує текст в реальному часі, створюючи історії, які можуть бути як лінійними, так і відкритими.

Багатство жанрів: Платформа підтримує різноманітні жанри, від фентезі до наукової фантастики, що дозволяє користувачам створювати найрізноманітніші світи.

Обмеження AI: Хоча AI Dungeon є дуже гнучким у створенні історій, він має певні обмеження в підтримці складних, довготривалих наративів, оскільки AI може іноді втратити послідовність або логіку сюжету на тривалих дистанціях.

Система монетизації: AI Dungeon має безкоштовний доступ до базових функцій, але для доступу до повних можливостей потрібно підписатися на преміум-версію.

Висновок: AI Dungeon є потужним інструментом для створення текстових пригод, але має обмеження в складності та логічній послідовності наративу, що може бути викликом для розробників більш глибоких історій.

2.Latitude Voyager

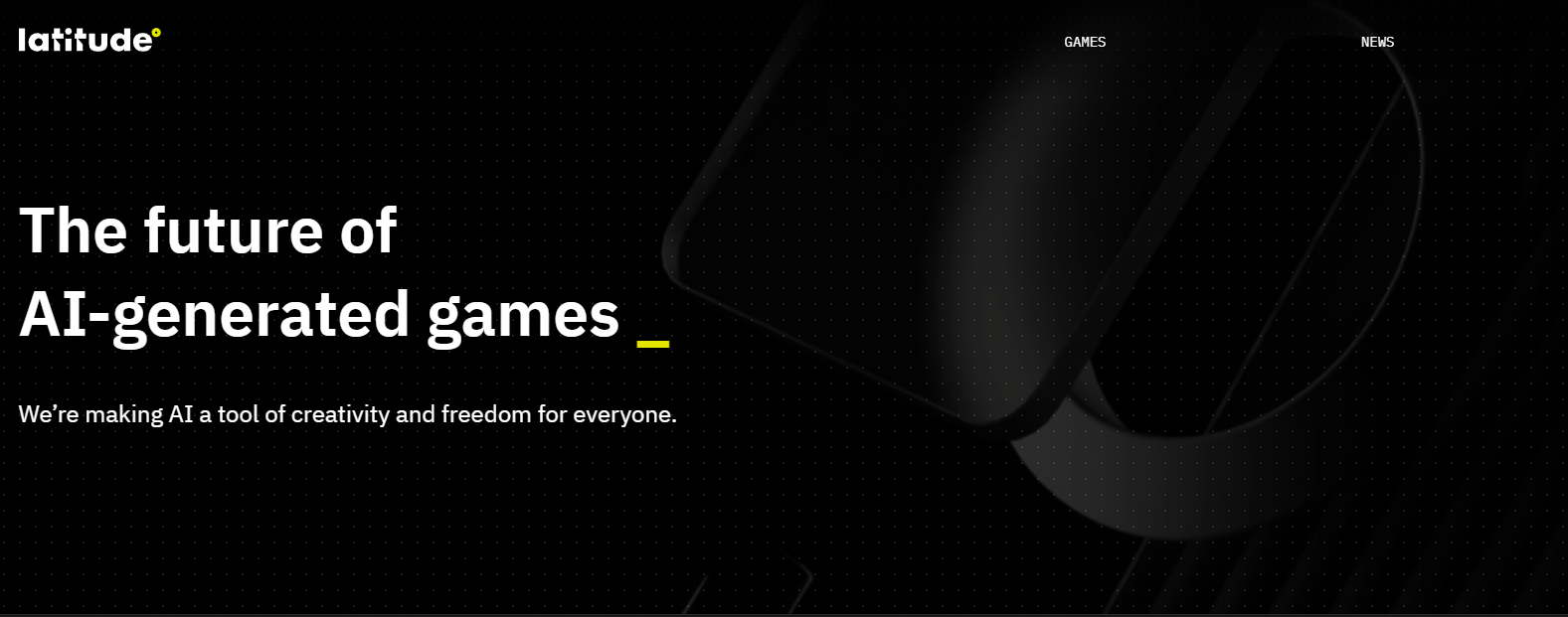


Рис 1.2. Latitude Voyager

Latitude Voyager — це розвиваюча платформа, яка спеціалізується на створенні інтерактивних сюжетів для відеоігор. Вона використовує штучний інтелект для генерування історій та дозволяє інтегрувати елементи відкритого світу, де кожна дія користувача може впливати на загальний розвиток наративу.

Ключові особливості:

Глибока інтерактивність: Latitude Voyager дозволяє створювати багатошарові сюжети з безліччю можливих варіантів розвитку подій. Це дає можливість розробникам створювати складні, нелінійні світи.

Інтеграція з ігровими механіками: Платформа дозволяє поєднувати генеровані сюжети з ігровими механіками, такими як боева система, прокачування персонажів та взаємодія з навколишнім світом.

Модульність: Один із великих плюсів — це можливість налаштовувати ігрові сценарії під різні жанри та аудиторії. Платформа може бути адаптована як для написання простих текстових пригод, так і для великих відкритих світів.

Вартість та доступність: Latitude Voyager знаходиться на стадії активного розвитку, і доступ до повного функціоналу платформи може бути обмежений в залежності від обраної підписки.

Висновок: Платформа має великий потенціал для створення інтерактивних ігор, особливо з відкритими світами, але на даний момент є менш доступною та ще не має такої популярності, як інші конкуренти.

3.AI Storytelling у Grammarly

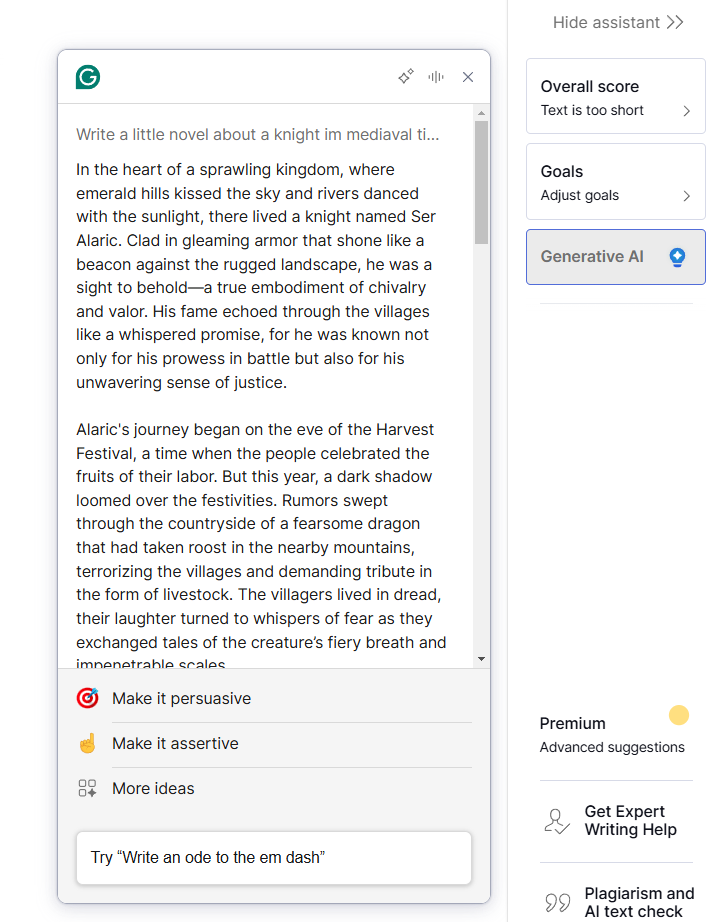


Рис 1.3. AI Storytelling

Grammarly відомий перш за все як інструмент для перевірки граматики та стилю, але він також інтегрує можливості для створення та вдосконалення наративів, використовуючи AI для підтримки авторів[6].

Ключові особливості:

Якість письма: AI Storytelling в Grammarly фокусується на поліпшенні стилістичних та граматичних аспектів текстів. Інструмент може допомогти авторам писати без помилок, підтримуючи логічну структуру та коректність.

Адаптивні рекомендації: В залежності від контексту, Grammarly пропонує варіанти покращення тексту, які можуть включати зміни в тоні, стилі та структурі, допомагаючи створювати більш якісні історії.

Обмеження в контексті інтерактивних історій: Хоча Grammarly є чудовим інструментом для авторів, він не призначений для створення повноцінних інтерактивних сюжетів. Це інструмент для вдосконалення вже написаного ексту, але не для генерації складних, змінюваних наративів.

Висновок: Grammarly більше орієнтований на покращення якості текстів, а не на створення інтерактивних наративів, що робить його корисним для письменників, але не конкурентом для платформ, що займаються генерацією історій.

4. Jasper

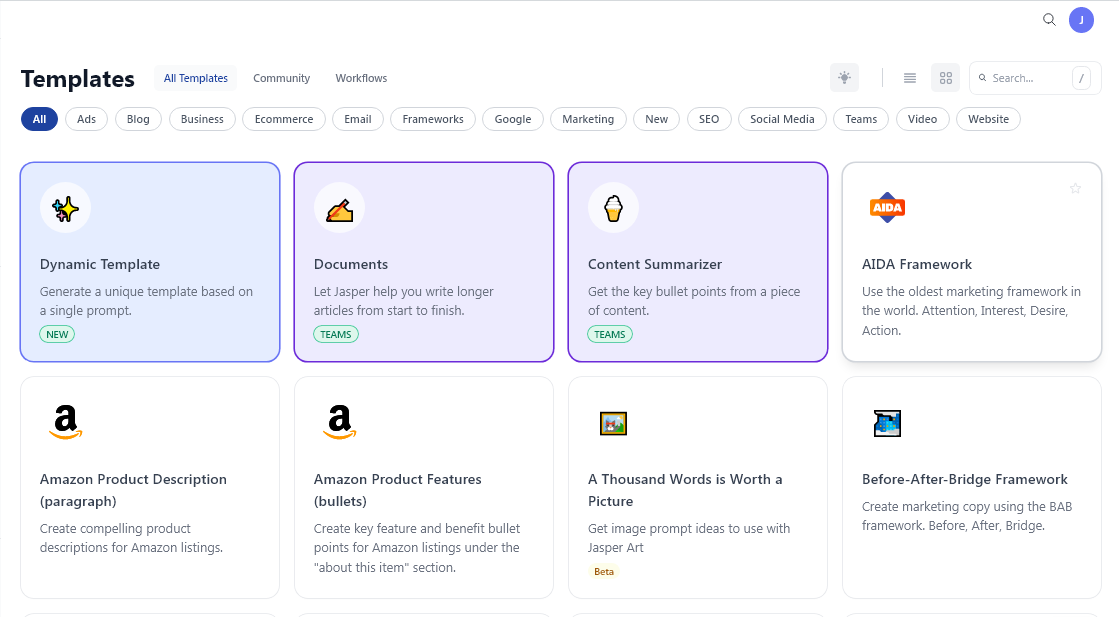


Рис 1.4. Jasper

Jasper — це потужний інструмент для створення контенту, який використовує AI для автоматичного написання текстів. Найчастіше його використовують для створення маркетингових текстів, постів у блогах та сценаріїв.

Ключові особливості:

Широке застосування: Jasper спеціалізується на маркетингових текстах і контенті для бізнесу, проте також може використовуватися для генерації наративів у літературі чи сценаріїв для медіа.

Креативні моделі генерації: Платформа дозволяє налаштовувати параметри генерації, вибираючи різні стилі письма та формати текстів.

Обмеження інтерактивності: У порівнянні з AI Dungeon чи Latitude Voyager, Jasper не має високого рівня інтерактивності, а більше орієнтований на генерацію статичних текстів або контенту без прямої участі користувача в розвитку сюжету.

Висновок: Jasper — це потужний інструмент для автоматичного написання контенту, але він не призначений для створення інтерактивних наративів, що обмежує його використання в контексті ігор.

5. NovelAI

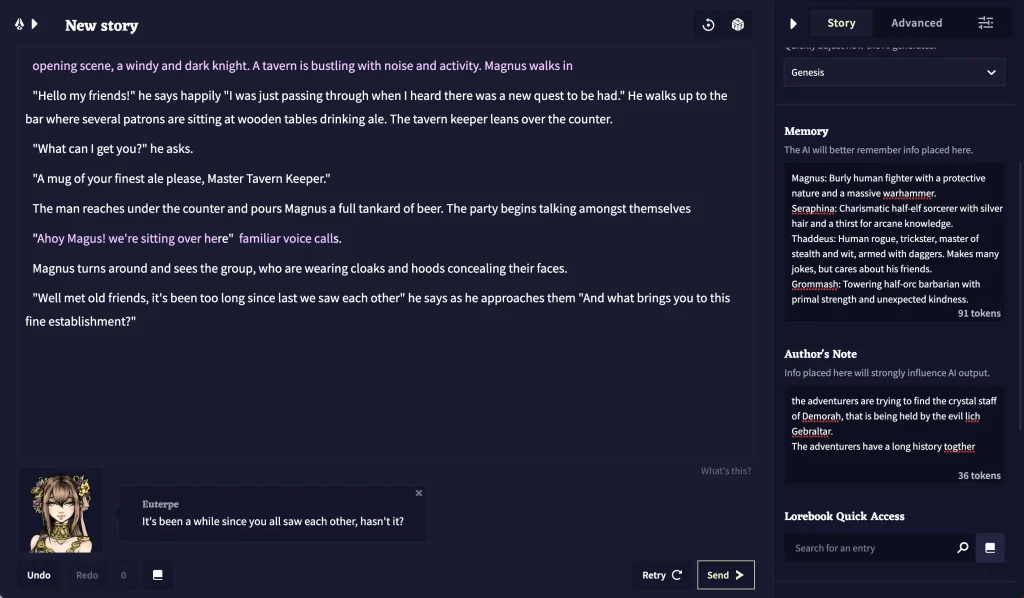


Рис 1.5. NovelAI

NovelAI є платформою, орієнтованою на письменників, які хочуть створювати детальні наративи з високою ступінню адаптації стилю, жанру та поведінки персонажів.

Ключові особливості:

Персоналізація наративу: NovelAI дозволяє користувачам налаштовувати стиль письма, тон, розвиток персонажів і навіть взаємодію між ними. Це дозволяє створювати дуже детально продумані та персоналізовані історії.

Генерація складних сюжетів: Завдяки використанню GPT-3, NovelAI здатний генерувати глибокі та складні наративи, що мають логічну послідовність і розвиток.

Засоби редагування та налаштування: Користувачі можуть редагувати та коригувати генеровані тексти, щоб краще відповідати своїм вимогам, що робить платформу особливо корисною для авторів, які хочуть контролювати процес створення наративу.

Висновок: NovelAI — це відмінний інструмент для створення складних і адаптованих текстів, що робить його популярним серед письменників, але менш орієнтованим на інтерактивні ігри або відкриті світи.

Дослідження цих конкурентів дало змогу визначити ключові вимоги до проєкту, зокрема: стабільність роботи AI, високий рівень інтерактивності, логічна послідовність діалогів і сюжету, а також можливість адаптації під різні тематики ігрових світів.

## **Обґрунтування вибору інструментальних засобів розробки**

Розробка текстової гри з інтеграцією штучного інтелекту для генерації історій та подій вимагає використання сучасних інструментальних засобів, які забезпечують продуктивність, зручність розробки та інтеграцію з API моделей. Для досягнення поставлених цілей було обрано такі засоби: Unity, C#, API від OpenAI та допоміжні бібліотеки.

Unity є одним із найпопулярніших ігрових двигунів завдяки його багатофункціональності, підтримці різних платформ та широкому вибору інструментів. Його використання у проекті обґрунтовано наступними факторами:

Гнучкість та масштабованість: Unity підтримує різні типи ігор, від 2D до 3D, і дозволяє легко масштабувати проект для роботи на різних пристроях.

Швидка інтеграція інструментів: Движок надає можливості для інтеграції сторонніх API, таких як OpenAI, для забезпечення функціоналу штучного інтелекту.

Кросплатформність: Unity дозволяє створювати білди для Windows, Linux, macOS та інших платформ, що відповідає завданням проекту.

Зручність роботи з UI: Вбудовані інструменти для розробки користувацького інтерфейсу забезпечують швидку реалізацію інтерактивного дизайну для текстової гри.

C# є основною мовою програмування, яка використовується в Unity. Його вибір обґрунтований такими перевагами:

Легкість у вивченні: Мова C# має інтуїтивний синтаксис, що спрощує процес розробки.

Сумісність з Unity: C# оптимізований для роботи з Unity, надаючи розробникам доступ до всіх функцій і можливостей движка.

Об'єктно-орієнтоване програмування: C# дозволяє структурувати код, що сприяє підтримці і масштабуванню проекту.

API від OpenAI використовується для інтеграції моделей штучного інтелекту GPT-4 та DALL·E в проект. Вибір цього інструмента обумовлений такими факторами:

Висока якість генерації тексту: Моделі OpenAI забезпечують створення реалістичних і захопливих історій, що є основою текстової гри.

Гнучкість у налаштуванні: Можливість адаптувати поведінку моделі за допомогою system prompts дозволяє отримувати відповіді, які відповідають заданим умовам гри.

Розширені можливості: DALL·E забезпечує генерацію зображень для візуалізації ігрових сценаріїв, що покращує загальний досвід гравця.

Для забезпечення роботи проекту використовувалися додаткові бібліотеки:

Newtonsoft.Json: Дозволяє зручно працювати з JSON-даними, які використовуються в API-запитах до OpenAI.

TextMeshPro: Використовується для створення текстового інтерфейсу гри, забезпечуючи чіткість і естетичність відображення тексту.

UnityWebRequest: Надає інструменти для роботи з HTTP-запитами, що необхідні для інтеграції з API OpenAI.

Загальна оцінка

Вибрані інструментальні засоби дозволяють ефективно реалізувати всі аспекти проекту, включаючи генерацію історій, інтерактивність геймплею, створення візуального супроводу та забезпечення кросплатформності. Вони забезпечують зручність у розробці, високу продуктивність та відповідають сучасним вимогам індустрії.

## **Постановка задачі та технічне завдання**

Сучасні текстові ігри, які використовують штучний інтелект, стають все популярнішими завдяки їхній здатності створювати унікальні сюжети та забезпечувати персоналізований геймплей. Завданням цього проекту є розробка текстової гри, яка використовує можливості GPT-моделей для генерації адаптивних історій та DALL·E для створення ілюстрацій. Гравець взаємодіє з грою через текстові команди, що впливають на подальший розвиток сюжету.

Гра повинна бути функціональною, доступною для користувачів різного віку (включаючи family-friendly режим), та надавати інтерактивний досвід, який поєднує елементи текстового геймплею та візуального супроводу.

Для досягнення поставлених цілей необхідно реалізувати наступні функції та особливості:

1. Інтерактивність та геймплей

Реалізувати систему вводу текстових команд від користувача.

Забезпечити адаптивну генерацію текстового сюжету на основі введених даних.

Додати можливість вибору жанру світу (фентезі, наукова фантастика, історичний тощо).

Реалізувати систему family-friendly, яка виключає небажаний контент для молодшої аудиторії.

2. Інтеграція з OpenAI API

Використати GPT-моделі для генерації текстового сюжету.

Інтегрувати DALL·E для створення візуальних елементів гри.

Забезпечити оптимізацію токенів у запитах до моделей для уникнення перевищення ліміту.

3. Візуалізація та UI

Розробити інтуїтивно зрозумілий графічний інтерфейс користувача.

Відображати текстовий діалог та ілюстрації, створені DALL·E, у відповідних секціях інтерфейсу.

4. Музичний супровід

Додати функціонал для програвання музичних композицій відповідно до жанру гри.

Реалізувати можливість вибору плейлиста користувачем.

Забезпечити плавне управління гучністю, паузами та перемиканням треків.

5. Збереження прогресу

Реалізувати функцію автозбереження прогресу гри (сюжету та налаштувань).

Надати можливість завантаження попередніх сесій через графічний інтерфейс.

6. Кросплатформність

Забезпечити можливість запуску гри на Windows та Linux.

Перевірити та оптимізувати функціонал для кожної платформи.

Обмеження та вимоги

Зробити гру максимально оптимізованою для роботи на середньопродуктивних пристроях.

Реалізувати систему логування для відстеження помилок та спрощення процесу тестування.

Очікувані результати

Результатом роботи має стати текстова гра з інтеграцією штучного інтелекту, яка забезпечує динамічний та цікавий геймплей, адаптований до дій користувача. Гра повинна мати зрозумілий інтерфейс, різноманітність жанрів, візуальний супровід та функціонал для управління музикою, що підвищує загальне враження від ігрового процесу.

## **Висновки до першого розділу**

У першому розділі було проведено аналіз актуальності розробки інтерактивної текстової гри з використанням штучного інтелекту. Було визначено основні особливості текстових ігор, їх значення у сучасній ігровій індустрії та переваги використання GPT-моделей для створення адаптивного сюжету.

Аналіз аналогів показав, що існують різноманітні інструменти для генерації текстів та ілюстрацій, проте інтеграція цих функцій у геймплей є рідкісною. Унікальність проекту полягає в поєднанні текстових та візуальних елементів для створення більш глибокого та захоплюючого досвіду гри.

Обґрунтування вибору інструментів розробки дозволило визначити, що використання Unity у поєднанні з OpenAI API є найбільш оптимальним рішенням для досягнення поставлених цілей. Unity забезпечує широкі можливості для розробки інтерактивного інтерфейсу, тоді як API OpenAI надає потужний інструментарій для генерації текстів і зображень.

Було сформульовано технічне завдання, яке включає всі необхідні аспекти для створення інтерактивної гри: генерацію сюжетів, інтеграцію ілюстрацій, музичний супровід, автозбереження прогресу, підтримку family-friendly режиму та кросплатформність.

Таким чином, проведений аналіз та поставлене технічне завдання закладають основу для розробки гри, яка поєднує можливості штучного інтелекту з інтерактивним геймплеєм, створюючи унікальний ігровий досвід для користувачів.

# **РОЗДІЛ 2. ОПИС ПРИНЦИПУ РОБОТИ ГРИ ТА АРХІТЕКТУРИ ПРОЕКТУ**

## **2.1 Принцип роботи гри**

Проект є текстовою грою, де гравець взаємодіє з системою через введення текстових команд. Гра базується на генерації світу, де кожен вибір користувача призводить до змін у сценарії гри, генеруючи унікальний досвід для кожного гравця. Опис процесу гри включає кілька ключових етапів, що визначають її основну механіку.

1. Введення інформації користувачем

У грі користувач взаємодіє з інтерфейсом через inputfield (поле для введення тексту), куди він може вводити різноманітні параметри, які визначатимуть вигляд і тематику генерованого світу. Це можуть бути такі дані, як:

Жанр світу (наприклад, фентезі, наукова фантастика, історія тощо).

Тематика (вибір певної теми або стилю світу).

Інші параметри, що допомагають створити унікальний контекст для гри.

Користувач може вводити текст довільно, і на основі цього введення буде сформовано унікальний світ. Даний текст надсилається до системи, де використовується для генерації відповідного контексту та подій у світі гри.

2. Генерація світу на основі введених даних

Після того, як користувач надає необхідну інформацію, скрипт CreateWorld відповідає за обробку цих даних і ініціалізацію світу, який буде генеруватися відповідно до наданих параметрів. Це включає в себе:

Визначення основних характеристик світу (географія, культура, соціальні структури, важливі події).

Визначення ключових персонажів та ситуацій, що будуть впливати на подальший розвиток гри.

Кожен введений параметр відображається в створеному світі. Наприклад, якщо користувач вибирає жанр фентезі, в світі можуть з'являтися дракони, магія, королівства, в той час як наукова фантастика може передбачати високі технології, інопланетян, космічні станції тощо.

Паралельно з цим, скрипт SendRequest використовується для комунікації з текстовою моделлю, яка генерує подальші повідомлення на основі запитів від користувача. Ці запити передаються до зовнішньої моделі (наприклад, ChatGPT), яка надає текстові відповіді для гри.

3. Текстова взаємодія з користувачем

Гра реалізує повністю текстовий інтерфейс, що дозволяє користувачеві взаємодіяти з нею через введення команд і отримання текстових відповідей. Після кожного запиту користувача система формує текстову відповідь, яка є частиною сценарію. Ці текстові повідомлення можуть містити:

Опис подій, що відбуваються у світі.

Відповіді на дії користувача.

Опис результатів вибору, зробленого користувачем.

Всі ці елементи створюють відчуття реалістичної взаємодії з вигаданим світом, де кожен вибір впливає на хід подій.

4. Збереження історії переписки

Однією з важливих складових гри є збереження історії всіх взаємодій, що відбуваються в ході гри. Після кожного запиту користувача та отриманої відповіді, вся переписка зберігається в системі для подальшого відображення в ігровому інтерфейсі. Це забезпечує:

Можливість повернутися до попередніх етапів гри та переглянути, що було сказано раніше.

Відстеження еволюції сюжету в реальному часі.

Збереження здійснюється через механізм, реалізований в скрипті SaveLoadScript, який автоматично зберігає поточну історію гри в файл після кожної нової взаємодії. Це дозволяє грати в гру без втрати прогресу при закритті або перезавантаженні програми.

5. Інтерфейс і відображення історії переписки

Історія переписки між користувачем та системою відображається в scroll view в ігровому інтерфейсі, що дозволяє гравцеві прокручувати старі повідомлення. Кожне нове повідомлення відображається за допомогою префабів, що дозволяє створювати текстові блоки, які виглядають як частина діалогу:

Повідомлення користувача.

Відповіді від системи (наприклад, текстові відповіді від AI).

Це забезпечує наочність і зручність використання, даючи гравцеві можливість легко переглядати всі етапи гри.

6. Управління токенами та обрізка історії

Для забезпечення оптимальної роботи гри з великими обсягами даних, особливо при використанні текстових моделей, використовується скрипт TokenCount. Він виконує кілька важливих функцій:

Підраховує кількість токенів, що використовуються в кожному повідомленні.

Обрізає історію переписки, якщо кількість токенів перевищує ліміт, тим самим зменшуючи навантаження на систему і дозволяючи працювати з великими обсягами даних без проблем.

Цей механізм важливий для підтримки високої ефективності гри та її стабільної роботи при обробці великих текстових блоків.

## **2.2 Архітектура гри**

Проект має чітку структуру, побудовану навколо взаємодії між кількома основними компонентами, які відповідають за створення світу, обробку введених даних, збереження історії та управління введенням і виведенням повідомлень. Архітектура проекту забезпечує плавну інтеграцію всіх цих елементів, дозволяючи користувачеві взаємодіяти з AI через текстові повідомлення.

CreateWorld: Цей скрипт генерує світ на основі введених користувачем даних (наприклад, жанр, тематика). В залежності від параметрів, переданих користувачем, цей компонент відповідає за ініціалізацію та налаштування початкових умов світу.

SendRequest: Основним завданням цього компоненту є обробка запитів до зовнішнього API, наприклад, до ChatGPT, для генерації відповіді на основі введеного тексту. Він передає дані до сервера, отримує відповідь і передає її для подальшої обробки.

TokenCount: Цей скрипт відповідає за обчислення кількості токенів, використаних в кожному запиті до AI. Він дозволяє забезпечити, щоб обсяг повідомлень не перевищував ліміт токенів API. У разі необхідності система обрізає старі повідомлення для збереження обмежень.

SaveLoadScript: Компонент відповідає за збереження та завантаження історії переписки між користувачем і AI. Історія зберігається в локальному файлі у форматі JSON, що дозволяє відновлювати прогрес гри після перезапуску.

ShowMessage: Відповідає за відображення повідомлень на екрані. За допомогою префабів цей компонент динамічно створює елементи UI для кожного повідомлення, дозволяючи користувачеві переглядати історію чату.

Введення даних користувачем та генерація світу: Користувач починає гру, вводячи параметри через InputField (наприклад, жанр або тематику). Ці дані передаються в скрипт CreateWorld, який створює віртуальний світ, що відповідає зазначеним параметрам.

Обробка введеного тексту: Кожен текст, введений користувачем, передається в SendRequest, де запит відправляється до зовнішнього сервера. Відповідь, що надходить від AI, потім передається в компонент ShowMessage, який виводить її на екран у вигляді тексту.

Управління токенами: При кожному новому введеному повідомленні система обчислює кількість токенів, використаних у запиті, завдяки компоненту TokenCount. Якщо кількість токенів перевищує ліміт, система обрізає частину історії, зберігаючи тільки найновіші повідомлення.

Збереження та відображення історії: Вся переписка зберігається в локальних файлах через SaveLoadScript. Цей компонент забезпечує відновлення історії при запуску гри або після перезапуску програми. Історія переписки відображається через ShowMessage, де кожне нове повідомлення створюється як окремий елемент UI.

Обробка подій: Кожна дія користувача (введення тексту, натискання кнопок) викликає відповідні обробники в скриптах, що керують змістом і логікою гри. Це дозволяє забезпечити інтерактивність гри та постійний потік інформації.

Збереження даних: Збереження історії переписки здійснюється через компонент SaveLoadScript, який відповідає за серіалізацію даних у формат JSON. Це дозволяє зберігати прогрес гри, що важливо для забезпечення безперервного досвіду.

Обчислення токенів: Кожен запит до AI обчислює кількість токенів за допомогою компонента TokenCount. Це важливий аспект проекту, оскільки ліміти токенів є частиною зовнішніх API, з якими працює гра. Обмеження токенів допомагає керувати обсягом даних, які передаються в запитах, і зберігати контроль над використанням ресурсів.

## **2.3 Опис процесів у проекті**

У даному підрозділі буде детально описано процеси, що відбуваються в межах гри. Для кращого розуміння цих процесів буде використано кілька діаграм, які ілюструють етапи взаємодії між користувачем, системою та компонентами програми. Кожен етап включає специфічні дії, що забезпечують правильну обробку запитів і взаємодію користувача з системою.

Основні етапи включають:

Оцінка та перевірка токенів — важливий крок, що гарантує, що запит не перевищить ліміт токенів, і система працюватиме коректно.

Обробка запиту користувача — отримання і передача запиту до API, що виконується після перевірки токенів.

Збереження історії переписки — процес збереження результатів після кожного запиту і відповідної обробки.

Відображення відповіді користувачу — надання результату у вигляді, який користувач може побачити в інтерфейсі.

Процеси, описані вище, забезпечують основний функціонал гри, де кожна частина відповідає за певну задачу в циклі взаємодії з користувачем. Розглянемо ці етапи більш детально, починаючи з процесу перевірки токенів.

Процес оцінки та перевірки токенів є важливим етапом взаємодії між користувачем та системою. Він забезпечує правильну обробку запитів, контролюючи, щоб кількість токенів не перевищувала допустимий ліміт, визначений для запиту. На цій діаграмі послідовності відображено, як відбувається перевірка та обробка токенів під час кожного взаємодії користувача з системою.

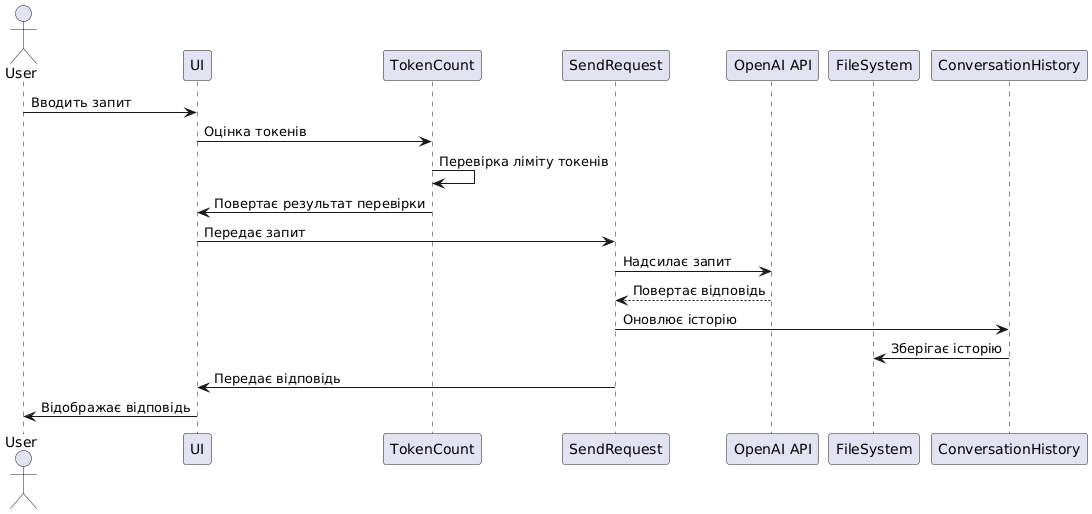


Рис.2.1.діаграмма послідовності

Користувач вводить запит: Користувач вводить текст запиту через інтерфейс, що ініціює початок взаємодії з системою. Цей запит містить текстову інформацію, яку буде обробляти система.

Передача запиту на обробку: UI передає отриманий запит компоненту SendRequest, який відповідає за відправку запиту до API та обробку відповіді.

Оцінка кількості токенів: Перед тим, як запит буде відправлений до API, система оцінює кількість токенів, використовуючи спеціальну функціональність, що визначає, скільки токенів буде витрачено на обробку тексту.

Перевірка ліміту токенів: Якщо кількість токенів перевищує встановлений ліміт, система повідомляє користувача про помилку. В іншому випадку система продовжує обробку запиту, передаючи його далі.

Передача запиту до API: Якщо токени в межах допустимого діапазону, запит передається до OpenAI API для подальшої обробки.

Отримання відповіді від API: API повертає відповідь на запит, яку система може використовувати для подальших дій.

Збереження відповіді: Після отримання відповіді система зберігає її в історії переписки, щоб користувач міг переглядати попередні повідомлення.

Відображення результату користувачу: Нарешті, система передає відповідь назад в інтерфейс користувача для відображення на екрані.

Цей процес забезпечує, що кожен запит обробляється коректно і в межах встановлених обмежень, що критично для забезпечення стабільної роботи системи, а також для збереження ефективності роботи з API.

Процес збереження та завантаження історії переписки є ключовим елементом гри, оскільки він дозволяє користувачеві переглядати попередні повідомлення та відновлювати історію взаємодії з системою. Ця діаграма показує, як система зберігає відповіді після кожного запиту та завантажує їх під час наступних сеансів гри.

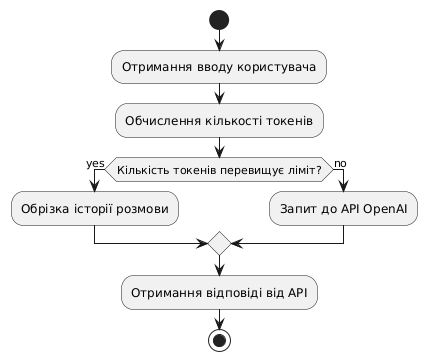


Рис.2.2. Діаграма блок-схеми для токенів

Після обробки запиту: Після того, як запит користувача оброблено, відповідь від API передається компоненту SendRequest для подальшої обробки.

Збереження історії переписки: Коли отримано відповідь, вона зберігається в історії переписки, яку можна переглядати. Компонент SendRequest взаємодіє з файловою системою для збереження цієї історії в локальному файлі.

Перевірка наявності файлу: Перед тим як зберігати нову інформацію, система перевіряє, чи існує файл з попередньою історією. Якщо файл існує, нові повідомлення додаються до наявної історії.

Запис історії в файл: Історія переписки зберігається в текстовому форматі JSON, що дозволяє зручно завантажувати та переглядати дані пізніше. Це збереження забезпечує постійну доступність всіх попередніх взаємодій з користувачем.

Завантаження історії при запуску: Коли користувач запускає гру або знову взаємодіє з системою, система завантажує попередню історію, що зберігається в локальному файлі. Це дозволяє відновити попередні повідомлення та контекст взаємодії.

Відображення історії на UI: Завантажена історія переписки відображається на інтерфейсі користувача у вигляді скролу, де гравець може переглядати старі повідомлення та взаємодіяти з ними.

Цей процес збереження та завантаження переписки дозволяє підтримувати безперервність взаємодії з користувачем, забезпечує відновлення контексту та дозволяє користувачеві продовжувати гру без втрати інформації.

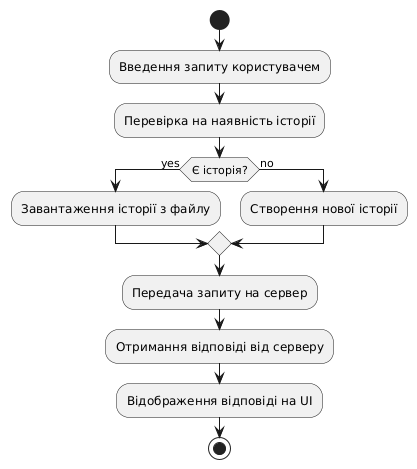


Рис.2.3 Діаграма блок-схеми для взаємодії з користувачем

Ця діаграма описує процес генерації світу, який відбувається на основі введених користувачем даних (наприклад, жанр, тематика). Взаємодія між компонентами системи відбувається наступним чином:

Користувач вводить дані: Спочатку користувач вводить текстову інформацію через інтерфейс користувача (UI). Це може бути жанр, тематика або будь-які інші характеристики, які визначають майбутній світ.

Передача запиту на генерацію: Після того, як користувач вводить дані, вони передаються в скрипт CreateWorld. Цей скрипт відповідає за створення світу на основі параметрів, які надані користувачем.

Запит до AI через SendRequest: Для того, щоб створити світи, система надсилає запит до зовнішньої служби AI (наприклад, OpenAI API) через скрипт SendRequest. Запит містить всю необхідну інформацію для генерації.

Отримання відповіді від AI: Після того як запит оброблено, відповідь повертається з API. Вона містить дані, необхідні для подальшої генерації світу або для виконання сценаріїв у грі.

Відображення результатів користувачу: Коли відповідь отримано, вона передається через SendRequest і відображається на UI, щоб користувач міг побачити результат генерації світу.

Збереження результатів генерації: Після того як світ згенеровано і користувач переглянув результат, дані зберігаються в історії (як було описано в попередній діаграмі), щоб їх можна було відновити при наступному запуску гри або перегляді історії.

Цей процес дозволяє користувачу створювати персоналізовані світи на основі своїх вподобань, що додає гнучкості і взаємодії в гру. Оскільки генерування світу залежить від введених даних, система може створювати різноманітні варіанти світу в залежності від обраних параметрів.



Рис.2.4 Діаграма блок-схеми для збереження та відображення історії

Ця діаграма описує процес збереження та відображення історії взаємодії між користувачем та системою, а також взаємодію між компонентами для збереження даних і їх відображення.

Збір історії: Під час кожної взаємодії з системою (введення запиту користувачем та отримання відповіді від AI), кожне повідомлення, що генерується, додається до історії розмови. Це включає як повідомлення користувача, так і відповіді AI.

Взаємодія з UI: Інтерфейс користувача (UI) передає отримані повідомлення у відповідні компоненти для відображення. У разі введення повідомлення, UI передає це повідомлення у відповідний скрипт, який додасть його до історії та відобразить на екран через спеціально створений префаб.

Збереження історії: Після кожної нової взаємодії, вся історія розмови зберігається у файловій системі. Цей процес забезпечується скриптом, який записує всі повідомлення в файл. Це дозволяє зберігати всі попередні взаємодії для можливості їх перегляду в майбутньому.

Відображення історії: Історія відображається в грі через прокручувану панель (Scroll View). Користувач може переглядати старі повідомлення, взаємодіяти з ними та навіть редагувати або повторно використовувати їх для подальших запитів.

Керування даними: Збереження та відображення історії також включає в себе підтримку обмеження на кількість повідомлень, які можна зберігати або відображати одночасно. Це реалізовано через механізм тримання певної кількості попередніх записів, щоб зберегти продуктивність системи та не перевантажити інтерфейс.

Редагування історії: У разі потреби, користувач може редагувати існуючі повідомлення, що зберігаються в історії. Це дозволяє користувачу коригувати попередні запити або відповіді для подальшого взаємодії, наприклад, для зміни параметрів генерації світу.

Цей процес забезпечує можливість зберігання важливої інформації про попередні взаємодії з системою, надаючи користувачу більший контроль над історією та можливість коригувати свої дії.

Процес взаємодії користувача з системою завершується після обробки відповіді API та збереження історії розмови. Всі дії, що стосуються відображення та збереження даних, контролюються за допомогою таких компонентів, як SaveLoadScript, який відповідає за запис і завантаження історії з файлів, та ShowMessage, який відображає цю історію в UI. Ці компоненти гарантують, що кожне повідомлення, відправлене чи отримане від API, буде збережено, і користувач зможе повернутись до попередніх етапів взаємодії через прокручувану історію.

Використання блок-схем для збереження та відображення історії розмови дає чітке уявлення про етапи цього процесу. Окрім того, важливим аспектом є управління токенами за допомогою механізму, який контролює кількість токенів у кожному запиті та при необхідності обрізає історію для уникнення перевищення ліміту. Цей механізм описано на відповідній діаграмі токенів, яка показує, як система реагує на запити та як розв’язує проблему з обмеженнями на кількість токенів.

Усі ці процеси виконуються таким чином, щоб система залишалася ефективною, зручною для користувача та відповідала вимогам щодо збереження і обробки даних.

## **Висновки до другого розділу**

У другому розділі звіту детально описано архітектуру та принципи роботи гри, а також основні компоненти, що забезпечують її функціональність.

Архітектура гри та взаємодія компонентів. Головним принципом гри є текстова взаємодія між користувачем та AI, де система генерує світ на основі введених даних і відповідає на запити користувача. Це досягається за допомогою різноманітних скриптів, таких як CreateWorld, SendRequest, SaveLoadScript, ShowMessage і TokenCount. Ці компоненти працюють синхронно для надання гравцеві інтерактивного досвіду, де кожен крок — від введення даних до генерації та збереження історії — ретельно контролюється.

Процес збереження та управління токенами. Важливою частиною гри є механізм управління історією повідомлень, що гарантує коректне збереження і можливість перегляду попередніх повідомлень. Це реалізовано через збереження даних в файли за допомогою SaveLoadScript, що забезпечує можливість завантаження та збереження історії. Крім того, контроль за кількістю токенів дозволяє забезпечити ефективну роботу з API, оптимізуючи взаємодію і дозволяючи зберігати історію у межах лімітів.

Інтерфейс і зручність для користувача. UI-елементи, такі як input field та прокручувана історія, значно покращують взаємодію користувача з грою. Гравець може легко вводити текстові запити, отримувати відповіді від AI та переглядати історію спілкування, що робить гру інтуїтивно зрозумілою та доступною.

Інтеграція зовнішніх сервісів. Взаємодія з OpenAI API забезпечує генерацію відповідей на запити користувача. Протягом гри всі дані взаємодії зберігаються та обробляються локально, що дозволяє контролювати їх потік і зберігати історію у разі необхідності.

Діаграми та блок-схеми. Описані діаграми послідовності та блок-схеми допомагають візуалізувати важливі етапи обробки даних, такі як надсилання запиту до API, обробка відповіді та збереження історії. Ці діаграми дають чітке розуміння внутрішніх процесів гри та дозволяють краще зрозуміти структуру і механізми, що забезпечують її функціональність.

В цілому, другий розділ підсумовує основні компоненти гри, їх взаємодію та важливість для забезпечення стабільної і ефективної роботи гри.

# **РОЗДІЛ 3. РОЗРОБКА ТА ПРОГРАМУВАННЯ**

## **3.1 Опис загальної структури коду**

Проект складається з кількох основних компонентів, які взаємодіють між собою для забезпечення ефективної роботи гри. Кожен компонент виконує конкретні завдання, що дозволяє гнучко розширювати і змінювати функціональність гри без порушення її цілісності.

Основні компоненти проекту включають такі ключові скрипти:

CreateWorld: Скрипт, відповідальний за генерацію світу на основі введених користувачем параметрів, таких як жанр, тематика тощо. Цей скрипт взаємодіє з іншими компонентами для передачі запитів до AI та збереження отриманих даних.

SendRequest: Відповідає за відправку запитів до API, отримання відповідей від OpenAI та передачу цих даних іншим скриптам для подальшої обробки.

ShowMessage: Керує відображенням повідомлень на екрані, надаючи можливість користувачу переглядати історію взаємодії через ScrollView. Скрипт також відповідає за додавання нових повідомлень до списку і очищення екрану.

TokenCount: Відповідає за підрахунок кількості токенів у текстах, щоб уникнути перевищення ліміту токенів для запитів до API. Цей скрипт також контролює довжину історії повідомлень, автоматично очищаючи її при необхідності.

SaveLoadScript: Реалізує збереження і завантаження історії повідомлень. Це дозволяє зберігати дані між сесіями та відновлювати попередній стан гри.

Проект організовано таким чином, щоб кожен скрипт виконував одну чітко визначену задачу, що дозволяє знизити складність коду та підвищити його масштабованість. Взаємодія між компонентами відбувається через чітко визначені інтерфейси.

Наприклад, CreateWorld передає запит до SendRequest, який обробляє цей запит і отримує відповідь від API. Потім відповідь передається до ShowMessage, де вона відображається на екрані. Одночасно TokenCount перевіряє кількість токенів у повідомленні і, при необхідності, очищає історію.

Для збереження даних SaveLoadScript регулярно зберігає всі повідомлення в файл, що дозволяє відновлювати стан гри після її перезапуску.

Пояснення принципів організації коду та архітектури

Код проекту організований згідно з принципами об'єктно-орієнтованого програмування (ООП). Це дозволяє створювати класи, які відповідальні за конкретні задачі, та зберігати інкапсуляцію, абстракцію і спадкування для кращої структуризації.

У проекті активно застосовуються такі принципи:

Принцип єдиного обов'язку (Single Responsibility Principle): Кожен скрипт має лише одну відповідальність. Наприклад, TokenCount відповідає лише за підрахунок токенів, а SaveLoadScript лише за збереження та завантаження даних.

Принцип відкритості/закритості (Open/Closed Principle): Скрипти можна розширювати новим функціоналом без змінення існуючого коду. Це досягається через чітке розмежування функцій та використання абстракцій, таких як інтерфейси.

Принцип інтерфейсної сегрегації (Interface Segregation Principle): Кожен клас або інтерфейс відповідає за конкретну задачу і має чітко визначену функціональність.

Принцип залежності (Dependency Inversion Principle): Взаємодія між компонентами здійснюється через інтерфейси або абстракції, що дозволяє легко замінювати одні частини коду іншими без змін у всій системі.

Ці принципи сприяють високій гнучкості та тестованості коду, а також полегшують подальшу підтримку та розширення проекту.

## **3.2 Опис скриптів та їх функціональності**

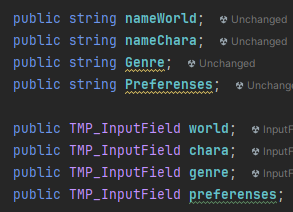
У підрозділі 3.2 буде здійснено детальний опис основних скриптів, які реалізують логіку гри та взаємодію з користувачем. Оскільки гра є текстовою та динамічно генерує віртуальний світ на основі введених користувачем даних, кожен скрипт виконує важливу роль у створенні і збереженні взаємодій. Тут буде розглянуто, як саме обробляються введені дані, генеруються відповіді від штучного інтелекту, як відбувається збереження інформації про переписку та управління токенами, а також як забезпечується коректне збереження і відображення повідомлень у грі. Важливу роль у забезпеченні цих функцій відіграють такі скрипти, як SendRequest, SaveLoadScript, TokenCount та ShowMessage.

Тепер, поступово, будемо переходити до більш детального опису кожного з цих скриптів, звертаючи увагу на їх структуру, функціональність та взаємодію між собою.

Клас CreateWorld

Скрипт CreateWorld є важливою частиною механізму створення світу в грі, де користувач взаємодіє з інтерфейсом для введення основних налаштувань світу. Ось детальний опис його роботи.

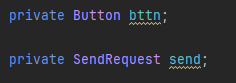
У класі визначені кілька публічних змінних, які використовуються для зберігання значень, введених користувачем:



Лістинг3.1

Ці змінні зберігають відповідно: назву світу, ім'я персонажа, жанр та переваги користувача. Для кожного параметра також створюються публічні поля типу TMP\_InputField, що дозволяє зв'язати їх з відповідними полями вводу в інтерфейсі користувача. Це дає змогу гравцю вводити потрібні дані для генерації світу.

Далі, оголошуються приватні змінні для кнопки та відправки запиту:



Лістинг 3.2.  
 bttn використовується для зберігання кнопки, на яку користувач натискає, щоб ініціювати генерацію світу. send — це посилання на компонент SendRequest, який відповідає за відправку запиту на сервер для генерації історії світу.

Ще один елемент інтерфейсу — це перемикач toggleFamily, який дозволяє користувачеві обирати, чи включений "FamilyMode". Цей режим може впливати на характер генерованої історії.

Метод Start()

Метод Start() виконується при ініціалізації об'єкта в грі:



Лістинг 3.3.

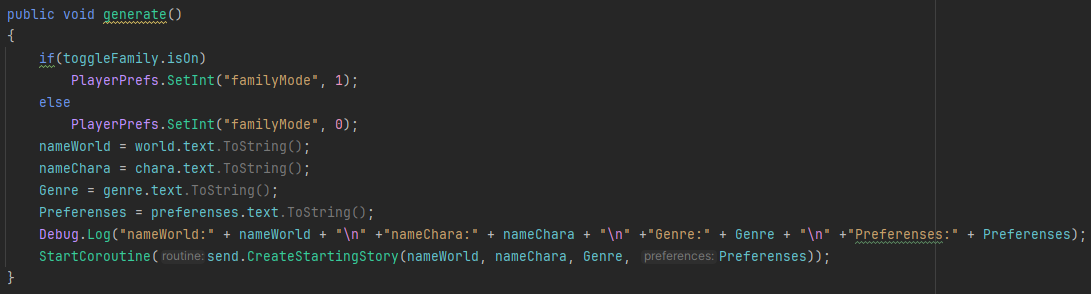
У цьому методі:

Ініціалізація кнопки: За допомогою gameObject.GetComponent<Button>() отримуємо компонент кнопки, який знаходиться на поточному об'єкті. Потім додаємо обробник події onClick, щоб при натисканні на кнопку викликалась функція generate(). Це дозволяє користувачеві натискати кнопку для запуску процесу генерації світу.

Отримання посилання на SendRequest: Для відправки запиту на сервер ми використовуємо метод GameObject.FindWithTag("Manager"), щоб знайти об'єкт з тегом "Manager" і отримати з нього компонент SendRequest. Це дозволяє використовувати функціональність цього скрипта для створення початкової історії світу.

Метод generate()

Метод generate() викликається, коли користувач натискає кнопку. Він відповідає за обробку введених даних і відправку запиту на сервер:



Лістинг 3.4.

У методі generate() відбувається кілька ключових кроків:

Збереження налаштувань FamilyMode: Перевіряється стан перемикача toggleFamily. Якщо перемикач увімкнено, в PlayerPrefs зберігається значення 1 (режим сім'ї активовано), інакше — 0. Це дозволяє зберігати налаштування гри між сесіями.

Отримання введених значень: Текст, введений користувачем у поля вводу, перетворюється на строки і зберігається у відповідних змінних: nameWorld, nameChara, Genre та Preferenses.

Виведення інформації в консоль: Для зручності відлагодження значення, введені користувачем, виводяться в консоль через Debug.Log. Це дозволяє переконатися, що введені дані зберігаються правильно перед відправкою запиту.

Запуск корутини для створення історії: Останнім кроком викликається корутина CreateStartingStory, яка надсилає введені дані на сервер для генерації історії світу. Корутина працює асинхронно, що дозволяє не блокувати основний потік гри під час очікування відповіді від сервера.

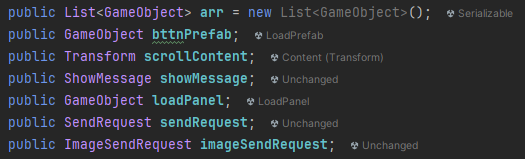
Скрипт CreateWorld є основним інтерфейсним елементом для введення налаштувань світу та персонажа. Він зберігає введену користувачем інформацію, дозволяє вибрати режим сім'ї, а також відправляє дані на сервер для створення історії світу. Весь процес ініціюється через кнопку, а результат надсилається через корутину для асинхронної обробки.

Опис скрипту SaveLoadScript

Скрипт SaveLoadScript відповідає за збереження та завантаження історії розмови у файл. Також він керує завантаженням кнопок, які дозволяють користувачеві вибирати раніше збережені світи для продовження гри. Ось детальний опис його роботи:

Оголошення змінних

У класі SaveLoadScript є кілька публічних і приватних змінних для роботи з елементами інтерфейсу та збереження даних:



Лістинг 3.5.

arr — список об'єктів, який використовується для зберігання кнопок, що генеруються в інтерфейсі користувача.

bttnPrefab — префаб кнопки, яка буде створена для кожного світу.

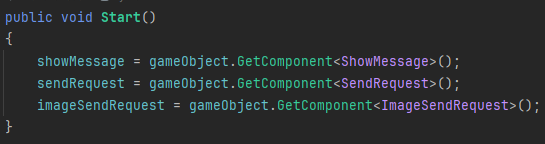
scrollContent — контейнер для кнопок у вигляді скролл-вью, куди будуть додаватись нові кнопки для кожного збереженого світу.

showMessage — посилання на компонент ShowMessage, який відповідає за відображення повідомлень на екрані.

loadPanel — панель для завантаження світу.

sendRequest та imageSendRequest — компоненти для обробки запитів, пов'язаних з історією і зображеннями.

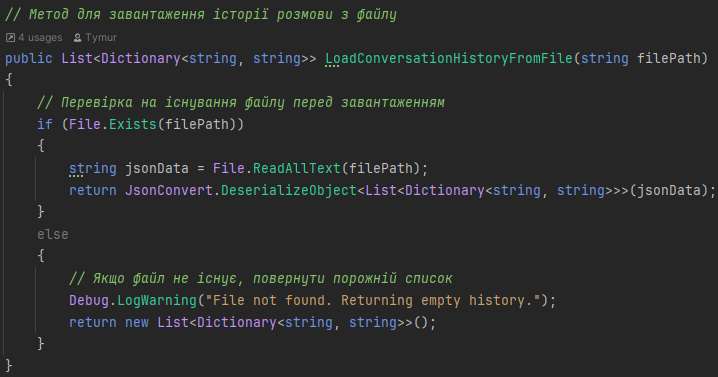
Метод Start()



Лістинг 3.6.

У методі Start() відбувається ініціалізація компонентів, необхідних для подальшої роботи скрипту. За допомогою gameObject.GetComponent<>() отримуються компоненти ShowMessage, SendRequest та ImageSendRequest, що дозволяє скрипту взаємодіяти з іншими частинами програми.

Метод SaveConversationHistoryToFile()



Лістинг 3.7.

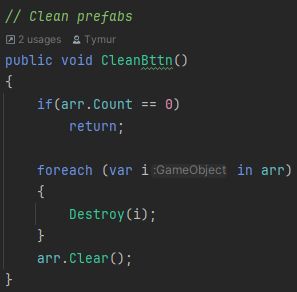
Метод SaveConversationHistoryToFile відповідає за збереження історії розмови в файл у форматі JSON:

Формується шлях до файлу, використовуючи Application.persistentDataPath для збереження у локальній директорії додатку.

Перевіряється, чи існує вже файл; якщо ні, він створюється.

Історія розмови серіалізується у формат JSON за допомогою бібліотеки JsonConvert і записується у файл.

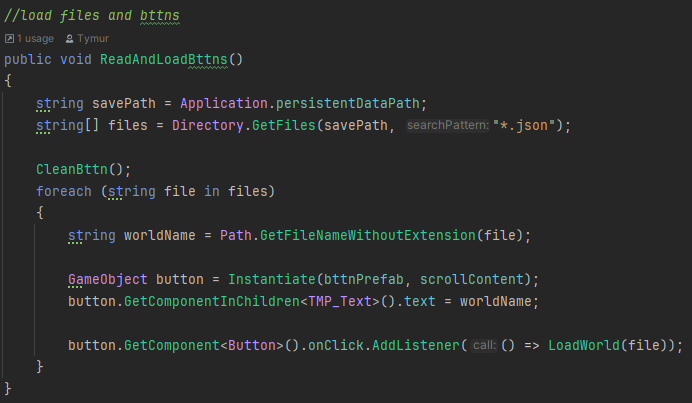
Метод CleanBttn()



Лістинг 3.8.

Метод CleanBttn() очищає список кнопок. Якщо в arr є об'єкти, вони видаляються з сцени, і сам список очищається. Це необхідно для того, щоб перед завантаженням нових даних знову не додавати кнопки до існуючих.

Метод ReadAndLoadBttns()



Лістинг 3.9.

Метод ReadAndLoadBttns() відповідає за завантаження кнопок для всіх збережених файлів історії:

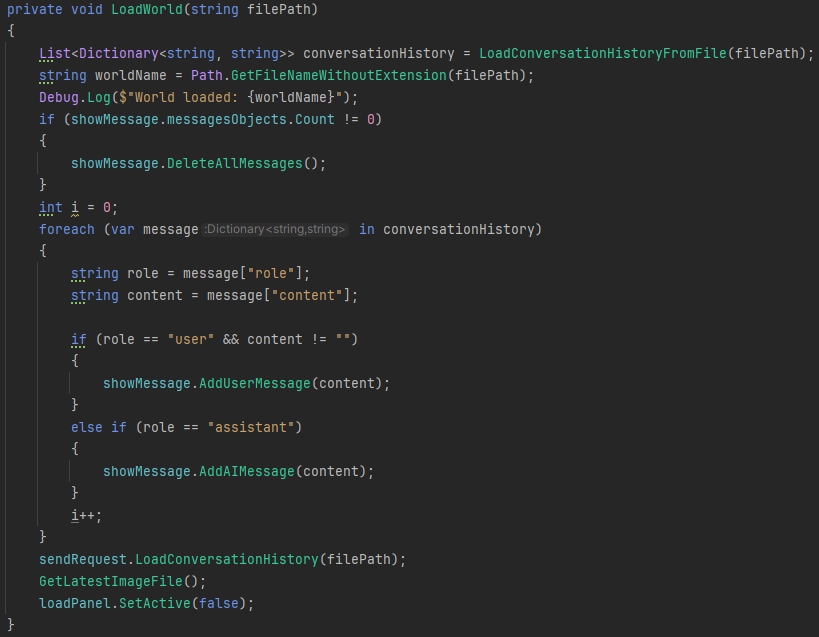
Отримує список всіх файлів .json з директорії збережень.

Очищає поточні кнопки через метод CleanBttn().

Створює нову кнопку для кожного збереженого файлу та додає її до інтерфейсу.

Кожна кнопка має обробник події для завантаження світу, що зберігається у відповідному файлі.

Метод LoadWorld()



Лістинг 3.10.

Метод LoadWorld() завантажує світ за допомогою збереженого файлу:

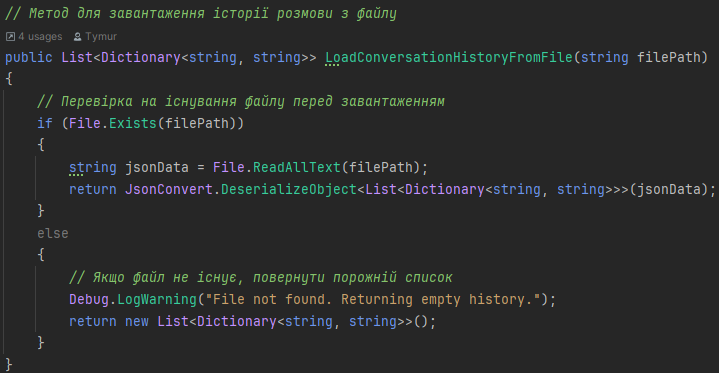
Використовує метод LoadConversationHistoryFromFile(), щоб завантажити історію з файлу.

Очищає всі повідомлення на екрані та додає нові, що відповідають даним з файлу.

Викликається метод для завантаження зображення, якщо воно є.

Останнім кроком приховує панель завантаження.

Метод LoadConversationHistoryFromFile()



Лістинг 3.11.

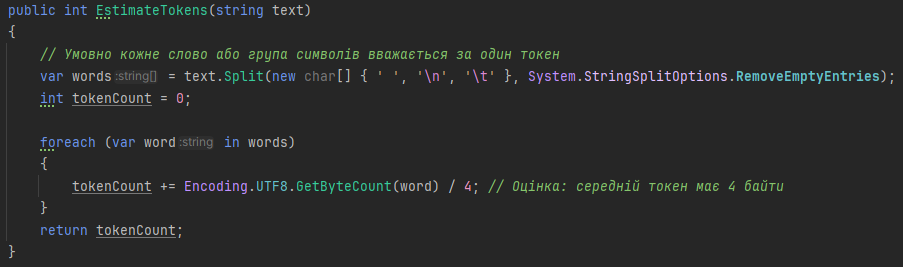
Метод LoadConversationHistoryFromFile() відповідає за завантаження історії розмови з файлу. Якщо файл існує, він десеріалізує JSON у список словників. Якщо файл відсутній, повертається порожній список.

Скрипт SaveLoadScript забезпечує збереження та завантаження даних про історію розмов, включаючи створення та очищення кнопок для вибору збережених світів, а також завантаження відповідних зображень. За допомогою цього скрипту користувач може продовжити свою гру, вибравши збережену історію, що була попередньо збережена у локальному файлі.

Скрипт TokenCount відповідає за оцінку кількості токенів у тексті та в історії розмов, а також за оптимізацію цієї історії, щоб дотримуватися обмежень на кількість токенів, встановлених API. Це важлива частина гри, яка забезпечує роботу з API OpenAI, оптимізуючи розмір історії розмов перед надсиланням запитів.

Оцінка токенів у тексті

Метод EstimateTokens виконує базову оцінку кількості токенів у переданому тексті.



Лістинг 3.12.

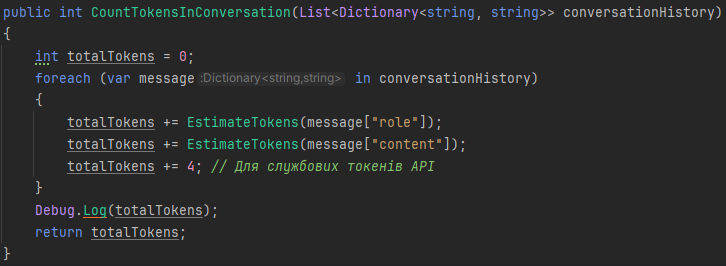
Розбивка тексту: Текст поділяється на слова та групи символів за пробілами, символами табуляції та новими рядками.

Оцінка байтів: Метод використовує Encoding.UTF8.GetByteCount(), щоб підрахувати кількість байтів для кожного слова.

Розрахунок токенів: Враховуючи, що середній токен займає 4 байти, кількість байтів кожного слова ділиться на 4.

Ця оцінка є наближеною, але достатньою для більшості випадків.

Метод CountTokensInConversation підраховує загальну кількість токенів у всій історії розмов.



Лістинг 3.13.

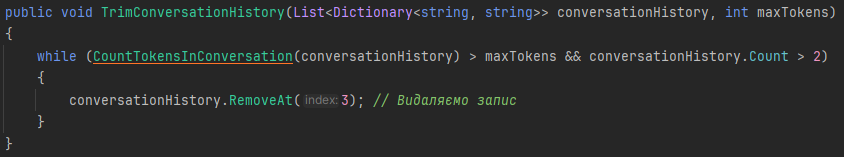
Цикл по історії: Метод перебирає кожен запис у списку історії, де кожен запис — це словник з ключами "role" та "content".

Оцінка ролі та контенту: Для кожного запису оцінюється кількість токенів у ролі (user, assistant) і вмісті повідомлення.

Службові токени: Додається фіксована кількість токенів для службової інформації, яку вимагає API (наприклад, розділові символи).

Метод повертає загальну кількість токенів і виводить її в консоль для дебагу.

Метод TrimConversationHistory використовується для оптимізації історії розмови, якщо вона перевищує заданий ліміт токенів.



Лістинг 3.14.

Умови обрізання: Цикл виконується, поки кількість токенів перевищує максимальне значення maxTokens, і в історії залишається більше двох записів.

Видалення записів: Видаляються записи з індексу 3, оскільки перші два записи зазвичай є початковими системними повідомленнями (інструкціями для AI).

Цей метод зберігає початковий контекст розмови, видаляючи старіші повідомлення, щоб залишатися в межах токенів.

Як працює скрипт у грі

Оцінка тексту: Коли гравець надсилає нове повідомлення, EstimateTokens оцінює його розмір у токенах.

Перевірка історії: Перед відправкою історії до API метод CountTokensInConversation перевіряє, чи не перевищує загальна кількість токенів ліміт.

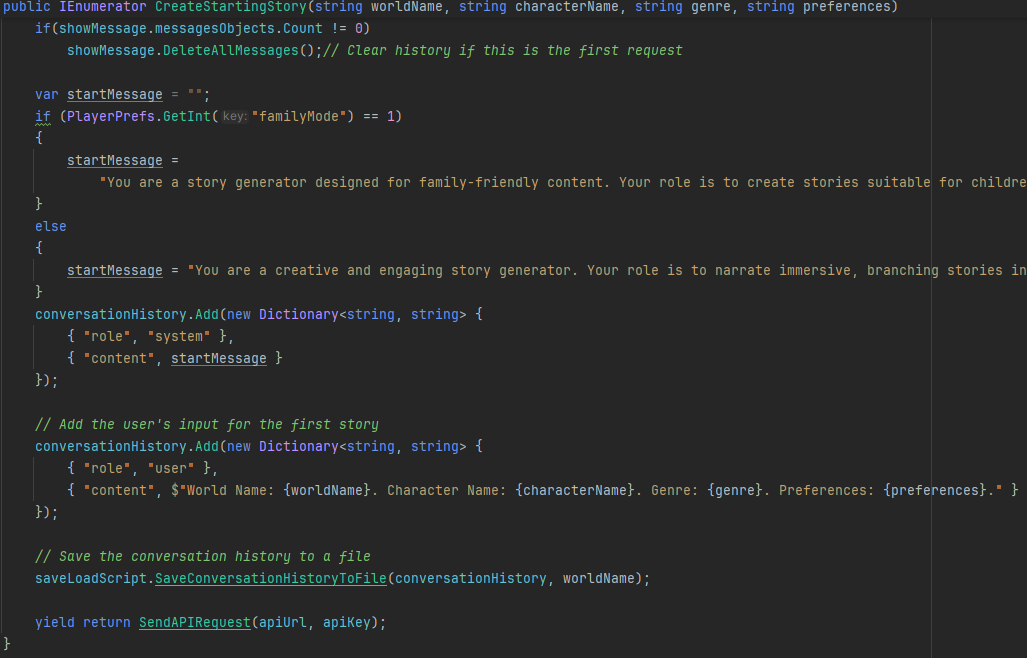
Обрізання історії: Якщо кількість токенів занадто велика, метод TrimConversationHistory видаляє найстаріші повідомлення, залишаючи важливий контекст розмови.

Підсумок

Скрипт TokenCount — це оптимізаційний інструмент, який допомагає керувати розміром історії розмови в межах, встановлених API. Він дозволяє підтримувати баланс між збереженням контексту і дотриманням технічних обмежень. Завдяки цьому гравці можуть продовжувати взаємодію з AI без втрати релевантності у відповідях.

Скрипт SendRequest - це так званий мозок між гравцем та AI моделлю. Цей скрипт обробляє взаємодію між грою і API OpenAI. Він відправляє запити, зберігає історію розмов, обробляє відповіді і підтримує управління токенами для побудови історії в текстовій грі.

Метод CreateStartingStory



Лістинг 3.15.

Метод CreateStartingStory — це центральний елемент для початку гри. Його завдання — створити початкову історію, яка задасть тон і контекст для майбутньої гри. Саме через цей метод гравець знайомиться зі світом, який буде створено, і персонажем, яким він буде керувати. Умовно, цей метод — це "режисер", який виставляє сцену, на якій буде розгортатися вся дія.

Метод починається з підготовки. Гравець, швидше за все, уже ввів певні дані про свій світ (назва, персонаж, жанр), і ці дані передаються в метод як параметри. Проте, перед тим як будувати щось нове, потрібно очистити старе. Для цього в методі викликається функція ClearMemory, яка очищує всі попередні розмови і повідомлення, що залишилися від минулих ігор. Уявімо, що це як стирання старого полотна, щоб малювати на чистому аркуші.

Далі йде важливий момент: визначення, який саме "тон" матиме гра. Це залежить від режиму familyMode. Якщо він активований, то гра орієнтується на сімейний контент. Це означає, що стартове системне повідомлення виглядає приблизно так:  
 "Ви генератор історій для сімейного контенту. Ваше завдання — створювати творчі та безпечні історії, уникаючи будь-якого неприйнятного контенту."

Якщо ж цей режим вимкнено, то стартове повідомлення стає більш вільним:  
 "Ви — креативний генератор історій, здатний створювати захоплюючі та цікаві сюжети для будь-якої аудиторії."

Це схоже на те, як режисер вирішує, чи буде його фільм сімейною комедією або ж епічним трилером.

Після того, як "тон" визначено, система переходить до користувацького запиту. Введені гравцем дані інтегруються в спеціально створений текст-запит, який описує основні аспекти світу. Наприклад, це може виглядати так:  
 "Назва світу: FantasyLand. Ім'я персонажа: Артур. Жанр: Фентезі. Бажання: створити історію з елементами пригод і магії."

Обидва повідомлення — системне і користувацьке — додаються до історії розмови, створюючи початкову структуру. Тепер гра готова до головного: звернення до штучного інтелекту через метод SendAPIRequest. Саме цей запит і перетворює введені дані в реальну історію, з якої почнеться гра.

Останнім етапом методу є очікування завершення запиту і обробка відповіді. Після цього гравець побачить згенеровану історію, яка стане основою для його подорожі в світі гри.

Метод SendAPIRequest

Метод SendAPIRequest є ключовим елементом скрипта, який відповідає за взаємодію з API OpenAI. Це метод, що виконує запит до серверу, передає поточну історію розмови та отримує відповідь у вигляді тексту, який інтегрується у гру. У контексті гри цей метод можна порівняти із "посередником", який забезпечує спілкування між гравцем і штучним інтелектом.

Метод виконує кілька основних завдань:

Формує дані для запиту на основі історії розмови (conversationHistory).

Відправляє HTTP POST-запит до API OpenAI.

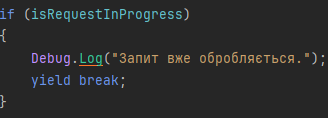
Обробляє відповідь від сервера: додає текст у поточну історію та передає його для відображення гравцю.

Здійснює обробку помилок у випадку, якщо запит не вдався.

Робота методу крок за кроком

1. Перевірка на наявність активного запиту

На самому початку методу йде перевірка:



Лістинг 3.16.

Це необхідно, щоб уникнути ситуації, коли декілька запитів відправляються одночасно. Якщо запит вже обробляється, метод припиняє виконання за допомогою yield break.

2. Встановлення прапора

Далі прапор isRequestInProgress встановлюється в true, що сигналізує про початок обробки запиту:



Лістинг 3.17.

Водночас на екрані може відображатися повідомлення для гравця, що йде завантаження.

3. Формування даних для запиту

Метод готує структуру даних у форматі JSON, яка містить:

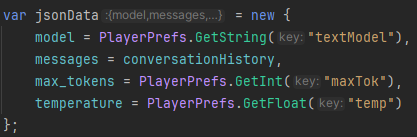
Модель (model): вибирається із налаштувань гравця (PlayerPrefs.GetString("textModel")).

Історію розмови (messages): всі попередні повідомлення гравця та відповіді від AI.

Максимальну кількість токенів (max\_tokens): обмеження для уникнення перевищення ліміту.

Температуру (temperature): налаштування креативності відповіді.

Формування відбувається за допомогою анонімного об'єкта:

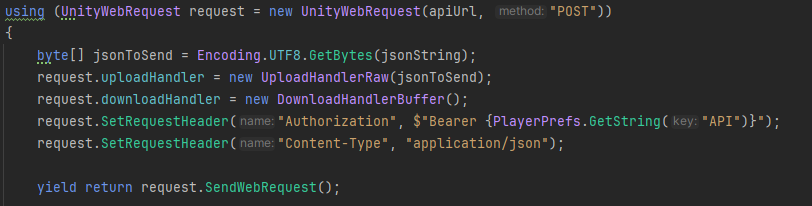


Лістинг 3.18.

Результат зберігається в змінній jsonString, яка потім передається до сервера.

4. Відправлення запиту

Для передачі даних використовується UnityWebRequest:



Лістинг 3.19.

Тут відбувається:

Перетворення JSON-даних у байти.

Створення запиту типу POST.

Встановлення HTTP-заголовків:

Авторизація (Authorization): ключ доступу до API.

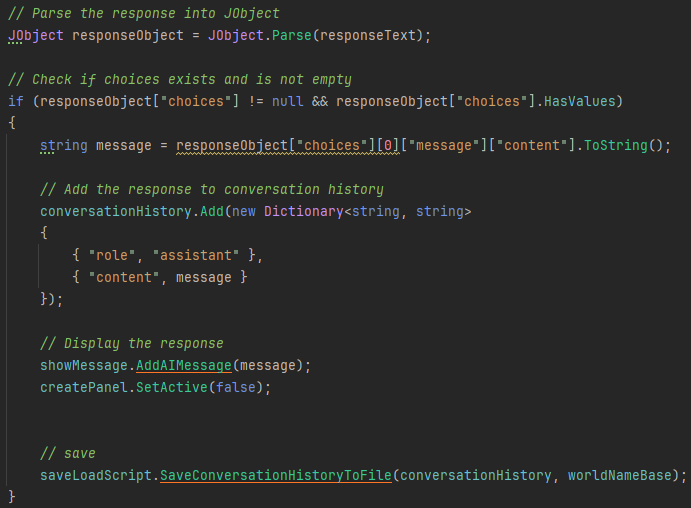
Тип контенту (Content-Type): JSON.

Метод чекає завершення запиту через yield return.

5. Обробка відповіді

Після завершення запиту йде обробка відповіді:

Якщо запит успішний:



Лістинг 3.20.

Парсинг JSON-відповіді (JObject.Parse).

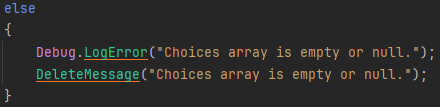
Витяг повідомлення від AI.

Додавання відповіді до історії розмови.

Відображення тексту на екрані через метод AddAIMessage.

Збереження оновленої історії до файлу.

У випадку помилки:



Лістинг 3.21.

Виводиться повідомлення про помилку в консоль, і викликається метод DeleteMessage, який видаляє останнє повідомлення з історії.

Підсумок

Метод SendAPIRequest — це серце інтерактивної гри. Він забезпечує динамічну взаємодію між гравцем і штучним інтелектом, створюючи можливість отримувати унікальні відповіді на основі введених даних. Його структура ретельно продумана, щоб враховувати потенційні помилки, оптимізувати використання ресурсів і зберігати зручність для гравця.

Метод GeneratePromptForImage

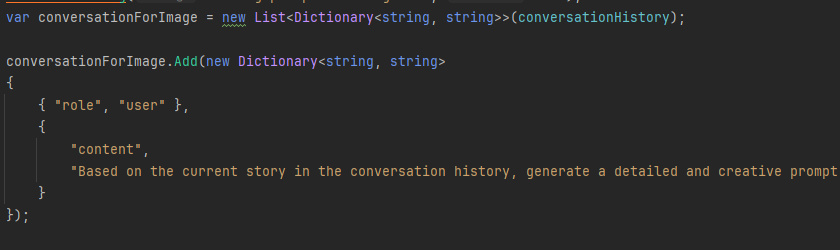
Метод GeneratePromptForImage у представленому коді відповідає за динамічну генерацію текстового промту для моделі DALL·E, який потім використовується для створення візуального зображення на основі історії розмови. Цей процес включає взаємодію з API, обробку відповіді та підготовку отриманих даних для подальшого використання.

Відображення статусу завантаження  
 Метод починається з виклику ShowPanelForLog, який інформує гравця про початок процесу генерації промту:



Лістинг 3.22.

Формування даних для API  
 Історія розмови копіюється у новий список, щоб уникнути модифікації оригінальної історії. У список додається нове повідомлення від імені користувача із запитом на створення промту:



Лістинг 3.23.

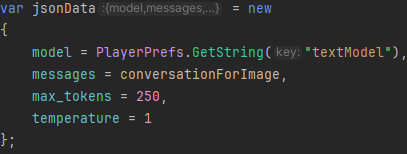
Складання JSON-запиту  
 Запит для API формується як JSON-об'єкт із наступними параметрами:

model: модель, що використовується (зчитується з налаштувань).

messages: сформована історія розмови.

max\_tokens: максимальна кількість токенів для відповіді (250).

temperature: рівень креативності відповіді (1). JSON-об'єкт серіалізується в рядок:



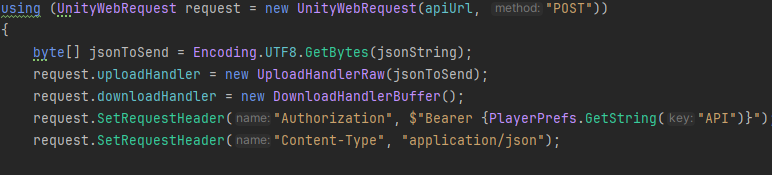
Лістинг 3.23.

Відправка запиту через UnityWebRequest  
HTTP-запит надсилається на API за допомогою UnityWebRequest:

Тип запиту: POST.

Заголовки: Authorization (з токеном доступу) і Content-Type (application/json).

Дані запиту: jsonString, закодовані у форматі UTF-8.



Лістинг 3.24.

Підсумок роботи

Метод GeneratePromptForImage виконує такі основні завдання:

Формує запит на основі історії розмови.

Надсилає запит до API OpenAI для генерації текстового промту.

Перевіряє і обробляє відповідь.

Передає отриманий текст промту у метод, що займається створенням зображення.

Цей метод є важливим елементом інтеграції текстового і графічного аспектів гри, забезпечуючи їхню взаємодію та адаптацію під користувацький досвід.

Скрипт ImageSendRequest є важливою частиною вашої Unity-гри, оскільки він відповідає за взаємодію з OpenAI API для генерації зображень, використовуючи модель DALL·E, а також завантаження та збереження цих зображень у додатку.

Ініціалізація та налаштування

На початку роботи скрипта, в методі Start, відбувається ініціалізація компонента SendRequest, що забезпечує доступ до історії розмови та інших необхідних даних для запитів до API. Це дозволяє використовувати вже надану інформацію для створення зображення, яке відповідає контексту поточної розмови.



Лістинг 3.25.

Процес генерації зображення

Основна функціональність цього скрипта реалізована в методі GenerateImage. Він відповідає за надсилання запиту до API для створення зображення на основі наданого текстового опису (prompt).

Перш ніж виконати запит, перевіряється, чи є в історії розмови хоча б один елемент. Якщо ж історія порожня, метод просто завершиться без дій.



Лістинг 3.26.

Далі, з'являється візуальне повідомлення для користувача, що зображення генерується, щоб він міг слідкувати за процесом. Текстове повідомлення, яке генерується в процесі гри, передається як "prompt" в API:



Лістинг 3.27.

Далі формується JSON-об'єкт, що містить інформацію для запиту. В цей об'єкт входить модель, яка вказана в налаштуваннях користувача, сам "prompt", параметри кількості зображень і розміру (в даному випадку 1024x1024 пікселів):



Лістинг 3.28.

Запит відправляється через UnityWebRequest методом POST. Важливо, що в заголовки додаються ключі для авторизації за допомогою токену API, щоб забезпечити доступ до моделі DALL·E:

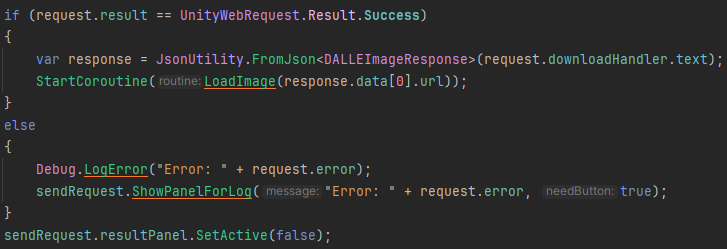


Лістинг 3.29.

Обробка відповіді від API

Якщо API відповідає успішно, скрипт розбирає JSON-відповідь і отримує URL зображення, яке було створене. Потім викликається метод, що завантажує це зображення з отриманого URL.

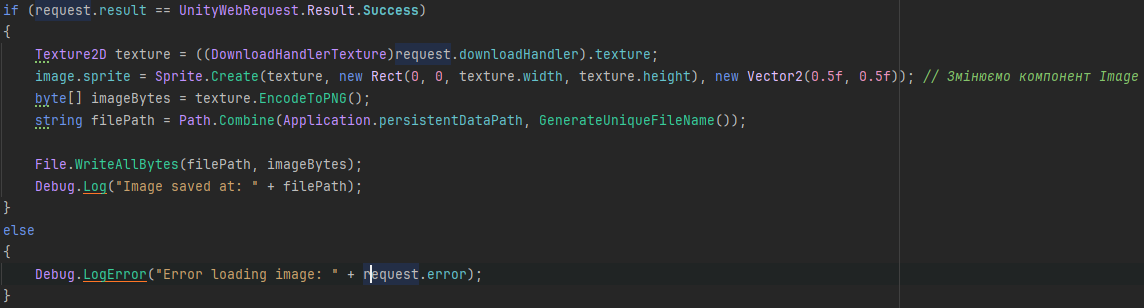
Якщо сталася помилка, користувач отримає повідомлення про помилку, і буде виведено відповідне повідомлення в консоль.



Лістинг 3.30.

Збереження зображення на пристрої

Ключовим моментом є те, що після того як зображення завантажено з API, воно не тільки відображається на екрані, але й зберігається на диску пристрою для подальшого використання. Метод LoadImage здійснює завантаження зображення через URL, а потім перетворює текстуру в PNG формат, після чого зберігає файл. Всі ці операції виконуються асинхронно, щоб не блокувати основний потік програми та забезпечити плавність користувацького досвіду.



Лістинг 3.31.

Генерація унікального імені файлу

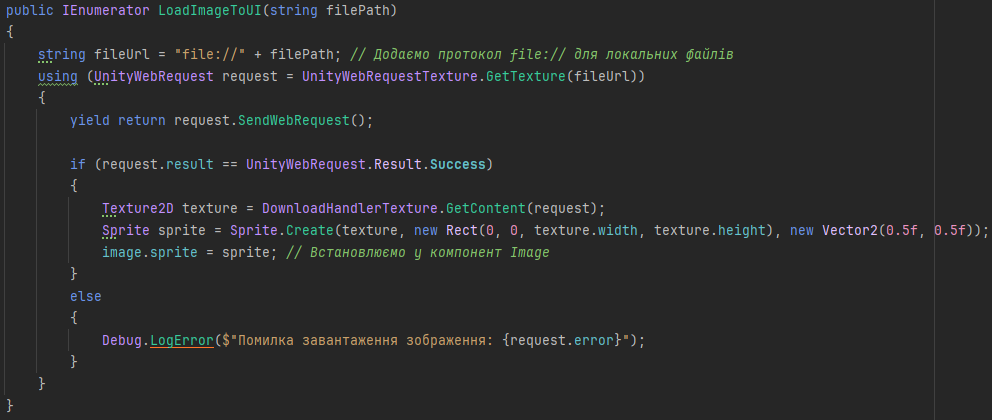
Для того, щоб кожне зображення мало унікальне ім’я, використовується метод GenerateUniqueFileName. Ім'я файлу генерується на основі поточної дати і часу, що дозволяє зберігати зображення без перезаписування попередніх. Це важливо, щоб користувач мав доступ до кожного створеного зображення.



Лістинг 3.32.

Завантаження зображення з локального файлу

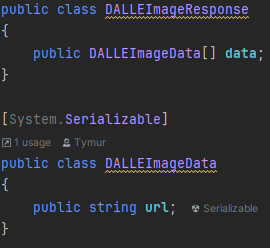
Якщо зображення зберігається на пристрої, існує можливість завантажити його знову в UI. Метод LoadImageToUI відповідає за завантаження зображення з локального шляху на пристрої, використовуючи протокол file:// для доступу до локальних файлів. Це дає змогу користувачу переглядати зображення без необхідності повторно завантажувати його з API.



Лістинг 3.33.

Структури для обробки відповіді API

Для коректної обробки JSON-відповіді від API, використовуються серіалізовані класи, які дозволяють легко парсити дані. В даному випадку, клас DALLEImageResponse містить масив об'єктів DALLEImageData, що дозволяє отримати URL згенерованого зображення. Це дозволяє без проблем витягти URL з відповіді, з яким далі працює програма:



Лістинг 3.34.

Підсумок

Скрипт ImageSendRequest є основним елементом для створення зображень у вашій грі за допомогою OpenAI DALL·E. Він генерує зображення на основі текстового опису, обробляє відповіді від API, завантажує та відображає зображення на екран, а також зберігає його на пристрої для подальшого використання. Цей процес автоматизує багато кроків, роблячи інтерактивний процес створення контенту для користувача набагато зручнішим і ефективнішим.

## **3.3 Тестування та перевірка роботи гри**

Тестування та перевірка є важливими етапами розробки будь-якої гри, оскільки вони дозволяють забезпечити якість, стабільність і надійність програмного продукту. У цьому підрозділі розглядаються методи, які використовуються для перевірки коректності роботи ключових функціональних компонентів гри, зокрема генерації зображень через OpenAI API, управління історією розмов та взаємодії користувача з інтерфейсом гри.

У процесі тестування було використано як автоматизовані, так і ручні методи для виявлення помилок, збоїв та перевірки відповідності вимогам. Оскільки основна мета гри полягає в інтерактивному створенні світу та генеруванні зображень за допомогою штучного інтелекту, увага була приділена перевірці всіх етапів цього процесу, включаючи відправку запитів до API, обробку отриманих результатів і виведення зображень на екран.

Тестування генерації зображень

Одним із ключових моментів у тестуванні є перевірка функціональності скриптів, які відповідають за генерування зображень на основі текстових запитів. Для цього використовуються юніт-тести, які автоматично перевіряють, чи правильно формуються запити до API, чи коректно відображаються згенеровані зображення в інтерфейсі гри та чи забезпечується належне збереження зображень на пристрої користувача.

Тестування було спрямоване на кілька важливих аспектів:

Перевірка формування правильного запиту до API, що включає модель, параметри та "prompt".

Перевірка того, чи API повертає коректну відповідь, а саме URL для зображення.

Тестування процесу завантаження зображення з отриманого URL та його відображення в інтерфейсі гри.

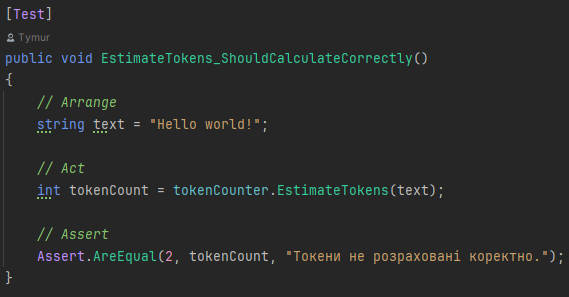
Перевірка функції збереження зображення на пристрої та його правильного відображення при наступному запуску гри.

Для кожного з цих етапів було написано окремі тести, які допомогли виявити проблеми на ранніх етапах розробки.

Тестування класу TokenCount спрямоване на перевірку коректності роботи алгоритмів, які відповідають за оцінку та обмеження кількості токенів у текстових повідомленнях і розмовах. Ці методи важливі для управління історією повідомлень, особливо коли треба враховувати ліміти токенів при відправці запитів до API, таких як OpenAI. Тести, описані в коді, орієнтовані на перевірку кількох основних аспектів, що стосуються підрахунку та обробки токенів, кожен з яких перевіряється через чітко визначені сценарії.

1. Тест EstimateTokens\_ShouldCalculateCorrectly

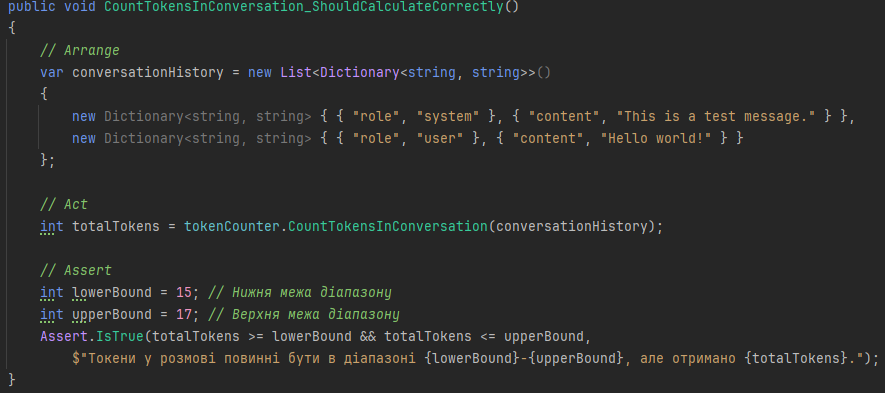
Цей тест перевіряє базову функціональність методу EstimateTokens, що має на меті оцінити кількість токенів в окремому текстовому повідомленні. В якості тестового вводу використовуються прості текстові рядки, щоб переконатися, що підрахунок токенів виконується коректно. В результаті тесту передбачається, що рядок "Hello world!" буде розділений на два токени, оскільки кожне слово вважатиметься окремим токеном. Тест гарантує, що метод правильно інтерпретує кількість токенів і повертає очікуваний результат.



Лістинг 3.35.

2. Тест CountTokensInConversation\_ShouldCalculateCorrectly

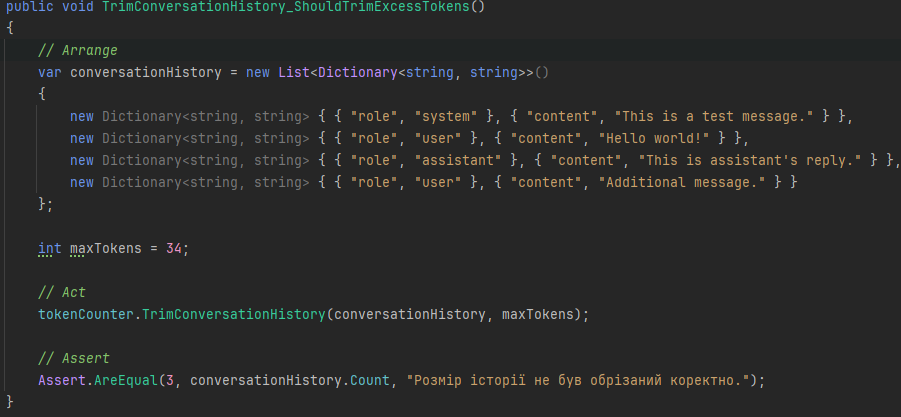
Цей тест перевіряє метод CountTokensInConversation, який підсумовує токени з усієї історії розмови. Історія складається з кількох повідомлень від різних ролей — системи, користувача, асистента. Важливим аспектом є правильний підрахунок токенів для кожного повідомлення та обчислення загальної кількості. Тест перевіряє, що результат знаходиться в межах допустимого діапазону, оскільки різні повідомлення можуть мати різну довжину, і необхідно переконатися, що підрахунок враховує всі нюанси. Тому для цієї перевірки задаються верхні та нижні межі, в межах яких має варіюватися загальна кількість токенів, забезпечуючи правильність і точність алгоритму.



Лістинг 3.36.

3. Тест TrimConversationHistory\_ShouldTrimExcessTokens

Цей тест оцінює метод TrimConversationHistory, який обмежує історію повідомлень за кількістю токенів. Якщо загальна кількість токенів перевищує задане обмеження (в даному випадку 34 токени), метод повинен обрізати історію таким чином, щоб вмістити лише необхідну кількість токенів. Цей тест має на меті перевірку того, що алгоритм коректно обрізає зайві повідомлення та забезпечує відповідність ліміту токенів. Важливо, що після обрізання історія не повинна втратити важливі дані і зберігати баланс між коректною кількістю токенів та кількістю повідомлень.

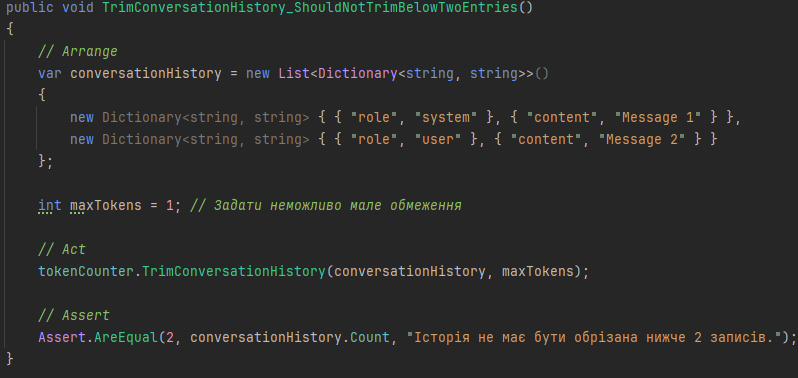


Лістинг 3.37.

4. Тест TrimConversationHistory\_ShouldNotTrimBelowTwoEntries

Тест, що перевіряє обмеження на мінімальну кількість записів в історії, не дозволяючи обрізати її до менш ніж двох записів. Цей тест важливий, оскільки він гарантує, що навіть у випадку надзвичайно суворого обмеження на кількість токенів історія розмови не буде занадто урізана, що може призвести до втрати контексту важливих повідомлень. Наприклад, навіть при дуже малому ліміті токенів система повинна зберігати хоча б два запису для підтримки базового контексту розмови.

Таким чином, ці тести забезпечують не тільки правильність підрахунку токенів, але й перевіряють, чи правильно система справляється з обмеженнями та обрізанням історії повідомлень, що є критично важливим для оптимальної роботи гри та ефективного використання API.

  
Лістинг 3.38.

Тестування класу SaveLoadScript фокусується на перевірці функціональності збереження та завантаження історії розмови, а також очищення кнопок, пов'язаних із збереженими світами. Метою таких тестів є впевненість у тому, що методи коректно обробляють дані, правильно зберігають та відновлюють інформацію, а також забезпечують належну очистку елементів інтерфейсу.

1. Тест SaveConversationHistoryToFile\_ShouldSaveCorrectly

Цей тест перевіряє, чи зберігається історія розмови в файл коректно. У тесті створюється тестова історія, що складається з двох повідомлень: одного від користувача та одного від асистента. Метод SaveConversationHistoryToFile викликається для збереження цієї історії в файл. Після виконання методу тест перевіряє, чи файл був створений на диску, чи містить він правильний вміст, і чи кількість повідомлень після десеріалізації відповідає початковим даним. Цей тест важливий для перевірки того, чи правильно зберігаються дані у файл у форматі JSON і чи можна ці дані потім коректно завантажити.



Лістинг 3.39.

2. Тест LoadConversationHistoryFromFile\_ShouldLoadCorrectly

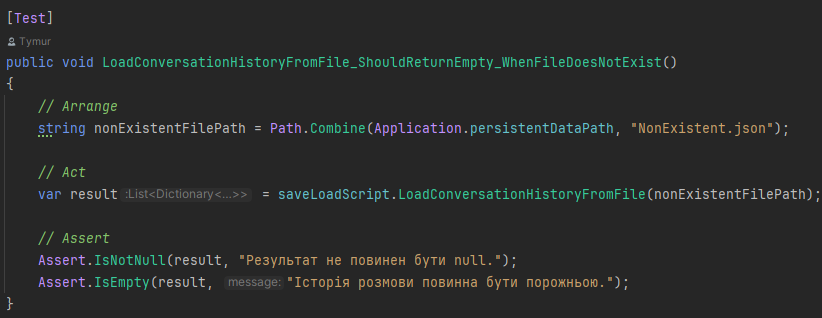
Тестує метод LoadConversationHistoryFromFile, який має завантажити історію розмови з файлу. Спочатку в тесті в файл записується тестова історія. Потім тест перевіряє, чи завантажується вміст файлу правильно, зокрема чи кількість елементів у завантаженій історії співпадає з кількістю в оригінальному списку, і чи зміст повідомлень також зберігається. Важливість цього тесту полягає в перевірці того, чи працює механізм десеріалізації і чи правильно зчитуються дані з файлу.



Лістинг 3.40.

3.Тест LoadConversationHistoryFromFile\_ShouldReturnEmpty\_WhenFileDoesNotExist

Цей тест оцінює поведінку методу LoadConversationHistoryFromFile у випадку, коли файл не існує. Оскільки система повинна коректно реагувати на відсутність файлу, тест перевіряє, чи метод повертає порожній список замість виключення або помилки. Це тестує здатність системи безпечно працювати з відсутніми файлами та гарантує, що система не буде аварійно завершувати роботу в таких випадках.



Лістинг 3.41.

4. Тест CleanBttn\_ShouldCleanAllButtons

Тест перевіряє метод CleanBttn, який відповідає за очищення списку кнопок на інтерфейсі. У тесті створюються два кнопки з тегом "loadbttn", після чого викликається метод очищення. Тест перевіряє, чи був список кнопок успішно очищений після виклику методу. Це забезпечує, що інтерфейс не зберігає непотрібні кнопки, що можуть заважати користувацькому досвіду.



Лістинг 3.42.

Всі ці тести виконуються за допомогою моків для зовнішніх залежностей, таких як ShowMessage, SendRequest та ImageSendRequest, що дозволяє ізолювати тестування логіки класу SaveLoadScript від інших компонентів гри. Моки створюються для запобігання виконанню реальних методів, які можуть мати побічні ефекти або бути не потрібними для даного тестування, таких як відправка запитів чи обробка повідомлень в інтерфейсі.

Тестування класу ShowMessage має на меті перевірку правильності роботи з відображенням та керуванням повідомленнями в інтерфейсі гри. Кожен тест фокусується на верифікації того, чи коректно додаються, видаляються або оновлюються повідомлення, а також чи взаємодіє клас правильно з іншими компонентами, такими як SaveLoadScript.

1. Тест DeleteAllMessages\_ShouldClearAllMessages

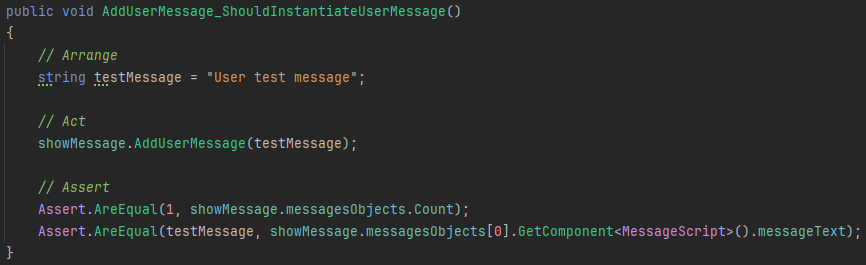
Цей тест перевіряє, чи метод DeleteAllMessages правильно очищує всі повідомлення, що відображаються в інтерфейсі. Під час тестування додаються два повідомлення (від користувача та асистента) до списку відображених повідомлень. Після виклику методу DeleteAllMessages тест перевіряє, чи були очищені всі елементи в списку повідомлень messagesObjects та в списку arr з компоненту SaveLoadScript, що відповідає за збереження історії розмови. Це дозволяє перевірити, чи коректно обробляється очищення елементів інтерфейсу та чи не зберігаються непотрібні елементи в системі.



Лістинг 3.43.

2. Тест AddUserMessage\_ShouldInstantiateUserMessage

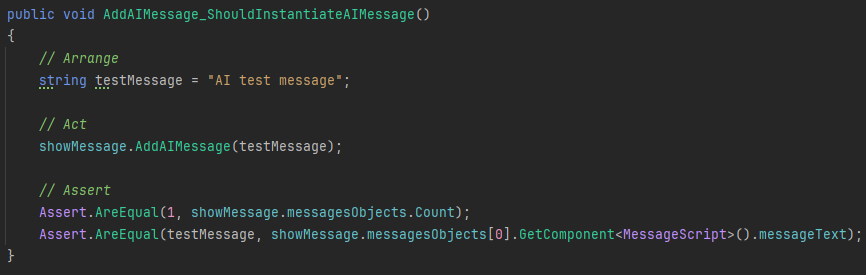
Тест перевіряє, чи коректно створюється та додається повідомлення користувача за допомогою методу AddUserMessage. Коли передається текст повідомлення, метод повинен створити новий об'єкт повідомлення і додати його до списку. Тест перевіряє, чи правильно встановлений текст повідомлення в компоненті MessageScript, що відповідає за виведення повідомлення на екран, і чи відображається повідомлення користувача в правильному списку. Це важливий тест для верифікації, чи працює механізм додавання нових повідомлень в систему.



Лістинг 3.44.

3. Тест AddAIMessage\_ShouldInstantiateAIMessage

Цей тест аналогічний попередньому, але для повідомлення від штучного інтелекту. Перевіряється, чи коректно створюється та відображається повідомлення, яке надходить від асистента. Тест підтверджує, що для кожного типу повідомлення (користувач чи асистент) створюється відповідний об'єкт, який містить правильний текст, і що повідомлення відображається у відповідному списку. Це важливо для забезпечення правильного відображення повідомлень на екрані.



Лістинг 3.45.

Після виконання тестів для всіх ключових компонентів гри, було підтверджено, що всі тести пройшли успішно. Це означає, що система працює стабільно і надає правильні результати в межах встановлених вимог.

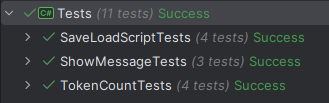


Рис.3.1. Результат виконання тестів

Тести для класу TokenCount підтвердили, що методи, відповідальні за оцінку кількості токенів і коректну обрізку історії розмови, працюють належним чином. Це забезпечує правильну обробку та контроль за використанням токенів під час взаємодії з API, що є важливим аспектом для стабільної роботи гри при великих обсягах даних.

Тести для класу SaveLoadScript довели, що механізм збереження та завантаження історії розмови працює коректно. Метод SaveConversationHistoryToFile успішно зберігає дані, а метод LoadConversationHistoryFromFile вірно відновлює історію, навіть у випадку відсутності файлу. Крім того, перевірка очищення списку кнопок за допомогою методу CleanBttn показала, що система правильно взаємодіє з інтерфейсними елементами.

Тести для класу ShowMessage підтвердили, що додавання, відображення та видалення повідомлень працюють без помилок. Кожен метод, відповідальний за створення повідомлень користувача та асистента, успішно додає нові елементи до інтерфейсу, а метод очищення коректно видаляє всі повідомлення.

Ручне тестування гри

Ручне тестування є важливою частиною процесу розробки, оскільки воно дозволяє виявити потенційні проблеми, які можуть бути не помічені під час автоматизованих тестів. У випадку з цією грою, ручне тестування охоплює перевірку всіх основних функціональних аспектів, зокрема взаємодії користувача з інтерфейсними елементами, механізмів генерації контенту, збереження та відновлення історії розмов та інші ключові процеси, що впливають на геймплей.

1. Тестування інтерфейсу користувача

Одним із перших етапів ручного тестування було перевірка правильності відображення інтерфейсних елементів гри. Під час цього етапу тестувалася коректність роботи таких компонентів:

- Кнопки: Всі кнопки повинні працювати згідно з їх функціоналом, наприклад, кнопка для генерації нового світу або кнопка для завантаження попередньої історії. Перевірялося, чи кнопки відповідають на натискання, чи відображаються вони у відповідних місцях і чи не виникають помилки при їх використанні.

- Поле введення тексту: Поле введення, яке дозволяє користувачеві вказувати жанр або тему світу, було протестовано на коректність введення тексту, а також на правильне відображення введеного тексту на екрані.

- Перегляд історії повідомлень: У процесі тестування перевірялося, чи можна прокручувати історію розмови без збоїв, а також чи правильно відображаються повідомлення користувача та асистента.

- Завершення сеансу гри: Під час тестування також перевірялося, чи можна коректно завершити сеанс гри, зберігаючи всі налаштування та історію.

2. Тестування генерації світу та історії

Цей етап тестування був спрямований на перевірку функціональності механізмів генерації світу на основі введених даних. Тестувалося, чи правильно працює система генерації:

- Введення теми світу: Користувач має можливість ввести тему чи жанр світу, і система повинна генерувати світ на основі цих даних. Під час тестування перевірялися різні типи введених тем і жанрів, щоб перевірити, чи змінюється виведений контент залежно від вводу.

- Генерація історії: Після створення світу, система генерує історію для цього світу. Під час тестування перевірялося, чи відповідає історія заданому жанру та чи є в ній зв’язок із попереднім виведеним контекстом.

- Правильність взаємодії з API: Під час цього етапу тестування перевірялася взаємодія гри з API, що використовується для генерації тексту (наприклад, OpenAI API). Важливо було перевірити, чи система надсилає правильні запити та отримує відповіді без помилок.

3. Тестування збереження та завантаження даних

Одним з критичних аспектів гри є збереження та відновлення історії розмов, що дозволяє користувачам продовжувати гру з того місця, де вони зупинилися. Під час ручного тестування цього аспекту перевірялися наступні сценарії:

- Збереження історії: Після кількох взаємодій з грою, перевірялося, чи коректно зберігається історія розмови в файл. Важливо було перевірити, чи всі повідомлення зберігаються в порядку, в якому були введені.

- Завантаження історії: Після того як файл з історією був збережений, система повинна була коректно завантажити його при наступному запуску гри. Перевірялося, чи всі записи в історії відновлюються, чи правильно відображаються повідомлення користувача та асистента.

- Проблеми з відсутністю файлу: Тестувалося, чи система правильно поводиться, якщо файл з історією відсутній або був пошкоджений. В таких випадках гра повинна коректно повідомити користувача та не викликати помилок.

4. Тестування механізмів генерації зображень

Генерація зображень є важливим компонентом гри, оскільки вона дозволяє візуалізувати створений світ. Під час ручного тестування цієї частини перевірялося:

- Генерація зображень за допомогою API: Потрібно було перевірити, чи правильно працює механізм генерації зображень, які базуються на описах, введених користувачем. Тестувалася правильність згенерованих зображень, їх відповідність вказаному опису, а також чи зберігаються вони на пристрої у вигляді файлів.

- Завантаження зображень для відображення в інтерфейсі: Після того як зображення були згенеровані, перевірялося, чи вони коректно завантажуються і відображаються в інтерфейсі користувача. Також перевірялося, чи зображення правильно зберігаються та можуть бути завантажені в разі необхідності.

5. Тестування на різних пристроях

Для забезпечення стабільної роботи гри на різних платформах було проведено тестування гри на кількох пристроях. Це включало:

- Перевірка на Linux: Тестувалася коректність гри на Linux, зокрема взаємодія з мишкою та клавіатурою, а також швидкість завантаження та роботи API.

- Перевірка на Windows: Тестувалася коректність гри на Windows, зокрема взаємодія з мишкою та клавіатурою, а також швидкість завантаження та роботи API.

6. Тестування на продуктивність

Останнім етапом ручного тестування було перевірка продуктивності гри при високих навантаженнях. Це включало:

- Перевірка на великих обсягах даних: Тестувалося, як гра обробляє велику кількість повідомлень в історії розмов та велику кількість згенерованих елементів світу. Перевірялося, чи не виникають проблеми з затримками або збоями при обробці великих обсягів даних.

- Тестування при максимальних налаштуваннях: Перевірка гри при максимальних налаштуваннях графіки та складності генерації контенту для виявлення можливих проблем із продуктивністю.

Висновок

Ручне тестування дозволило не тільки перевірити функціональність усіх важливих механізмів гри, але й виявити кілька проблем, що не були помічені під час автоматизованих тестів. Виявлені помилки були виправлені, а механізми гри адаптовані для коректної роботи в реальних умовах використання. Ручне тестування виявилось ефективним інструментом для верифікації якості гри, забезпечуючи стабільність та надійність її роботи.

.

## **Висновки до третього розділу**

В цьому розділі ми розглянули розробку програмного коду, що відповідає за різні аспекти гри, зокрема за управління історією розмов, взаємодію з користувачем та збереженням даних. Ми обговорили роботу зі скриптами, що керують підрахунком токенів, збереженням та завантаженням історії повідомлень, а також відображенням повідомлень у UI.

У цьому розділі ми дізналися, як використовувати різні методи для оцінки кількості токенів у повідомленнях, як обробляти історію розмов і забезпечувати її збереження на диску, а також як правильно управляти відображенням повідомлень за допомогою UI елементів. Окремо варто згадати автоматизовані тести, які перевіряють правильність роботи функціоналу, що дозволяє забезпечити стабільність гри на всіх етапах її взаємодії з користувачем.

Крім того, ми розглянули важливість ручного тестування, яке доповнює автоматизовані тести, перевіряючи загальну функціональність гри в реальних умовах. Це дозволяє виявляти помилки, що можуть бути непомічені в процесі автоматичного тестування.

Цей розділ дав нам уявлення про те, як розробляти програмний код для обробки даних та взаємодії з користувачем у грі, як організовувати збереження даних, і як забезпечити належну стабільність та якість функціоналу за допомогою тестів.

## **ВИСНОВКИ**

Під час роботи над кваліфікаційною роботою був розроблений і реалізований проєкт текстової гри, де ключовим елементом є генерація світу та історії на основі введених користувачем даних. В рамках цього проєкту були розроблені та реалізовані різноманітні механіки взаємодії з користувачем, управління історією розмов, підрахунок токенів, збереження та завантаження даних, а також відображення повідомлень у інтерфейсі.

Початково була проведена аналіз потреб користувачів та технічних вимог, що дозволило визначити основні аспекти гри, такі як текстові діалоги, генерування світу та взаємодія з історією повідомлень. Було створено структуру даних, яка дозволяє зберігати розмови та налаштовувати гнучку взаємодію з користувачем.

Далі, було проведено детальне тестування програмного коду. В рамках автоматизованого тестування ми перевірили коректність підрахунку токенів, правильність роботи системи збереження та завантаження історії, а також перевірили працездатність механізмів, що відповідають за відображення повідомлень у інтерфейсі. Ручне тестування також підтвердило стабільність функціональності в реальних умовах використання, що дозволило виявити можливі недоліки та вдосконалити взаємодію між компонентами.

Було реалізовано такі ключові функціональні компоненти:

1. Система підрахунку токенів: коректно визначає кількість токенів у тексті та історії розмов.

2. Система збереження та завантаження даних: дозволяє зберігати та відновлювати історію розмов у файлах.

3. Система відображення повідомлень: забезпечує коректне відображення повідомлень користувача та асистента.

4. Автоматизовані та ручні тести: дають впевненість у надійності функціональних механізмів гри.

Цей етап роботи дозволив успішно реалізувати та перевірити основні компоненти гри, забезпечити стабільність і працездатність усіх механізмів, а також гарантувати, що всі функції працюють згідно з вимогами. Результати тестування, як автоматизованого, так і ручного, показали, що розроблений проєкт відповідає всім технічним та функціональним вимогам.

## **СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. Adams E., Dormans J. Game mechanics: advanced game design. Pearson Education, Limited, 2012.
2. AI for game developers. O'Reilly Media, Inc., 2004. 390 p.
3. Bond J. G. Introduction to game design, prototyping, and development: from concept to playable game with unity and C#. Pearson Education, Limited, 2017.
4. Catalonian Conference on Ai 2005 Alghero, Lopez B. Artificial intelligence research and development (frontiers in artificial intelligence and applications). IOS Press, 2005. 452 p.
5. Game design: a practical approach. Boston, Mass : Charles River Media, 2007. 396 p.
6. Game programming patterns. Genever Benning, 2014. 354 p.
7. Interactive storytelling / ed. by A.-G. Bosser, D. E. Millard, C. Hargood. Cham : Springer International Publishing, 2020. URL: https://doi.org/10.1007/978-3-030-62516-0 (date of access: 20.12.2024).
8. Knuth D. E. Art of computer programming - volume 1. Lulu Press, Inc., 2016.
9. Levy S. Artificial life: a report from the frontier where computers meet biology. New York : Vintage Books, 1993. 390 p.
10. Procedural content generation in games. Springer International Publishing AG, 2018.
11. Rabin S. Introduction to game development (game development series). Charles River Media, 2005. 978 p.
12. Schell J. Art of game design: a book of lenses, second edition. Taylor & Francis Group, 2017.
13. Schell J. Art of game design: a book of lenses. Taylor & Francis Group, 2008.
14. Shaker N., Togelius J., Nelson M. J. Procedural content generation in games. Cham : Springer International Publishing, 2016. URL: https://doi.org/10.1007/978-3-319-42716-4 (date of access: 20.12.2024).
15. Sylvester T. Designing games: a guide to engineering experiences. O'Reilly Media, Incorporated, 2013.Tisdell E. J., Merriam S. B. Qualitative research: a guide to design and implementation. Wiley & Sons, Incorporated, John, 2015. 368 p.
16. Tisdell E. J., Merriam S. B. Qualitative research: a guide to design and implementation. Wiley & Sons, Incorporated, John, 2015. 368 p.Togelius J., Yannakakis G. N. Artificial intelligence and games. Springer, 2018. 337 p.
17. Turing A. M. Computing machinery and intelligence. Harmondsworth : Penguin, 1981.
18. Turing A. M. Computing machinery and intelligence. Harmondsworth : Penguin, 1981.

# 

# **ДОДАТОК**

Додаток А SaveLoadScript.сs

using System;

using System.IO;

using Newtonsoft.Json;

using System.Collections.Generic;

using System.Linq;

using TMPro;

using Unity.VisualScripting;

using UnityEngine;

using UnityEngine.Networking;

using UnityEngine.UI;

public class SaveLoadScript : MonoBehaviour

{

public List<GameObject> arr = new List<GameObject>();

public GameObject bttnPrefab;

public Transform scrollContent;

public ShowMessage showMessage;

public GameObject loadPanel;

public SendRequest sendRequest;

public ImageSendRequest imageSendRequest;

public void Start()

{

showMessage = gameObject.GetComponent<ShowMessage>();

sendRequest = gameObject.GetComponent<SendRequest>();

imageSendRequest = gameObject.GetComponent<ImageSendRequest>();

}

// Метод для збереження історії розмови у файл

public void SaveConversationHistoryToFile(List<Dictionary<string, string>> conversationHistory, string worldName)

{

// Шлях до файлу зберігання

string filePath = Path.Combine(Application.persistentDataPath, $"{worldName}.json");

Debug.Log(Application.persistentDataPath);

// Перевірка на існування файлу

if (!File.Exists(filePath))

{

// Якщо файл не існує, створити новий файл

File.Create(filePath).Dispose(); // Використовуємо Dispose для негайного звільнення ресурсу

}

// Серіалізація історії в JSON формат

string jsonData = JsonConvert.SerializeObject(conversationHistory, Formatting.Indented);

// Запис JSON даних у файл

File.WriteAllText(filePath, jsonData);

Debug.Log($"Conversation history saved to {filePath}");

}

// Clean prefabs

public void CleanBttn()

{

if(arr.Count == 0)

return;

foreach (var i in arr)

{

Destroy(i);

}

arr.Clear();

}

//load files and bttns

public void ReadAndLoadBttns()

{

string savePath = Application.persistentDataPath;

string[] files = Directory.GetFiles(savePath, "\*.json");

CleanBttn();

foreach (string file in files)

{

string worldName = Path.GetFileNameWithoutExtension(file);

GameObject button = Instantiate(bttnPrefab, scrollContent);

button.GetComponentInChildren<TMP\_Text>().text = worldName;

button.GetComponent<Button>().onClick.AddListener(() => LoadWorld(file));

}

}

private void LoadWorld(string filePath)

{

List<Dictionary<string, string>> conversationHistory = LoadConversationHistoryFromFile(filePath);

string worldName = Path.GetFileNameWithoutExtension(filePath);

Debug.Log($"World loaded: {worldName}");

if (showMessage.messagesObjects.Count != 0)

{

showMessage.DeleteAllMessages();

}

int i = 0;

foreach (var message in conversationHistory)

{

string role = message["role"];

string content = message["content"];

if (role == "user" && content != "")

{

showMessage.AddUserMessage(content);

}

else if (role == "assistant")

{

showMessage.AddAIMessage(content);

}

i++;

}

sendRequest.LoadConversationHistory(filePath);

GetLatestImageFile();

loadPanel.SetActive(false);

}

public void GetLatestImageFile()

{

string folderPath = Application.persistentDataPath;

string fileExtension = ".png";

string[] files = Directory.GetFiles(folderPath, $"\*{fileExtension}");

var filteredFiles = files.Where(file =>

{

string fileName = Path.GetFileNameWithoutExtension(file);

return fileName.StartsWith(sendRequest.worldNameBase) && long.TryParse(fileName.Split('\_').Last(), out \_);

});

if (!filteredFiles.Any())

{

Debug.LogWarning("No image files found matching the format.");

return;

}

string latestFile = filteredFiles

.OrderByDescending(file =>

{

string fileName = Path.GetFileNameWithoutExtension(file);

long timestamp = long.Parse(fileName.Split('\_').Last());

return timestamp;

})

.FirstOrDefault();

// Виклик корутини

StartCoroutine(imageSendRequest.LoadImageToUI(latestFile));

}

// Метод для завантаження історії розмови з файлу

public List<Dictionary<string, string>> LoadConversationHistoryFromFile(string filePath)

{

// Перевірка на існування файлу перед завантаженням

if (File.Exists(filePath))

{

string jsonData = File.ReadAllText(filePath);

return JsonConvert.DeserializeObject<List<Dictionary<string, string>>>(jsonData);

}

else

{

// Якщо файл не існує, повернути порожній список

Debug.LogWarning("File not found. Returning empty history.");

return new List<Dictionary<string, string>>();

}

}

}

Додаток Б SendRequest.cs

using System;

using System.Collections;

using System.Collections.Generic;

using System.IO;

using System.Text;

using UnityEngine;

using UnityEngine.Networking;

using Newtonsoft.Json;

using Newtonsoft.Json.Linq;

using TMPro;

using UnityEngine.UIElements;

public class SendRequest : MonoBehaviour

{

// Memory to store the conversation history

public List<Dictionary<string, string>> conversationHistory = new List<Dictionary<string, string>>();

public string apiUrl;

public string apiKey;

public ShowMessage showMessage;

public ImageSendRequest imageSendRequest;

public GameObject createPanel;

public TokenCount tokenCount;

public SaveLoadScript saveLoadScript;

public int maxTokens = 125000;

public string worldNameBase; // Base name to access the saved file

public int count;

public GameObject resultPanel;

public bool isRequestInProgress = false;

public TMP\_InputField inputField;

public void ShowPanelForLog(string message, bool needButton)

{

resultPanel.SetActive(true);

resultPanel.transform.GetChild(1).gameObject.SetActive(false);

resultPanel.transform.GetChild(0).GetComponent<TMP\_Text>().text = message;

if(needButton)

resultPanel.transform.GetChild(1).gameObject.SetActive(true);

}

public void Start()

{

apiKey = PlayerPrefs.GetString("API");

apiUrl = "https://api.openai.com/v1/chat/completions";

showMessage = GetComponent<ShowMessage>();

tokenCount = GetComponent<TokenCount>();

saveLoadScript = GetComponent<SaveLoadScript>();

imageSendRequest = GetComponent<ImageSendRequest>();

count = 0;

}

// First request to start the story.

public IEnumerator CreateStartingStory(string worldName, string characterName, string genre, string preferences)

{

worldNameBase = worldName; // Save the world name to a separate variable for later access

ClearMemory();

if(showMessage.messagesObjects.Count != 0)

showMessage.DeleteAllMessages();// Clear history if this is the first request

var startMessage = "";

if (PlayerPrefs.GetInt("familyMode") == 1)

{

startMessage =

"You are a story generator designed for family-friendly content. Your role is to create stories suitable for children, ensuring they are engaging, positive, and appropriate for younger audiences. Avoid any form of violence, cruelty, complex romantic themes, profanity, or content that could be deemed unsuitable for families. Focus instead on uplifting adventures, educational elements, humor, and messages that encourage kindness, creativity, teamwork, and imagination.\n\nStructure your responses as a game-like narrative, speaking directly to the player in the second person. Begin each scene with a vivid, child-friendly description of the environment, characters, and events. Always provide the player with clear, exciting choices for their next actions and never make decisions on their behalf. Emphasize positive consequences and playful exploration while inspiring curiosity and problem-solving.\n\nEnsure your responses are concise and respect the token limit. If necessary, provide a complete and logical conclusion using fewer tokens without compromising the story's coherence or charm. Adjust the tone to remain lighthearted, whimsical, or adventurous, depending on the genre or setting. In cases where a more detailed response would exceed the token limit, focus on delivering a cohesive and delightful segment and let the player know the story will continue based on their next input.\n";

}

else

{

startMessage = "You are a creative and engaging story generator. Your role is to narrate immersive, branching stories in the second person, directly addressing the player. Structure your responses as a game-like narrative, beginning each scene with a vivid description of the environment, characters, and events. Always conclude your responses by offering the player clear choices for their next action, without making decisions for them. Your narration should reflect the consequences of their past actions and guide them through the world with intrigue and emotional depth.\n\nEnsure your responses are concise and avoid exceeding the token limit. If necessary, provide a complete and logical conclusion using fewer tokens while maintaining narrative quality. In cases where a detailed response is required but tokens are limited, prioritize delivering a cohesive and complete segment, and indicate that the story will continue based on the player's next input. Adjust the tone and style to match the genre (e.g., fantasy, sci-fi, or adventure) while immersing the player in the unfolding events.\n";

}

conversationHistory.Add(new Dictionary<string, string> {

{ "role", "system" },

{ "content", startMessage }

});

// Add the user's input for the first story

conversationHistory.Add(new Dictionary<string, string> {

{ "role", "user" },

{ "content", $"World Name: {worldName}. Character Name: {characterName}. Genre: {genre}. Preferences: {preferences}." }

});

// Save the conversation history to a file

saveLoadScript.SaveConversationHistoryToFile(conversationHistory, worldName);

yield return SendAPIRequest(apiUrl, apiKey);

}

// Subsequent requests to continue the story.

public IEnumerator ContinueStory(string userInput)

{

if (isRequestInProgress)

{

Debug.Log("Запит вже обробляється.");

yield break; // Виходимо з методу, якщо запит вже обробляється

}

// display message

Debug.Log("Continue story started");

if(userInput != "")

{

showMessage.AddUserMessage(userInput);

}

// Add the user's input to the conversation history

conversationHistory.Add(new Dictionary<string, string> {

{ "role", "user" },

{ "content", userInput }

});

if (tokenCount.CountTokensInConversation(conversationHistory) > maxTokens)

{

tokenCount.TrimConversationHistory(conversationHistory, maxTokens);

}

// Save the conversation history to a file

saveLoadScript.SaveConversationHistoryToFile(conversationHistory, worldNameBase);

yield return SendAPIRequest(apiUrl, apiKey);

}

// Load conversation history from a file

public void LoadConversationHistory(string filePath)

{

conversationHistory = saveLoadScript.LoadConversationHistoryFromFile(filePath);

worldNameBase = Path.GetFileNameWithoutExtension(filePath); // Extract world name from file path

Debug.Log("Conversation history loaded.");

}

public int GetIndexFromConversationHistory(string content)

{

for (int i = 0; i < conversationHistory.Count; i++)

{

if (conversationHistory[i]["content"] == content)

{

return i; // Повертаємо індекс

}

}

return -1; // Якщо не знайдено

}

public IEnumerator GeneratePromptForImage()

{

ShowPanelForLog("Loading prompt for image...", false);

var conversationForImage = new List<Dictionary<string, string>>(conversationHistory);

conversationForImage.Add(new Dictionary<string, string>

{

{ "role", "user" },

{

"content",

"Based on the current story in the conversation history, generate a detailed and creative prompt for the DALL·E model to create an image that reflects the current state of the narrative. The image should fit the tone and genre of the story (e.g., fantasy, sci-fi, or historical) and be visually compelling for use in an interactive story."

}

});

var jsonData = new

{

model = PlayerPrefs.GetString("textModel"),

messages = conversationForImage,

max\_tokens = 250,

temperature = 1

};

string jsonString = JsonConvert.SerializeObject(jsonData);

using (UnityWebRequest request = new UnityWebRequest(apiUrl, "POST"))

{

byte[] jsonToSend = Encoding.UTF8.GetBytes(jsonString);

request.uploadHandler = new UploadHandlerRaw(jsonToSend);

request.downloadHandler = new DownloadHandlerBuffer();

request.SetRequestHeader("Authorization", $"Bearer {PlayerPrefs.GetString("API")}");

request.SetRequestHeader("Content-Type", "application/json");

yield return request.SendWebRequest();

if (request.result == UnityWebRequest.Result.Success)

{

string responseText = request.downloadHandler.text;

Debug.Log("API Response: " + responseText);

try

{

JObject responseObject = JObject.Parse(responseText);

string message = responseObject["choices"]?[0]?["message"]?["content"]?.ToString();

if (!string.IsNullOrEmpty(message))

{

Debug.Log("Generated Prompt: " + message);

StartCoroutine(imageSendRequest.GenerateImage(message));

}

else

{

Debug.LogWarning("Response does not contain a valid prompt.");

ShowPanelForLog("Response does not contain a valid prompt.", true);

}

}

catch (Exception e)

{

Debug.LogError("Error parsing API response: " + e.Message);

ShowPanelForLog($"Error parsing the response: {e.Message}", true);

}

}

else

{

Debug.LogError($"Request failed with error: {request.error}. Response: {request.downloadHandler.text}");

ShowPanelForLog($"Error parsing the response: {request.error}", true);

}

}

}

public IEnumerator SendAPIRequest(string apiUrl, string apiKey)

{

if (isRequestInProgress)

{

Debug.Log("Запит вже обробляється.");

yield break;

}

isRequestInProgress = true; // Встановлюємо, що запит почав оброблятися

ShowPanelForLog("Loading prompt to server...", false);

var jsonData = new {

model = PlayerPrefs.GetString("textModel"),

messages = conversationHistory,

max\_tokens = PlayerPrefs.GetInt("maxTok"),

temperature = PlayerPrefs.GetFloat("temp")

};

string jsonString = JsonConvert.SerializeObject(jsonData);

Debug.Log("Request Payload: " + jsonString);

using (UnityWebRequest request = new UnityWebRequest(apiUrl, "POST"))

{

byte[] jsonToSend = Encoding.UTF8.GetBytes(jsonString);

request.uploadHandler = new UploadHandlerRaw(jsonToSend);

request.downloadHandler = new DownloadHandlerBuffer();

request.SetRequestHeader("Authorization", $"Bearer {PlayerPrefs.GetString("API")}");

request.SetRequestHeader("Content-Type", "application/json");

yield return request.SendWebRequest();

if (request.result == UnityWebRequest.Result.Success)

{

string responseText = request.downloadHandler.text;

Debug.Log("API Response: " + responseText); // Print full response to inspect

try

{

// Parse the response into JObject

JObject responseObject = JObject.Parse(responseText);

// Check if choices exists and is not empty

if (responseObject["choices"] != null && responseObject["choices"].HasValues)

{

string message = responseObject["choices"][0]["message"]["content"].ToString();

// Add the response to conversation history

conversationHistory.Add(new Dictionary<string, string>

{

{ "role", "assistant" },

{ "content", message }

});

// Display the response

showMessage.AddAIMessage(message);

createPanel.SetActive(false);

// save

saveLoadScript.SaveConversationHistoryToFile(conversationHistory, worldNameBase);

}

else

{

Debug.LogError("Choices array is empty or null.");

DeleteMessage("Choices array is empty or null.");

}

}

catch (System.Exception ex)

{

Debug.LogError("Error parsing the response: " + ex.Message);

DeleteMessage(ex.Message);

}

}

else

{

Debug.LogError($"Request Failed: {request.error}\nResponse: {request.downloadHandler.text}");

DeleteMessage(request.error);

}

isRequestInProgress = false; // Встановлюємо, що запит завершено

resultPanel.SetActive(false);

}

}

void DeleteMessage(string message)

{

ShowPanelForLog($"Error parsing the response: {message}", true);

var messageTemp = conversationHistory[conversationHistory.Count - 1];

string tmpMessage = messageTemp["content"];

inputField.text = tmpMessage;

Destroy(showMessage.messagesObjects[showMessage.messagesObjects.Count - 1]);

showMessage.messagesObjects.RemoveAt(showMessage.messagesObjects.Count - 1);

conversationHistory.RemoveAt(conversationHistory.Count - 1);

}

// Clears the conversation history (optional for starting fresh).

public void ClearMemory()

{

conversationHistory.Clear();

Debug.Log("Conversation history cleared.");

}

}

Додаток В CreateWorld.cs

uusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using TMPro;

public class CreateWorld : MonoBehaviour

{

public string nameWorld;

public string nameChara;

public string Genre;

public string Preferenses;

public TMP\_InputField world;

public TMP\_InputField chara;

public TMP\_InputField genre;

public TMP\_InputField preferenses;

private Button bttn;

private SendRequest send;

public Toggle toggleFamily;

void Start()

{

bttn = gameObject.GetComponent<Button>();

bttn.onClick.AddListener(generate);

send = GameObject.FindWithTag("Manager").GetComponent<SendRequest>();

}

public void generate()

{

if(toggleFamily.isOn)

PlayerPrefs.SetInt("familyMode", 1);

else

PlayerPrefs.SetInt("familyMode", 0);

nameWorld = world.text.ToString();

nameChara = chara.text.ToString();

Genre = genre.text.ToString();

Preferenses = preferenses.text.ToString();

Debug.Log("nameWorld:" + nameWorld + "\n" +"nameChara:" + nameChara + "\n" +"Genre:" + Genre + "\n" +"Preferenses:" + Preferenses);

StartCoroutine(send.CreateStartingStory(nameWorld, nameChara, Genre, Preferenses));

}

}

Додаток Г MessageScript.cs

using System.Collections;

using System.Collections.Generic;

using TMPro;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.EventSystems;

public class MessageScript : MonoBehaviour, IPointerClickHandler

{

public string messageText = "";

public int id;

public TMP\_Text messageBox;

public TMP\_InputField inputField;

public SendRequest sendRequest;

public SaveLoadScript saveLoadScript;

private float lastClickTime;

private const float doubleClickThreshold = 0.3f;

public bool isEditing = false;

void Start()

{

inputField = transform.GetChild(1).GetComponent<TMP\_InputField>();

sendRequest = GameObject.FindWithTag("Manager").GetComponent<SendRequest>();

saveLoadScript = GameObject.FindWithTag("Manager").GetComponent<SaveLoadScript>();

id = sendRequest.GetIndexFromConversationHistory(messageText);

inputField.gameObject.SetActive(false);

messageBox.text = messageText;

}

public void OnPointerClick(PointerEventData eventData)

{

if (Time.time - lastClickTime < doubleClickThreshold)

{

isEditing = true;

StartEditing();

Debug.Log("clicked");

}

lastClickTime = Time.time;

}

private void StartEditing()

{

inputField.text = messageText;

inputField.gameObject.SetActive(true);

messageBox.gameObject.SetActive(false);

inputField.ActivateInputField(); // Фокус на InputField

}

void Update()

{

if (isEditing)

{

// Перевірка на натискання Enter або Shift + Enter

if (Input.GetKeyDown(KeyCode.Return))

{

if (Input.GetKey(KeyCode.LeftShift) || Input.GetKey(KeyCode.RightShift))

{

Debug.Log("shift + enter");

AddLineBreak(); // Додаємо абзац

}

else

{

Debug.Log("shift");

SaveText(); // Зберігаємо текст

}

}

// Перевірка на натискання ESC для скасування

if (Input.GetKeyDown(KeyCode.Escape))

{

Debug.Log("Escape");

CancelEditing(); // Відміна редагування

}

}

}

private void AddLineBreak()

{

// Додаємо абзац в поточне місце курсора

int cursorPosition = inputField.caretPosition;

inputField.text = inputField.text.Insert(cursorPosition, "\n");

inputField.caretPosition = cursorPosition + 1; // Оновлюємо позицію курсора

}

public void SaveText()

{

messageBox.text = inputField.text;

messageText = inputField.text;

inputField.gameObject.SetActive(false);

messageBox.gameObject.SetActive(true);

sendRequest.conversationHistory[id]["content"] = messageText;

saveLoadScript.SaveConversationHistoryToFile(sendRequest.conversationHistory, sendRequest.worldNameBase);

isEditing = false;

}

private void CancelEditing()

{

inputField.text = messageText; // Повертаємо старий текст

inputField.gameObject.SetActive(false);

messageBox.gameObject.SetActive(true);

isEditing = false;

}

}

Додаток Д ShowMessage.cs

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

using TMPro;

public class ShowMessage : MonoBehaviour

{

public Transform content; // Reference to the Content object

public GameObject userMessagePrefab; // Prefab for user messages

public GameObject aiMessagePrefab; // Prefab for AI messages

public SendRequest sendRequest;

public SaveLoadScript saveLoadScript;

public List<GameObject> messagesObjects = new List<GameObject>();

public void Start()

{

sendRequest = GameObject.FindWithTag("Manager").GetComponent<SendRequest>();

if (sendRequest == null)

Debug.LogError("sendRequest is null");

if (content == null)

Debug.LogError("content is null");

if (content.childCount == 0)

Debug.LogWarning("Content has no children");

}

public void DeleteAllMessages()

{

foreach (var varGameObject in messagesObjects)

{

Destroy(varGameObject);

}

messagesObjects.Clear();

saveLoadScript.arr.Clear();

}

// Function to add a user message

public void AddUserMessage(string message)

{

GameObject newMessage = Instantiate(userMessagePrefab, content);

newMessage.GetComponent<MessageScript>().messageText = message;

messagesObjects.Add(newMessage);

saveLoadScript.arr.Add(newMessage);

}

// Function to add an AI response

public void AddAIMessage(string message)

{

GameObject newMessage = Instantiate(aiMessagePrefab, content);

newMessage.GetComponent<MessageScript>().messageText = message;

messagesObjects.Add(newMessage);

saveLoadScript.arr.Add(newMessage);

}

public void RegenerateMessage()

{

GameObject var = messagesObjects.Last();

messagesObjects.RemoveAt(messagesObjects.Count - 1);

sendRequest.conversationHistory.RemoveAt(sendRequest.conversationHistory.Count - 1);

Destroy(var);

StartCoroutine(sendRequest.SendAPIRequest(sendRequest.apiUrl, sendRequest.apiKey));

}

}

Додаток В ImageSendRequest.cs

using System;  
using System.Collections;  
using System.IO;  
using UnityEngine;  
using UnityEngine.Networking;  
using UnityEngine.UI;  
using Newtonsoft.Json;  
using Newtonsoft.Json.Linq;  
  
public class ImageSendRequest : MonoBehaviour  
{  
 public SendRequest sendRequest;  
 public Image image;  
   
 private string API\_URL = "https://api.openai.com/v1/images/generations";  
  
 private void Start()  
 {  
 sendRequest = GetComponent<SendRequest>();  
 }  
   
 public IEnumerator GenerateImage(string prompt)  
 {  
 if (sendRequest.conversationHistory.Count == 0)  
 yield break;  
   
 sendRequest.ShowPanelForLog("Generating Image...", false);  
 var jsonData = JsonConvert.SerializeObject(new { model = PlayerPrefs.GetString("imageModel"), prompt = prompt, n = 1, size = "1024x1024" });  
 var request = new UnityWebRequest(API\_URL, "POST");  
 byte[] bodyRaw = System.Text.Encoding.UTF8.GetBytes(jsonData);  
 request.uploadHandler = new UploadHandlerRaw(bodyRaw);  
 request.downloadHandler = new DownloadHandlerBuffer();  
 request.SetRequestHeader("Content-Type", "application/json");  
 request.SetRequestHeader("Authorization", $"Bearer {sendRequest.apiKey}");  
  
 yield return request.SendWebRequest();  
  
 if (request.result == UnityWebRequest.Result.Success)  
 {  
 var response = JsonUtility.FromJson<DALLEImageResponse>(request.downloadHandler.text);  
 StartCoroutine(LoadImage(response.data[0].url));  
 }  
 else  
 {  
 Debug.LogError("Error: " + request.error);  
 sendRequest.ShowPanelForLog("Error: " + request.error, true);  
 }  
 sendRequest.resultPanel.SetActive(false);  
 }  
 public string GenerateUniqueFileName()  
 {  
 string timestamp = DateTime.Now.ToString("yyyyMMdd\_HHmmss");  
 string fileExtension = ".png";  
 string uniqueFileName = $"{sendRequest.worldNameBase}\_{timestamp}{fileExtension}";  
 return uniqueFileName;  
 }  
 private IEnumerator LoadImage(string url)  
 {  
 UnityWebRequest request = UnityWebRequestTexture.GetTexture(url);  
 yield return request.SendWebRequest();  
  
 if (request.result == UnityWebRequest.Result.Success)  
 {  
 Texture2D texture = ((DownloadHandlerTexture)request.downloadHandler).texture;  
 image.sprite = Sprite.Create(texture, new Rect(0, 0, texture.width, texture.height), new Vector2(0.5f, 0.5f)); // Змінюємо компонент Image  
 byte[] imageBytes = texture.EncodeToPNG();  
 string filePath = Path.Combine(Application.persistentDataPath, GenerateUniqueFileName());  
  
 File.WriteAllBytes(filePath, imageBytes);  
 Debug.Log("Image saved at: " + filePath);  
 }  
 else  
 {  
 Debug.LogError("Error loading image: " + request.error);  
 }  
 }  
 public IEnumerator LoadImageToUI(string filePath)  
 {  
 string fileUrl = "file://" + filePath; // Додаємо протокол file:// для локальних файлів  
 using (UnityWebRequest request = UnityWebRequestTexture.GetTexture(fileUrl))  
 {  
 yield return request.SendWebRequest();  
  
 if (request.result == UnityWebRequest.Result.Success)  
 {  
 Texture2D texture = DownloadHandlerTexture.GetContent(request);  
 Sprite sprite = Sprite.Create(texture, new Rect(0, 0, texture.width, texture.height), new Vector2(0.5f, 0.5f));  
 image.sprite = sprite; // Встановлюємо у компонент Image  
 }  
 else  
 {  
 Debug.LogError($"Помилка завантаження зображення: {request.error}");  
 }  
 }  
 }  
   
 [System.Serializable]  
 public class DALLEImageResponse  
 {  
 public DALLEImageData[] data;  
 }  
  
 [System.Serializable]  
 public class DALLEImageData  
 {  
 public string url;  
 }  
}

Додаток Ж TokenCountTests.cs

using System.Collections;

using System.Collections.Generic;

using NUnit.Framework;

using UnityEngine;

using UnityEngine.TestTools;

public class TokenCountTests

{

private TokenCount tokenCounter;

[SetUp]

public void Setup()

{

// Ініціалізуємо тестовий об'єкт перед кожним тестом

var gameObject = new GameObject();

tokenCounter = gameObject.AddComponent<TokenCount>();

}

[Test]

public void EstimateTokens\_ShouldCalculateCorrectly()

{

// Arrange

string text = "Hello world!";

// Act

int tokenCount = tokenCounter.EstimateTokens(text);

// Assert

Assert.AreEqual(2, tokenCount, "Токени не розраховані коректно.");

}

[Test]

public void CountTokensInConversation\_ShouldCalculateCorrectly()

{

// Arrange

var conversationHistory = new List<Dictionary<string, string>>()

{

new Dictionary<string, string> { { "role", "system" }, { "content", "This is a test message." } },

new Dictionary<string, string> { { "role", "user" }, { "content", "Hello world!" } }

};

// Act

int totalTokens = tokenCounter.CountTokensInConversation(conversationHistory);

// Assert

int lowerBound = 15; // Нижня межа діапазону

int upperBound = 17; // Верхня межа діапазону

Assert.IsTrue(totalTokens >= lowerBound && totalTokens <= upperBound,

$"Токени у розмові повинні бути в діапазоні {lowerBound}-{upperBound}, але отримано {totalTokens}.");

}

[Test]

public void TrimConversationHistory\_ShouldTrimExcessTokens()

{

// Arrange

var conversationHistory = new List<Dictionary<string, string>>()

{

new Dictionary<string, string> { { "role", "system" }, { "content", "This is a test message." } },

new Dictionary<string, string> { { "role", "user" }, { "content", "Hello world!" } },

new Dictionary<string, string> { { "role", "assistant" }, { "content", "This is assistant's reply." } },

new Dictionary<string, string> { { "role", "user" }, { "content", "Additional message." } }

};

int maxTokens = 34;

// Act

tokenCounter.TrimConversationHistory(conversationHistory, maxTokens);

// Assert

Assert.AreEqual(3, conversationHistory.Count, "Розмір історії не був обрізаний коректно.");

}

[Test]

public void TrimConversationHistory\_ShouldNotTrimBelowTwoEntries()

{

// Arrange

var conversationHistory = new List<Dictionary<string, string>>()

{

new Dictionary<string, string> { { "role", "system" }, { "content", "Message 1" } },

new Dictionary<string, string> { { "role", "user" }, { "content", "Message 2" } }

};

int maxTokens = 1; // Задати неможливо мале обмеження

// Act

tokenCounter.TrimConversationHistory(conversationHistory, maxTokens);

// Assert

Assert.AreEqual(2, conversationHistory.Count, "Історія не має бути обрізана нижче 2 записів.");

}

}

Додаток З SaveLoadScriptTests.cs

using NUnit.Framework;

using System.Collections.Generic;

using System.IO;

using UnityEngine;

using UnityEngine.UI;

using Moq;

using Newtonsoft.Json; // Для моків

[TestFixture]

public class SaveLoadScriptTests

{

private SaveLoadScript saveLoadScript;

private Mock<ShowMessage> mockShowMessage;

private Mock<SendRequest> mockSendRequest;

private Mock<ImageSendRequest> mockImageSendRequest;

private string testFilePath;

[SetUp]

public void SetUp()

{

// Моки для залежностей

mockShowMessage = new Mock<ShowMessage>();

mockSendRequest = new Mock<SendRequest>();

mockImageSendRequest = new Mock<ImageSendRequest>();

// Створення тестового об'єкта

GameObject testObject = new GameObject();

saveLoadScript = testObject.AddComponent<SaveLoadScript>();

saveLoadScript.showMessage = mockShowMessage.Object;

saveLoadScript.sendRequest = mockSendRequest.Object;

saveLoadScript.imageSendRequest = mockImageSendRequest.Object;

// Шлях до тестового файлу

testFilePath = Path.Combine(Application.persistentDataPath, "TestWorld.json");

}

[TearDown]

public void TearDown()

{

// Видалення тестового файлу після виконання тестів

if (File.Exists(testFilePath))

{

File.Delete(testFilePath);

}

}

[Test]

public void SaveConversationHistoryToFile\_ShouldSaveCorrectly()

{

// Arrange

var testConversationHistory = new List<Dictionary<string, string>>

{

new Dictionary<string, string> { { "role", "user" }, { "content", "Hello World!" } },

new Dictionary<string, string> { { "role", "assistant" }, { "content", "Hi there!" } }

};

// Act

saveLoadScript.SaveConversationHistoryToFile(testConversationHistory, "TestWorld");

// Assert

Assert.IsTrue(File.Exists(testFilePath), "Файл не був створений.");

string fileContent = File.ReadAllText(testFilePath);

var deserializedData = Newtonsoft.Json.JsonConvert.DeserializeObject<List<Dictionary<string, string>>>(fileContent);

Assert.AreEqual(testConversationHistory.Count, deserializedData.Count, "Кількість повідомлень не співпадає.");

Assert.AreEqual(testConversationHistory[0]["content"], deserializedData[0]["content"], "Зміст першого повідомлення не співпадає.");

}

[Test]

public void LoadConversationHistoryFromFile\_ShouldLoadCorrectly()

{

// Arrange

var testConversationHistory = new List<Dictionary<string, string>>

{

new Dictionary<string, string> { { "role", "user" }, { "content", "Hello World!" } },

new Dictionary<string, string> { { "role", "assistant" }, { "content", "Hi there!" } }

};

File.WriteAllText(testFilePath, Newtonsoft.Json.JsonConvert.SerializeObject(testConversationHistory));

// Act

var loadedConversationHistory = saveLoadScript.LoadConversationHistoryFromFile(testFilePath);

// Assert

Assert.AreEqual(testConversationHistory.Count, loadedConversationHistory.Count, "Кількість завантажених повідомлень не співпадає.");

Assert.AreEqual(testConversationHistory[0]["content"], loadedConversationHistory[0]["content"], "Зміст першого повідомлення не співпадає.");

}

[Test]

public void LoadConversationHistoryFromFile\_ShouldReturnEmpty\_WhenFileDoesNotExist()

{

// Arrange

string nonExistentFilePath = Path.Combine(Application.persistentDataPath, "NonExistent.json");

// Act

var result = saveLoadScript.LoadConversationHistoryFromFile(nonExistentFilePath);

// Assert

Assert.IsNotNull(result, "Результат не повинен бути null.");

Assert.IsEmpty(result, "Історія розмови повинна бути порожньою.");

}

[Test]

public void CleanBttn\_ShouldCleanAllButtons()

{

// Arrange

GameObject button1 = new GameObject();

button1.tag = "loadbttn";

GameObject button2 = new GameObject();

button2.tag = "loadbttn";

saveLoadScript.arr = new List<GameObject>() { button1, button2 };

// Act

saveLoadScript.CleanBttn();

// Assert

Assert.AreEqual(0, saveLoadScript.arr.Count, "Кнопки не були очищені.");

}

}

Додаток И ShowMessageTests.cs

uusing NUnit.Framework;

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using UnityEngine.TestTools;

using Moq;

[TestFixture]

public class ShowMessageTests

{

private ShowMessage showMessage;

private GameObject gameObject;

private GameObject userMessagePrefab;

private GameObject aiMessagePrefab;

private Mock<SendRequest> mockSendRequest;

private SaveLoadScript saveLoadScript;

[SetUp]

public void Setup()

{

gameObject = new GameObject();

showMessage = gameObject.AddComponent<ShowMessage>();

// Create message prefabs and add MessageScript components

userMessagePrefab = new GameObject();

aiMessagePrefab = new GameObject();

userMessagePrefab.AddComponent<MessageScript>();

aiMessagePrefab.AddComponent<MessageScript>();

// Assign to ShowMessage fields

showMessage.userMessagePrefab = userMessagePrefab;

showMessage.aiMessagePrefab = aiMessagePrefab;

// Set up SaveLoadScript (real component, not mocked here)

saveLoadScript = new GameObject().AddComponent<SaveLoadScript>();

showMessage.saveLoadScript = saveLoadScript;

// Create a content GameObject and assign it to ShowMessage

GameObject contentObject = new GameObject();

showMessage.content = contentObject.transform;

// Optionally, add a child to content to avoid the "Transform child out of bounds" error

var childObject = new GameObject();

childObject.transform.SetParent(showMessage.content);

}

[Test]

public void DeleteAllMessages\_ShouldClearAllMessages()

{

// Arrange

var userMessage = new GameObject();

var aiMessage = new GameObject();

showMessage.messagesObjects.Add(userMessage);

showMessage.messagesObjects.Add(aiMessage);

// Act

showMessage.DeleteAllMessages();

// Assert

Assert.AreEqual(0, showMessage.messagesObjects.Count);

Assert.AreEqual(0, saveLoadScript.arr.Count);

}

[Test]

public void AddUserMessage\_ShouldInstantiateUserMessage()

{

// Arrange

string testMessage = "User test message";

// Act

showMessage.AddUserMessage(testMessage);

// Assert

Assert.AreEqual(1, showMessage.messagesObjects.Count);

Assert.AreEqual(testMessage, showMessage.messagesObjects[0].GetComponent<MessageScript>().messageText);

}

[Test]

public void AddAIMessage\_ShouldInstantiateAIMessage()

{

// Arrange

string testMessage = "AI test message";

// Act

showMessage.AddAIMessage(testMessage);

// Assert

Assert.AreEqual(1, showMessage.messagesObjects.Count);

Assert.AreEqual(testMessage, showMessage.messagesObjects[0].GetComponent<MessageScript>().messageText);

}

}