

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

«17» жовтня 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 **«17» жовтня 2024 р., №505/с**

Термін здачі закінченої роботи: **«10» грудня 2024 р.**

Вихідні дані роботи: об'єктом дослідження є процес генерації контенту в ігрових додатках за допомогою ШІ. Предметом дослідження є використання сучасних текстових моделей ШІ, зокрема GPT-4 версій, для генерації ігрової історії, моделей DALL-E та генерації зображень в контексті історії гри.

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

Розділ	Консультант	Дата	
		Завдання видав	Завдання прийняв
1	Сугоняк І.І.	10.09.24	16.10.24
2	Сугоняк І.І.	16.10.24	21.11.24
3	Сугоняк І.І.	21.11.24	20.12.24

**Календарний план**

№ з/п	Назва етапів випускної кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі. Опрацювання літературних джерел. Пошук, огляд та аналіз аналогічних розробок.	2.09.24 – 1.10.24	Виконано
2	Проектування структури системи	2.10.24 – 16.10.24	Виконано
3	Написання програмного коду	17.10.24 – 10.11.24	Виконано
4	Тестування системи	11.11.24 – 14.11.24	Виконано
5	Оформлення пояснювальної записки	15.11.24 – 1.12.24	Виконано
6	Захист	30.12.24	Виконано

Здобувач вищої освіти **Тимур ГАЛАС**  
Керівник **Руслан ПЕТРОСЯН**

## **РЕФЕРАТ**

Основний зміст роботи викладено на **75** сторінках тексту, 82 ілюстрацій, 9 додатків, налічує 19 найменувань літературних джерел.

Метою кваліфікаційної роботи є розробка ігрового додатку для створення адаптивних ігрових сценаріїв та реалізація системи, яка використовує ШІ для створення текстового контенту та історії для інтерактивної гри.

В цьому додатку гравець, використовуючи моделі від OpenAI, буде отримувати унікальні текстову гру, в якій зможе отримати унікальний ігровий досвід та прямо впливати на хід наративу.

У роботі запропоновано реалізацію ігрового додатку на рушії Unity з використанням OpenAI API для створення світу та історії, який, в залежності від дій користувача, буде адаптувати історію та рухати її в тому напрямку, який обрав гравець, даючи унікальний досвід для кожного користувача.

Ключові слова: AI / ШІ, UI, JSON, C#, API, GUI, Prefab, Фідбек, smm, промпт.

## **ABSTRACT**

The main content of the work is laid out on 75 pages of text, 82 illustrations, 9 appendices, and includes 19 names of literary sources.

The goal of the qualification work is the development of a game application for creating adaptive game scenarios and the implementation of a system that uses AI to create text content and a story for an interactive game.

In this application, the player will receive a unique text game using models from OpenAI, in which they will be able to get a unique gaming experience and directly influence the course of the narrative.

The work proposes the implementation of a game application on the Unity engine using the OpenAI API to create a world and story, which, depending on the user's actions, will adapt the story and move it in the direction chosen by the player, giving a unique experience to each user.

Keywords: AI, UI, JSON, C#, API, GUI, Prefab, Feedback, smm, prompt.

## ЗМІСТ

РЕФЕРАТ .....	3
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ .....	7
ВСТУП .....	8
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ТА ПЕРЕГЛЯД ТЕХНОЛОГІЙ ДЛЯ СТВОРЕННЯ ГРИ .....	10
1.1 Постановка мети розробки додатку .....	10
1.2 Аналіз існуючих технологій реалізації ігрових додатків .....	11
1.3 Принцип роботи гри .....	17
1.4 Постановка задачі та технічне завдання .....	20
1.5 Обґрунтування вибору інструментальних засобів розробки .....	21
Висновки до першого розділу .....	24
РОЗДІЛ 2. ПРОЄКТУВАННЯ, РОЗРОБКА ТА ПОБУДОВА ДОДАТКУ .....	25
2.1 Проєктування архітектури гри .....	25
2.2 Опис процесів у проєкті .....	27
2.3 Робробка та написання коду гри .....	33
Висновки до другого розділу .....	56
РОЗДІЛ 3. РОЗГОРТАННЯ СИСТЕМИ, НАЛАШТУВАННЯ GUI ТА ТЕСТУВАННЯ .....	58
3.1 Розгортання системи та налаштування інтерфейсу користувача в ігровому рушії та інспекторі Unity. ....	58
3.2 Інтерфейс та порядок роботи .....	65
3.3 Тестування ігрового додатку .....	73
Висновки до третього розділу .....	82
ВИСНОВКИ .....	83
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....	85
ДОДАТКИ .....	87
Додаток А .....	4
Додаток Б .....	8

Додаток В .....	17
Додаток Г .....	19
Додаток Д .....	22
Додаток В .....	24
Додаток Ж .....	27
Додаток З .....	30
Додаток И .....	34

## **ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ**

AI / ШІ – Artificial Intelligence / Штучний інтелект

UI – User Interface (Користувацький інтерфейс)

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

C# – C Sharp (Мова програмування C#)

Промпт -запитання чи завдання, яке користувач ставить під час спілкування з мовною моделлю штучного розуму.

API – Application Programming Interface (Інтерфейс програмування додатків)

GUI – Graphical User Interface (Графічний інтерфейс користувача)

Prefab - макет, заготовка ігрового об'єкту

Фідбек - відгук

SMM - маркетинговий відділ.

## ВСТУП

**Актуальність теми.** Стрімкий розвиток ШІ дозволяє підняти інтерактивність та адаптивність до дій гравця на новий рівень та дозволить створювати унікальний досвід для кожного користувача. Це дозволяє створювати сюжети, що відрізняються в залежності від вибору кожного гравця та має потенціал змінити традиційні підходи до розробки ігрових додатків. Інтеграція ШІ в індустрію відеоігор є важливим кроком.

**Мета і завдання дослідження.** Метою дослідження є відповідь на запитання, чи можна використовувати ШІ для генерації історії в ігровому додатку на рушії Unity, який, у вигляді текстової гри/історії буде генерувати історію в залежності від дій гравця та рухати її саме в тому напрямку, в якому дії гравця її привели.

Поставлена мета обумовлює необхідність вирішення таких завдань дослідження:

- розробка ігрового додатку для генерації унікальних історій і подій у залежності від вибору гравця та його дій;
- обрати модель ШІ, яка здатна створювати історії та здатна адаптуватися до дій гравця;
- протестувати готову гру. Перевірити, чи вдається їй захопити інтерес гравця, чи коректно вона реагує на дії користувача та чи можна вважати цей додаток успішним;

**Об'єкт дослідження.** Процес генерації контенту в ігрових додатках за допомогою ШІ.

**Предмет дослідження.** Використання сучасних текстових моделей ШІ, зокрема GPT-4 версій, для генерації ігрової історії та моделей DALL-E, для генерації зображень в контексті історії гри.

**Методи дослідження.** У цій кваліфікаційній роботі використовувалися аналітичні методи, щоб вивчити уже існуючі рішення у ШІ, задля інтеграції їх в ігровий додаток на рушії Unity. Використовувалися методи системного аналізу для проектування та розробки структури та архітектури гри, і також

експериментальні методи для проведення тестування функціональності, зацікавленості гравця та графічного інтерфейсу.

**Наукова новизна.** Полягає у створенні та реалізації підходу до інтеграції ІІІ в ігровий процес для генерації світу та історії у ігровому додатку. Запропонована реалізація враховує дії гравця та продовжує історію, опираючись на його дії та створює унікальний досвідж для кожного користувача за допомогою тексту історії та зображень, які передають теперішній стан історії.

Практичне значення полягає в тому, що результати роботи можуть бути використані:

- для створення ігрового додатку у вигляді текстової гри для генерації історії, яка адаптується до гравця;
- для використання додатку як ігрового навчального інструменту для дітей, щоб навчати їх у вигляді казкових сімейних історій;
- у використанні додатку як асистента або помічника, для розробки власних історій, допомога при створенні контенту для рольових ігор;
- як основа для подальших досліджень у галузі інтеграції нейронних мереж у творчі та розважальні системи;

**Публікації.** За темою кваліфікаційної роботи було опубліковано одну публікацію:

1. Галас Т. Т., Петросян Р.В. Генерація ігрового контенту за допомогою штучного інтелекту: адаптивні сценарії та світи.Тези доповідей VII Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення», м. Житомир, 02–03 грудня 2024 р. Житомир: Житомирська політехніка, 2024.



# РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ТА ПЕРЕГЛЯД ТЕХНОЛОГІЙ ДЛЯ СТВОРЕННЯ ГРИ

## 1.1 Постановка мети розробки додатку

У результаті виконання даної кваліфікаційної роботи буде створено ігровий додаток, який буде виконувати такі функції:

- гра буде автоматично створювати світ та історію, після отримання початкової інформації від гравця, та буде продовжувати її, опираючись на жанр, світ, персонажа гравця та його дії;

- даний додаток дасть можливість гравцям впливати на сюжет через введення тексту чи дій свого персонажа. Всі дії гравця будуть обраховуватися ШІ та історія може кардинально змінити свій виток, якщо це спричинить гравець;

- структура додатку була обрана модульна. Це дозволить легше, в майбутньому, додавати новий функціонал та полегшить процес тестування нових модулів та читабельність коду, що полегшить повторну адаптацію в разі довгої перерви;

- інтерфейс гри буде спроектований з акцентом на зручність та мінімалістичність, щоб за потреби в майбутньому, при доданні нового функціоналу, більш дружнелюбноше додати нові елементи інтерфейсу та не перезавантажити інтерфейс;

- використання OpenAI API та їх моделей гарантує високу якість тексту та зображень, стабільність та підтримку і легкість адаптацій нових моделей в додаток. Гра буде акцентувати увагу на адаптивну та точну історію;

- в залежності від початкових параметрів користувача, модель ШІ(типу GPT-4) створить світ та персонажа гравця і почне, у вигляді диктору, розповідати історію;

- додаток буде протестовано як ручними, так і автоматичними тестами, щоб підтвердити роботу та здатність створювати продуктивні та адаптивні світи. В кінці тестування очікується позитивний фідбек від користувачів, що гра приносить задоволення, зацікавлює їх та працює коректно;

- додаток такого плану надасть іншим геймдизайнерам та сценаристам

зручний інструмент, який полегшить та прискорить роботу над ігровими сценаріями та допоможе їм зосередитись на творчих аспектах історії, без необхідності описувати дрібні деталі або неважливі частини історій;

- фінальна версія продукту сприятиме підвищенню інтерактивності, адаптивності та персоналізації ігрових історій. Спрощення процесу створення для розробників сприятиме пошвидшенню розвитку ігрової індустрії та ШІ в цілому;

Зазначені результати очікуються як важливий внесок у вдосконалення інноваційних підходів до створення інтерактивного контенту.

## 1.2 Аналіз існуючих технологій реалізації ігрових додатків

Перед початком проектування та розробки ігрового додатку, було проведено пошук та аналіз подібного ПЗ, який вже наявний на ринку та використовує схожі технології або принципи. Серед усіх переглянутих аналогів, було обрано 6 додатків, які можуть конкурувати: AI Dungeon, Latitude Voyager, AI Storytelling в Grammarly, Jasper, та NovelAI.

### 1. AI Dungeon

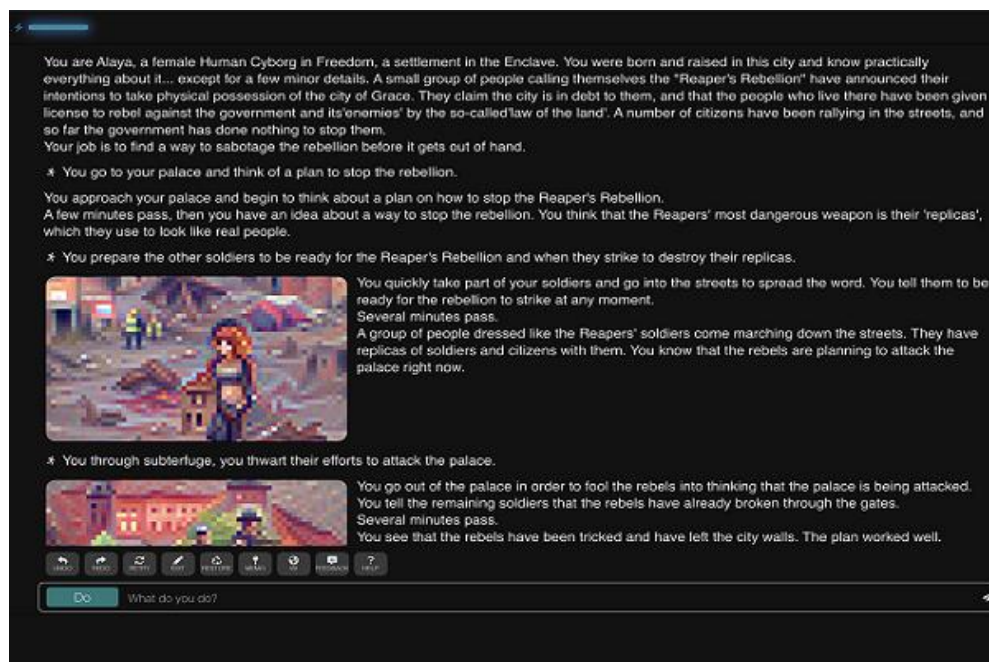


Рис 1.1. AI Dungeon

AI Dungeon - це один із найпопулярніших ігрових веб/мобільних застосунків, який використовує GPT-3 від OpenAI як основну модель для генерації історії та текстового контенту[3]. Одною з головних можливостей є створення відкритого світу, в якому гравці зможуть робити будь які дії та спостерігати за результатами після обробки системою їх запитів.

Багатство жанрів - AI Dungeons може створювати історії різноманітних жанрів: від фентезі до звичайного сучасного світу. Завдяки цьому, гравці можуть створювати безліч різних світів та історій.

Непередбачуваність сюжету - додаток дозволяє гравцям впливати на сюжет та історію за допомогою запитів. Модель ШІ генерує текст як лінійних, так і відкритих, історій в реальному часі.

Обмеження AI - AI Dungeon вважається гарним засобом для створення історій, але у нього є обмеження в пам'яті, через що довгі та складні історії та світи через деякий час починають втрачати логічність подій та адекватну послідовність, або події просто починають повторюватися, але це норма для будь якої моделі ШІ на даний момент через обмеження в кількості токенів (пам'ять мережі, максимальна кількість слів).

Система монетизації - Монетизація сервісу AI Dungeon працює за таким принципом - всі користувачі отримують доступ до базового функціоналу сайту, але більш детальні налаштування чи режими потребують від користувача платної підписки. До цього функціоналу належать більший об'єм контексту(пам'ять), кількість генерацій зображень на місяць, різні моделі і т.д.

Висновок: AI Dungeon це чудовий ігровий додаток і веб браузері або мобільному додатку для користувача, якого не цікавлять довгі та складні історії, оскільки обмеження на контекст дуже велике і задля глибокої та логічної історії треба буде заплатити дорогу ціну підписки[4].

## 2.Latitude Voyager

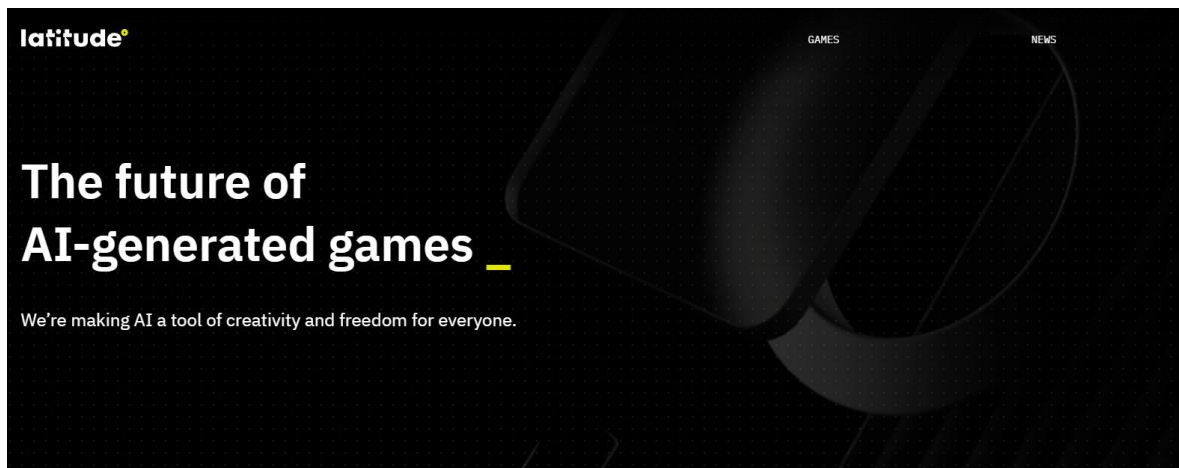


Рис 1.2. Latitude Voyager

Latitude Voyager - це веб-додаток від розробників AI Dungeons, який, як і AI Dungeons, спеціалізується на створенні інтерактивних історій за допомогою ШІ, але подає історію у вигляді, більше схожою на гру, аніж історія[5].

Глибока інтерактивність - Latitude Voyager має здатність створювати складні та мультишарові сюжети. Це дає користувачам можливість генерувати глибокі та детальні нелінійні історії та світи.

Інтеграція з ігровими механіками - веб додаток має функціонал для подачі історії у вигляді гри. Це дозволяє гравцям відчувати повноцінно бойову систему, прокачку персонажа, детальну взаємодію зі світом та інші функції, доступні лише у іграх.

Модульність - Latitude Voyager надає детальну та глибоку кастомізацію при створенні сценаріїв. Гравці можуть адаптувати її як і для написання простої текстової історії, так і для великої глибокої гри у відкритому світі.

Вартість та доступність - платформа знаходить на стадії активної розробки та доступна лише користувачам, які мають найвищий рівень підписки в AI Dungeon та були обрані для тестування.

Висновок: Latitude Voyager це потенційно революційна платформа для історій або інтерактивних текстових ігор. На жаль, на даний момент є закритою для більшості користувачів та знаходиться на етапі закритого тестування.

### 3.AI Storytelling у Grammarly

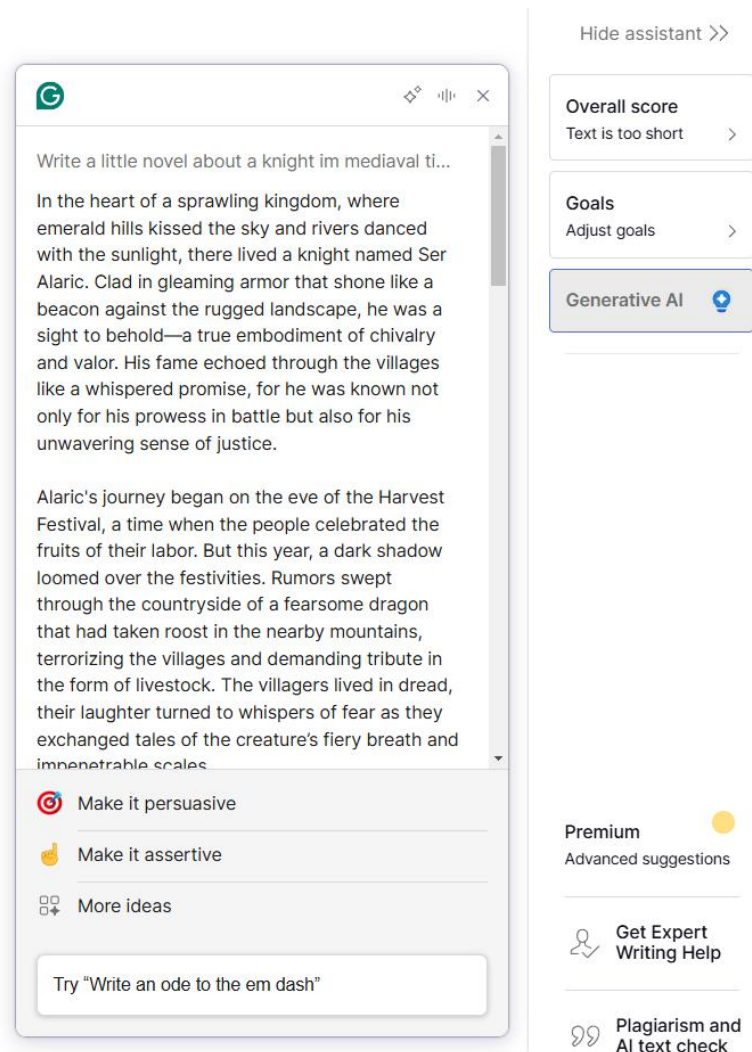


Рис 1.3. AI Storytelling

Grammarly це інструмент, який більшість користувачів знають як інструмент перевірки правопису, граматики та стилю, але в нього є також власна ШІ, AI Storytelling, яка спеціалізується на створенні та покращенні художніх історій або наративів[6].

Якість письма - Основна мета AI Storytelling направлена на дотримання стилістичних та граматичних правил, написання логічно структурованих історій та коретних історій.

Адаптивні рекомендації - Grammarly пропонує користувачу варіанти покращення історій та тексту. Зазвичай, це тон, структура або стиль історії, аби коректніше передати настрій.

Обмеження в контексті інтерактивних історій - незважаючи на те, що AI Storytelling пропонує чудовий функціонал, він не призначений для створення

історій з нуля. Він чудово підходить під роль рецензента, який відредагує вже написану історію, додасть художні засоби, більш детальні епітети і виправить ТОН.

Висновок: AI Storytelling від Grammarly це корисний та цікавий засіб для письменників та сценаристів, але ніяк не інструмент для генерації глибоких та детальних історій та світів.

#### 4. Jasper

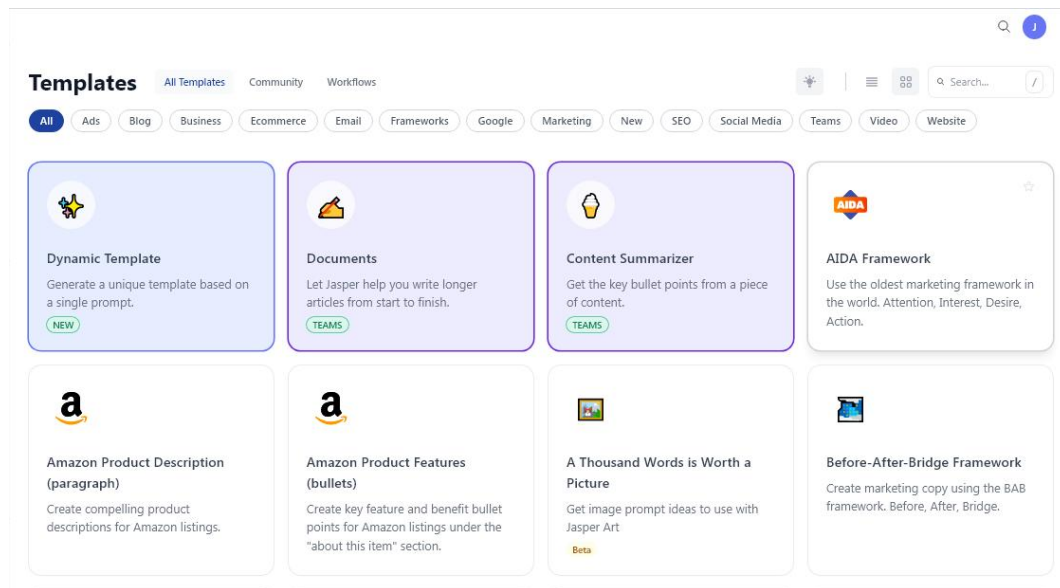


Рис 1.4. Jasper

Jasper - це відомий інструментарій для створення текстового контенту, який інтегрує ШІ для свого функціоналу з автоматичного написання текстів. Зазвичай, використовується для маркетинговому полі та smm.

Широке застосування - хоча Jasper позиціонує себе як асистента для маркетингу та бізнесу, він може викоритсовуватися для написання художньої літератури та сценаріїв.

Креативні моделі генерації - платформа Jasper дає можливість користувачу гнучко та детально налаштувати генерацію. Серед параметрів є такі, як стиль формат, тональність тексту.

Обмеження інтерактивності - У порівнянні з AI Dungeon чи Latitude Voyager, у Jasper відсутній будь який намік на інтерактивність чи адаптивність. Він орієнтований на створення статичного тексту без його розвитку.

Висновок: Jasper - це зручний та доступний інструментарій для створення текстового контенту у сфері бізнесу та маркетингу, або невеличких творчих текстів. Він не призначений для інтерактивних історій, сюжетів чи розповідей, що робить його використання в контексті інтерактивної гри абсолютно непотрібним та зайвим[7].

## 5. NovelAI

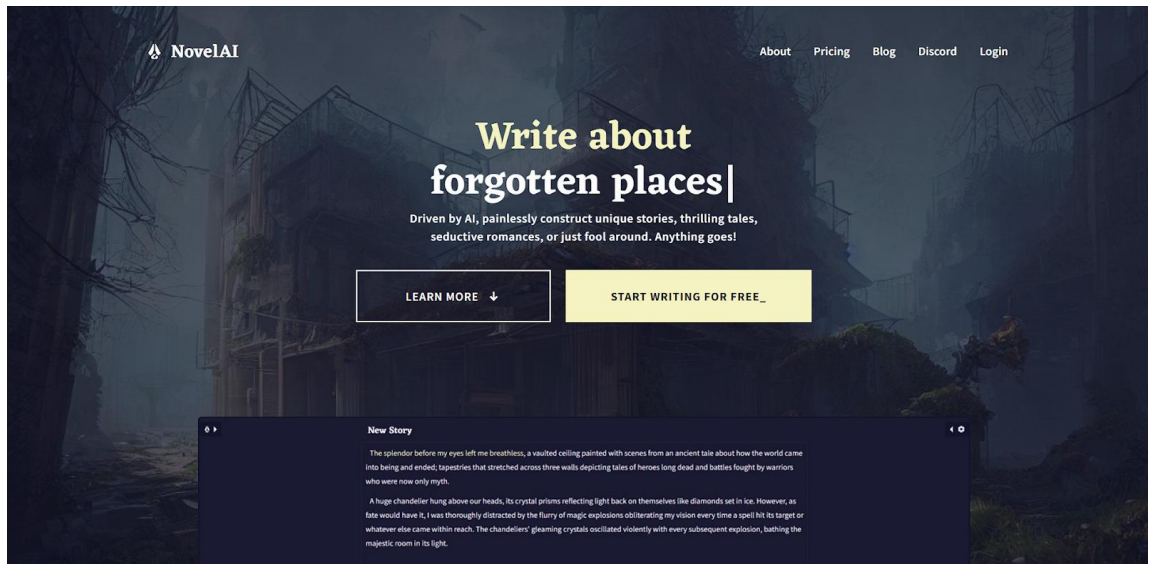


Рис 1.5. NovelAI

NovelAI - це веб-платформа для створення художньої текстів, детальних історій. Генерує текст з високою адаптацією до стилю, жанру, які запропонував користувач, та детально і “по живому” відіграє поведінку персонажів в історії.

Персоналізація наративу - NovelAI дає своїм гравцям/користувачам детальні налаштування для параметрів стилю, тон чи розвитку та взаємодії з персонажами між собою, через що гравець отримує детальну, продуману та унікальну історію.

Генерація складних сюжетів - NovelAI використовує модель ШІ на базі GPT-3. Це дає йому змогу генерувати складну та детальну історію, не забуваючи при цьому про логіку та послідовність історії.

Засоби редагування та налаштування - користувач надається функціонал для редагування згенерованої текстової частини від ШІ, що дає можливість внести зміни або детально контролювати історію та виправляти нелогічні моменти.



Висновок: NovelAI - це незамінний інструмент для письменників чи творців для допомоги при написанні художньої історії. На жаль, його лінійність та відсутність інтерактивності відштовхує більшість користувачів, які шукають саме інтерактивну текстову гру.

Дослідження ринку та конкурентів/аналогів показало, що рикон з інтерактивними текстовими іграми ще занадто молодий і тільки розвивається, що дає простір для початківців. Було визначено вимоги на базі аналізу конкурентів, такі як стабільність, висока інтерактивність, логіка/послідовність сюжету.

### **1.3 Принцип роботи гри**

Даний додаток це текстова гра, в якій гравець взаємодіє з моделлю ШІ через введення тексту з діями, фразами його персонажа. Гра базується на генерації світу, де потім кожен вибір користувача, кожна його дія призводить до зміни руху сюжету чи напакі продовжує його в тому самому русі. Завдяки цим змінам на ходу, гравці отримують унікальний ігровий досвід, який ніякий інший гравець не зможе отримати[11]. Ось стислий опис процесу гри, який можна поділити на декілька етапів:

#### **1. Введення інформації користувачем.**

Користувач грає в гру за допомогою інтерфейсу, а саме через inputfield (поле для введення тексту). В це поле він буде вводити початкові дані про світ та історію:

- жанр світу (наприклад, фентезі, наукова фантастика, історія тощо);
- тематика (вибір певної теми або стилю світу);
- ім'я персонажа;
- інший параметр, куди користувач може ввести будь-що: будь то напрям історії чи невеличка історія його персонажу;

Користувач може вводити текст довільно, тобто на нього не накладаються ніякі обмеження, і на основі його інформації буде згенеровано унікальний світ.

#### **2. Генерація світу на основі введених даних.**



Після збору необхідної інформації, скрипт CreateWorld починає обробляти її та формувати початковий запит до серверу OpenAI, який вже, в свою чергу, буде генерувати історію за допомогою обраної моделі користувачем.

Кожен введений параметр має вплив на світ та історію. 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.

Методи для надіслання запиту до серверу знаходяться в скрипті SendRequest. В цьому фомується та зберігається вся історія повідомлень та методи для обробки відповідей від серверу. Вся історія зберігається в форматі списку словарів та зберігається у json формат для зручності роботи з нею та серіалізацією/десеріалізацією. Ця історія передається до текстової моделі GPT-4(на вибір користувача).

### 3. Текстова взаємодія з користувачем.

Гра працює через текстовий інтерфейс та показує історію користувачу у вигляді тексту. Гравець вводить своїх подальші повідомлення, дії чи події у текстове поле та отримує відповідь після кожного такого запиту, яка, так само, виводиться на екран. Відповідь може містити:

- опис подій, що відбуваються у світі;
- відповіді на дії користувача;
- опис результатів вибору, зробленого користувачем;
- опис місця, куди потрапив гравець;
- тощо;

Всі ці елементи створюють відчуття реалістичної взаємодії з вигаданим світом та диктором, який розповідає тобі історію від другої особи, де кожен вибір впливає на хід подій.

### 4. Збереження історії переписки.

Важливою складовою є збереження всієї історії та переписки з моделлю ШІ для подальшого використання та збереження контексту історії, яка

відбувалась від час сеансу. Після кожної отриманої відповіді буде відбуватися автозбереження, яке запише інформацію у json файл, що дасть можливість використати її в майбутньому, якщо сеанс прервався. Це дає нам такий функціонал:

- можливість повернутися до попередніх повідомлень від ІІІ та переглянути, що було сказано раніше;
- відстеження прогресу сюжету в реальному часі;

Функціонал збереження та завантаження реалізований у скрипті SaveLoadScript, який зберіє весь список словарів, в якому в зручному форматі записано всі повідомлення, після кожної відповіді від серверу. Це дає можливість грати без втрати прогресу при закритті додатку.

#### 5. Інтерфейс і відображення історії переписки.

Всі повідомлення між користувачем та системою відображаються в scroll view, що дає можливість зручного розташування на екрані та гортання для перегляду минулих повідомлень. Кожне нове повідомлення буде створюватися за допомогою префабів, які будуть наповнюватися необхідним текстом за з'єкономить ресурси комп'ютера[15]. Вони будуть вигляді у вигляді діалогу, аби яскравіше передати атмосферу розмови з диктором історії так гравцем:

- повідомлення користувача;
- відповіді від системи;

Це забезпечує наочність і зручність використання, даючи гравцеві можливість легко переглядати всі попередні повідомлення між ним та ІІІ.

#### 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 та втратою логіки при створенні історії;

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

#### **1.4 Постановка задачі та технічне завдання**

Тестові ігри, які активно використовують ШІ для генерації історії, набувають широкої популярності, оскільки вони дають гравцям унікальний персоніфікований досвід та дають можливість відчувати свій вплив на історію[17]. Головною задачею є текстова гра, яка використовує моделі GPT-4 та DALL E від OpenAI для генерації історії та візуального зображення/ілюстрації, яка буде розвивати історію від другої особи завдяки діям гравця.

Гра буде орієнтована на весь спектр категорій модливих аудиторій та надає можливість ввімкнути family-friendly режим для дітей чи молодих користувачів, що дозволить використовувати додаток в навчальних цілях через гру, що дуже заохочує дітей.

Основні завдання було обрано такими:

##### **1) Інтерактивний ігровий процес.**

- розробка системи для обробки текстових команд;
- адаптація сюжету відповідно до дій користувача;
- можливість вибору жанру світу (наприклад, фентезі, наукова фантастика, історичний, тощо);
- впровадження режиму family-friendly для забезпечення безпеки контенту;

##### **2) Інтеграція технологій OpenAI.**

- використання GPT-моделей для створення текстових сценаріїв;
- застосування DALL·E для формування візуальних елементів гри;
- оптимізація запитів для ефективного використання токенів;

##### **3) Графічний інтерфейс та візуалізація.**

- розробка зрозумілого інтерфейсу для користувачів;
- інтеграція текстових діалогів і візуальних ілюстрацій у відповідні розділи екрану;

#### 4)Музичний супровід

- додавання музичних композицій, які відповідають обраному жанру гри;
- налаштування звуку та можливість вибору плейлиста;

#### 5) Збереження прогресу гри.

- реалізація функції автозбереження;
- додання можливості відновлення попередніх сесій через інтерфейс;

#### 6) Сумісність з платформами.

- оптимізація гри для роботи на Windows і Linux;
- тестування функціональності на кожній платформі;

В результаті виконання всіх цих завдань буде отримано ігровий додаток, який, у вигляді текстової гри, дає гравцю адаптивний сюжет, широкий спектр жанрів, графічний та музичний супровід.

### **1.5 Обґрунтування вибору інструментальних засобів розробки**

Для розробки гри необхідно буде використати сучасні інструменти, які забезпечують стабільність, зручність та можливість інтеграцій сторонніх бібліотек/фреймворків для специфічних задач, таких як обробка відповіді від серверу наприклад. Для реалізації текстової гри було обрано: рушій Unity, мову програмування C#, моделі ШІ від OpenAI та їх інтеграція за допомогою API та інші бібліотеки.

Unity - один з найпопулярніших ігрових рушіїв на ринку на сьогоднішній день[19]. Його доступність та зручність використання робить його чудовим вибором. У ньому наявний широкий спектр інструментів та функціоналу, таких як:

Гнучкість та масштабованість - Unity має функціонал для розробки як 2D, так і 3D ігор, що дає змогу адаптувати його під різні девайси.

Швидка інтеграція інструменті - даний рушій надає функціонал інтеграції сторонніх бібліотек та API для розширення і без того широкого спектру можливостей

Кросплатформність - Unity підтримує створення ігор на всі популярні платформи: від Windows до IOS та веб-додатків.

Зручність роботи з UI - двигун дає широкий спектр графічних налаштувань, що дає можливість створити гарний та дружлюбний інтерфейс для гри.

C# є мовою програмування, яку використовує Unity. За допомогою сторонньої бібліотеки також можна використовувати C++, але в цьому немає потреби, оскільки функціонал C# нас повністю задовільняє:

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

Сумісність з Unity - C# повністю оптимізований для роботи з Unity, що дає нам доступ до всіх можливостей та функціоналу рушія.

Об'єктно-орієнтоване програмування - C# дозволяє використовувати класову систему, що є дуже корисним при розширенні функціоналу в майбутньому, оскільки це спрощує написання нових методів.

Для генерації контенту було обрано моделі GPT-4 для генерації тексту та історій, та моделі DALL·E 2/3 для створення зображень на основі контексту історії. Вибір зумовлений:

Висока якість генерації тексту - моделі від OpenAI зарекомендували себе як одні з найкращих на ринку. Їх широкий спектр можливостей повністю охоплює всі вимоги, які нам потрібні.

Гнучкість у налаштуванні - використовуючи system prompts ми можемо задати поведінку та правила, які має дотримуватися модель, що робить її гнучкою та дає можливість внести зміни в майбутньому за потреби.

Розширені можливості - моделі DALL·E задовільняють нашу потребу у створенні детальних зображень для візуального зображення історії для покращення занурення гравця.

Вибраний спектр інструментів та засобів дає можливість повністю реалізувати поставлену задачу по створенню гри з адаптивним сюжетом та геймплеєм, генерації текстової історії, візуалізації історії за допомогою зображень в контексті історії, зручний та детальний інтерфейс та кросплатформеність. Данні засоби забезпечують зручність розробки та відповідають найм вимогам.

## **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. ПРОЄКТУВАННЯ, РОЗРОБКА ТА ПОБУДОВА ДОДАТКУ

### 2.1 Проєктування архітектури гри

Проєкт побудований навколо взаємодії між різними основними об'єктами, кожен з яких має своє призначення, створення світу, збереження та завантаження світу, обробка введених даних, відправлення запиту на сервер тощо. За допомогою чіткої структури та правильно побудованої архітектури владося плавно та модульно інтегрувати ці компоненти.

**CreateWorld:** відповідальність цього скрипта лежить на створенні першого повідомлення з роллю `system`, в якому буде описано всі необхідні вказівки та правила для моделі ШІ. Він буде брати дані, введені гравцем при створенні світу, формувати коректний запит та передавати його до скрипту `SendRequest`.

**SendRequest:** цей скрипт відповідає за “діалог” між гравцем та моделлю. Він отримує дані від гравця, оброблює їх у коректний формат та надсилає до серверу `OpenAI`. На його плечі також лягає обробка відповіді від серверу та передача відповіді іншим скриптам для подальших дій.

**TokenCount:** оскільки у всіх моделей типу `GPT` є обмеження в пам'яті, то необхідно контролювати обсяг та розмір всієї історії повідомлень між гравцем та моделлю, щоб вона не перевищувала ліміт (125 000 токенів у моделей `GPT-4`). Методи в цьому скрипті підраховують загальну кількість токенів у всій історії повідомлень та, за потреби, видаляють старіші повідомлення.

**SaveLoadScript:** скрипт `SaveLoadScript` має в собі код та методи, які виконують збереження та запис історії повідомлень між гравцем та ШІ у локальний `json` формат, що дозволить гравцям повертатися до вже створених історій та продовжувати їх.

**ShowMessage:** У скрипті `ShowMessage` реалізовано створення та виведення на екран гравця повідомлень від ШІ чи його повідомлення у вигляді префабів, стилізований як діалог/переписка між оповідачем(ШІ) та слухачем(гравцем).



Введення даних користувачем та генерація світу: гравець, починаючи нову гру, вводить початкові параметри світу за допомогою `InputField`(наприклад, жанр, тематику, назву т.д.). Після введення, ці дані передаються у скрипт `CreateWorld`, який, в свою чергу, формує коректний запит та передає його до `SendRequest` для надіслання на сервер.

Обробка введеного тексту: всі текстові повідомлення гравця проходять через скрипт `SendRequest`, який відправляє запит до серверу, очукує відповідь та передає її до `ShowMessage`, який вже, в свою чергу, виводить повідомлення на екран гравця у зручному форматі.

Управління токенами: при кожному введенні повідомлення метод обраховує його розмір в токенах у скрипті `TokenCount`. Якщо історія повідомлення близька до ліміту, то найстаріші повідомлення видаляються, що дозволяє зберегти “пам’ять” про найновіші події.

Збереження та відображення історії: всі повідомлення зберігаються у вигляді переписки на локальному файлі за допомогою компонента `SaveLoadScript`. Цей скрипт також відповідає за завантаження всієї історії при повторному завантаженні після перезапуску гри. Самі ж повідомлення виводяться на екран за допомогою методів у скрипті `ShowMessage`.

Обробка подій: будь яка дія гравця, будь то введення тексту чи натискання кнопок, викликає відповідні частини коду в скриптах. Це дозволяє забезпечити інтерактивність гри.

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.

У підрозділі 2.3 буде здійснено детальний опис основних скриптів, які реалізують логіку гри та взаємодію з користувачем.

## 2.2 Опис процесів у проекті

У даному підрозділі буде детальний опис процесів, що відбуваються у грі. Для більш детально та візуального опису будуть використовуватися діаграми, які покажуть взаємодію користувача з системою та іншими компонентами додатку.

Основні етапи включають:

Оцінка та перевірка токенів - важливий крок, оскільки він гарантує, що запит не перевищує обмеження по токенам і система з моделлю ШІ буде працювати коректно.

Обробка запиту користувача - отримання, обробка та передача зпиту до API, яке відбувається одразу після перевірки на кількість токенів.

Збереження історії переписки — процес збереження результатів після кожного запиту.

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

Процеси, описані вище, забезпечують основний функціонал гри, де кожна частина відповідає за певну задачу в циклі взаємодії з користувачем.

Перейдемо до розгляду цих всіх етапів детальніше, почавши з процесу перевірки токенів.

Оцінка та перевірка кількості токенів це важлива частина обробки взаємодії між гравцем та системою. За допомогою неї можна гарантувати коректну обробку запитів від ШІ, щоб кількість токенів не перевищувала ліміт, визначений для запиту. На данній діаграммі активності зображено як відбувається перевірка та обробка токенів під час кожної взаємодії з системою.

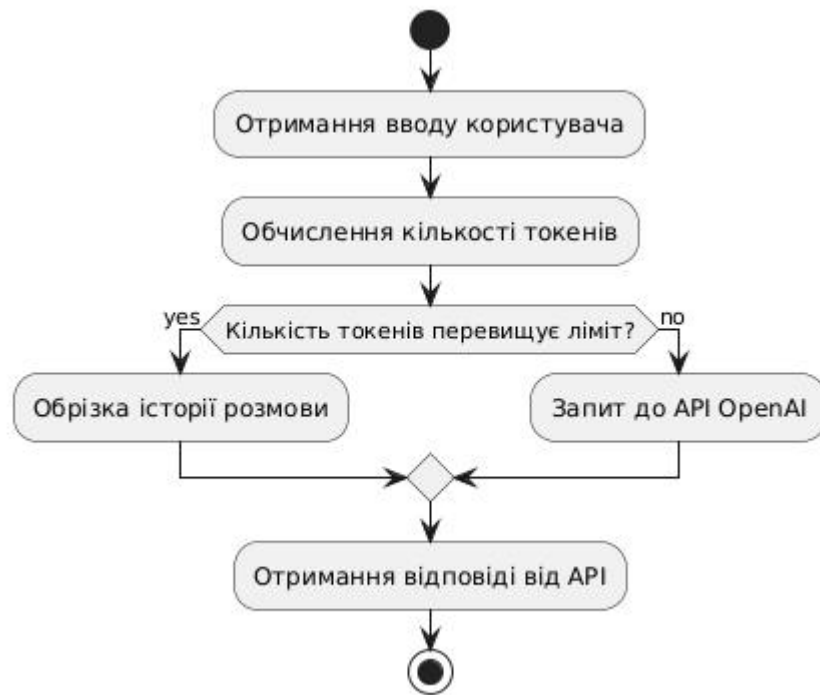


Рис.2.1. Діаграма активності для токенів

Далі приведено діаграму послідовності роботи між користувачем та системою.

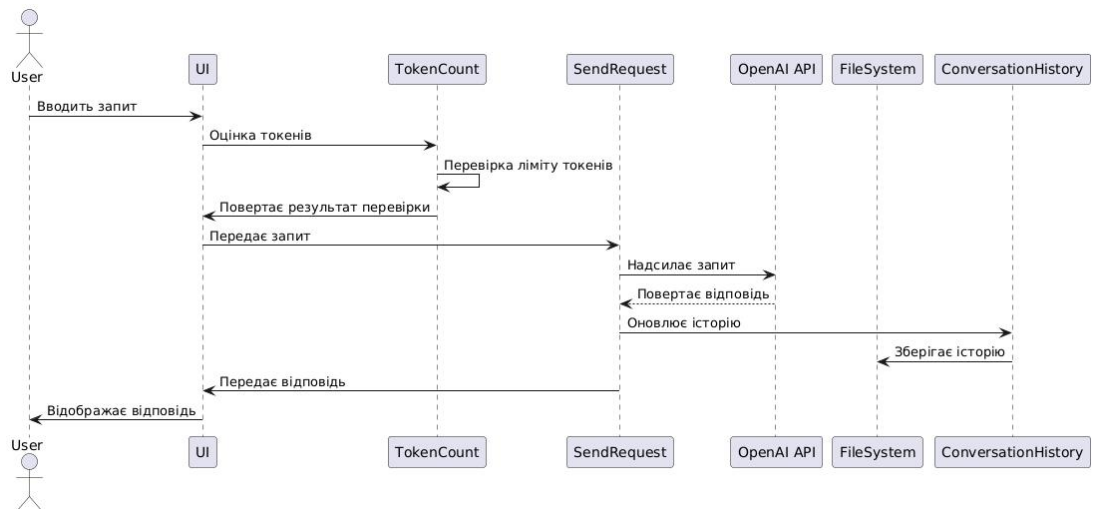


Рис.2.2. Діаграма послідовності

Процес взаємодії з системою починається, коли користувач вводить свій запит через інтерфейс гри. Цей запит ініціює обробку та передається в компонент SendRequest для подальшої обробки.

Перш ніж запит потрапить до зовнішнього API, система спочатку оцінює кількість токенів, які будуть витрачені на його обробку. Це важливо для того, щоб перевірити, чи не перевищує кількість токенів встановлений ліміт.

Якщо кількість токенів перевищує ліміт, система повідомляє користувача про помилку. Якщо ж ліміт не порушено, система продовжує обробку запиту та передає його до OpenAI API.

API обробляє запит і повертає відповідь, яку система використовує для подальших дій. З отриманою відповіддю система зберігає її в історії переписки, щоб користувач міг переглядати попередні повідомлення.

Нарешті, результат передається назад в інтерфейс користувача для відображення на екрані.

Цей процес дозволяє обробляти кожен запит коректно та у межах встановлених обмежень, що важливо для стабільної роботи системи та ефективного використання API.

Процес збереження та завантаження історії переписки є важливою частиною гри, оскільки він дозволяє користувачам переглядати попередні повідомлення та відновлювати контекст взаємодії з системою. Ця діаграма демонструє, як система зберігає відповіді після кожного запиту та завантажує їх під час наступних сеансів гри.

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

Збереження історії переписки: Коли відповідь отримано, система зберігає її в історії переписки. Компонент SendRequest взаємодіє з файловою системою, щоб зберегти історію в локальному файлі.

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

Запис історії в файл: Історія переписки зберігається у форматі JSON, що дозволяє легко зберігати, завантажувати та переглядати ці дані пізніше.

Завантаження історії при запуску: Коли користувач запускає гру або продовжує взаємодіяти з системою, система завантажує попередню історію з локального файлу. Це дозволяє відновити контекст попередніх розмов і продовжити без втрат.

Відображення історії на UI: Завантажена історія відображається на інтерфейсі користувача у вигляді списку повідомлень, що дозволяє гравцю переглядати старі повідомлення та взаємодіяти з ними.

Цей процес збереження та завантаження історії забезпечує безперервність взаємодії з користувачем і дозволяє зберігати важливу інформацію, щоб продовжити гру без втрат контексту.

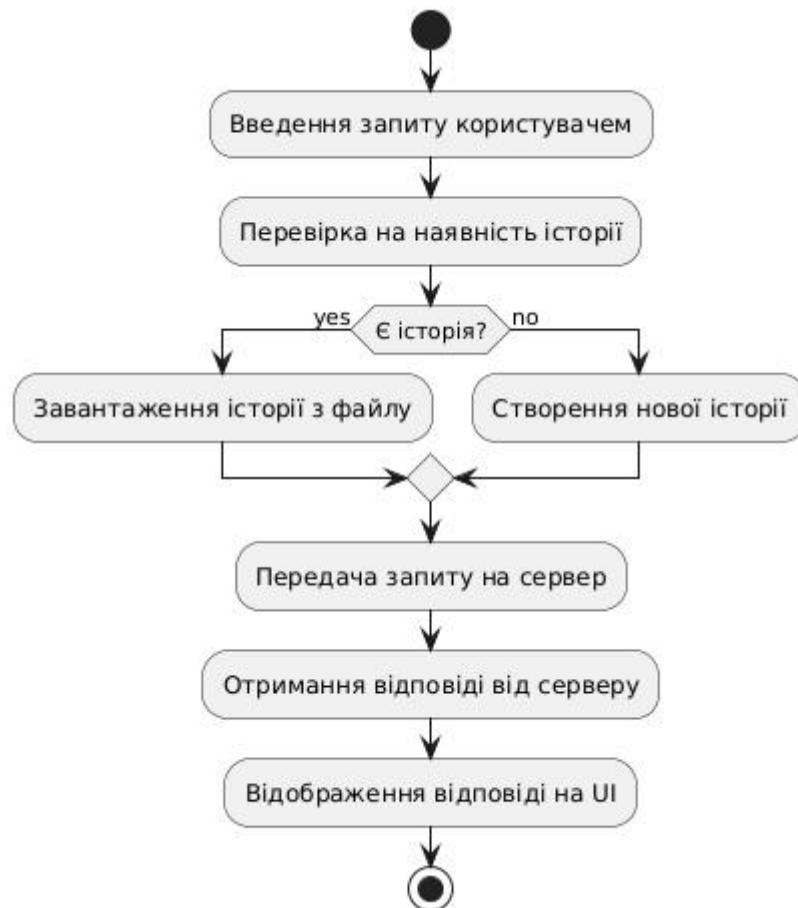


Рис.2.3 Діаграма активності для взаємодії з користувачем

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 активності 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 діаграм 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.

## **2.3 Робробка та написання коду гри**

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

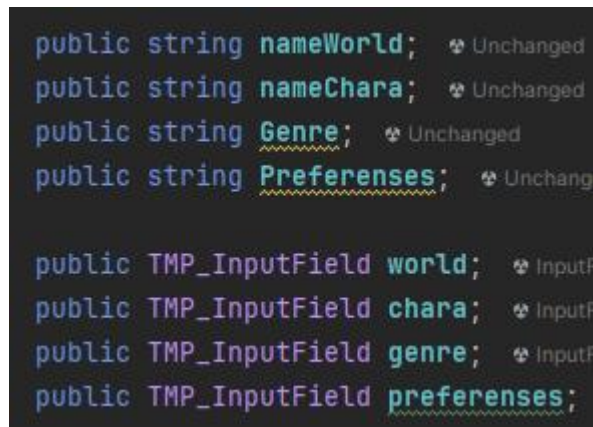


забезпеченні цих функцій відіграють такі скрипти, як `SendRequest`, `SaveLoadScript`, `TokenCount` та `ShowMessage`.

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

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:



```
public string nameWorld;  ⚙️ Unchanged
public string nameChara;  ⚙️ Unchanged
public string Genre;      ⚙️ Unchanged
public string Preferences; ⚙️ Unchang

public TMP_InputField world; ⚙️ InputF
public TMP_InputField chara; ⚙️ InputF
public TMP_InputField genre; ⚙️ InputF
public TMP_InputField preferences;
```

Рис.2.5

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:



```
private Button btn;
private SendRequest send;
```

Рис. 2.6.

`btn` 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() виконується при ініціалізації об'єкта в грі:

```
void Start()
{
    btn = gameObject.GetComponent<Button>();
    btn.onClick.AddListener(generate);
    send = GameObject.FindWithTag("Manager").GetComponent<SendRequest>();
}
```

Рис. 2.7.

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

```

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();
    Preferences = preferences.text.ToString();
    Debug.Log("nameWorld: " + nameWorld + "\n" + "nameChara: " + nameChara + "\n" + "Genre: " + Genre + "\n" + "Preferences: " + Preferences);
    StartCoroutine(routine: send.CreateStartingStory(nameWorld, nameChara, Genre, preferences: Preferences));
}

```

Рис. 2.8.

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 Preferences.

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.

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:

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

```
public List<GameObject> arr = new List<GameObject>(); // Serializable
public GameObject btnPrefab; // LoadPrefab
public Transform scrollContent; // Content (Transform)
public ShowMessage showMessage; // Unchanged
public GameObject loadPanel; // LoadPanel
public SendRequest sendRequest; // Unchanged
public ImageSendRequest imageSendRequest; // Unchanged
```

Рис. 2.9.

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

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

scrollContent — контейнер для кнопок у вигляді скролл-в'ю, 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.

```
public void Start()
{
    showMessage = gameObject.GetComponent<ShowMessage>();
    sendRequest = gameObject.GetComponent<SendRequest>();
    imageSendRequest = gameObject.GetComponent<ImageSendRequest>();
}
```

Рис. 2.10.

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()

```
// Метод для завантаження історії розмови з файлу
4 usages Tymur
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>>();
    }
}
```

Рис. 2.11.

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.

```
// Clean prefabs
2 usages Tymur
public void CleanBttn()
{
    if(arr.Count == 0)
        return;

    foreach (var i :GameObject in arr)
    {
        Destroy(i);
    }
    arr.Clear();
}
```

Рис. 2.12.

Метод `CleanBtn()` очищає список кнопок. Якщо в `arg` 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.

```
//load files and bttns
1 usage  Tymur
public void ReadAndLoadBttns()
{
    string savePath = Application.persistentDataPath;
    string[] files = Directory.GetFiles(savePath, searchPattern: "*.json");

    CleanBtn();
    foreach (string file in files)
    {
        string worldName = Path.GetFileNameWithoutExtension(file);

        GameObject button = Instantiate(btnPrefab, scrollContent);
        button.GetComponentInChildren<TMP_Text>().text = worldName;

        button.GetComponent<Button>().onClick.AddListener(call: () => LoadWorld(file));
    }
}
```

Рис. 2.13.

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

- отримує список всіх файлів `.json` з директорії збережень.
- очищає поточні кнопки через метод `CleanBtn()`.
- 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.



```

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 : Dictionary<string, string> 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);
}

```

Рис. 2.14.

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 для завантаження історії розмови з файлу
4 usages  Tumor
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>>();
    }
}

```

Рис. 2.15.

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.

Метод `EstimateTokens` виконує базову оцінку кількості токенів у переданому тексті.

```
public int EstimateTokens(string text)
{
    // Умовно кожне слово або група символів вважається за один токен
    var words:string[] = text.Split(new char[] { ' ', '\n', '\t' }, System.StringSplitOptions.RemoveEmptyEntries);
    int tokenCount = 0;

    foreach (var word:string in words)
    {
        tokenCount += Encoding.UTF8.GetByteCount(word) / 4; // Оцінка: середній токен має 4 байти
    }
    return tokenCount;
}
```

Рис. 2.16.

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.

```
public int CountTokensInConversation(List<Dictionary<string, string>> conversationHistory)
{
    int totalTokens = 0;
    foreach (var message :Dictionary<string,string> in conversationHistory)
    {
        totalTokens += EstimateTokens(message["role"]);
        totalTokens += EstimateTokens(message["content"]);
        totalTokens += 4; // Для службових токенів API
    }
    Debug.Log(totalTokens);
    return totalTokens;
}
```

Рис.2.16.

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.

```
public void TrimConversationHistory(List<Dictionary<string, string>> conversationHistory, int maxTokens)
{
    while (CountTokensInConversation(conversationHistory) > maxTokens && conversationHistory.Count > 2)
    {
        conversationHistory.RemoveAt(3); // Видаляємо запис
    }
}
```

Рис. 2.17.

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.

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.

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.

```

public IEnumerator CreateStartingStory(string worldName, string characterName, string genre, string preferences)
{
    if(showMessage.messagesObjects.Count != 0)
        showMessage.DeleteAllMessages();// Clear history if this is the first request

    var startMessage = "";
    if (PlayerPrefs.GetInt( key: "familyMode") == 1)
    {
        startMessage =
            "You are a story generator designed for family-friendly content. Your role is to create stories suitable for children."
    }
    else
    {
        startMessage = "You are a creative and engaging story generator. Your role is to narrate immersive, branching stories in the world of {worldName}."
    }
    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);
}

```

Рис. 2.18.

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 неприйняттого 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` 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 вдавсь;

На самому початку методу йде перевірка:

```
if (isRequestInProgress)
{
    Debug.Log("Запит вже обробляється.");
    yield break;
}
```

Рис. 2.19.

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

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

```
isRequestInProgress = true; // Встановлюємо, що запит почав оброблятися
ShowPanelForLog( message: "Loading prompt to server...", needButton: false);
```

Рис. 2.20.

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

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:

```
var jsonData :{model,messages,...} = new {
    model = PlayerPrefs.GetString( key: "textModel"),
    messages = conversationHistory,
    max_tokens = PlayerPrefs.GetInt( key: "maxTok"),
    temperature = PlayerPrefs.GetFloat( key: "temp")
};
```

Рис. 2.21.

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

Для передачі даних використовується UnityWebRequest:

```
using (UnityWebRequest request = new UnityWebRequest(apiUrl, method: "POST"))
{
    byte[] jsonToSend = Encoding.UTF8.GetBytes(jsonString);
    request.uploadHandler = new UploadHandlerRaw(jsonToSend);
    request.downloadHandler = new DownloadHandlerBuffer();
    request.SetRequestHeader( name: "Authorization", $"Bearer {PlayerPrefs.GetString( key: "API")}");
    request.SetRequestHeader( name: "Content-Type", "application/json");

    yield return request.SendWebRequest();
}
```

Рис. 2.22.

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;

Після завершення запиту йде обробка відповіді:

Якщо запит успішний:

```

// 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);
}

```

Рис. 2.23.

- 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:

```

else
{
    Debug.LogError("Choices array is empty or null.");
    DeleteMessage("Choices array is empty or null.");
}

```

Рис. 2.24.

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

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` 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.

Метод починається з виклику `ShowPanelForLog`, який інформує гравця про початок процесу генерації промту:

```
ShowPanelForLog( message: "Loading prompt for image...", needButton: false);
```

Рис. 2.25.

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

```
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  
    }  
});
```

Рис. 2.26.

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:

```
var jsonData :{model,messages,...} = new
{
    model = PlayerPrefs.GetString( key: "textModel"),
    messages = conversationForImage,
    max_tokens = 250,
    temperature = 1
};
```

Рис. 2.27.

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;

```
using (UnityWebRequest request = new UnityWebRequest(apiUrl, method: "POST"))
{
    byte[] jsonToSend = Encoding.UTF8.GetBytes(jsonString);
    request.uploadHandler = new UploadHandlerRaw(jsonToSend);
    request.downloadHandler = new DownloadHandlerBuffer();
    request.SetRequestHeader( name: "Authorization", $"Bearer {PlayerPrefs.GetString( key: "API")}");
    request.SetRequestHeader( name: "Content-Type", "application/json");
}
```

Рис. 2.28.

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 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.

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.

```
sendRequest = GetComponent<SendRequest>();
```

Рис. 2.29.

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.

```
if (sendRequest.conversationHistory.Count == 0)  
    yield break;
```

Рис. 2.30.

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`:

```
sendRequest.ShowPanelForLog( message: "Generating Image...", needButton: false);
```

Рис. 2.31.

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):

```
var jsonData:string = JsonConvert.SerializeObject(new { model = PlayerPrefs.GetString(key: "imageModel"), prompt = prompt, n = 1, size = "1024x1024" });
```

Рис. 2.32.

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

```
sendRequest.ShowPanelForLog( message: "Generating Image...", needButton: false);  
var jsonData:string = JsonConvert.SerializeObject(new { model = PlayerPrefs.GetString(key: "imageModel"), prompt = prompt, n = 1, size = "1024x1024" });  
var request = new UnityWebRequest(API_URL, method: "POST");  
byte[] bodyRaw = System.Text.Encoding.UTF8.GetBytes(jsonData);  
request.uploadHandler = new UploadHandlerRaw(bodyRaw);  
request.downloadHandler = new DownloadHandlerBuffer();  
request.SetRequestHeader( name: "Content-Type", "application/json");  
request.SetRequestHeader( name: "Authorization", $"Bearer {sendRequest.apiKey}");  
yield return request.SendWebRequest();
```

Рис. 2.33.

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.

```
if (request.result == UnityWebRequest.Result.Success)  
{  
    var response = JsonUtility.FromJson<DALLEImageResponse>(request.downloadHandler.text);  
    StartCoroutine( routine: LoadImage(response.data[0].url));  
}  
else  
{  
    Debug.LogError("Error: " + request.error);  
    sendRequest.ShowPanelForLog( message: "Error: " + request.error, needButton: true);  
}  
sendRequest.resultPanel.SetActive(false);
```

Рис. 2.34.

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.

```
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);
}
```

Рис. 2.35.

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.

```
public string GenerateUniqueFileName()
{
    string timestamp = DateTime.Now.ToString( format: "yyyyMMdd_HH:mm:ss");
    string fileExtension = ".png";
    string uniqueFileName = $"{sendRequest.worldNameBase}_{timestamp}{fileExtension}";
    return uniqueFileName;
}
```

Рис. 2.36.

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.

```
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}");
        }
    }
}
```

Рис. 2.37.

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:

```
public class DALLEImageResponse
{
    public DALLEImageData[] data;
}

[System.Serializable]
public class DALLEImageData
{
    public string url;
}
```

Рис. 2.38.

Скрипт `ImageSendRequest` є основним елементом для створення зображень у 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.

## Висновки до другого розділу

У другому розділі звіту розглядаються архітектура гри та її основні компоненти, що забезпечують її функціональність.

Архітектура гри та взаємодія компонентів: Гра заснована на текстовій взаємодії між користувачем і штучним інтелектом. Система генерує світ на основі введених користувачем даних і відповідає на запити. Це реалізується через використання різних скриптів, таких як `CreateWorld`, `SendRequest`, `SaveLoadScript`, `ShowMessage` та `TokenCount`. Всі ці компоненти працюють разом, щоб створити інтерактивний досвід для гравця, де кожен етап — від введення даних до генерації та збереження історії — чітко контролюється.

Процес збереження та управління токенами: Важливим аспектом гри є механізм управління історією повідомлень, що гарантує коректне збереження та перегляд попередніх повідомлень. Це досягається через збереження даних у файли за допомогою скрипта `SaveLoadScript`, що дозволяє завантажувати та зберігати історію. Також контролювання кількості токенів забезпечує оптимальну роботу з API, що дозволяє зберігати історію в межах лімітів.

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

Інтеграція зовнішніх сервісів: Взаємодія з OpenAI API дозволяє генерувати відповіді на запити користувача. Всі дані взаємодії обробляються локально, що дає змогу контролювати потік інформації та зберігати історію при необхідності.

Діаграми: У розділі наведені діаграми послідовності та блок-схеми, які допомагають візуалізувати основні етапи обробки даних: надсилання запиту до API, обробка відповіді та збереження історії. Ці діаграми дають чітке розуміння внутрішніх процесів гри та її функціонування.

В загальному, другий розділ підсумовує важливі компоненти гри, їх взаємодію та роль у забезпеченні стабільної й ефективної роботи гри.



## РОЗДІЛ 3. РОЗГОРТАННЯ СИСТЕМИ, НАЛАШТУВАННЯ GUI ТА ТЕСТУВАННЯ

### 3.1 Розгортання системи та налаштування інтерфейсу користувача в ігровому рушії та інспекторі Unity.

У цьому підрозділі буде розглянуто налаштування та побудова інтерфейсу у рушії Unity для зручної роботи з системою. Інтерфейс це важлива частина будь якої програми, оскільки вона відповідає за взаємодію між користувачем та системою, подаючи всі обрахунки та код у вигляді красивого екрану додатку.

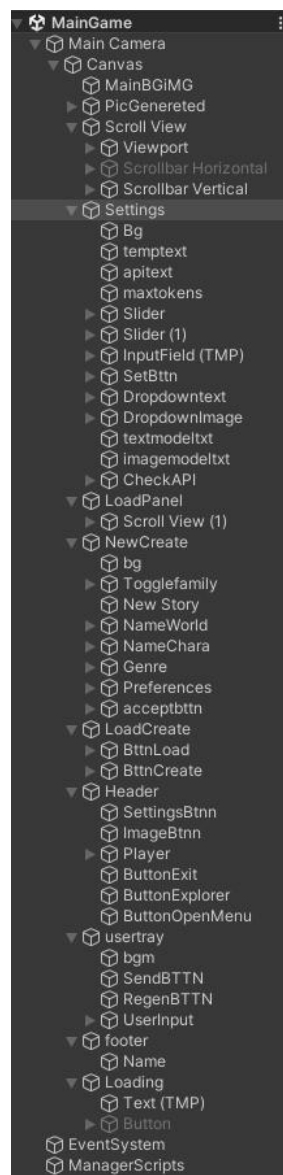


Рис.3.1. Ієрархія сцени

Інтерфейс додатку побудований на основі Canvas — стандартного компоненту Unity для роботи з UI-елементами.

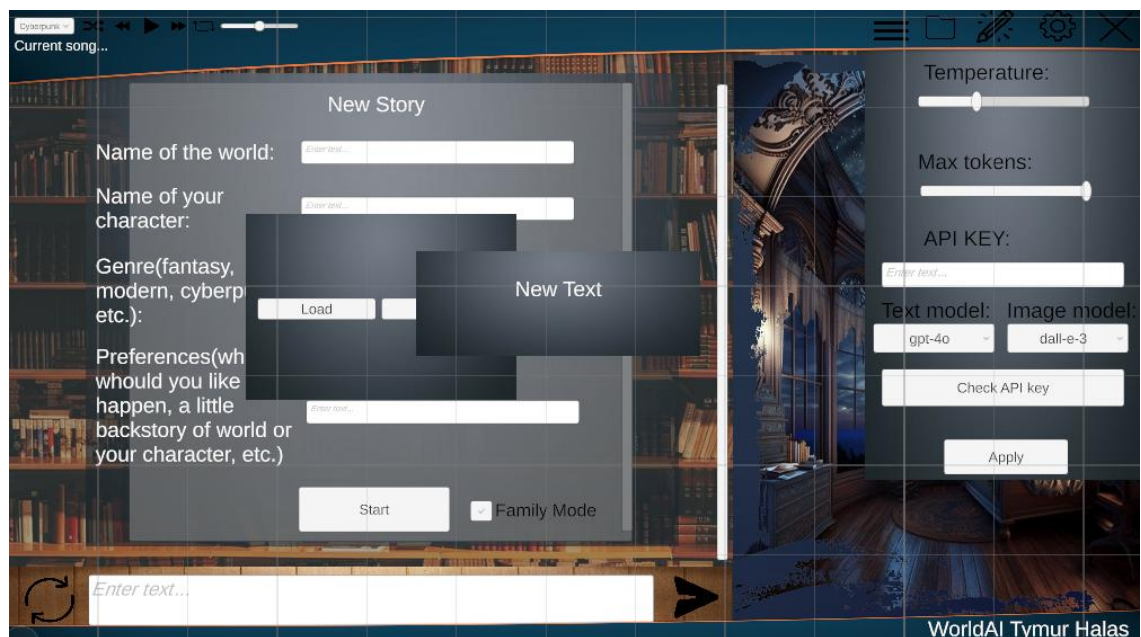


Рис.3.2. Вигляд сцени у редакторі  
У структурі інтерфейсу реалізовано кілька основних модулів:

Головне меню — панель для вибору початкових налаштувань світу та персонажа. Включає в себе текстові поля для введення назви світу, імені персонажа, вибору жанру (наприклад, фентезі, кіберпанк), а також поле для вводу побажань гравця.

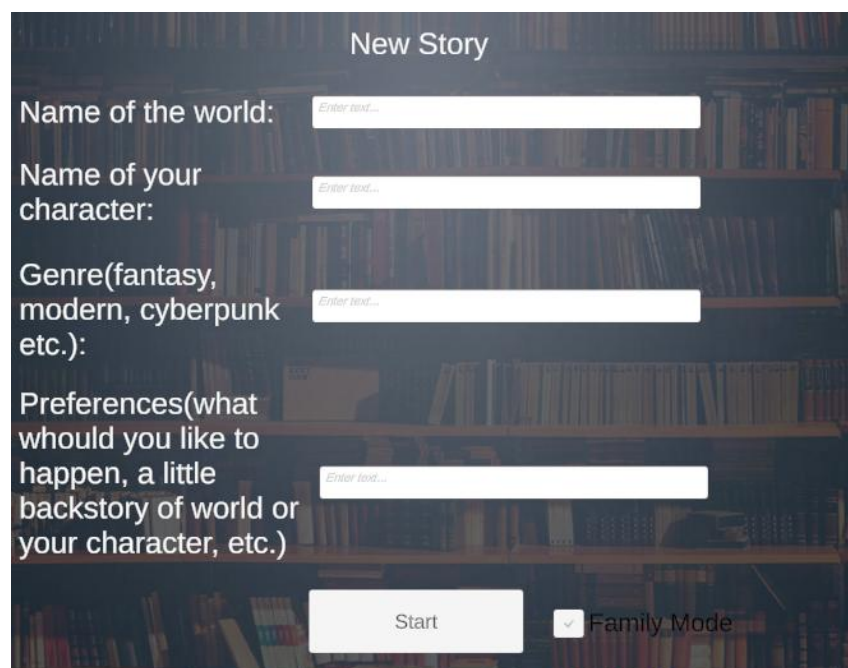


Рис.3.3. Створення світу

Панель параметрів AI — панель, що дозволяє гнучко налаштовувати параметри генерації, такі як температура моделі, максимальна кількість токенів, вибір текстової та візуальної моделей, а також введення API-ключа та його перевірка на дійсність.



Рис.3.4. Вікно налаштувань

Кнопки управління — інтерактивні елементи для виконання операцій, таких як завантаження/створення, відкриття папки з сейвами, генерація зображення, налаштувань та вихід з гри



Рис.3.5. Кнопки операцій

Зона результатів — текстова область, у якій відображається згенерована історія у вигляді діалогу за допомогою префабів у scroll view.

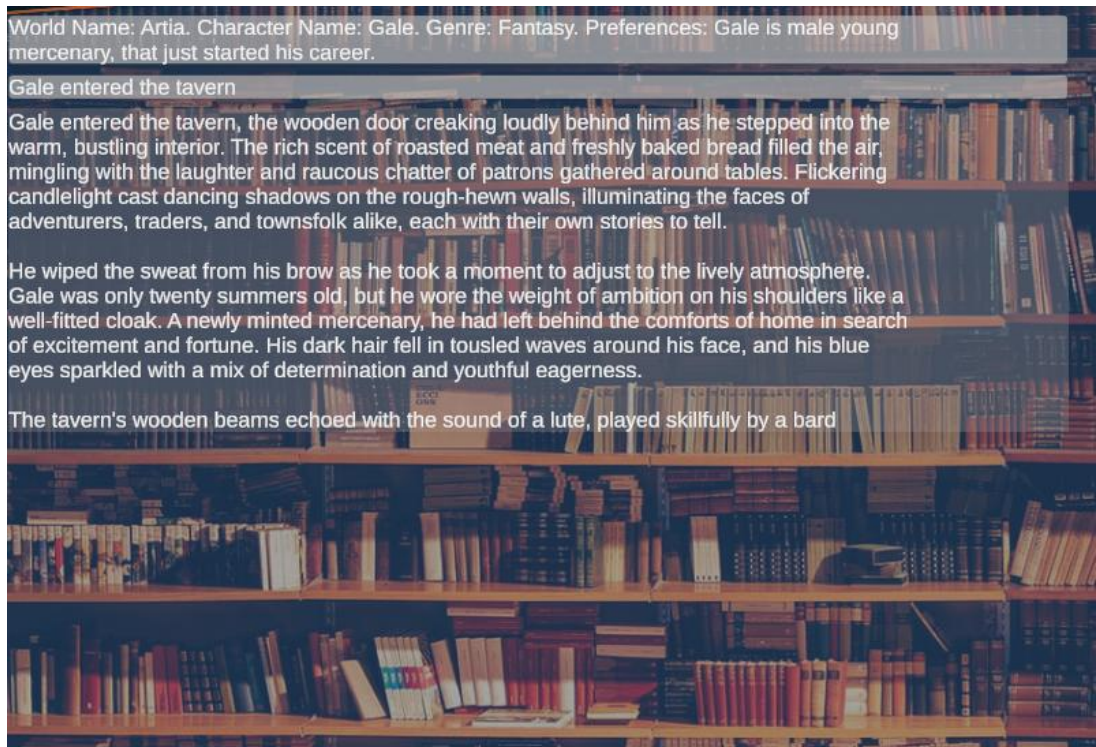


Рис.3.6. Поле повідомлень

Панель вводу - це інструменти та поле, за допомогою якого користувач спілкується з грою та рухає історію вперед. В ній він може ввести текст, надіслати його або повторно згенерувати минуле повідомлення від ШІ.

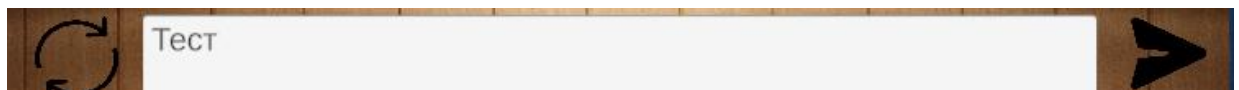


Рис.3.7.Панель вводу

Рамка зображення - за бажанням, користувач може створити зображення на основі історії, яке буде згенероване та поміщене у цю область.



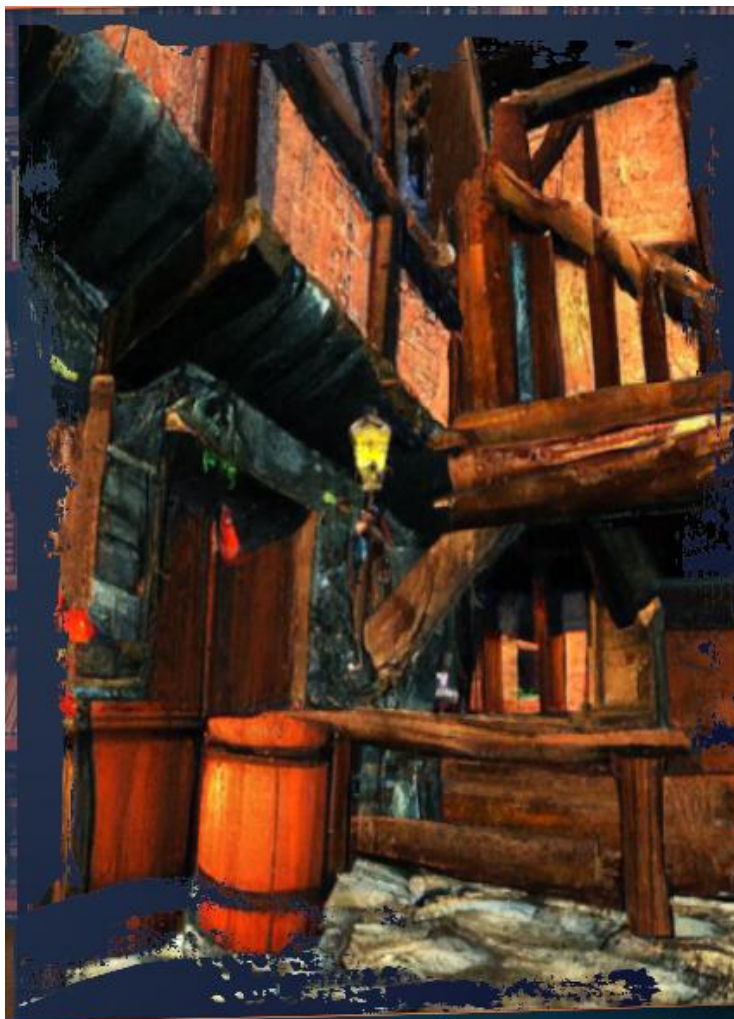


Рис.3.8.Рамка зображення

Панель плеєра - це вбудований музикальний плеєр, за допомогою якого гравець може слухати музичне супроводження для глибшого занурення в атмосферу. Гравець може обрати жанр композицій, аби воно співдало з сеттінгом історії.

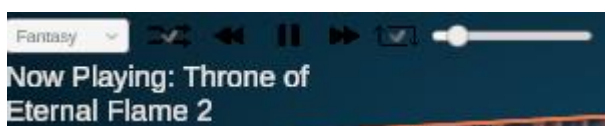


Рис.3.9.Панель плеєра

Панель завантаження - у користувача може бути декілька сеансів, і для цього було зроблено функціонал для завантаження минулих світів.



Рис.3.10.Панель завантаження

Також було створено пустий об'єкт ManagerScripts, в якому знаходяться всі необхідні компоненти для виконання функції під час гри.

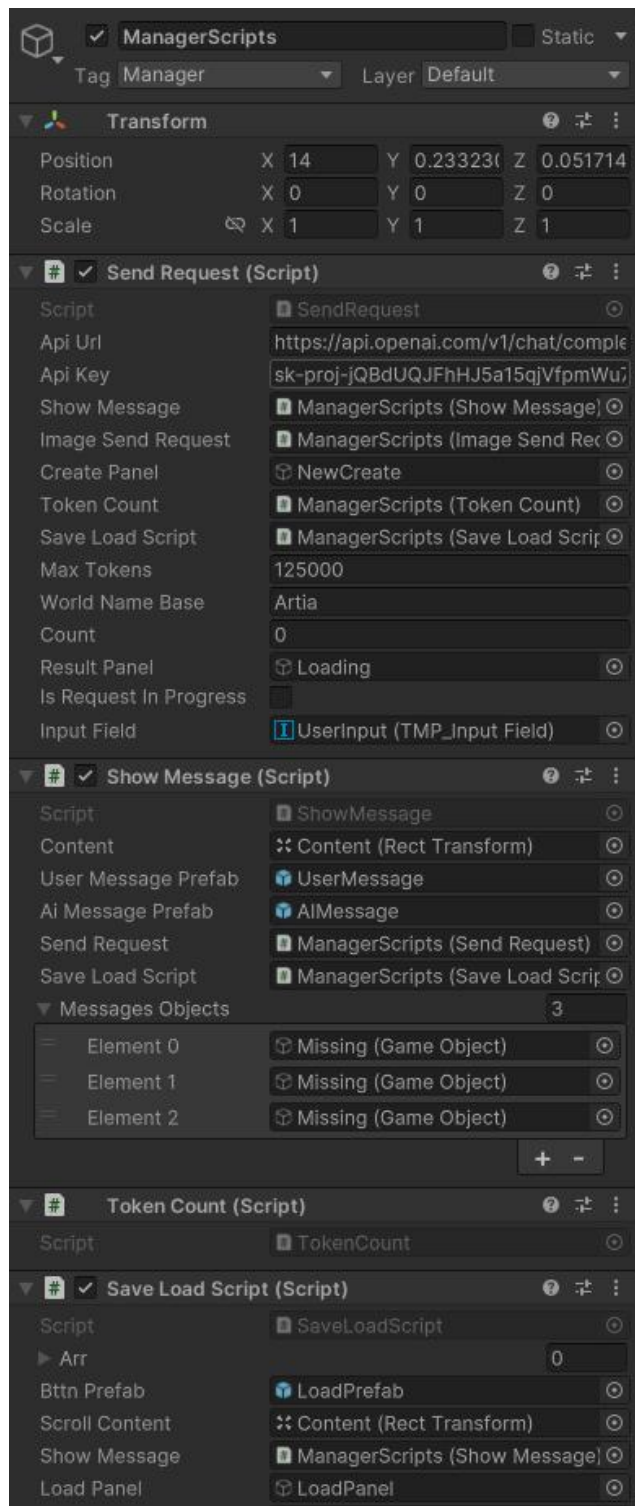


Рис.3.11.Панель завантаження

Для побудови інтерфейсу використовувалися базові компоненти Unity, такі як TextMeshPro для роботи з текстом, Button для інтерактивних кнопок, та Slider для налаштування числових параметрів. Рушій Unity дозволив налаштувати ці елементи відповідно до вимог функціоналу, забезпечуючи інтеграцію між GUI та основною логікою програми.

### 3.2 Інтерфейс та порядок роботи

У цьому підрозділі буде описано принцип роботи з додатком, що і як потрібно натискати, яка послідовність роботи користувача з додатком і заодно перевірено його коректність роботи(ручне тестування). Додаток має інтуїтивно зрозумілий інтерфейс, що дозволить новим користувачам легше адаптуватися та звикнути до нього.

При запуску гри нас зустрічає панель з 2 кнопками, за допомогою яких можна або завантажити та продовжити попередній сеанс або створити повністю новий світ. Почнемо з створення нового світу для початку, оскільки в нас немає ще попередньої історії. При натисканні кнопки створення у нас з'являється нова панель з налаштуваннями нового світу, де гравець може ввести:

- назву світу;
- назву свого персонажа;
- жанр світу(фентезі, наукова фантастика, тощо.);
- побажання(поле, куди користувач може ввести будь яку інформацію, будь то побажання для сюжету чи опис свого персонажа.);
- family mode(спеціальний режим для молодшої аудиторії);

Після введення інформації та натискання кнопки підтвердити, запит створюється на надсилається на сервер, після чого гравець отримує початок історії.

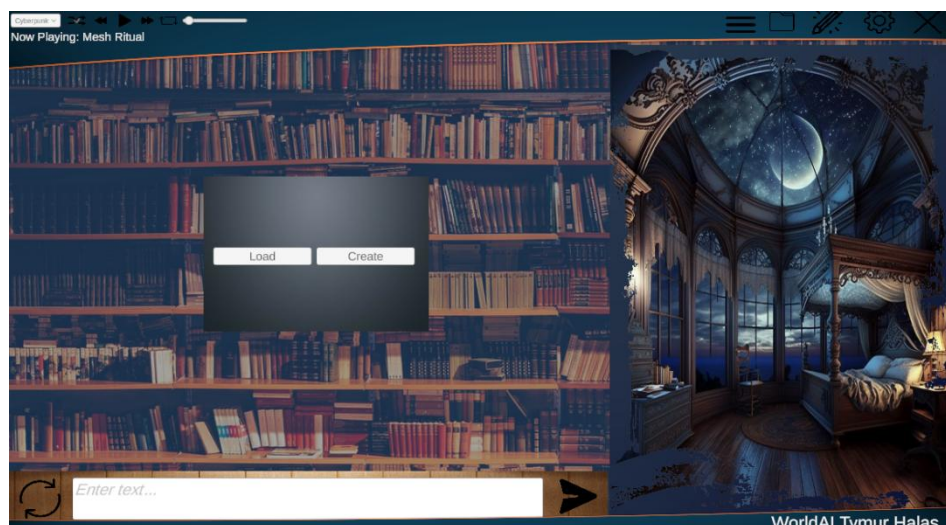
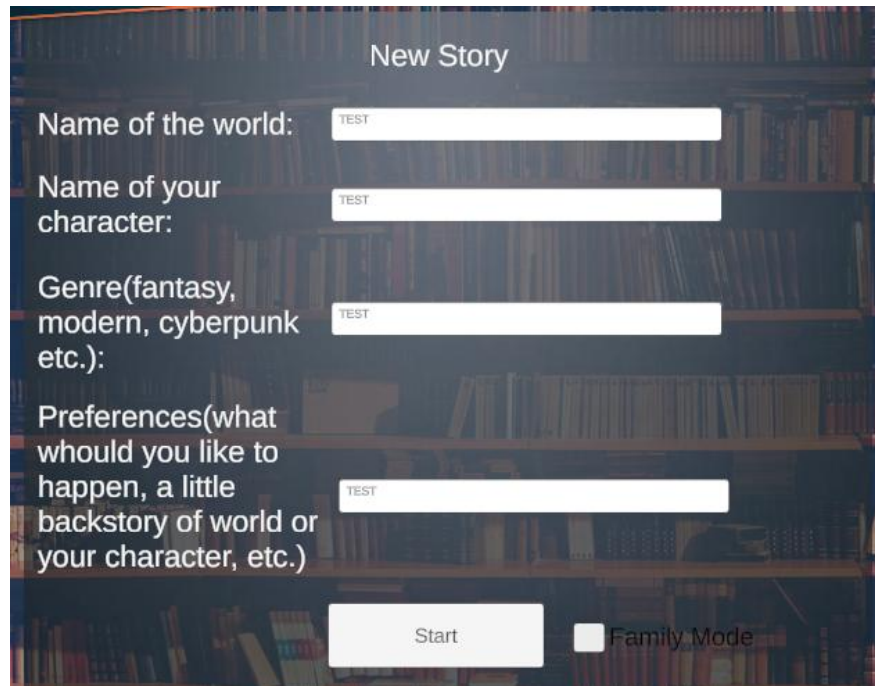


Рис.3.12. Головне меню гри





New Story

Name of the world:

Name of your character:

Genre(fantasy, modern, cyberpunk etc.):

Preferences(what would you like to happen, a little backstory of world or your character, etc.):

☐ Family Mode

Рис.3.13. Панель створення

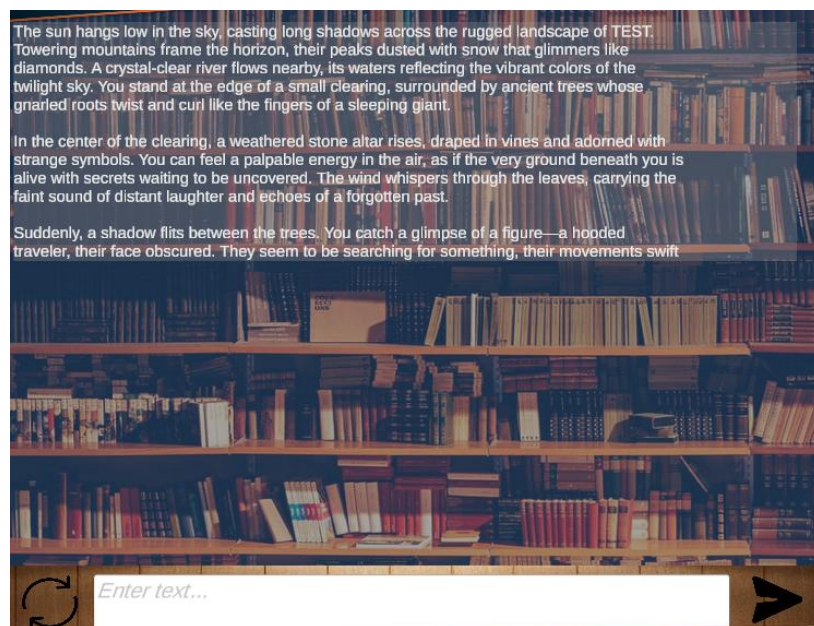


Рис.3.14. Успішне створення світу

Якщо ж гравець вимкне гру та запустить її знову, він, натиснувши на кнопку завантаження, отримає список усіх попередніх світів та зможе повертися до нього, щоб продовжити гру.



Рис.3.15. Кнопка з минулим світом

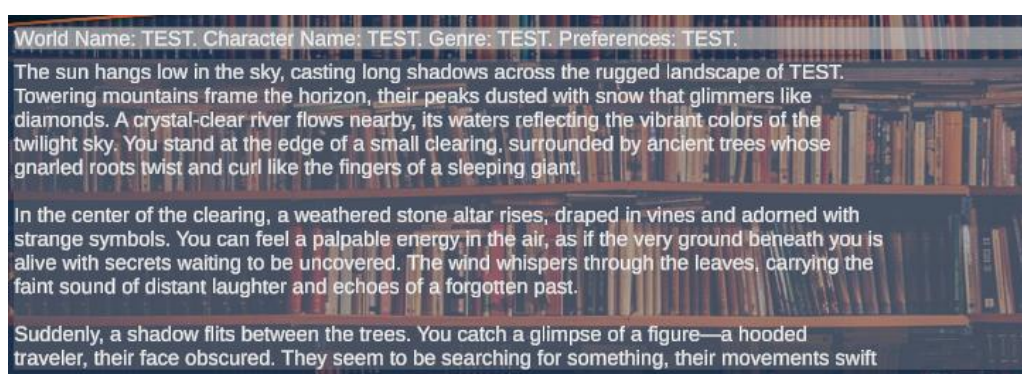


Рис.3.16. Успішне завантаження попередньої сесії

Якщо гравцю не сподобався текст історії, він може натиснути на спеціальну кнопку регенерації повідомлення, яке видалить останнє повідомлення та та перепише цю частину історії.

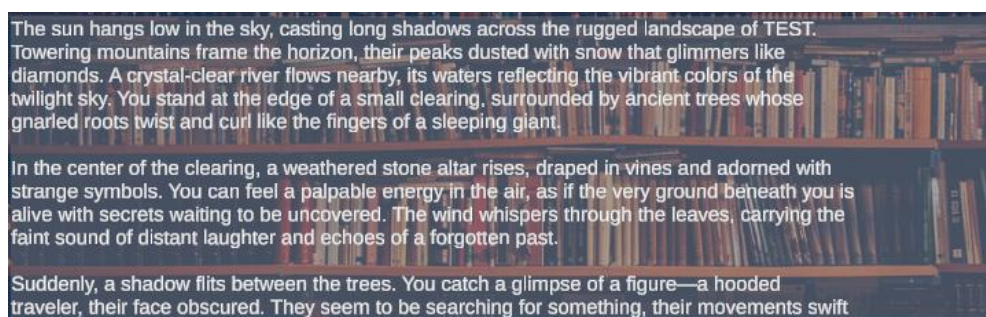


Рис.3.17. Старе повідомлення



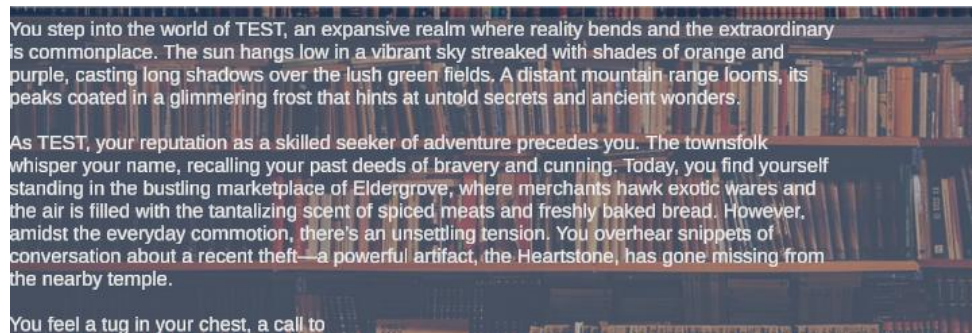


Рис.3.18. Регенерації повідомлення

Також гравець може власноруч відредагувати повідомлення, натиснувши на нього двічі, відкриється редим релагування та гравець зможе додати/видалити/відредагувати текст.

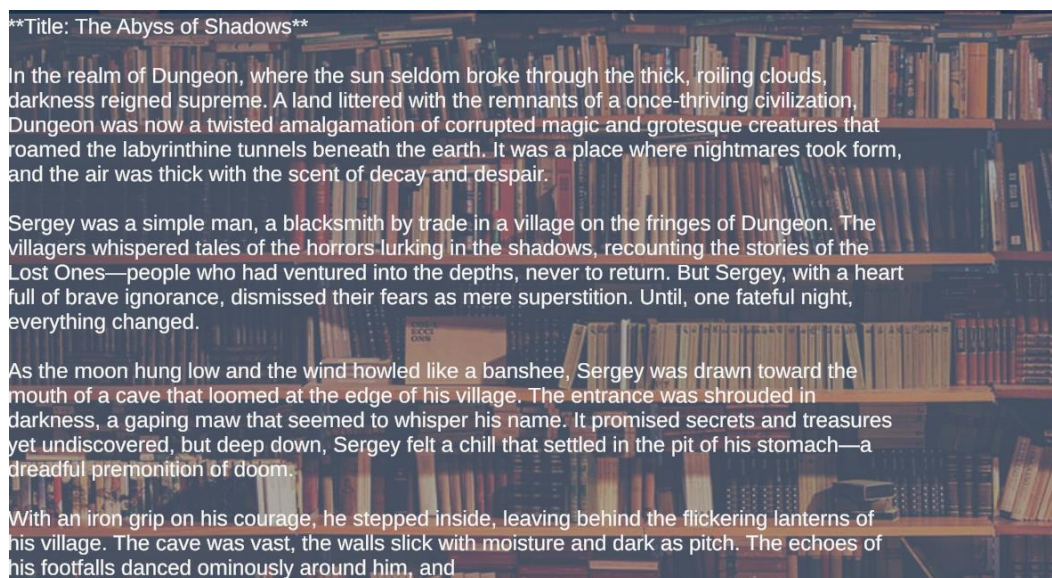


Рис.3.19. Повідомлення для редагування



Рис.3.20. Вікно редактора



Рис.3.21. Відредаговане повідомлення

Також гравець може використати вбудований у гру музичний плеєр у лівій верхній частині екрану для прослуховування музики. Його функціонал:

- відтворення;
- пауза;
- наступний трек/попередній;
- зміна плейлісту;
- випадкова пісня;
- повтор;
- гучність;



Рис.3.22. Програвання музики

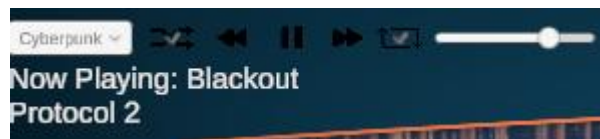


Рис.3.23. Перемикання музики

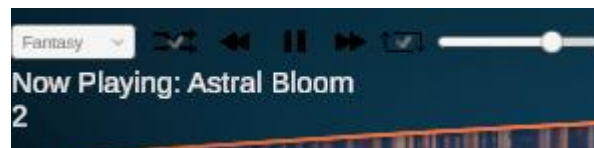


Рис.3.24. Зміна плейлісту

Також гравцю дана панель налаштувань в правій верхній частині екрану, в якій він може налаштувати:

- температуру моделі;
- максимальну кількість токенів для відповіді від ШІ;
- ввести свій API ключ для доступу до моделей ШІ;
- обрати моделі для генерації тексту чи зображень;
- спеціальна кнопка для перевірки дійсності API ключа;
- кнопка застосування змін;

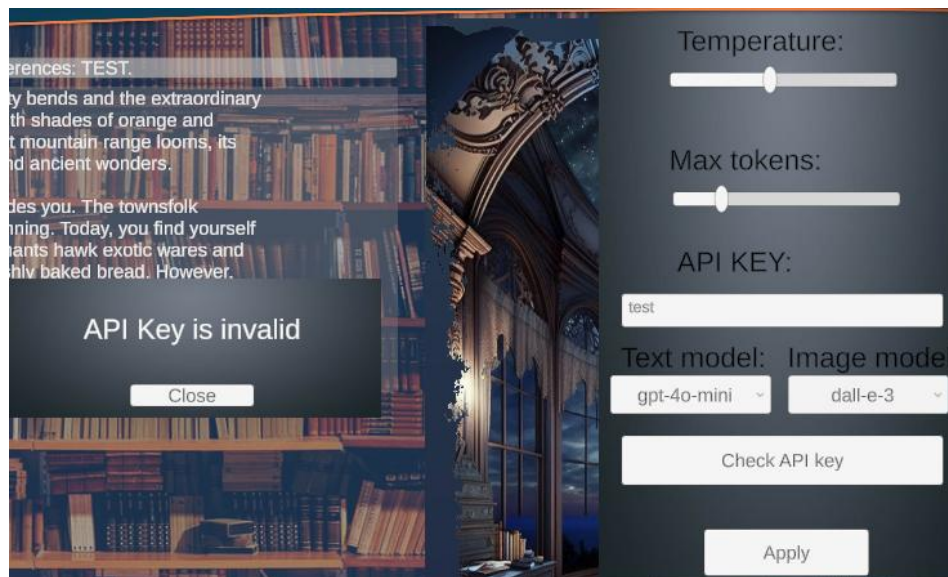


Рис.3.25. Некоректний ключ

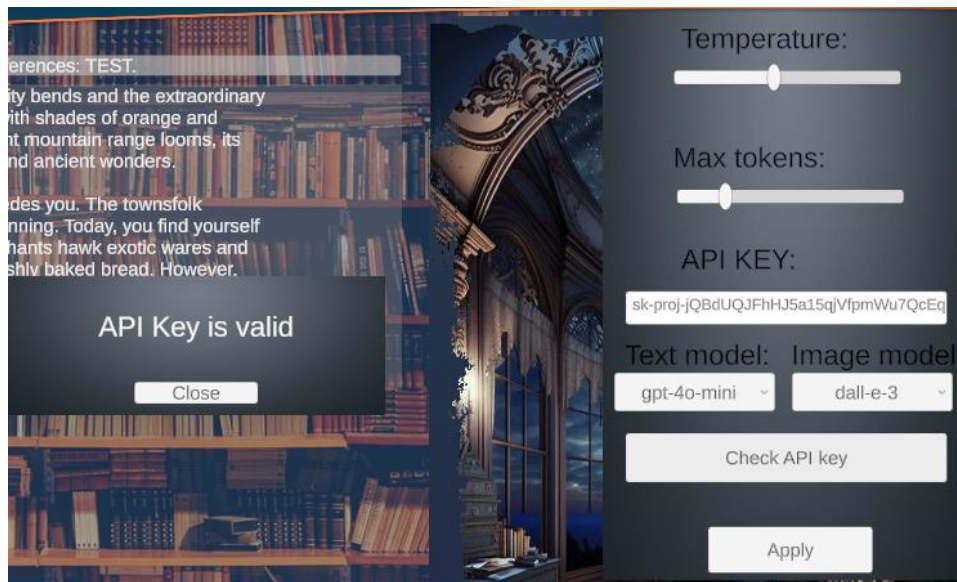


Рис.3.26. Коректний ключ

Аби продовжити історію, у гравця є текстове поле знизу екрана, де він може писати продовження історії(що зробив його персонаж, що трапилося у світі, або просто залиши полу порожнім, щоб ШІ продовжив свою історію).



Після натискання клавіші правіше від поля, повідомлення надсилається та отримується відповідь.



Рис.3.27. Введення тексту

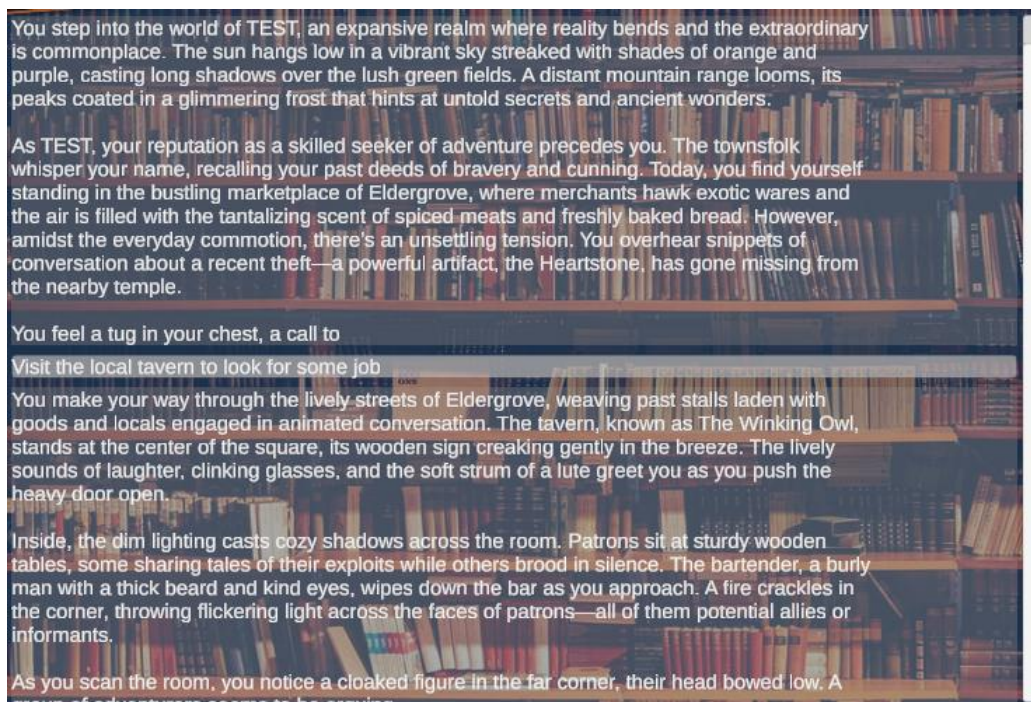


Рис.3.28. Продовження історії через повідомлення від гравця

Також, натиснувши на кнопку олівця в верхній правій частині екрану, почнеться генерація зображення до цієї історії.



Рис.3.29. Зображення до генерації



Рис.3.30. Згенероване зображення в контексті історії

### 3.3 Тестування ігрового додатку

Тестування - важлива частина будь якої розробки, будь то ІТ сфера чи щось інше. Завдяки детальній перевірці розробник може бути впевненим, що його додаток працює правильно або знайти помилки на етапі розробці, що полегшить його усунення. У цьому підрозділі буде описано тестування гри, а саме: підрахунок токенів, збереження/завантаження, виведення повідомлень на екран.

Під час тестування використовувалися як ручні, так і автоматизовані методи тестування продукту. Було перевірено та ми впевнилися у стабільності та коректності роботи додатку. Було приділено увагу кожній частині гри, щоб знайти всі помилки та збої та оперативно їх усунути на етапі розробки та дебагінгу.

На початку було приступлено до написання атоматизовних UNIT тестів для перевірки роботи класу `TokenCount`. Було написано декілька тестів, аби впевнитися у коректній роботі алгоритмів обрахунку кількості токенів у всій історії повідомлень між гравцем та ШІ. Потрібно дуже пильно перевірити коректність роботи цього функціоналу, оскільки від цього залежить досвід гравця на довгій дистанції користування грою. Якщо все буде працювати некоректно, то модель не буде працювати коректно або взагалі перестане давати відповіді на дії користувача. У тестах, описаних у коді, перевіряються основні аспекти підрахунку та обробки токенів. Для цього використовуються чітко визначені сценарії, які охоплюють різні можливі ситуації.

#### 1. Тест `EstimateTokens_ShouldCalculateCorrectly`:

У цьому тесті тестується функціональність методу `EstimateToken`, призначеного для підрахунку кількості токенів (символів чи слів) у текстовому повідомленні. Як приклад, якщо передати методу рядок `"Hello world!"`, ми очікуємо, що текст розіб'ється на два токени — окремі слова. Тест перевіряє коректність підрахунку токенів у різних прикладах, зокрема для звичайних текстових повідомлень.



```

[Test]
2 Tymur
public void EstimateTokens_ShouldCalculateCorrectly()
{
    // Arrange
    string text = "Hello world!";

    // Act
    int tokenCount = tokenCounter.EstimateTokens(text);

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

```

Рис. 3.31.

## 2. Тест CountTokensInConversation\_ShouldCalculateCorrectly:

Цей тест перевіряє метод CountTokensInConversation, який підраховує кількість токенів у всій історії розмови. Історія складається з повідомлень від трьох ролей: системи, користувача та асистента. Основна мета тесту - впевнитися, що токени для кожного повідомлення враховуються правильно, а загальна сума не виходить за допустимі межі. Для цього задаються граничні значення, які дозволяють перевірити точність і коректність роботи алгоритму навіть при різній довжині повідомлень.

```

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}.");
}

```

Рис. 3.32.

## 3. Тест TrimConversationHistory\_ShouldTrimExcessTokens:

Даний тест перевіряє метод TrimConversationHistory, який відповідає за видалення старих повідомлень, якщо ліміт токенів був перевищений та необхідно оптимізувати історію. Якщо історія перевищує 34 токени, то метод має обрізати старіші повідомлення, щоб історія не переходила за поріг у 34 токени.

```
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.");
}
```

Рис. 3.33.

#### 4. Тест TrimConversationHistory\_ShouldNotTrimBelowTwoEntries:

Цей тест перевіряє обмеження на мінімальну кількість записів в історії. Перші 2 записи в кожній історії несуть системний рахактер, тому їх видалення призведе до втрати контексту для майбутніх повідомлень і ші почне генерувати текст, не дотримуючись правил гри.

```
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.");
}
```

Рис. 3.34.

Тестування класу SaveLoadScript зосереджене на перевірці збереження та завантаження історії розмов, а також очищення кнопок, пов'язаних із збереженими світами. Метою тестів є впевнитися, що методи коректно зберігають і відновлюють дані та належно очищають елементи інтерфейсу.

### 1. Тест SaveConversationHistoryToFile\_ShouldSaveCorrectly:

Тест перевіряє коректність збереження історії розмови в файл. Для цього створюється тестова історія з двох повідомлень — від користувача та асистента. Метод SaveConversationHistoryToFile зберігає історію у файл, а після виконання тест перевіряє, чи файл створено, чи містить він правильний вміст і чи відповідає кількість повідомлень після десеріалізації початковим даним. Тест важливий для перевірки правильності збереження даних у форматі JSON та їх коректного завантаження.

```
[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, worldName: "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.");
}
```

Рис. 3.35.

### 2. Тест LoadConversationHistoryFromFile\_ShouldLoadCorrectly:

Тест перевіряє метод LoadConversationHistoryFromFile, який завантажує історію розмови з файлу. Спочатку в файл записується тестова історія, після чого тест перевіряє, чи правильно завантажено вміст. Зокрема, перевіряється, чи кількість елементів у завантаженій історії співпадає з оригінальним списком, а також чи зберігається зміст повідомлень. Цей тест важливий для перевірки коректної роботи механізму десеріалізації та правильності зчитування даних з файлу.

```

[Test]
a Tymur
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, contents: Newtonsoft.Json.JsonConvert.SerializeObject(testConversationHistory));

    // Act
    var loadedConversationHistory :List<Dictionary<string, string>> = 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.");
}

```

Рис. 3.36.

### 3.Тест

#### LoadConversationHistoryFromFile\_ShouldReturnEmpty\_WhenFileDoesNotExist:

Тест перевіряє поведінку методу LoadConversationHistoryFromFile, коли файл не існує. Оскільки система повинна коректно обробляти відсутність файлу, тест перевіряє, чи повертає метод порожній список замість виключення або помилки. Це дозволяє перевірити, що система безпечно працює з відсутніми файлами і не завершується аварійно.

```

[Test]
a Tymur
public void LoadConversationHistoryFromFile_ShouldReturnEmpty_WhenFileDoesNotExist()
{
    // Arrange
    string nonExistentFilePath = Path.Combine(Application.persistentDataPath, "NonExistent.json");

    // Act
    var result :List<Dictionary<string, string>> = saveLoadScript.LoadConversationHistoryFromFile(nonExistentFilePath);

    // Assert
    Assert.IsNotNull(result, "Результат не повинен бути null.");
    Assert.IsEmpty(result, message: "Історія розмови повинна бути порожньою.");
}

```

Рис. 3.37.

#### 4. Тест CleanBtnn\_ShouldCleanAllButtons:

Тест перевіряє метод CleanBtnn, який очищає список кнопок на інтерфейсі. Спочатку створюються дві кнопки з тегом "loadbtnn", після чого викликається метод очищення. Тест перевіряє, чи успішно очищено список кнопок. Це гарантує, що інтерфейс не містить непотрібних кнопок, які можуть погіршити користувацький досвід.

```

[Test]
 Tymur
public void CleanBtnn_ShouldCleanAllButtons()
{
    // Arrange
    GameObject button1 = new GameObject();
    button1.tag = "loadbtnn";
    GameObject button2 = new GameObject();
    button2.tag = "loadbtnn";
    saveLoadScript.arr = new List<GameObject>() { button1, button2 };

    // Act
    saveLoadScript.CleanBtnn();

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

```

Рис. 3.38.

Всі ці тести виконуються з використанням моків для зовнішніх залежностей, таких як ShowMessage, SendRequest та ImageSendRequest, що дозволяє ізолювати тестування логіки класу SaveLoadScript від інших компонентів гри. Моки використовуються, щоб уникнути виконання реальних методів, які можуть мати побічні ефекти або бути непотрібними для цього тесту, наприклад, відправка запитів чи обробка повідомлень в інтерфейсі.

Тестування класу ShowMessage спрямоване на перевірку коректності відображення та керування повідомленнями в інтерфейсі гри. Кожен тест перевіряє, чи правильно додаються, видаляються або оновлюються повідомлення, а також чи правильно клас взаємодіє з іншими компонентами, такими як SaveLoadScript.

#### 1. Тест DeleteAllMessages\_ShouldClearAllMessages:

Тест перевіряє, чи метод DeleteAllMessages правильно очищує всі повідомлення в інтерфейсі. Під час тестування додаються два повідомлення (від користувача та асистента) до списку відображених повідомлень. Після виклику методу тест перевіряє, чи очищено всі елементи в списках messagesObjects та arr з компоненту SaveLoadScript, який відповідає за збереження історії розмови. Це дозволяє переконатися, що очищення елементів



інтерфейсу відбувається коректно і що непотрібні елементи не зберігаються в системі.

```
[Test]
Tymur
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);
}
```

Рис. 3.39.

## 2. Тест AddUserMessage\_ShouldInstantiateUserMessage:

Тест перевіряє правильність створення та додавання повідомлення користувачем за допомогою методу AddUserMessage. Коли передаються дані у вигляді тексту, метод має створити новий об'єкт повідомлення та додати його до списку. Тест перевіряє правильність встановлення тексту у компоненті MessageScript - цей компонент відповідає за виведення повідомлення на екран користувача - і з'явлення повідомлень у списку користувача у потрібний час і порядку.

```
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);
}
```

Рис. 3.40.

### 3. Тест AddAIMessage\_ShouldInstantiateAIMessage:

Цей тест аналогічний попередньому, але перевіряє повідомлення від штучного інтелекту. Тестує, чи коректно створюється та відображається повідомлення від асистента. Він підтверджує, що для кожного типу повідомлення (користувач чи асистент) створюється відповідний об'єкт з правильним текстом, а повідомлення відображається в правильному списку. Це важливо для забезпечення коректного відображення повідомлень на екрані.

```
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);
}
```

Рис. 3.41.

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.

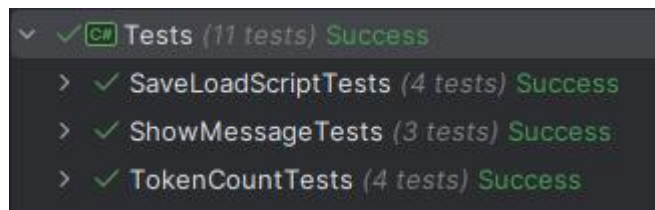


Рис.3.42. Результат виконання тестів

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 підтвердили, що механізм збереження та завантаження історії розмови працює коректно. Метод SaveConversationHistoryToFile успішно зберігає дані, а метод LoadConversationHistoryFromFile правильно відновлює історію, навіть коли

файл відсутній. Крім того, перевірка методу CleanBtn, що очищує список кнопок, показала, що система правильно взаємодіє з елементами інтерфейсу.

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.



## **Висновки до третього розділу**

У цьому розділі ми розглянули розробку програмного коду, який відповідає за ключові аспекти гри, зокрема за управління історією розмов, взаємодію з користувачем та збереженням даних. Ми детально обговорили роботу скриптів, що керують підрахунком токенів, збереженням та завантаженням історії повідомлень, а також відображенням повідомлень у інтерфейсі користувача.

Ми ознайомилися з різними методами для оцінки кількості токенів у повідомленнях, а також з техніками обробки історії розмов, що включають її збереження на диску. Окрему увагу було приділено управлінню відображенням повідомлень через елементи UI. Важливим етапом процесу є використання автоматизованих тестів, які дозволяють перевірити правильність функціонування цих компонентів і гарантують стабільність гри на всіх етапах взаємодії з користувачем.

Окрім цього, ми підкреслили важливість ручного тестування, яке доповнює автоматизовані перевірки та дозволяє оцінити загальну функціональність гри в реальних умовах. Це тестування допомагає виявити помилки, які можуть бути непомічені в процесі автоматичної перевірки.

Таким чином, цей розділ надає цілісне уявлення про розробку програмного коду для обробки даних і взаємодії з користувачем у грі, організацію збереження даних, а також про забезпечення належної стабільності та якості функціоналу за допомогою комплексних тестів.

## ВИСНОВКИ

В процесі виконання кваліфікаційної роботи був розроблений і реалізований проєкт текстової гри, основною особливістю якого є генерація світу та історії, що адаптуються залежно від даних, введених користувачем. У рамках цього проєкту було реалізовано кілька важливих механізмів, зокрема управління історією розмов, підрахунок токенів, збереження та завантаження даних, а також відображення повідомлень в інтерфейсі гри.

На початковому етапі було проведено аналіз потреб користувачів та технічних вимог, що дозволило визначити основні аспекти гри, такі як текстові діалоги, генерація світу та взаємодія з історією повідомлень. Створено структуру даних, яка забезпечує збереження розмов і налаштовує гнучку взаємодію з користувачем.

Далі було проведено детальне тестування програмного коду. У рамках автоматизованого тестування перевірено коректність підрахунку токенів, правильність роботи механізмів збереження та завантаження історії, а також ефективність системи відображення повідомлень в інтерфейсі. Ручне тестування також підтвердило стабільність функціональності в реальних умовах, що дало змогу виявити недоліки та покращити взаємодію між компонентами.

В результаті було реалізовано наступні ключові компоненти:

- система підрахунку токенів: ефективно визначає кількість токенів у тексті та історії розмов;
- система збереження та завантаження даних: забезпечує збереження та відновлення історії розмов у файлах;
- система відображення повідомлень: гарантує коректне відображення повідомлень користувача та асистента;
- автоматизовані та ручні тести: забезпечують надійність і стабільність функціональних механізмів гри;

Цей етап роботи дозволив успішно реалізувати основні компоненти гри, забезпечити стабільність і працездатність усіх механізмів, а також підтвердити,

що всі функції працюють відповідно до вимог. Результати тестування показали, що розроблений проєкт відповідає як технічним, так і 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. Catalanian 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.
  19. Галас Т. Т., Петросян Р.В. Генерація ігрового контенту за допомогою штучного інтелекту: адаптивні сценарії та світи. Тези доповідей VII Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення», м. Житомир, 02–03 грудня 2024 р. Житомир: Житомирська політехніка, 2024.

# ДОДАТКИ

## Додаток А

### SaveLoadScript.cs

```
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 btnnPrefab;
    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 CleanBtnn()  
{  
    if(arr.Count == 0)  
        return;  
  
    foreach (var i in arr)  
    {  
        Destroy(i);  
    }  
    arr.Clear();  
}
```

```
//load files and btnns  
public void ReadAndLoadBtnns()  
{  
    string savePath = Application.persistentDataPath;  
    string[] files = Directory.GetFiles(savePath, "*.json");  
  
    CleanBtnn();  
    foreach (string file in files)  
    {  
        string worldName = Path.GetFileNameWithoutExtension(file);  
  
        GameObject button = Instantiate(btnnPrefab, 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

```
using 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 btnn;

    private SendRequest send;

    public Toggle toggleFamily;
    void Start()
    {
        btnn = gameObject.GetComponent<Button>();
        btnn.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,  
Preferences));  
    }  
}
```

## 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_HH:mm:ss");
    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 3

### 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

```
using 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);
}
}
```