МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»

ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

**ПОЯСНЮВАЛЬНА ЗАПИСКА**

до кваліфікаційної роботи освітнього ступеня «магістр»

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

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

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

Житомир – 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»

ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП’ЮТЕРНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП’ЮТЕРНИХ НАУК

ЗАТВЕРДЖУЮ

Завідувач кафедри комп’ютерних наук

Марина ГРАФ

«25» січня 2024 р.

**ЗАВДАННЯ**

на кваліфікаційну роботу освітнього ступеня «магістр»

за спеціальністю 122 «Комп'ютерні науки»

(освітня програма «Комп'ютерні науки»)

Здобувач вищої освіти: **ГАЛАС Тимур Тимурович**

Керівник роботи: **ПЕТРОСЯН Руслан Валерікович**

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

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

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

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

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

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

**Календарний план**

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

Здобувач вищої освіти **Тимур ГАЛАС**

Керівник **Руслан ПЕТРОСЯН**

# **РЕФЕРАТ**

Пояснювальна записка до кваліфікаційної роботи на тему «Механіки ігрових процесів в слайдскроллерах з елементами платформінгу» складається з переліку умовних скорочень, вступу, трьох розділів, висновків, списку використаної літератури та додатку.

Текстова частина викладена на **47** сторінках друкованого тексту.

Пояснювальна записка має **31** сторінки додатків. Список використаних джерел містить **20** найменування і займає **2** сторінки. В роботі наведено **11** рисунків. Загальний обсяг роботи – **88** сторінок.

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

У другому розділі було проведено проектування гри та аналіз механік.

У третьому розділі був розроблено ігрові механіки, додано аудіо-візуальну складову гри, був описаний код головних механік гри.

Висновок містить в собі результати виконаної роботи по розробці гри в жанрі сайдскролер.

ABSTRACT

The explanatory note to the qualification work on the topic "Game Mechanics in Side-scrolling Platformers" consists of a list of abbreviations, introduction, three chapters, conclusions, a list of references and an additions.

The text part is set out on 47 pages of printed text.

The explanatory note has 31 pages of appendices. The list of references includes 20 titles and occupies 2 pages. The paper contains 11 pictures. The total volume of the work is 88 pages.

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

The second chapter focuses on game design and mechanics analysis.

The third chapter involves the development of game mechanics, the addition of audio-visual components to the game, and a description of the code for the main game mechanics.

The conclusion presents the results of the work done on the development of a side-scrolling platformer game.

**ЗМІСТ**

[РЕФЕРАТ 3](#_Toc17774)

[ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 7](#_Toc1184)

[ВСТУП 8](#_Toc13515)

[РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ТА ПЕРЕГЛЯД ТЕХНОЛОГІЙ ДЛЯ 9](#_Toc15288)

[СТВОРЕННЯ ГРИ 9](#_Toc32404)

[1.1 Постановка задачі 9](#_Toc5444)

[1.2 Аналіз існуючих технологій реалізації ігрових додатків 11](#_Toc16600)

[1.3 Обґрунтування вибору інструментальних засобів розробки 16](#_Toc1765)

[1.4 Постановка задачі та технічне завдання 18](#_Toc21330)

[Висновки до першого розділу 20](#_Toc26494)

[РОЗДІЛ 2. ОПИС ПРИНЦИПУ РОБОТИ ГРИ ТА АРХІТЕКТУРИ ПРОЕКТУ 22](#_Toc32522)

[2.1 Принцип роботи гри 22](#_Toc15079)

[2.2 Архітектура гри 25](#_Toc17344)

[2.3 Опис процесів у проекті 27](#_Toc26648)

[Висновки до другого розділу 34](#_Toc8666)

[РОЗДІЛ 3. РОЗРОБКА ТА ПРОГРАМУВАННЯ САЙДСКРОЛЛЕР ГРИ 36](#_Toc5607)

[3.1 Розробка механік, пов’язаних з ігровим персонажем 36](#_Toc21686)

[3.2 Розробка інших скриптів та механік 46](#_Toc2708)

[3.3 Розробка інших аспектів гри 53](#_Toc2186)

[Висновки до третього розділу 55](#_Toc3242)

[ВИСНОВКИ 57](#_Toc12339)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 58](#_Toc30727)

[ДОДАТОК 60](#_Toc685)

# **ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ**

HP - health point, показник здоров’я.

SP - stamina point, показник витривалості.

JSON - JavaScript Object Notation

.TXT - розширення текстового файлу

Лор — це фонова інформація про світ гри, його історію, культуру, традиції та інші деталі.

SFX - звукові єффекти.

MFX - музика та музичні єффекти..

Бектрекінг - повернення на вже вивчені гравцем локації.

Хаб - сленг в ігровій спільноті, який використовують для назви безпечної для гравця локації, де можна спілкуватися, купляти, продавати та займатися іншими речами в безпеці..

Quality of life features - в контексті відеоігор стосується функцій або аспектів ігрового дизайну, які гарантують, що гравці мають плавний ігровий процес і не вигорають так швидко.

Souls-like - це жанр комп'ютерних ігор, що виник як результат популярності серії ігор Dark Souls, розробленої компанією FromSoftware.

ГГ - Головний Герой / Головна Героїня.

# **ВСТУП**

Актуальність теми. Розвиток штучного інтелекту (ШІ) відкриває нові

горизонти в розробці інтерактивних додатків, зокрема в індустрії відеоігор.

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

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

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

# **СТВОРЕННЯ ГРИ**

## **Постановка задачі**

Результатом виконання проєкту має стати програмне забезпечення, яке забезпечує:

Динамічну генерацію сюжетів:

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

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

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

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

Інтеграція сучасних технологій ШІ:

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

Тестування і оптимізація:

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

Розширення можливостей для геймдизайнерів:

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

Вплив результатів:

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

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

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

1.AI Dungeon

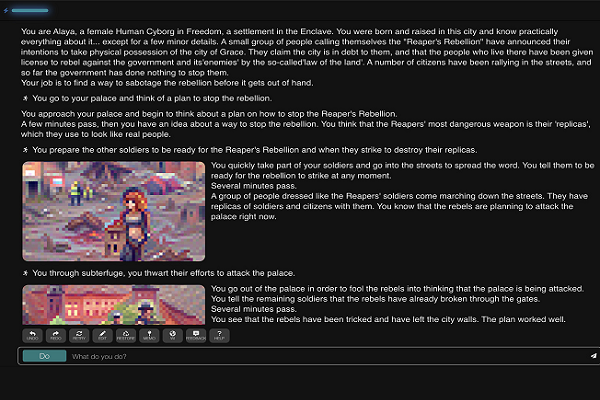


Рис 1.1. AI Dungeon

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

Ключові особливості:

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

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

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

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

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

2.Latitude Voyager

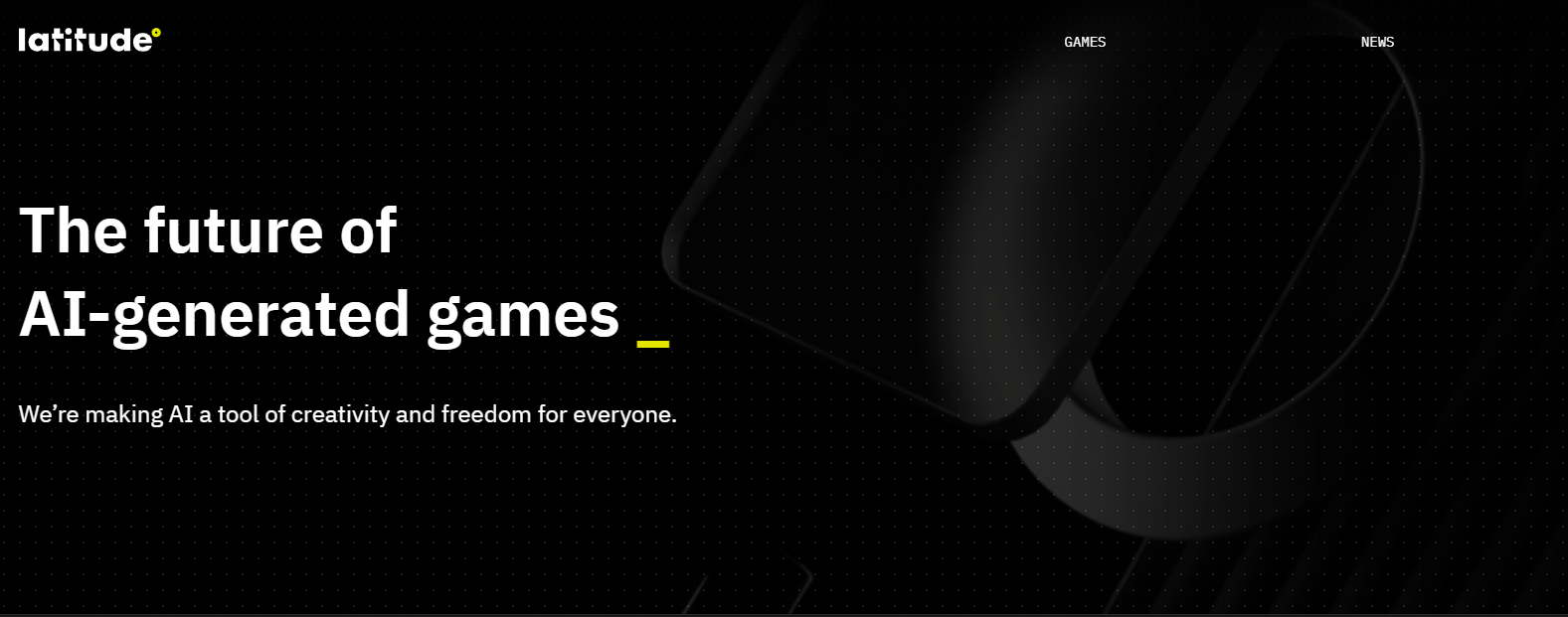


Рис 1.2. Latitude Voyager

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

Ключові особливості:

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

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

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

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

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

3.AI Storytelling у Grammarly

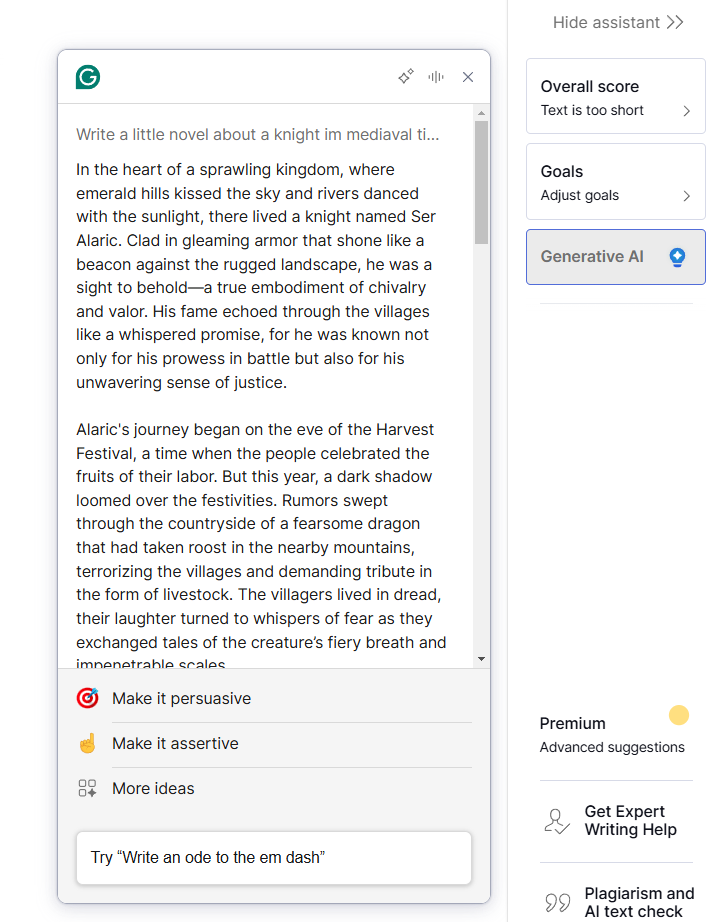


Рис 1.3. AI Storytelling

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

Ключові особливості:

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

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

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

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

4. Jasper

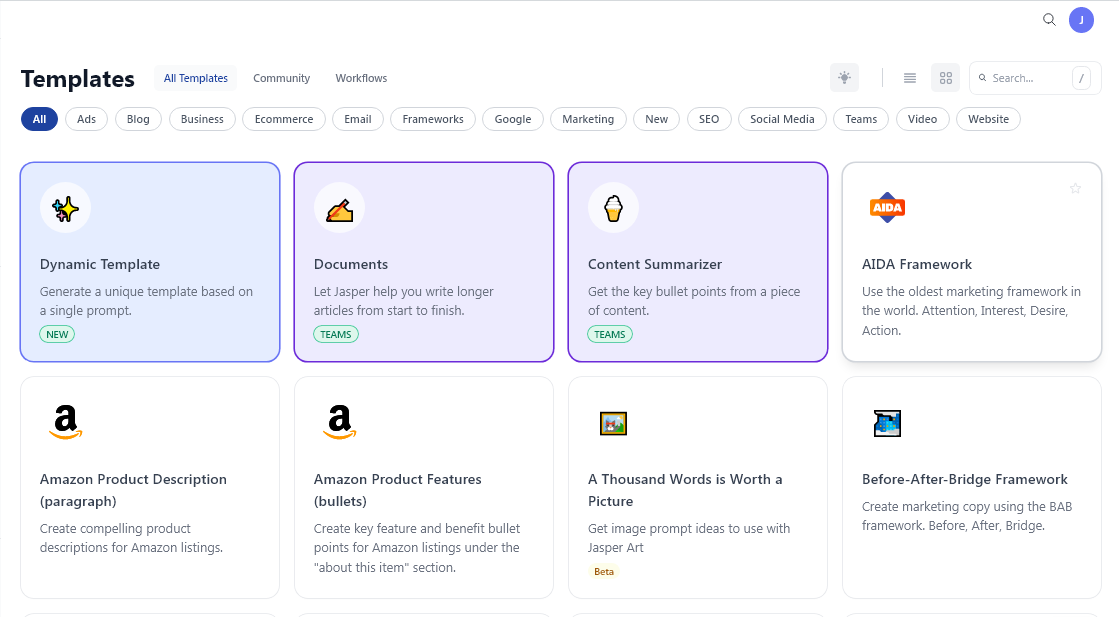


Рис 1.4. Jasper

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

Ключові особливості:

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

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

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

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

5. NovelAI

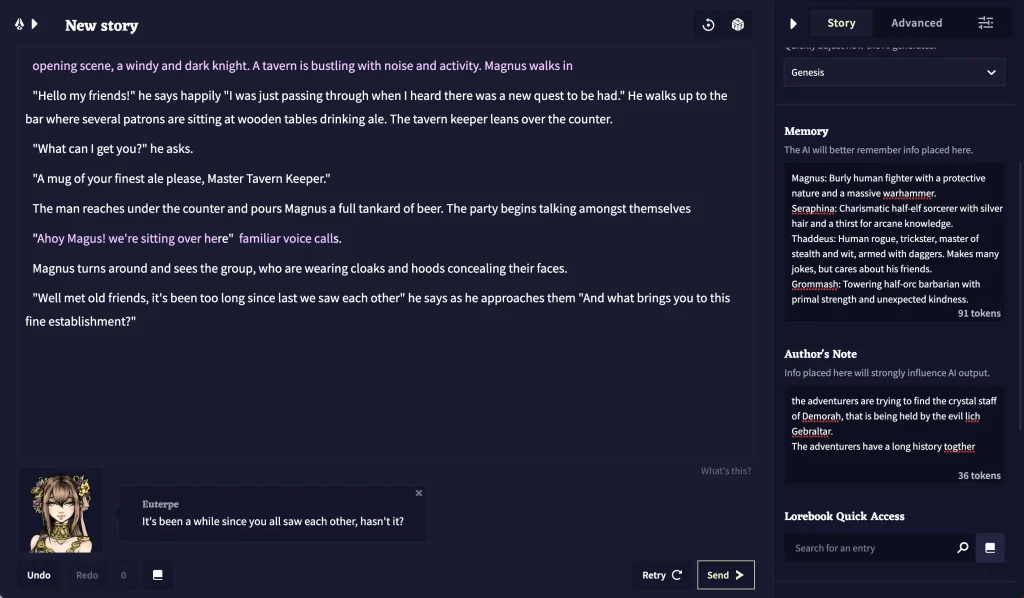


Рис 1.5. NovelAI

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

Ключові особливості:

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

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

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

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

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

## **Обґрунтування вибору інструментальних засобів розробки**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Для забезпечення роботи проекту використовувалися додаткові бібліотеки:

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

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

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

Загальна оцінка

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

## **Постановка задачі та технічне завдання**

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

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

Для досягнення поставлених цілей необхідно реалізувати наступні функції та особливості:

1. Інтерактивність та геймплей

Реалізувати систему вводу текстових команд від користувача.

Забезпечити адаптивну генерацію текстового сюжету на основі введених даних.

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

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

2. Інтеграція з OpenAI API

Використати GPT-моделі для генерації текстового сюжету.

Інтегрувати DALL·E для створення візуальних елементів гри.

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

3. Візуалізація та UI

Розробити інтуїтивно зрозумілий графічний інтерфейс користувача.

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

4. Музичний супровід

Додати функціонал для програвання музичних композицій відповідно до жанру гри.

Реалізувати можливість вибору плейлиста користувачем.

Забезпечити плавне управління гучністю, паузами та перемиканням треків.

5. Збереження прогресу

Реалізувати функцію автозбереження прогресу гри (сюжету та налаштувань).

Надати можливість завантаження попередніх сесій через графічний інтерфейс.

6. Кросплатформність

Забезпечити можливість запуску гри на Windows та Linux.

Перевірити та оптимізувати функціонал для кожної платформи.

Обмеження та вимоги

Зробити гру максимально оптимізованою для роботи на середньопродуктивних пристроях.

Реалізувати систему логування для відстеження помилок та спрощення процесу тестування.

Очікувані результати

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

## **Висновки до першого розділу**

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

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

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

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

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

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

## **2.1 Принцип роботи гри**

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

1. Введення інформації користувачем

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

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

Тематика (вибір певної теми або стилю світу).

Інші параметри, що допомагають створити унікальний контекст для гри.

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

2. Генерація світу на основі введених даних

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

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

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

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

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

3. Текстова взаємодія з користувачем

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

Опис подій, що відбуваються у світі.

Відповіді на дії користувача.

Опис результатів вибору, зробленого користувачем.

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

4. Збереження історії переписки

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

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

Відстеження еволюції сюжету в реальному часі.

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

5. Інтерфейс і відображення історії переписки

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

Повідомлення користувача.

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

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

6. Управління токенами та обрізка історії

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

Підраховує кількість токенів, що використовуються в кожному повідомленні.

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

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

## **2.2 Архітектура гри**

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

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

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

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

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

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

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

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

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

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

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

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

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

## **2.3 Опис процесів у проекті**

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

Основні етапи включають:

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

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

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

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

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

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

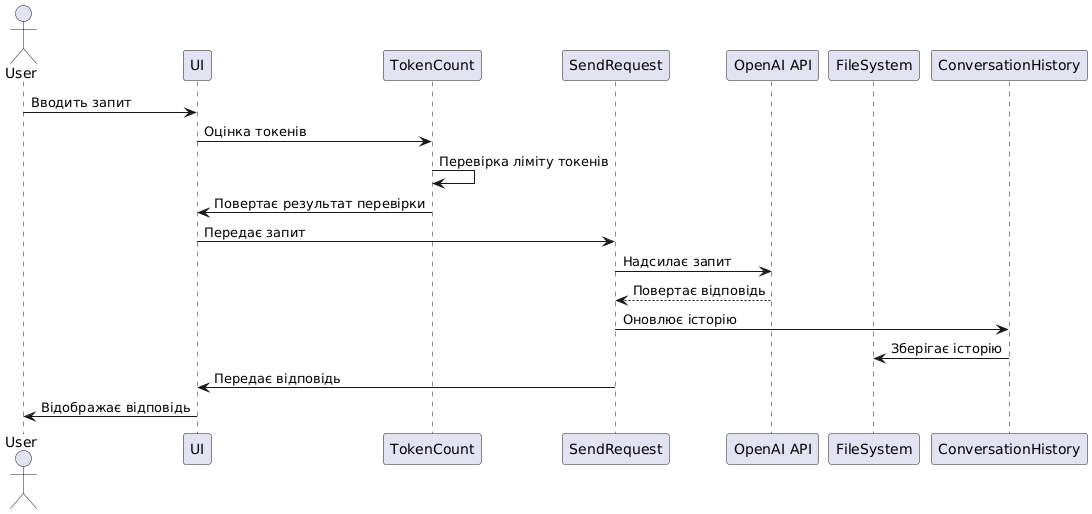


Рис.2.1.діаграмма послідовності

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

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

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

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

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

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

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

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

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

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

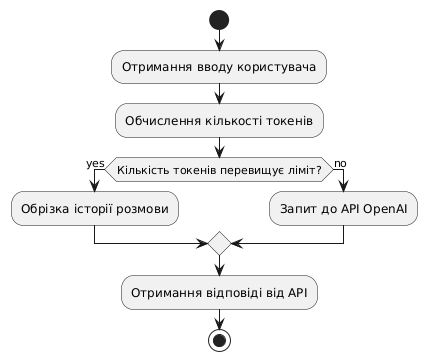


Рис.2.2. Діаграма блок-схеми для токенів

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

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

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

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

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

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

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

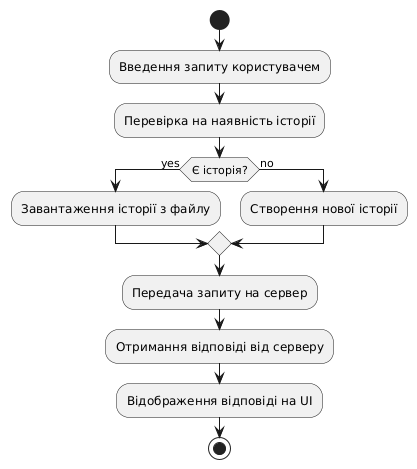


Рис.2.3 Діаграма блок-схеми для взаємодії з користувачем

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

## **Висновки до другого розділу**

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

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

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

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

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

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

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

# **РОЗДІЛ 3. РОЗРОБКА ТА ПРОГРАМУВАННЯ**

## **3.1 Опис загальної структури коду**

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

Основні компоненти проекту включають такі ключові скрипти:

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

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

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

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

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

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

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

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

Пояснення принципів організації коду та архітектури

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

У проекті активно застосовуються такі принципи:

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

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

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

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

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

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

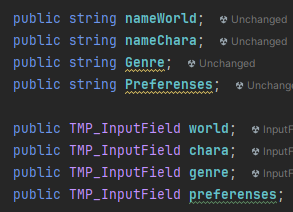
У підрозділі 3.2 буде здійснено детальний опис основних скриптів, які реалізують логіку гри та взаємодію з користувачем. Оскільки гра є текстовою та динамічно генерує віртуальний світ на основі введених користувачем даних, кожен скрипт виконує важливу роль у створенні і збереженні взаємодій. Тут буде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

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

Клас CreateWorld

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

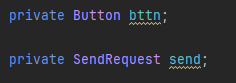
У класі визначені кілька публічних змінних, які використовуються для зберігання значень, введених користувачем:



Лістинг3.1

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

Далі, оголошуються приватні змінні для кнопки та відправки запиту:



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

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

Метод Start()

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



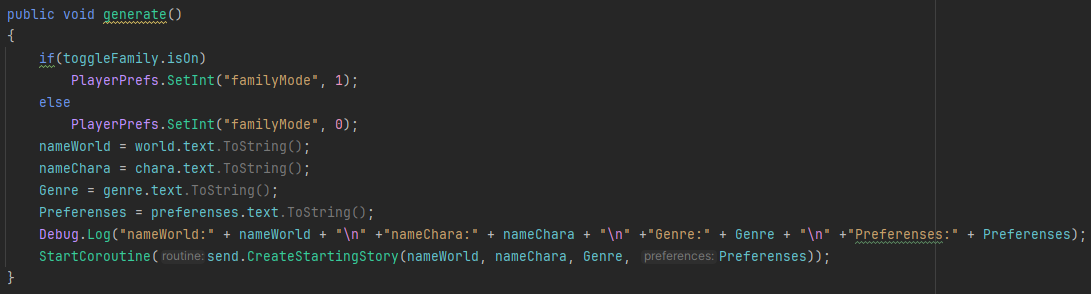
У цьому методі:

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

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

Метод generate()

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



У методі generate() відбувається кілька ключових кроків:

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

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

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

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

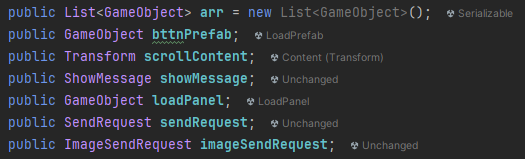
Скрипт CreateWorld є основним інтерфейсним елементом для введення налаштувань світу та персонажа. Він зберігає введену користувачем інформацію, дозволяє вибрати режим сім'ї, а також відправляє дані на сервер для створення історії світу. Весь процес ініціюється через кнопку, а результат надсилається через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

Опис скрипту SaveLoadScript

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

Оголошення змінних

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



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

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

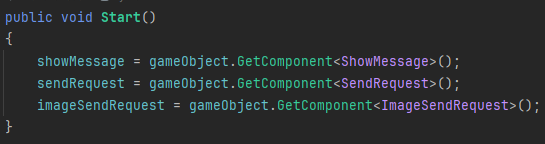
scrollContent — контейнер для кнопок у вигляді скролл-вью, куди будуть додаватись нові кнопки для кожного збереженого світу.

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

loadPanel — панель для завантаження світу.

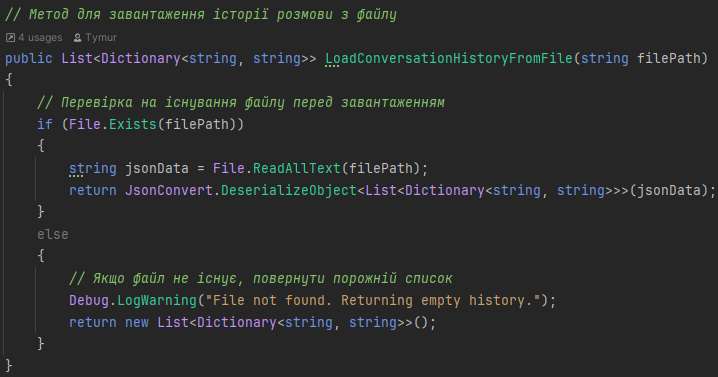
sendRequest та imageSendRequest — компоненти для обробки запитів, пов'язаних з історією і зображеннями.

Метод Start()



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()



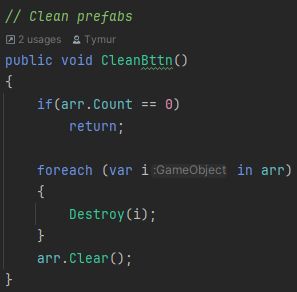
Метод SaveConversationHistoryToFile відповідає за збереження історії розмови в файл у форматі JSON:

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

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

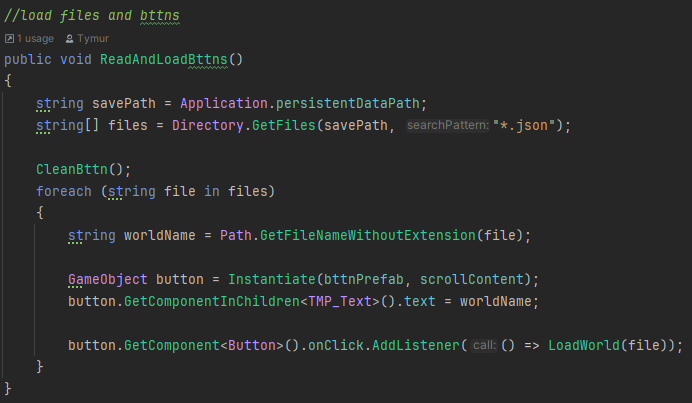
Історія розмови серіалізується у формат JSON за допомогою бібліотеки JsonConvert і записується у файл.

Метод CleanBttn()



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

Метод ReadAndLoadBttns()



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

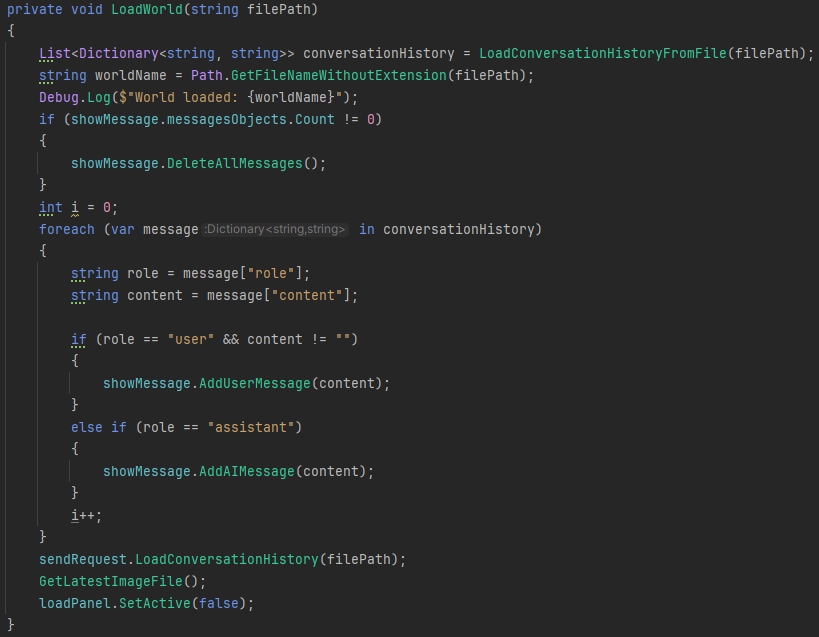
Отримує список всіх файлів .json з директорії збережень.

Очищає поточні кнопки через метод CleanBttn().

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

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

Метод LoadWorld()



Метод LoadWorld() завантажує світ за допомогою збереженого файлу:

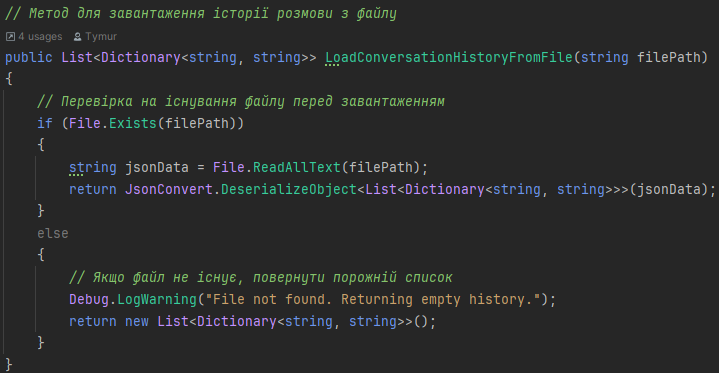
Використовує метод LoadConversationHistoryFromFile(), щоб завантажити історію з файлу.

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

Викликається метод для завантаження зображення, якщо воно є.

Останнім кроком приховує панель завантаження.

Метод LoadConversationHistoryFromFile()



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.

## **3.3 Розробка інших аспектів гри**

Окрім реалізації механік та коду гри, було відповідально за звуковий дизайн, сюжет та вигляд екранів завантаження. Кожен з цих аспектів можна коротко описати:

- Звуковий дизайн: було створено та інтегровано звукові ефекти, музику та атмосферні звуки у гру. Звукові ефекти слугували для підкреслення дій гравця, створення атмосфери навколишнього середовища та особливих подій. Музика додавала настрою та підтримувала емоційну атмосферу геймплею.

Реалізацію звуків та музики не потребує детального опису, оскільки працює завдяки одному коду рядку - AudioSource.Play() (Лістинг.3.17). Завдяки їй ми вмикаємо джерело звуку і воно програється один раз - саме те, що нам і потрібно.

attacksfx.Play();

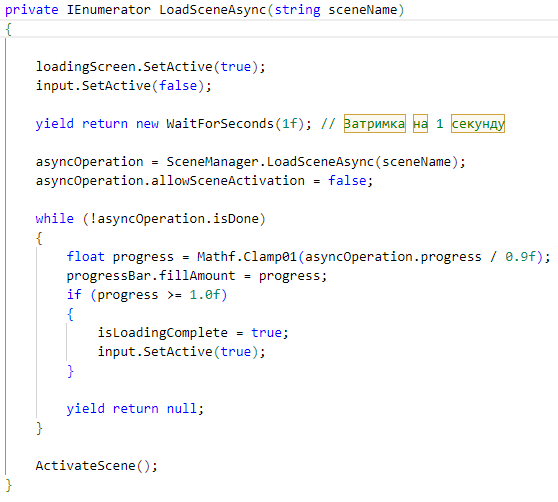
Лістинг.3.17. приклад програвання звуку та музики.

- Сюжет: було розроблено та впроваджено сюжетні елементи, діалоги персонажів та оповідання. Це включало створення захопливої історії, характерів персонажів, ключових моментів сюжету та взаємодію гравця з ними. Метою було створення захоплюючого та занурюючого ігрового досвіду.

- Екрани завантаження: було зайнятося створенням та оформленням екранів завантаження, які з'являлися перед початком гри або при переході між рівнями. Основна мета полягала в створенні привабливого та професійного вигляду, який відповідав загальному стилю гри та привертав увагу гравця.

Ми використовуємо асинхронне завантаження для виконання процесу завантаження. Спочатку ми активуємо екран завантаження, а потім, використовуючи AsyncOperation, ми отримуємо стан завантаження сцени та візуально відображаємо його у вигляді колеса завантаження.

Коли сцена повністю завантажена і готова до відображення, ми повідомляємо гравця і просимо натиснути клавішу Enter для активації цієї сцени (Лістинг.3.18).



Лістинг.3.18. AsyncLoad

Кожен з цих аспектів - звуковий дизайн, сюжет та екрани завантаження - є важливою складовою геймплею та допомагає створити цілісне та захоплююче враження від гри.

## **Висновки до третього розділу**

В цьому підрозділі ми обговорили розробку програмного коду для реалізації різних аспектів гри. Ми розглянули роботу зі скриптами, які керують поведінкою ворогів та камери, а також з UI елементами. Ми також поговорили про механізм збереження даних та реалізацію рівнів.

У цьому розділі ми дізналися, як використовувати різні класи та методи для створення ворожих персонажів, як керувати камерою для плавного слідкування за гравцем, як працювати з UI елементами, включаючи їх вимкнення та активацію, і як зберігати та завантажувати дані гри.

Крім того, ми розглянули роль звукового дизайну, сюжету та екранів завантаження у грі. Звуковий дизайн створює атмосферу та підкреслює дії гравця, сюжет розкриває історію та персонажів, а екрани завантаження додають професійного вигляду грі перед початком геймплею.

Цей розділ дав нам уявлення про те, як розробляти програмний код для різних аспектів гри та як їх взаємодіяти між собою. Детальний розгляд коду та опис функціональності допоможуть нам зрозуміти, як гра працює та які можливості вона надає.

## **ВИСНОВКИ**

Під час роботи над кваліфікаційною роботою було створено і реалізовано ігровий проект у жанрі сайдскролер екшн з елементами платформінгу. Для успішної реалізації цього проекту були виконані послідовні кроки, які були попередньо сплановані.

Початково був проведений аналіз ігрового ринку та конкурентів, що допомогло визначитися з аспектом гри, жанром, механіками та естетикою. Було продумано та створено маршрути для різних рівнів гри. Також були досліджені різні типи інтерфейсу користувача, що допомогло вирішити питання, що виникали при створенні UI для власного проекту.

Далі, була проведена аналітична перевірка функціональності самої гри, переконавшись, що запланований маршрут може бути успішно пройдений гравцем. Було перевірено коректність прокладення маршруту та можливість гравця досягти своєї мети.

Були реалізовані наступні системи:

1. Система переміщення: механіка переміщення відчувається реактивною та енергійною.

2. Система бойових механік: бойова система належним чином реалізована та ставить перед гравцем виклик, вимагаючи прийняття важливих рішень.

3. Система інвентаризації: гравець має можливість збирати предмети на локаціях та використовувати їх для полегшення проходження гри.

4. Система діалогів з персонажами: діалоги виконані у стилі візуальних новел, задовольняючи потреби гравця.

5. Механіки, які не входять в жодну з категорій: інші механіки, спрямовані на різні аспекти гри та зручність, працюють бездоганно.

Цей етап дозволив переконатися, що проект розвивається відповідно до задуму та всі його основні функціональні елементи працюють бездоганно.

## **СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. Steam [Електронний ресурс] – Режим доступу до ресурсу: https://store.steampowered.com/app/283640/Salt\_and\_Sanctuary/
2. Steam [Електронний ресурс] – Режим доступу до ресурсу: [https://store.steampowered.com/app/268910/Cuphead/](https://www.ign.com/)
3. Steam [Електронний ресурс] – Режим доступу до ресурсу: [https://store.steampowered.com/app/367520/Hollow\_Knight/](https://playua.net/)
4. Unity in Action: Multiplatform Game Development in C# / Joe Hocking, 2015. - 53с.
5. Mastering Unity 2D Game Development / Simon Jackson, 2014. – 121c.
6. Unity Game Development Cookbook / Michael G. Flores, 2019. – 402с.
7. Learning C# by Developing Games with Unity / Harrison Ferrone, 2022. – 99с.
8. Unity Game Optimization / Chris Dickinson, 2019. – 188с.
9. Unity in Action: Multiplatform Game Development in C# / Joe Hocking, 2019. – 98с.
10. Unity 2018 Cookbook / Matt Smith, 2018. – 10с.
11. Документація Unity [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com
12. Документація Unity Libraries [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/Manual/>
13. Документація Unity for Beginers [Електронний ресурс] – Режим доступу до ресурсу: <https://unity.com/how-to/beginner/5-unity-tutorials-new-game-developers>
14. Unity Tutorials [Електронний ресурс] - Режим доступу до ресурсу: <https://www.tutorialspoint.com/unity/index.htm>
15. ItProger [Електронний ресурс] - Режим доступу до ресурсу: <https://itproger.com/ua/course/unity-csharp>
16. Stack Overflow [Електронний ресурс] - Режим доступу до ресурсу: <https://stackoverflow.com/>
17. DotNetPearls [Електронний ресурс] - Режим доступу до ресурсу: <https://www.dotnetperls.com/between-before-after>
18. C# Documentation [Електронний ресурс] - Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/csharp/
19. Souls-Like Games [Електронний ресурс] - Режим доступу до ресурсу: <https://en.wikipedia.org/wiki/Soulslike>
20. Unity discussions [Електронний ресурс] - Режим доступу до ресурсу: <https://discussions.unity.com/>

# 

# **ДОДАТОК**

Додаток А PlayerClass.cs

using System;

using System.Collections;

using System.Collections.Generic;

using Unity.Mathematics;

using UnityEditorInternal;

using UnityEngine;

using UnityEngine.SceneManagement;

public class PlayerClass : MonoBehaviour

{

#region PlayerChara

public float hpMax = 100f;

public float spMax = 50f;

public int piecesToGrade = 10;

public int pieces = 0;

public int level = 1;

#endregion

#region FightBehaviour

public float damage = 10;

#endregion

#region Movement

public float speed = 8f;

public float jumpPower = 16f;

public float spintMult = 1.5f;

#endregion

#region Inventory

public int flaskMax = 2;

public List<int> inv;

public List<int> amount;

#endregion

#region Progress

public bool lvl\_0\_open = true;

public bool lvl\_hub\_open = true;

public bool lvl\_1\_open = false;

public bool lvl\_2\_open = false;

public bool lvl\_0\_completed = false;

public bool lvl\_1\_completed = false;

public bool lvl\_2\_completed = false;

//smith

public bool que\_01\_00\_smith\_open= true;

public bool que\_01\_00\_smith\_completed = false;

public bool que\_01\_01\_smith\_open = false;

public bool que\_01\_01\_smith\_completed = false;

//reaper

public bool que\_01\_00\_reaper\_open = true;

public bool que\_01\_00\_reaper\_completed = false;

public bool que\_01\_01\_reaper\_open = false;

public bool que\_01\_01\_reaper\_completed = false;

//alchemist

public bool que\_01\_00\_alchemist\_open = true;

public bool que\_01\_00\_alchemist\_completed = false;

public bool que\_01\_01\_alchemist\_open = false;

public bool que\_01\_01\_alchemist\_completed = false;

//lvl\_0\_items

public bool lvl\_0\_shard\_picked = false;

public bool lvl\_0\_flower\_picked = false;

public bool lvl\_0\_scroll\_picked = false;

//lvl\_1\_items

public bool lvl\_1\_shard\_picked = false;

public bool lvl\_1\_flower\_picked = false;

public bool lvl\_1\_scroll\_picked = false;

#endregion

public string currScene;

public Vector3 respPos;

public PlayerClass(PlayerClass data)

{

hpMax = data.hpMax;

spMax = data.spMax;

level = data.level;

damage = data.damage;

piecesToGrade = data.piecesToGrade;

pieces = data.pieces;

currScene = data.updateScene();

flaskMax = data.flaskMax;

inv = data.inv;

amount = data.amount;

lvl\_0\_open = data.lvl\_0\_open;

lvl\_hub\_open = data.lvl\_hub\_open;

lvl\_1\_open = data.lvl\_1\_open;

lvl\_2\_open = data.lvl\_2\_open;

lvl\_0\_completed = data.lvl\_0\_completed;

lvl\_1\_completed = data.lvl\_1\_completed;

lvl\_2\_completed = data.lvl\_2\_completed;

que\_01\_00\_smith\_open = data.que\_01\_00\_smith\_open;

que\_01\_00\_smith\_completed = data.que\_01\_00\_smith\_completed;

que\_01\_01\_smith\_open = data.que\_01\_01\_smith\_open;

que\_01\_01\_smith\_completed = data.que\_01\_01\_smith\_completed;

que\_01\_00\_reaper\_open = data.que\_01\_00\_reaper\_open;

que\_01\_00\_reaper\_completed = data.que\_01\_00\_reaper\_completed;

que\_01\_01\_reaper\_open = data.que\_01\_01\_reaper\_open;

que\_01\_01\_reaper\_completed = data.que\_01\_01\_reaper\_completed;

que\_01\_00\_alchemist\_open = data.que\_01\_00\_alchemist\_open;

que\_01\_00\_alchemist\_completed = data.que\_01\_00\_alchemist\_completed;

que\_01\_01\_alchemist\_open = data.que\_01\_01\_alchemist\_open;

que\_01\_01\_alchemist\_completed = data.que\_01\_01\_alchemist\_completed;

lvl\_0\_shard\_picked = data.lvl\_0\_shard\_picked;

lvl\_0\_flower\_picked = data.lvl\_0\_flower\_picked;

lvl\_0\_scroll\_picked = data.lvl\_0\_scroll\_picked;

lvl\_1\_shard\_picked = data.lvl\_1\_shard\_picked;

lvl\_1\_flower\_picked = data.lvl\_1\_flower\_picked;

lvl\_1\_scroll\_picked = data.lvl\_1\_scroll\_picked;

}

public string updateScene()

{

currScene = SceneManager.GetActiveScene().name;

return currScene;

}

public void GetPos()

{

var tmpPlayer = GameObject.FindWithTag("Player");

respPos = new Vector3(transform.position.x,transform.position.y,transform.position.z);

}

}

Додаток Б PlayerMovement.cs

using System;

using System.Collections;

using System.Collections.Generic;

using UnityEditor;

using UnityEngine;

public class PlayerMovement : MonoBehaviour

{

public Animator animator;

private SpriteRenderer sprite;

#region Variables

private float horizontal;

public float faceDir;

[SerializeField] public Rigidbody2D rb;

[SerializeField] private GameObject groundCheck;

[SerializeField] private GameObject wallCheck;

[SerializeField] private LayerMask groundLayer;

[SerializeField] private float dashingVelocity;

[SerializeField] private float dashingTime;

public Vector2 dashingDir;

public float dashForce = 5f;

public float dashCost = 25f;

public float dashCD = 1f;

public float dashCDcurr = 0f;

private float lastDodgeTime = -1f;

public bool isDashing;

private bool \_canDash = true;

#endregion

public CapsuleCollider2D coll;

public GameObject player;

[SerializeField] private PlayerChara stats;

[SerializeField] private FightBehaviour fight;

public PlayerClass playerClass;

public GameObject interestPoint;

public AudioSource dashSFX;

public AudioSource jumpSFX;

private void Start()

{

coll = GetComponent<CapsuleCollider2D>();

player = GameObject.FindWithTag("Player");

playerClass = GetComponent<PlayerClass>();

sprite = GetComponent<SpriteRenderer>();

animator = GetComponent<Animator>();

rb = GetComponent<Rigidbody2D>();

stats = GetComponent<PlayerChara>();

fight = GetComponent<FightBehaviour>();

groundCheck = GameObject.Find("groundCheck");

}

void Update()

{

if (stats.isDead || stats.isStunned || stats.inDialogue || fight.isParry || fight.isAttacking )

{

rb.velocity = Vector2.zero;

return;

}

if (GameObject.FindWithTag("MainCamera").GetComponent<CameraFollow>().inPause)

{

return;

}

if (isDashing && AtWall())

{

rb.velocity = Vector2.zero;

return;

}

if(isDashing)

return;

var inputX = Input.GetAxisRaw("Horizontal");

var jumpInput = Input.GetKeyDown(KeyCode.W);

var dodgeInput = Input.GetKeyDown(KeyCode.Space);

if (fight.isBlocking)

{

rb.velocity = Vector2.zero;

animator.SetFloat("Speed", 0);

return;

}

animator.SetFloat("Speed", Math.Abs(inputX));

if (inputX > 0)

{

player.transform.localScale = new Vector3(6,6,1);

}

else if (inputX < 0)

{

player.transform.localScale = new Vector3(-6,6,1);

}

if (!fight.isBlocking)

{

rb.velocity = new Vector2(inputX \* playerClass.speed, rb.velocity.y);

if (jumpInput && isGrounded() && stats.canJump())

{

jumpSFX.Play();

rb.velocity = new Vector2(rb.velocity.x, playerClass.jumpPower);

stats.spCurrent -= stats.jumpCost;

}

if (dodgeInput && CanDodge())

{

Dodge();

}

if (isGrounded())

{

\_canDash = true;

animator.SetBool("isJump", false);

}

else

{

animator.SetBool("isJump", true);

}

if (!fight.isFighting)

{

if (inputX == 1f || inputX == -1f)

{

faceDir = inputX;

}

}

}

}

private bool CanDodge()

{

return Time.time - lastDodgeTime > dashCD && stats.spCurrent >= dashCost && rb.velocity != Vector2.zero && !isDashing;

}

private void Dodge()

{

dashSFX.Play();

lastDodgeTime = Time.time;

stats.spCurrent -= dashCost;

isDashing = true;

coll.isTrigger = true;

rb.constraints = RigidbodyConstraints2D.FreezePositionY;

rb.AddForce(FaceDirection() \* dashForce, ForceMode2D.Impulse);

animator.SetBool("isDash", true);

StartCoroutine(StopDashing());

}

private IEnumerator StopDashing()

{

yield return new WaitForSeconds(dashingTime);

isDashing = false;

animator.SetBool("isDash", false);

coll.isTrigger = false;

rb.constraints &= ~RigidbodyConstraints2D.FreezePositionY;

rb.constraints = RigidbodyConstraints2D.FreezeRotation;

}

public bool isGrounded()

{

return Physics2D.OverlapCircle(groundCheck.transform.position, 0.5f,groundLayer);

}

public bool AtWall()

{

return Physics2D.OverlapCircle(wallCheck.transform.position, 0.1f, groundLayer);

}

public Vector2 FaceDirection()

{

if (faceDir == 1)

{

return Vector2.right;

}

else

{

return Vector2.left;

}

}

}

Додаток В FightBehavior.cs

using System;

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class FightBehaviour : MonoBehaviour

{

[SerializeField] private PlayerMovement move;

public PlayerClass playerClass;

private PlayerChara chara;

private LayerMask enemyMask;

private float rayDistanse = 5f;

public bool isFighting = false;

public bool isAttacking = false;

public bool isBlocking = false;

public bool isParry = false;

public bool isRecentlyHit = false;

private float unHitTime = 3f;

private float startUnHitTime = 0f;

public GameObject cam;

private float timeBtwAttack;

private float startTimeBtwAttack = 0.5f;

private int attackCount = 0;

private int attackCountMax = 3;

private float attackDelayTime = 1f;

private float parryTime = 0.5f;

private GameObject attackPos;

private bool parryCan = true;

public float attackRange = 2f;

//public float damage = 10f;

private bool isBlockClicked = false;

private bool canAttack = true;

private Animator animCont;

public AudioSource attacksfx;

void Start()

{

playerClass = GetComponent<PlayerClass>();

animCont = GetComponent<Animator>();

chara = GetComponent<PlayerChara>();

move = GetComponent<PlayerMovement>();

enemyMask = LayerMask.GetMask("Enemy");

attackPos = GameObject.Find("attackPos");

cam = GameObject.Find("Main Camera");

}

public bool canBlockCheck()

{

if (!isAttacking)

{

return true;

}

else

{

return false;

}

}

public IEnumerator attack()

{

attacksfx.Play();

animCont.SetTrigger("Attack");

isAttacking = true;

canAttack = false;

yield return new WaitForSeconds(0.25f);

timeBtwAttack = startTimeBtwAttack;

Collider2D[] enemyToDamage = Physics2D.OverlapCircleAll(attackPos.transform.position, attackRange, enemyMask);

for (int i = 0; i < enemyToDamage.Length; i++)

{

enemyToDamage[i].GetComponent<EnemyClass>().TakeDamage(playerClass.damage);

}

isAttacking = false;

StartCoroutine(AttackCD());

}

public IEnumerator CanParry()

{

if (!isAttacking && !isBlocking)

{

animCont.SetTrigger("Parry");

isParry = true;

parryCan = false;

yield return new WaitForSeconds(0.2f);

isParry = false;

StartCoroutine(ParryCD());

}

}

private IEnumerator AttackCD()

{

yield return new WaitForSeconds(startTimeBtwAttack);

canAttack = true;

}

public IEnumerator ParryCD()

{

yield return new WaitForSeconds(parryTime);

parryCan = true;

}

void Update()

{

if (chara.isDead || chara.isStunned || chara.inDialogue || cam.GetComponent<CameraFollow>().inPause)

return;

if (Input.GetKeyDown(KeyCode.Mouse0) && canAttack && move.rb.velocity == Vector2.zero)

{

StartCoroutine(attack());

}

else if (Input.GetKey(KeyCode.Mouse1))

{

if (canBlockCheck())

{

animCont.SetBool("Block", true);

isBlocking = true;

StartCoroutine(CanParry());

}

}

else if (Input.GetMouseButtonDown(2) && parryCan && move.rb.velocity == Vector2.zero)

{

StartCoroutine(CanParry());

}

else

{

animCont.SetBool("Block", false);

isBlocking = false;

}

if (isRecentlyHit)

{

startUnHitTime += Time.deltaTime;

if (startUnHitTime >= unHitTime)

{

startUnHitTime = 0;

isRecentlyHit = false;

}

}

}

}

Додаток Г PlayerChara.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PlayerChara : MonoBehaviour

{

#region BasicStats

public float hpCurrent = 100f;

public float spCurrent = 50f;

private bool spRegenerating = false;

private bool enduranceRegenerating = false;

public bool isStunned = false;

private float stunTime = 1f;

public GameObject RespawnPoint;

private LevelCreate levelCreateSRC;

private FightBehaviour fight;

public bool isDead = false;

public bool deathCourotine = false;

public bool inDialogue = false;

#endregion

#region MoveCost

[SerializeField] public float jumpCost = 10f;

[SerializeField] public float dashCost = 25f;

#endregion

public PlayerClass player;

public bool deacrese = false;

public Camera cam;

public CameraFollow camSRC;

private Animator animCont;

public AudioSource blocksfx;

public AudioSource parrysfx;

public AudioSource hitSFX;

public bool takeDmg(float dmg, int type)

{

if(GameObject.FindWithTag("Player").GetComponent<PlayerMovement>().isDashing && (type == 0 || type == 1)) // if dodging && onlyDodge attack or default attack

return true;

if (GameObject.FindWithTag("Player").GetComponent<PlayerMovement>().isDashing && type == 2) // if dodging && onlyParry attack

{

hpCurrent -= dmg;

return false;

}

if (fight.isBlocking && type == 0 && !fight.isParry) // if block default attack

{

spCurrent -= dmg;

blocksfx.Play();

return false;

}

if (fight.isParry && (type == 0 || type == 2)) // if parry default or parryOnly attack

{

parrysfx.Play();

StopCoroutine(fight.CanParry());

return true;

}

animCont.SetTrigger("Hurt"); // others

hpCurrent -= dmg;

hitSFX.Play();

return false;

}

public void getPieces(float amount)

{

var target = player.pieces + amount;

player.pieces += (int)amount;

}

public bool canJump()

{

if (spCurrent >= jumpCost)

return true;

else

return false;

}

public bool canDash()

{

if (spCurrent >= dashCost)

return true;

else

return false;

}

void Start()

{

player = GetComponent<PlayerClass>();

animCont = GetComponent<Animator>();

cam = Camera.main;

camSRC = cam.GetComponent<CameraFollow>();

levelCreateSRC = GameObject.FindWithTag("lvlScr").GetComponent<LevelCreate>();

RespawnPoint = GameObject.FindWithTag("playerPos");

fight = GetComponent<FightBehaviour>();

assignStats();

}

public void RespawnPointAssign(GameObject pos)

{

RespawnPoint = pos;

}

public void assignStats()

{

hpCurrent = player.hpMax;

spCurrent = player.spMax;

}

private IEnumerator RegenSP()

{

spRegenerating = true;

yield return new WaitForSeconds(3f);

while (true)

{

if (spCurrent < player.spMax)

{

spCurrent += 4;

yield return new WaitForSeconds(0.3f);

}

else

{

spRegenerating = false;

yield return null;

break;

}

}

}

public IEnumerator Death()

{

deathCourotine = true;

isDead = true;

animCont.SetBool("isDead", true);

cam.GetComponent<CameraFollow>().YouDied.SetActive(true);

camSRC.BlackScreenTransparency(1);

animCont.SetBool("isDead", false);

yield return new WaitForSeconds(3f);

assignStats();

player.pieces /= 2;

gameObject.transform.position = RespawnPoint.transform.position;

cam.GetComponent<CameraFollow>().YouDied.SetActive(false);

levelCreateSRC.GetComponent<LevelCreate>().forEachEnemyFindDestroy();

levelCreateSRC.GetComponent<LevelCreate>().foreachCycle(levelCreateSRC.GetComponent<LevelCreate>().Enemy, levelCreateSRC.GetComponent<LevelCreate>().EnemyPos);

isDead = false;

deathCourotine = false;

camSRC.BlackScreenTransparency(0);

}

private IEnumerator Stun()

{

isStunned = true;

yield return new WaitForSeconds(stunTime);

isStunned = false;

}

void Update()

{

if (hpCurrent <= 0 && !deathCourotine)

{

StartCoroutine(Death());

}

if (hpCurrent > player.hpMax)

{

hpCurrent = player.hpMax;

}

if (spCurrent < 0)

spCurrent = 0;

if (spCurrent < player.spMax && !spRegenerating)

{

StartCoroutine(RegenSP());

}

if (spCurrent > player.spMax)

spCurrent = player.spMax;

}

}

Додаток Д PlayerInventory.cs

using System.Collections;

using System.Collections.Generic;

using TMPro;

using UnityEngine;

using UnityEngine.UIElements;

using Image = UnityEngine.UI.Image;

public class PlayerInventory : MonoBehaviour

{

private PlayerChara chara;

private PlayerMovement move;

private FightBehaviour fight;

private PlayerClass playerClass;

private GameObject itemBar;

private Transform posItemUse;

public int quickCurrent = 0;

private GameObject knife;

private GameObject bomb;

public TextAsset itemsList;

public TextAsset itemsSprite;

private string itemsListString;

public AudioSource bombsfx;

public AudioSource knifesfx;

public AudioSource potionsfx;

void Start()

{

playerClass = GetComponent<PlayerClass>();

knife = Resources.Load("Knife") as GameObject;

bomb = Resources.Load("Bomb") as GameObject;

posItemUse = gameObject.transform.GetChild(1).transform;

itemBar = GameObject.Find("itemBar");

chara = GetComponent<PlayerChara>();

move = GetComponent<PlayerMovement>();

fight = GetComponent<FightBehaviour>();

itemsListString = itemsList.text;

if (!playerClass.inv.Contains(2))

{

playerClass.inv.Add(2);

playerClass.amount.Add(playerClass.flaskMax);

}

ChangeSprite();

}

public void TakeItem(int id)

{

if (playerClass.inv.Contains(id))

{

int i = playerClass.inv.IndexOf(id);

playerClass.amount[i]++;

}else

{

playerClass.inv.Add(id);

playerClass.amount.Add(1);

}

}

public void UseItem(int index)

{

var tmpID = playerClass.inv.IndexOf(index);

if (playerClass.inv[index] == 2 && playerClass.amount[index] > 0)

{

chara.hpCurrent += 25;

potionsfx.Play();

playerClass.amount[index]--;

}

else if (playerClass.inv[index] == 3 && playerClass.amount[index] > 0)

{

var tmppos = new Vector2(posItemUse.position.x, posItemUse.position.y);

GameObject knife\_throw = Instantiate(knife, tmppos, Quaternion.identity);

knifesfx.Play();

playerClass.amount[index]--;

}

else if (playerClass.inv[index] == 4 && playerClass.amount[index] > 0)

{

var tmppos = new Vector2(posItemUse.position.x, posItemUse.position.y);

GameObject bomb\_throw = Instantiate(bomb, tmppos, Quaternion.identity);

playerClass.amount[index]--;

bombsfx.Play();

}

ChangeSprite();

}

public void ResetFlask()

{

playerClass.amount[playerClass.inv.IndexOf(2)] = playerClass.flaskMax;

}

private void ChangeItem(int n)

{

if (quickCurrent == playerClass.inv.Count-1 && n == 1)

{

quickCurrent = 0;

return;

}

else if (quickCurrent == 0 && n == -1)

{

quickCurrent = playerClass.inv.Count-1;

return;

}

quickCurrent += n;

}

void Update()

{

if (GameObject.FindWithTag("MainCamera").GetComponent<CameraFollow>().inPause)

{

return;

}

if (Input.GetKeyDown(KeyCode.I))

{

Debug.Log(playerClass.inv);

}

else if (Input.GetKeyDown(KeyCode.Q) && !fight.isAttacking && !fight.isBlocking)

{

UseItem(quickCurrent);

}

else if (Input.GetAxis("Mouse ScrollWheel") > 0f ) // forward

{

ChangeItem(1);

ChangeSprite();

}

else if (Input.GetAxis("Mouse ScrollWheel") < 0f ) // backwards

{

ChangeItem(-1);

ChangeSprite();

}

}

private void ChangeSprite()

{

var tmp = playerClass.inv[quickCurrent];

var tmp2 = playerClass.inv.IndexOf(tmp);

var text = itemsSprite.text;

var spr\_path = getBetween(text,playerClass.inv[quickCurrent].ToString() + ":" , "\r");

Sprite itmSpr = LoadSpriteFromFile(spr\_path);

itemBar.transform.GetChild(2).GetComponent<TextMeshProUGUI>().text = playerClass.amount[tmp2].ToString();

itemBar.transform.GetChild(1).GetComponent<Image>().sprite = itmSpr;

}

private static string getBetween(string file, string strStart, string strEnd)

{

if (file.Contains(strStart) && file.Contains(strEnd))

{

int Start, End;

Start = file.IndexOf(strStart, 0) + strStart.Length;

End = file.IndexOf(strEnd, Start);

return file.Substring(Start, End - Start);

}

return "";

}

private Sprite LoadSpriteFromFile(string path)

{

// Load the texture from the file

Texture2D texture = LoadTextureFromFile(path);

// Create a sprite from the texture

Sprite sprite = Sprite.Create(texture, new Rect(0, 0, texture.width, texture.height), Vector2.zero);

return sprite;

}

private Texture2D LoadTextureFromFile(string path)

{

// Load the image data from the file

byte[] imageData = System.IO.File.ReadAllBytes(path);

// Create a new texture and load the image data into it

Texture2D texture = new Texture2D(2, 2);

texture.LoadImage(imageData);

return texture;

}

}

Додаток В FightBehavior.cs

using System;

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class FightBehaviour : MonoBehaviour

{

[SerializeField] private PlayerMovement move;

public PlayerClass playerClass;

private PlayerChara chara;

private LayerMask enemyMask;

private float rayDistanse = 5f;

public bool isFighting = false;

public bool isAttacking = false;

public bool isBlocking = false;

public bool isParry = false;

public bool isRecentlyHit = false;

private float unHitTime = 3f;

private float startUnHitTime = 0f;

public GameObject cam;

private float timeBtwAttack;

private float startTimeBtwAttack = 0.5f;

private int attackCount = 0;

private int attackCountMax = 3;

private float attackDelayTime = 1f;

private float parryTime = 0.5f;

private GameObject attackPos;

private bool parryCan = true;

public float attackRange = 2f;

//public float damage = 10f;

private bool isBlockClicked = false;

private bool canAttack = true;

private Animator animCont;

public AudioSource attacksfx;

void Start()

{

playerClass = GetComponent<PlayerClass>();

animCont = GetComponent<Animator>();

chara = GetComponent<PlayerChara>();

move = GetComponent<PlayerMovement>();

enemyMask = LayerMask.GetMask("Enemy");

attackPos = GameObject.Find("attackPos");

cam = GameObject.Find("Main Camera");

}

public bool canBlockCheck()

{

if (!isAttacking)

{

return true;

}

else

{

return false;

}

}

public IEnumerator attack()

{

attacksfx.Play();

animCont.SetTrigger("Attack");

isAttacking = true;

canAttack = false;

yield return new WaitForSeconds(0.25f);

timeBtwAttack = startTimeBtwAttack;

Collider2D[] enemyToDamage = Physics2D.OverlapCircleAll(attackPos.transform.position, attackRange, enemyMask);

for (int i = 0; i < enemyToDamage.Length; i++)

{

enemyToDamage[i].GetComponent<EnemyClass>().TakeDamage(playerClass.damage);

}

isAttacking = false;

StartCoroutine(AttackCD());

}

public IEnumerator CanParry()

{

if (!isAttacking && !isBlocking)

{

animCont.SetTrigger("Parry");

isParry = true;

parryCan = false;

yield return new WaitForSeconds(0.2f);

isParry = false;

StartCoroutine(ParryCD());

}

}

private IEnumerator AttackCD()

{

yield return new WaitForSeconds(startTimeBtwAttack);

canAttack = true;

}

public IEnumerator ParryCD()

{

yield return new WaitForSeconds(parryTime);

parryCan = true;

}

void Update()

{

if (chara.isDead || chara.isStunned || chara.inDialogue || cam.GetComponent<CameraFollow>().inPause)

return;

if (Input.GetKeyDown(KeyCode.Mouse0) && canAttack && move.rb.velocity == Vector2.zero)

{

StartCoroutine(attack());

}

else if (Input.GetKey(KeyCode.Mouse1))

{

if (canBlockCheck())

{

animCont.SetBool("Block", true);

isBlocking = true;

StartCoroutine(CanParry());

}

}

else if (Input.GetMouseButtonDown(2) && parryCan && move.rb.velocity == Vector2.zero)

{

StartCoroutine(CanParry());

}

else

{

animCont.SetBool("Block", false);

isBlocking = false;

}

if (isRecentlyHit)

{

startUnHitTime += Time.deltaTime;

if (startUnHitTime >= unHitTime)

{

startUnHitTime = 0;

isRecentlyHit = false;

}

}

}

}

Додаток Ж EnemyClass.cs

using System.Collections;

using System.Collections.Generic;

using System.Runtime.CompilerServices;

using UnityEditorInternal.Profiling.Memory.Experimental;

using UnityEngine;

using Random = System.Random;

public class EnemyClass : MonoBehaviour

{

#region stats

public float hpMax;

public float hpCurrent;

public float speed;

public bool isDead;

public float pieces;

#endregion

#region Attacks

public float dmg;

public float range;

public float attackCd;

public float attackDelay;

public bool isStunned;

public bool canOnlyDodge;

public bool canOnlyParry;

#endregion

#region Look

public float distanceToSee;

public float distanceToAttack;

public float distanceToPlayer;

#endregion

#region Others

Random rng = new Random();

public bool isAttacking = false;

public bool canAttack = true;

#endregion

#region References

public GameObject playerObj;

public GameObject itemDrop;

public Rigidbody2D rb;

public Animator animController;

public EnemyBehavior enemyBehavior;

public GameObject attackPoss;

public GameObject indicatorDodge;

public GameObject indicatorParry;

public LayerMask playerLayer;

#endregion

#region Methods

public IEnumerator AttackCd()

{

canAttack = false;

yield return new WaitForSeconds(attackCd);

canAttack = true;

}

public bool SeePlayer()

{

if (distanceToPlayer <= distanceToSee)

return true;

return false;

}

public bool CanAttack()

{

if (distanceToPlayer <= distanceToAttack && canAttack)

return true;

return false;

}

public void Death()

{

StopAllCoroutines();

rb.velocity = Vector2.zero;

enemyBehavior.enabled = false;

GetComponent<CapsuleCollider2D>().isTrigger = true;

rb.isKinematic = true;

isDead = true;

animController.SetTrigger("isDead");

int dice = rng.Next(1,100);

if (dice % 2 == 0)

{

var pos = new Vector3(transform.position.x, transform.position.y, transform.position.z);

GameObject pickUp = Instantiate(itemDrop, pos, Quaternion.identity);

dice = rng.Next(1, 100);

if (dice % 2 == 0) // knifeID == 3

{

pickUp.GetComponent<ItemPickUp>().id = 3;

}

else // bombID == 4

{

pickUp.GetComponent<ItemPickUp>().id = 4;

}

}

playerObj.GetComponent<PlayerChara>().getPieces(pieces);

this.enabled = false;

}

private IEnumerator StunCd()

{

isStunned = true;

animController.SetTrigger("isStunned");

yield return new WaitForSeconds(2f);

isStunned = false;

}

public void TakeDamage(float damage)

{

if (gameObject != null)

{

if (!isAttacking)

{

animController.SetTrigger("getHit");

}

if (isStunned) //hit after stun

{

hpCurrent -= damage \* 2f;

StopCoroutine(StunCd());

isStunned = false;

return;

}

hpCurrent -= damage; //normal hit

}

}

public bool InRangeAttack()

{

if (distanceToPlayer <= range)

{

return true;

}

return false;

}

public IEnumerator Attack()

{

// 0 - default attack

// 1 - only dodge

// 2 - only parry

isAttacking = true;

var dice = rng.Next(0, 100);

if (canOnlyDodge && 0 <= dice && dice <= 10) // onlyDodge

{

animController.SetTrigger("attackSpecial");

// Indicator HERE

indicatorDodge.SetActive(true);

var tmpColor = indicatorDodge.GetComponent<SpriteRenderer>().color;

tmpColor.a = 100f;

indicatorDodge.GetComponent<SpriteRenderer>().color = tmpColor;

yield return new WaitForSeconds(attackDelay - 0.2f);

// Indicator FLASH

tmpColor.a = 255f;

indicatorDodge.GetComponent<SpriteRenderer>().color = tmpColor;

yield return new WaitForSeconds(0.2f);

Collider2D[] colls = Physics2D.OverlapCircleAll(attackPoss.transform.position, range, playerLayer);

foreach (var i in colls)

{

if (i.GetComponent<PlayerChara>().takeDmg(dmg, 1))

{

StartCoroutine(StunCd());

}

}

indicatorDodge.SetActive(false);

}

else if (canOnlyParry && 90 <= dice && dice < 100) // onlyParry

{

animController.SetTrigger("attackSpecial");

// Indicator HERE

indicatorParry.SetActive(true);

var tmpColor = indicatorDodge.GetComponent<SpriteRenderer>().color;

tmpColor.a = 100f;

indicatorParry.GetComponent<SpriteRenderer>().color = tmpColor;

yield return new WaitForSeconds(attackDelay - 0.2f);

// Indicator FLASH

tmpColor.a = 255f;

indicatorParry.GetComponent<SpriteRenderer>().color = tmpColor;

yield return new WaitForSeconds(0.2f);

Collider2D[] colls = Physics2D.OverlapCircleAll(attackPoss.transform.position, range, playerLayer);

foreach (var i in colls)

{

if (i.GetComponent<PlayerChara>().takeDmg(dmg, 2))

{

StartCoroutine(StunCd());

}

}

indicatorParry.SetActive(false);

}

else // default

{

animController.SetTrigger("attack");

yield return new WaitForSeconds(attackDelay);

Collider2D[] colls = Physics2D.OverlapCircleAll(attackPoss.transform.position, range, playerLayer);

foreach (var i in colls)

{

if (i.GetComponent<PlayerChara>().takeDmg(dmg, 0))

{

StartCoroutine(StunCd());

}

}

}

isAttacking = false;

StartCoroutine(AttackCd());

}

public void Move()

{

Vector2 direction = (playerObj.transform.position - transform.position).normalized;

rb.velocity = new Vector2(direction.x \* speed, rb.velocity.y);

animController.SetTrigger("run");

if (rb.velocity.x > 0) // mirror gameobject

{

transform.localScale = new Vector3(1, 1, 1);

}

else if (rb.velocity.x < 0)

{

transform.localScale = new Vector3(-1, 1, 1);

}

}

public void DestroyObj()

{

Destroy(gameObject);

}

#endregion

}

Додаток З EnemyBehaviour.cs

using System.Collections;

using System.Collections.Generic;

using System.Net.Http.Headers;

using UnityEngine;

using UnityEngine.EventSystems;

public class EnemyBehavior : MonoBehaviour

{

public EnemyClass enemyClass;

public bool inAction = false;

void Start()

{

enemyClass = GetComponent<EnemyClass>();

enemyClass.playerObj = GameObject.FindWithTag("Player");

enemyClass.animController.SetTrigger("start");

transform.localScale = new Vector3(1, 1, 0);

}

void Update()

{

if (enemyClass.isDead || enemyClass.isStunned || !enemyClass.canAttack)

{

enemyClass.rb.velocity = Vector2.zero;

return;

}

if (enemyClass.hpCurrent <= 0 && !enemyClass.isDead)

{

enemyClass.rb.velocity = Vector2.zero;

enemyClass.Death();

}

enemyClass.distanceToPlayer = Vector2.Distance(transform.position, enemyClass.playerObj.transform.position);

if (enemyClass.SeePlayer() && !enemyClass.CanAttack())

{

enemyClass.Move();

}

else if (enemyClass.CanAttack() && !enemyClass.isAttacking && !inAction)

{

enemyClass.rb.velocity = Vector2.zero;

StartCoroutine(enemyClass.Attack());

}

}

private IEnumerator Pause(float time)

{

inAction = true;

yield return new WaitForSeconds(time);

inAction = false;

}

}