

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет "Запорізька політехніка"

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО
ЗАСТОСУНКУ. RESEARCH AND SOFTWARE IMPLEMENTATION OF GAME
APPLICATION

Виконала: студентка 2 курсу, групи КНТ—124м
Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ГРИШКО Н.С.

(ПРІЗВИЩЕ та ініціали)

Керівник ДУБРОВІН В.І.

(ПРІЗВИЩЕ та ініціали)

Рецензент БАБЕНКО Н.В.

(ПРІЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет "Запорізька політехніка"
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти магістр
 Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТКИ

ГРИШКО Ніки Сергіївни

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та програмна реалізація ігрового застосунку. Research and Software Implementation of Game Application.

керівник проєкту (роботи) к.т.н., професор, ДУБРОВІН Валерій Іванович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “30” вересня 2025 року № 447

2. Строк подання студентом проєкту (роботи) 1 грудня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково—пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області, стан проблеми та постановка задачі. 2. Розробка комп'ютерних ігор. 3. Проєктування комп'ютерного додатку. 4. Розробка комп'ютерного додатку “Маленькі пригоди в Балабурсі”

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1—4 Основна частина	ДУБРОВІН В.І., професор		
Нормоконтроль	БСЛОВА А.В., асистент		

7. Дата видачі завдання " 30 " вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1—2 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	3 тиждень	Розділ 1
4	Розробка архітектури програми.	3 тиждень	Розділ 3, 4
5	Розробка програми.	4—6 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	7 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	8—10 тиждень	Додатки
8	Нормоконтроль та рецензування.	11 тиждень	
9	Захист роботи.	12 тиждень	

Студентка

(підпис) Ніка ГРИШКО,
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Валерій ДУБРОВІН
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
125 с., 45 рис., 3 дод., 14 джерел.

UNITY, C#, RPG, ІГРОВІ КОМП'ЮТЕРНІ ДОДАТКИ.

Об'єктом дослідження є процес створення комп'ютерних ігрових застосунків, тобто програмних продуктів, призначених для розваги користувачів та взаємодії з віртуальним середовищем. Для розробки ігор сьогодні застосовується велика кількість технологій, інструментів і програмних платформ.

Предмет дослідження становлять технології проєктування та реалізації ігрових застосунків, що охоплюють основні етапи створення гри: формування концепції, програмну реалізацію, візуалізацію, розробку звукового супроводу та подальше тестування. Вивчення цих технологій дозволяє застосовувати сучасні методи та підходи у сфері розробки інтерактивних продуктів.

Метою кваліфікаційної роботи є покращення користувацького досвіду у двовимірних ігрових застосунках шляхом створення пригодницької гри з елементами RPG, що забезпечує інтерактивну взаємодію гравця з віртуальним світом через систему діалогів і завдань, реалізовану в середовищі Unity.

Графічні елементи гри розроблено в редакторі PixelArt, де створено спрайти персонажів. Структуру діалогів змодельовано у draw.io, що дозволило вибудувати сюжетні розгалуження. Архітектурну модель програмного продукту сформовано у Modelio, що забезпечило логічну узгодженість і гнучкість системи.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 125 pages, 45 figures, 3 appendices, 14 sources.

UNITY, C#, RPG, GAMING APPLICATIONS.

The object of the study is the process of creating computer game applications, i.e., software products designed for user entertainment and interaction within virtual environments. A wide range of technologies, tools, and software platforms are currently applied in game development.

The subject of the study is the technologies for designing and implementing game applications, covering the main stages of game creation: concept development, programming, visualization, sound design, and subsequent testing. Studying these technologies allows for the application of modern methods and approaches in the development of interactive products.

The aim of the qualification thesis is to improve the user experience in two-dimensional game applications by developing an adventure game with RPG elements that provides interactive player engagement with a virtual world through a system of dialogues and quests implemented in the Unity environment.

Graphical elements of the game were developed in the PixelArt editor, where character sprites were created. The dialogue structure was modeled in draw.io, allowing for branching storylines. The architectural model of the software product was designed in Modelio, providing logical consistency and system flexibility.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	8
ВСТУП	9
1 Аналіз предметної області, стан проблеми та постановка задачі	11
1.1 Аналіз сучасного стану ігрової індустрії	11
1.1.1 Історія розвитку відеоігор	11
1.1.2 Класифікація ігор	12
1.1.3 Сучасні тенденції в розробці ігор	15
1.1.4 Вплив ігрової індустрії на економіку, культуру та освіту	16
1.2 Програмні засоби для створення ігрових застосунків	18
1.2.1 Огляд сучасних ігрових рушіїв	18
1.2.2 Порівняння технологій	19
1.2.3 Особливості розробки 2D та 3D ігор	21
1.3 Підходи до розробки ігрових застосунків	22
1.3.1 Архітектурні шаблони	22
1.3.2 Особливості розробки ігрового процесу та систем взаємодії	23
1.4 Постановка задачі дослідження	28
1.4.1 Визначення основної мети та завдань роботи	28
1.4.2 Формулювання проблеми	30
1.4.3 Обґрунтування вибору інструментів	31
1.5 Висновки за розділом 1	32
2 Розробка комп'ютерних ігор	34
2.1 Етапи розробки гри	34
2.2 Пре—продакшн	36
2.3 Виробництво	38
2.4 Забезпечення якості	42
2.5 Випуск	44

2.6 Пост—продакшн	45
2.7 Висновки за розділом 2	46
3 Проєктування комп'ютерного додатку	48
3.1 Визначення структури та функціоналу	48
3.2 Визначення концепції	53
3.3 Проєктування гри “Маленькі пригоди в Балабурсі”	55
3.4 Проєктування першого ігрового рівня	59
3.5 Висновки за розділом 3	61
4 Розробка комп'ютерного додатку “маленькі пригоди в балабурсі”	62
4.1 Розробка сюжету гри	62
4.2 Розробка дизайну ігрових локацій	67
4.3 Розробка ігрових персонажів та їх спрайтів	68
4.4 Програмна реалізація комп'ютерного додатку	72
4.5 Висновки за розділом 4	80
Висновки	81
Перелік джерел посилання	83
ДОДАТОК А Текст програми	85
ДОДАТОК Б Слайди презентації	115
ДОДАТОК В Відео-трейлер гри	125

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

- RPG – Role—Playing game;
2D – 2—dimensional;
NPC – non—playable character;
GDD – game design document.

ВСТУП

У сучасному інформаційному суспільстві відеоігри посідають вагоме місце як у сфері розваг, так і в культурному, соціальному та освітньому середовищах. Стрімкий розвиток цифрових технологій сприяв формуванню глобальної індустрії, яка охоплює мільйони користувачів і значно впливає на різні сфери людської діяльності. Ігрові продукти сьогодні варіюються від простих мобільних застосунків до складних інтерактивних віртуальних світів, що функціонують на персональних комп'ютерах та спеціалізованих платформах. Відеоігри дедалі частіше виконують не лише розважальну функцію, а й сприяють розвитку творчих здібностей, стратегічного мислення та навичок прийняття рішень.

У межах даної дипломної роботи передбачається розроблення 2D пригодницької гри «Маленькі пригоди в Балабурсі», у якій реалізовано сюжетно—орієнтовану структуру, систему діалогів, набір завдань для гравця та можливість впливу рішень користувача на подальший розвиток подій. Основним інструментом для створення програмної частини проєкту обрано ігровий рушій Unity, що зумовлено його широкими функціональними можливостями, підтримкою мови програмування C#, високою продуктивністю та здатністю забезпечувати багатоплатформність кінцевого продукту.

Графічне наповнення гри розроблено за допомогою безкоштовного редактора PixelArt, який дає змогу створювати спрайтову графіку для двовимірних ігор. Сценарна структура та діалогові моделі формувалися у середовищі draw.io, що дозволило побудувати логічні гілки вибору та інтегрувати їх у внутрішню ігрову логіку. Архітектурні рішення та структурні моделі програмного продукту створено засобами Modelio, що забезпечило формування гнучкої, розширюваної та підтримуваної системи.

У процесі реалізації проєкту було розроблено систему діалогів, створено набір сюжетних завдань, інтегровано графічні елементи та

спроєктовано архітектуру гри відповідно до сучасних підходів у розробленні ігрових застосунків. Використання комплексного набору інструментів забезпечило цілісність програмного продукту, узгодженість його структурних компонентів та відповідність визначеним вимогам і завданням проєкту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, СТАН ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз сучасного стану ігрової індустрії

1.1.1 Історія розвитку відеоігор

Перші цифрові ігри з'явилися в середині XX століття, коли комп'ютери почали використовувати не лише для наукових обчислень, але й для експериментів з інтерактивними системами. Однією з перших відомих ігор вважають “Tennis for Two” (1958), створену фізиком Вільямом Гігінботамом для публічної демонстрації можливостей обчислювальної техніки. Іншим раннім та важливим прикладом є “Spacewar!” (1962), розроблена студентами MIT, яка стала першим проєктом, близьким до сучасного розуміння відеоігор [1].

У 1970—х роках почалася комерціалізація галузі: випуск аркадних ігор (зокрема “Pong”, 1972) і поява перших домашніх консолей започаткували масове поширення ігор як розважального продукту. Це сприяло формуванню перших бізнес—моделей, студій розробки та жанрових канонів [1]. 1980—ті роки стали періодом як бурхливого розвитку, так і серйозної кризи — «крах відеоігор» 1983 року, після якого індустрія відродилася завдяки появі якісних консолей (зокрема NES) та зміщенню акценту до контролю якості й розширення міжнародного ринку [2].

Перехід до 1980—х і початку 2000—х ознаменувався стрімким розвитком персональних комп'ютерів, 3D—графіки та мережевого геймінгу. Саме в цей період з'явилися рушії та інструменти, що стандартизували процес розробки ігор і дозволили створювати масштабні проєкти AAA—класу [2]. Одночасно сформувалися академічні підходи до вивчення ігор як культурного й технологічного явища.

У XXI столітті індустрія розвивається за двома основними напрямками: збільшення масштабів і комерціалізації великих проєктів і демократизація

розробки завдяки інді—руху та доступним рушіям і платформам дистрибуції. Сучасні тренди включають мобільні платформи, VR/AR, кросплатформені рішення, а також зростання ролі штучного інтелекту у створенні контенту й поведінці персонажів. Ігрова індустрія перетворилася з технічного експерименту на вагомий культурний і економічний феномен [3].

Історія відеоігор поєднує технічні інновації, ринкові трансформації й культурні зрушення. Для магістерської роботи важливо показати не лише хронологію подій, а й те, як ці події впливали на підходи до архітектури ігор та методи їх розробки.

1.1.2 Класифікація ігор

Класифікація відеоігор є важливим інструментом аналізу сучасної ігрової індустрії, оскільки дозволяє упорядкувати широкий спектр існуючих ігор, зрозуміти їх спільні риси та відмінності, а також визначити особливості взаємодії гравців із продуктом. У наукових дослідженнях і галузевих аналітичних звітах виділяють кілька ключових критеріїв класифікації, серед яких найбільш поширеними є жанрова приналежність, цільова платформа та тип аудиторії. Кожен із цих критеріїв по—своєму впливає на дизайн, технічну реалізацію та комерційну модель гри, формуючи її місце на ринку.

Жанрова класифікація дозволяє описати фундаментальні механіки та структуру ігрового процесу, тому є однією з базових у геймдизайні. Платформна класифікація підкреслює технічні можливості й обмеження, зумовлені типом пристрою, що визначає вимоги до оптимізації та управління. Класифікація за аудиторією відображає поведінкові моделі гравців, їхній досвід, мотивації та очікування від ігрового процесу. У сукупності ці критерії дають комплексне уявлення про структуру ринку відеоігор та дозволяють глибше зрозуміти принципи створення сучасних ігрових продуктів.

Для подальшого аналізу в цьому підрозділі наведено узагальнену схему жанрової, платформної класифікацій, а також розглянуті їх ключові характеристики та практичне значення (рис. 1.1).

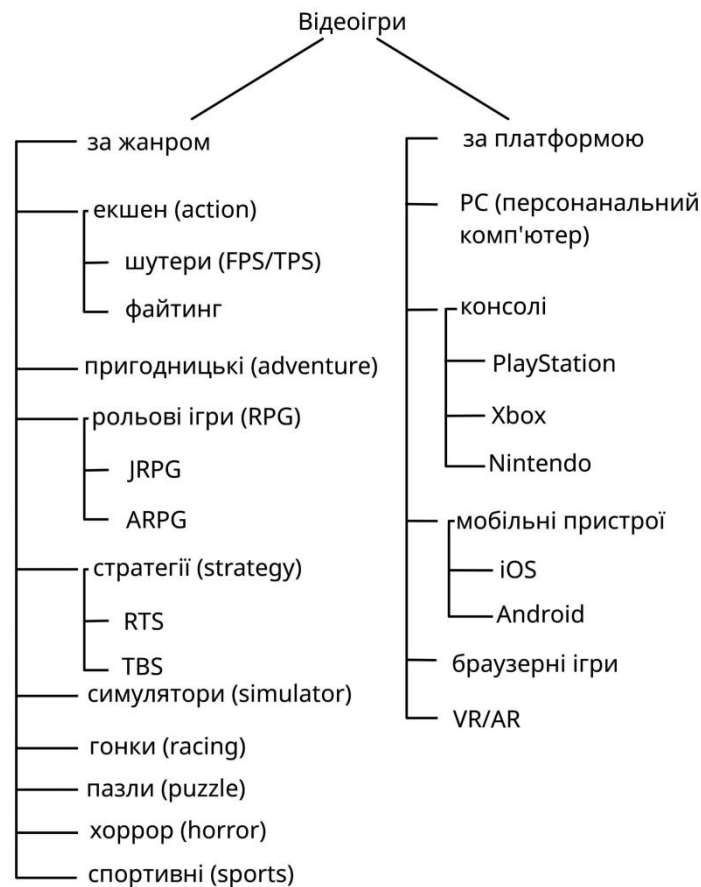


Рисунок 1.1 — Схема класифікації відеоігор за жанрами та платформами

Класифікація відеоігор є необхідною складовою аналізу сучасного стану ігрової індустрії, оскільки вона дозволяє систематизувати різноманіття розробок, визначити ключові напрями ринку та зрозуміти особливості взаємодії користувачів із продуктом. У науковій літературі та індустріальних дослідженнях виділяють три основні критерії класифікації: жанрова приналежність, цільова платформа, а також аудиторія гравців [3].

Жанр визначає структуру ігрового процесу, характер взаємодії гравця із системою та набір основних механік. У фундаментальних роботах К. Сален та Е. Циммермана жанр розглядається як комбінація геймплейних принципів

і типових сценаріїв поведінки гравця [3]. Розглянемо більш детально класифікацію ігор (рис. 1.2).

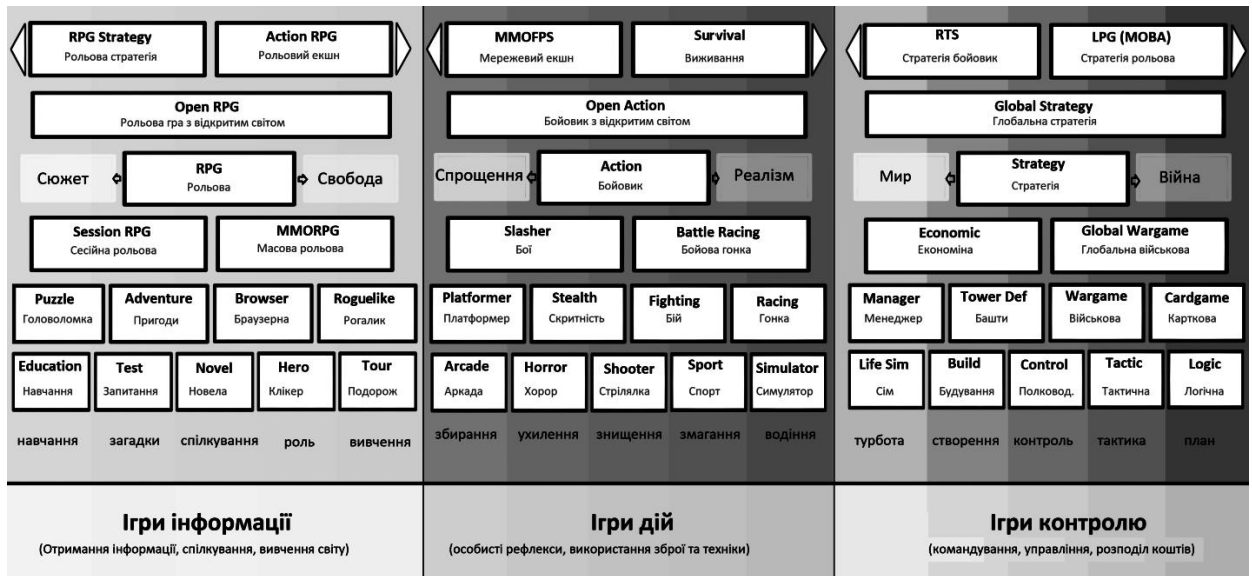


Рисунок 1.2 – Класифікація ігор по жанрам

Найпоширеніші жанри:

- Action (екшн) — швидка динаміка, високі вимоги до реакції (Doom, Devil May Cry);
- RPG — системи розвитку персонажа, сюжетна прогресія (Baldur's Gate 3, Skyrim);
- Adventure — дослідження і розв'язання задач (Uncharted, Life is Strange);
- Strategy — планування і управління ресурсами (StarCraft II, Civilization VI);
- Simulation — моделювання реальних або вигаданих систем (The Sims, Euro Truck Simulator);
- Shooter — бої від першої або третьої особи (Counter—Strike 2, Fortnite);
- Puzzle — логічні задачі (Tetris, Portal);

– Sports/Racing — спортивні симулятори і перегони (FIFA, Forza Horizon).

Сучасні ігри часто поєднують елементи різних жанрів, утворюючи гібридні моделі, що розширює творчі можливості розробників, наприклад: action-RPG, survival-horror тощо.

Платформа визначає доступні технічні можливості, характер управління та вимоги до оптимізації. Аналітичні звіти Newzoo підкреслюють, що саме платформний поділ суттєво впливає на бізнес—модель ігрових продуктів [2].

Цільова аудиторія визначає стиль подачі, рівень складності, тривалість сесій та загальні вимоги до дизайну. У звітах ESA (Entertainment Software Association) виділяють такі групи гравців [4].

Класифікація ігор за жанрами, платформами та аудиторіями дозволяє комплексно оцінити структуру ринку та визначити ключові тенденції його розвитку. Такий підхід є основою для подальшого аналізу індустрії та формування завдань магістерської роботи.

1.1.3 Сучасні тенденції в розробці ігор

Сучасна ігрова індустрія продовжує активно трансформуватися під впливом технологічних інновацій, змін ринку та розвитку нових моделей взаємодії з користувачами. Згідно зі звітами Newzoo та ESA, ключовими драйверами зростання залишаються мобільні проєкти, сегмент VR/AR, поширення кросплатформених технологій та впровадження систем штучного інтелекту [5], [6]. Мобільний ринок уже кілька років утримує позицію найбільшого сегмента, забезпечуючи близько половини всіх доходів індустрії, що пояснюється доступністю смартфонів, розвитком free—to—play моделі та вдосконаленням мобільних апаратних засобів. Розробники орієнтуються на короткі ігрові сесії, соціальні функції та високий рівень оптимізації, завдяки чому мобільні ігри успішно конкурують з більш традиційними платформами.

Паралельно розвивається напрям VR/AR, що отримав імпульс через здешевлення пристроїв та зростання інтересу до занурювального досвіду. Сучасні VR—проекти впроваджують природні фізичні взаємодії, контроль руху та хаптичні відчуття, що створює більш реалістичний ігровий простір. AR—технології, у свою чергу, активно інтегруються у мобільні застосунки, поєднуючи реальний та цифровий світи.

Ще однією формуючою тенденцією є кросплатформеність. Поширення ігрових рушіїв Unity та Unreal Engine дає змогу розробникам забезпечувати одночасний реліз гри на ПК, консолях і мобільних пристроях з єдиною базою коду [7]. Такий підхід знижує витрати виробництва, збільшує охоплення аудиторії та сприяє впровадженню функцій спільного прогресу й багатокористувацької гри між різними платформами. Додатковий вплив має розвиток хмарного геймінгу, який дозволяє запускати ресурсомісні ігри на слабших пристроях.

Особливої уваги заслуговує роль штучного інтелекту, який поступово стає одним із ключових інструментів сучасної розробки. Окрім традиційних задач — управління поведінкою NPC та адаптації складності — AI дедалі частіше застосовується у процедурній генерації рівнів, автоматизації створення контенту та оптимізації робочих процесів. Інструменти на основі машинного навчання допомагають генерувати анімації, текстури, діалоги та навіть динамічні сюжетні лінії, що суттєво скорочує виробничі витрати та розширює творчі можливості команд. Дослідження GDC свідчать, що інтеграція AI у найближчі роки стане стандартом галузі, адже дозволяє створювати складніші та інтерактивніші ігрові світи [7].

Сучасні тенденції демонструють, що розвиток індустрії рухається у напрямі мобільності, занурювальних технологій, універсальності платформ та інтелектуальної автоматизації виробництва, що визначає стратегічні орієнтири індустрії на найближче десятиліття.

1.1.4 Вплив ігрової індустрії на економіку, культуру та освіту

Ігрова індустрія сьогодні є одним із найбільш динамічних секторів цифрової економіки. Її вплив виходить далеко за межі розваг, формуючи нові економічні моделі, культурні практики та освітні підходи. За даними Entertainment Software Association, відеоігри щороку демонструють стабільне зростання та забезпечують робочими місцями мільйони спеціалістів у всьому світі [8].

Глобальний ринок відеоігор перевищує багато традиційних сегментів індустрії розваг, зокрема кіно та музичний бізнес. Доходи надходять не лише від продажу ігор, а й від моделей підписки, внутрішньоігрових транзакцій, електронного спорту та створення контенту стримерами. Розвиток індустрії стимулює суміжні сфери — ринок апаратного забезпечення, VR/AR технологій, хмарних сервісів та мобільних платформ. Крім того, індустрія підтримує високий рівень інновацій, що сприяє появі нових професій: ігрових дизайнерів, аналітиків поведінки користувачів, фахівців зі штучного інтелекту, інженерів XR—технологій тощо [8].

Відеоігри перетворилися на важливий культурний феномен, що формує нові форми сторітелінгу, мистецької виразності та взаємодії. Ігри поєднують елементи літератури, кінематографа, музики та інтерактивного дизайну, створюючи унікальні медіапростори. Глобальні ігрові спільноти сприяють поширенню культурних цінностей, змішанню традицій і виникненню нових форм комунікації. Такі ігри, як The Witcher 3, Journey, The Last of Us, стали культовими через глибокі сюжети та емоційний вплив на гравців. Відеоігри все частіше визнаються окремим видом цифрового мистецтва [8].

У сфері освіти відеоігри використовуються як інструмент для розвитку критичного мислення, просторової уяви, командної роботи та швидкості ухвалення рішень. Гейміфікація навчальних процесів робить засвоєння матеріалу ефективнішим, а симулятори дозволяють

відпрацьовувати навички в безпечному середовищі — від льотної підготовки до медичних операцій. Ігри також застосовуються у викладанні програмування, історії, мов, математики. Формується окрема дисципліна — *game—based learning*. Освітні продукти типу *Minecraft Education Edition* демонструють потенціал ігор у розвитку креативності та логіки в учнів різного віку [8].

Отже, вплив відеоігор виходить за межі індустрії розваг. Вони виступають потужним соціально—економічним чинником, формують сучасну цифрову культуру та відкривають нові можливості для навчання. Це робить дослідження технологій розробки ігор актуальним та науково значущим у контексті сучасного інформаційного суспільства.

1.2 Програмні засоби для створення ігрових застосунків

1.2.1 Огляд сучасних ігрових рушіїв

Сучасні ігрові рушії становлять основу індустрії цифрових ігор, забезпечуючи засоби для візуалізації графіки, моделювання фізики, роботи зі сценаріями та керування ресурсами. Згідно з дослідженням Alves та Roque, найбільш поширені рушії можна поділити на універсальні багатоплатформні системи, інструменти для високореалістичних 3D—проектів, а також спеціалізовані рішення для 2D—розробки [8].

Unity є одним із найпопулярніших рушіїв завдяки кросплатформеності, широкій екосистемі та підтримці як 2D, так і 3D середовищ. Автори зазначають, що саме гнучкість Unity, його модульна архітектура та доступність Asset Store роблять рушій оптимальним для інди—розробників, навчальних проектів і мобільних ігор [8].

Unreal Engine у дослідженні описується як рушій, орієнтований на графічно вимогливі проекти. Його ключовою перевагою є високоякісний рендеринг—ядро, що підтримує передові технології освітлення і матеріальних систем. Важливою особливістю Unreal Engine, за даними

авторів, є система Blueprints, яка дозволяє створювати логіку без необхідності глибокого програмування, що пришвидшує прототипування [8].

Godot у публікації розглядається як легкий та повністю відкритий рушій із підтримкою 2D та 3D. Його перевагою є безкоштовна ліцензія без роялті та незалежна екосистема, що робить його привабливим для освітніх інституцій і невеликих студій. GDScript, подібний до Python, знижує поріг входу для новачків [8].

GameMaker Studio та Construct характеризуються як рушії, що орієнтовані на швидку розробку 2D—ігор та навчальні проєкти. У статті підкреслюється, що вони забезпечують зручні засоби візуального програмування, що робить їх популярними у школах, серед початківців та незалежних розробників [8].

Як зазначають Alves і Roque, вибір рушія визначається не лише технічними можливостями, а й типом цільового продукту: Unity і Unreal підходять для комплексних 3D—проєктів, натомість Godot, GameMaker і Construct — для невеликих 2D—ігор, навчальних застосунків та швидкого прототипування. Отже, сучасні рушії формують інфраструктуру індустрії, визначаючи можливості та обмеження майбутніх ігор.

1.2.2 Порівняння технологій

У сучасних ігрових рушіях технологічні відмінності проявляються насамперед у мовах програмування, можливостях візуального редагування та у рівні підтримки цільових платформ. Alves і Roque підкреслюють, що саме ці параметри визначають швидкість розробки, гнучкість і доступність рушія для різних груп користувачів [8].

Unity використовує C#, що поєднує високу продуктивність із відносною простотою синтаксису, а також підтримує компонентно—орієнтовану модель, яка спрощує створення ігрової логіки. Візуальний редактор Unity вирізняється інтуїтивністю, а велика кількість інструментів

сцени, анімації та роботи з фізикою дозволяють швидко збирати прототипи. Unity також забезпечує широку платформну підтримку, охоплюючи Windows, Linux, macOS, Android, iOS, консолі та веб, що робить рушій універсальним для більшості комерційних ігор [8].

Unreal Engine базується на C++, але для спрощення роботи пропонує систему Blueprints — засіб візуального програмування, який дозволяє будувати складну логіку без написання коду. Дослідники зазначають, що така комбінація робить рушій придатним як для досвідчених програмістів, так і для дизайнерів. Unreal Engine має потужний редактор із розвиненими засобами роботи зі світлом, матеріалами та анімацією, що дає змогу створювати графічно насичені 3D—проекти. Підтримка платформ включає основні настільні операційні системи, мобільні пристрої та сучасні консолі [8].

Godot надає вибір між власною мовою GDScript, яка за стилем наближена до Python, а також C# і C++ для більш вимогливих завдань. Рушій має вбудований візуальний редактор сцени та окремі інструменти для 2D і 3D, що робить роботу з ним прозорою і структурованою. Основною відмінністю Godot є повністю відкрите програмне середовище, а також вільна ліцензія, завдяки чому розробники можуть адаптувати рушій до власних потреб. Підтримка експорту охоплює настільні платформи, мобільні ОС та веб [8].

GameMaker Studio, як зазначають Alves і Roque, використовує власну мову GML, орієнтовану на простоту й доступність. У поєднанні з візуальними інструментами та блоковим сценарним підходом рушій стає оптимальним для швидкої розробки 2D—проектів. У статті підкреслюється, що підтримка платформ у GameMaker включає настільні системи, мобільні платформи та HTML5, проте можливості 3D значно обмежені.

Construct — рішення, що базується майже повністю на візуальному програмуванні, що дозволяє створювати ігри без написання коду. Такий підхід робить рушій популярним у навчальному середовищі. Обмеженням

Construct є менша гнучкість у порівнянні з універсальними рушіями та залежність від веб—технологій [8].

Різниця між рушіями проявляється насамперед у балансі між програмуванням і візуальними засобами. Unity і Unreal пропонують універсальні середовища для складних проєктів із широкою платформною підтримкою; Godot забезпечує відкритість і гнучкість; GameMaker і Construct орієнтовані на швидкість створення 2D—ігор і навчальну доступність. У сукупності ці технологічні особливості формують різні сценарії використання рушіїв залежно від цілей і досвіду розробника [8].

1.2.3 Особливості розробки 2D та 3D ігор

Розробка ігор у двовимірному (2D) та тривимірному (3D) форматах має суттєві відмінності, що обумовлені технічними, графічними та логічними особливостями цих підходів.

У 2D—іграх об'єкти відображаються на площині, використовуючи спрайти, тайл—сети та фонові шари. Головна увага приділяється ефективному рендерингу спрайтів, організації анімаційних циклів та взаємодії між об'єктами у двовимірному просторі. Розробка таких ігор зазвичай є менш ресурсоємною, що дозволяє швидко створювати прототипи та адаптувати їх для мобільних платформ або веб—браузерів. Однак, навіть у 2D—проєктах важливими залишаються управління колізіями, фізичні взаємодії, управління станами персонажів та реалізація логіки гри [9].

У 3D—іграх об'єкти існують у тривимірному просторі, що вимагає додаткових компонентів, таких як моделі, текстури, освітлення, камери та системи фізики. Розробка 3D—ігор передбачає роботу з 3D—рушіями (наприклад, Unity або Unreal Engine), де необхідно враховувати тривимірну навігацію, обчислення колізій у тривимірному просторі, анімацію скелетних моделей, рендеринг світла та тіней, а також оптимізацію продуктивності для різних платформ. 3D—проєкти зазвичай більш ресурсоємні та потребують

ретельного планування архітектури сцени, логіки ігрового процесу та взаємодії об'єктів [9].

Незважаючи на відмінності, сучасні рушії дозволяють створювати як 2D, так і 3D ігри в єдиному середовищі, забезпечуючи спільні системи анімації, фізики та взаємодії. Основна різниця полягає в обсязі даних, графічній складності та вимогах до обчислювальних ресурсів, що визначає особливості проектування, тестування та оптимізації відповідного ігрового застосунку [9].

1.3 Підходи до розробки ігрових застосунків

1.3.1 Архітектурні шаблони

Під час проектування ігрових застосунків важливе місце займають архітектурні шаблони, що забезпечують структурованість, масштабованість та підтримуваність коду. Сучасні ігрові рушії використовують комбінацію кількох підходів, серед яких ключовими є шаблони Game Loop, Model—View—Controller (MVC) та Entity—Component System (ECS) [9].

Game Loop є фундаментальним механізмом будь—якого ігрового застосунку. Він представляє нескінченний цикл, у межах якого відбувається оновлення логіки гри, обробка введення користувача та рендеринг кадру. Головна мета цього шаблону — забезпечити плавну та синхронізовану роботу всіх підсистем гри при стабільній частоті оновлення. Класичний цикл гри складається з етапів: обробка подій, оновлення стану об'єктів, симуляція фізики та візуалізація сцени. Правильно реалізований Game Loop дозволяє підтримувати передбачувану поведінку, незалежність логіки від частоти кадрів і точне керування часом [9].

У контексті побудови ігрової архітектури логічним є також розгляд структурних шаблонів, які забезпечують впорядкування взаємодії між компонентами та підтримують зрозумілу організацію коду. До таких шаблонів належать наступні:

- MVC (Model—View—Controller) використовується в іграх для розділення логіки, даних та представлення;
- Model відповідає за внутрішні дані й правила гри;
- View забезпечує відображення, включаючи рендеринг та анімацію.

Controller обробляє взаємодію користувача. У контексті ігрових рушіїв MVC часто застосовується у внутрішніх підсистемах, таких як UI—фреймворки або система ігрових меню, де важливе відокремлення стану та представлення [9].

Entity—Component System (ECS) є сучасним та гнучким архітектурним підходом, що витісняє традиційні ієрархічні моделі (наприклад, об'єктно—орієнтовану модель зі складним спадкуванням). У ECS ігровий об'єкт (entity) містить лише ідентифікатор, тоді як поведінка визначається набором компонентів — дрібних модулів, що зберігають дані. Логіка реалізується системами (systems), які опрацьовують об'єкти з певними компонентами. Така структура дозволяє зменшити зв'язність, підвищує масштабованість і суттєво оптимізує роботу з пам'яттю, що особливо важливо при створенні великих ігрових світів або ігор, орієнтованих на високу продуктивність [9].

Застосування цих архітектурних шаблонів дає можливість будувати модульні та високопродуктивні ігрові застосунки, спрощує керування складністю та пришвидшує процес розробки. Більшість сучасних рушіїв (Unity, Unreal Engine, Godot) реалізують комбінацію цих підходів, що підкреслює їхню ефективність і універсальність.

1.3.2 Особливості розробки ігрового процесу та систем взаємодії

Розробка ігрового процесу є одним із ключових етапів створення ігрового застосунку, оскільки саме gameplay визначає взаємодію гравця з ігровим світом та формує загальний досвід користувача. Ігровий процес

включає правила, механіки, стан гри, поведінку об'єктів та реакцію системи на дії користувача. Для ефективної реалізації таких механізмів у сучасних ігрових рушіях застосовують спеціалізовані архітектурні підходи, серед яких одним із найбільш поширених є модель Entity–Component–System (ECS) [9].

Розглянемо проєктування та реалізацію логіки гри. Логіка ігрового процесу — це набір правил, що визначають, як змінюється стан гри залежно від дій гравця або внутрішніх подій. При проєктуванні цієї логіки враховується структура ігрового світу, способи взаємодії між об'єктами, механіки руху, завдання гравця, поведінка NPC та інші елементи.

У сучасних рушіях логіка гри зазвичай описується за допомогою систем, які працюють над даними, збереженими в компонентах. Такий підхід дозволяє відділити дані від поведінки та забезпечити можливість розширення механік без значних змін у коді.

Розглянемо архітектуру ігрового процесу: модель ECS. Модель ECS складається з трьох основних елементів.

Entity. Сутність — це унікальний об'єкт у грі, який зазвичай є лише ідентифікатором (ID). Сутність сама не містить логіки або даних, а виступає контейнером для компонентів. Приклади сутностей: гравець, ворог, платформа, куля, двері.

Component. Компонент містить дані, що визначають властивості сутності. Приклади компонентів:

- TransformComponent (позиція, обертання, масштаб);
- PhysicsComponent (швидкість, маса, прискорення);
- HealthComponent (рівень здоров'я);
- InputComponent (поточні дії гравця);
- CollisionComponent (форма колізії, радіус, bounding box).

Компоненти НЕ містять логіки — лише дані, що спрощує оновлення, тестування й оптимізацію [10].

System. Система містить поведінку, яка застосовується до сутностей з певними компонентами. Наприклад:

- PhysicsSystem оновлює положення об'єктів на основі їхньої швидкості;
- CollisionSystem перевіряє зіткнення між сутностями;
- AISystem визначає поведінку NPC;
- RenderSystem відповідає за відображення графіки.

Системи перебирають усі сутності, які містять потрібні компоненти, і застосовують логіку до їх стану.

Переваги ECS:

- низька зв'язаність між об'єктами;
- висока продуктивність (дані розміщуються компактно в пам'яті);
- зручність у створенні складних механік;
- можливість паралельного виконання систем;
- спрощена підтримка та розширення ігрового процесу.

Завдяки цим характеристикам ECS широко використовується як в інді—іграх, так і в масштабних проєктах AAA—класу [9].

Розглянемо реалізацію логіки ігрового процесу. Обробка дій користувача (Input System). Система обробки введення перетворює фізичні дії гравця (клавіатура, миша, геймпад, торкання) у внутрішні події:

- натискання клавіш — команди руху;
- кнопки миші — атака чи взаємодія;
- тач — жести — рух камери або переміщення.

InputSystem оновлює відповідні компоненти (наприклад, InputComponent), які потім використовуються іншими системами — руху, фізики або анімації.

Collision System (система зіткнень) визначає точки взаємодії між об'єктами:

- гравець не може пройти крізь стіну;
- куля знищує ворога;
- персонаж збирає предмет;

- об'єкти можуть відскакувати або ковзати.

Ця система складається з трьох частин:

- виявлення зіткнення (collision detection). Використовуються Bounding Box, Bounding Sphere або складніші поверхні;
- розв'язання колізій (collision response). Наприклад: відштовхування, втрата здоров'я, додавання предмету в інвентар;
- передача інформації в інші системи. Наприклад, система фізики обробляє відскок, система анімацій — ефект удару.

Тепер розглянемо поняття Game States (стан гри) та FSM. Ігровий процес складається зі станів:

- Main Menu (головне меню);
- Game (геймплей);
- Pause (пауза);
- Win (перемога);
- Lose (поразка).

Для керування станами використовується підхід FSM (Finite State Machine) — кінцевий автомат, де кожен стан має власну логіку, а переходи відбуваються за певних умов. FSM забезпечує:

- передбачуваність поведінки гри;
- простоту додавання нових станів;
- контроль складних сценаріїв (комбінації меню, кат—сцен, переходів).

Наступна тема — поведінка NPC: FSM, AI, Behavior Trees. У NPC (неігрових персонажів) можуть використовуватися різні моделі штучного інтелекту.

FSM для NPC. Приклад станів ворога:

- idle (очікування);
- patrol (патрулювання);
- chase (переслідування гравця);

- attack (атака);
- flee (втеча).

Перехід між станами залежить від умов, наприклад:

- якщо гравець ближче певної дистанції — chase;
- якщо здоров'я низьке — flee.

Behavior Trees (дерева поведінки). Використовуються у складніших іграх для моделювання багаторівневої поведінки. Переваги:

- логіка легко розширюється;
- поведінка виглядає «розумнішою»;
- немає «вибуху» станів, як у великих FSM.

AI Systems у ECS. У контексті ECS поведінка NPC реалізується у вигляді окремої системи, яка зчитує компоненти:

- VisionComponent (поле зору);
- AIStateComponent (поточний стан FSM);
- PathfindingComponent (маршрути).

Наступним іде взаємодія систем у межах ігрового процесу. Системи у ECS взаємодіють опосередковано через:

- компоненти;
- подієву модель (event bus);
- черги повідомлень.

Приклади взаємодії:

- InputSystem — MovementSystem;
- MovementSystem — PhysicsSystem;
- PhysicsSystem — CollisionSystem;
- CollisionSystem — AISystem або HealthSystem;
- AISystem — AnimationSystem.

Цей розподіл робить архітектуру модульною та забезпечує можливість незалежного розвитку окремих частин гри [9].

1.4 Постановка задачі дослідження

1.4.1 Визначення основної мети та завдань роботи

Метою даної магістерської роботи є дослідження архітектури та організації ігрового процесу на базі рушія Unity та практична реалізація 2D піксельної пригодницької RPG—гри «Маленькі пригоди в Балабурсі». Досягнення цієї мети передбачає комплексне вивчення сучасних технологій розробки ігор, аналіз ефективних методів організації логіки гри, побудови систем взаємодії між ігровими об'єктами та NPC, а також практичне застосування отриманих знань для створення функціонального прототипу гри.

Реалізація даної мети дозволяє поєднати наукове дослідження і практичну розробку: з одного боку, аналіз сучасних підходів та компонентної архітектури (Entity–Component–System, ECS) забезпечує теоретичну основу, а з іншого — розробка конкретного ігрового застосунку демонструє практичну придатність обраних рішень. Особлива увага приділяється організації ігрового процесу, створенню механік взаємодії, обробці подій, логіці поведінки персонажів та NPC, а також інтеграції сюжету та ігрових механік у цілісний, динамічний світ.

Практична значущість дослідження полягає у формуванні методології розробки 2D пригодницьких ігор із елементами RPG, що дозволяє швидко прототипувати ігрові рішення, оцінювати ефективність архітектури та механік і забезпечувати масштабованість подальших проєктів. Очікуваним результатом є створення робочого прототипу гри, який демонструє досліджувані принципи: ефективну організацію ігрового процесу, взаємодію об'єктів та NPC, а також інтеграцію сюжетних ігрових механік у межах піксельного 2D світу.

Робота поєднує дослідження теоретичних аспектів і практичну реалізацію, що дозволяє не лише оцінити обрані технології та методи, але й

отримати конкретний результат у вигляді готового ігрового застосунку, придатного для подальшого розвитку або досліджень у сфері 2D RPG—ігор.

Головною метою магістерської роботи є дослідження архітектури та організації ігрового процесу на базі рушія Unity та програмна реалізація 2D піксельної пригодницької RPG—гри «Маленькі пригоди в Балабурсі». Для досягнення цієї мети визначено наступні завдання:

- аналіз сучасних технологій розробки комп'ютерних ігор — вивчення доступних ігрових рушіїв, мов програмування та підходів до створення ігрових застосунків, а також огляд сучасних тенденцій у 2D—іграх, жанрах RPG та механіках пригодницьких ігор. Це завдання дозволяє сформувати теоретичну базу для подальшої розробки та обґрунтувати вибір інструментів;
- вивчення засобів Unity та мови програмування C# — дослідження можливостей рушія Unity для розробки 2D—проектів, включаючи сцени, анімацію, фізичні системи та інтеграцію компонентів. Вивчення синтаксису та особливостей C# необхідне для реалізації ігрової логіки, обробки подій та взаємодії об'єктів у грі;
- проектування ігрової логіки та архітектури застосунку — розробка структури гри, визначення сутностей, компонентів та систем за моделлю Entity–Component–System (ECS), планування взаємодій між ігровими об'єктами та створення алгоритмів для керування подіями користувача, станами гри та поведінкою NPC;
- створення базових ігрових механік — реалізація фундаментальних елементів гри, таких як рух персонажа, система зіткнень, взаємодія з об'єктами середовища, основні бойові та пригодницькі механіки, що забезпечують базовий функціонал ігрового процесу;
- розробка NPC та системи діалогів — створення неігрових персонажів з відповідною поведінкою, алгоритмами штучного інтелекту (FSM або прості дерева поведінки), а також реалізація системи взаємодії з

гравцем через діалоги, що формують сюжетні лінії та допомагають досягати цілей гри;

– тестування та відлагодження прототипу гри — перевірка працездатності всіх механік, систем взаємодії та логіки гри, виявлення помилок, налагодження взаємодій між компонентами та забезпечення стабільного виконання гри на цільовій платформі.

Дослідження та використання архітектури ECS — оцінка ефективності застосування моделі Entity–Component–System у рамках гри, аналіз переваг і недоліків, а також демонстрація гнучкості та масштабованості при розширенні ігрових механік.

1.4.2 Формулювання проблеми

У сучасній розробці 2D пригодницьких ігор існує потреба у комплексному підході, який охоплює всі етапи створення продукту: від розробки сюжету та концептуальних схем до проектування механік, організації архітектури гри та реалізації логіки взаємодії об'єктів і персонажів у середовищі Unity. Відсутність узагальнених методологічних рекомендацій ускладнює швидке прототипування, впровадження нових механік та забезпечення масштабованості проєктів.

Особливо складним є поєднання сюжетних та ігрових елементів у межах єдиної логіки гри, що включає взаємодію гравця з NPC, обробку подій, керування станами гри та систему колізій. Відсутність чітко структурованого підходу до організації цих процесів ускладнює ефективне планування розробки, тестування та інтеграцію нових механік у вже існуючу структуру гри.

Актуальність дослідження полягає у необхідності формування методології, яка дозволяє комплексно та послідовно поєднувати етапи створення ігрового застосунку, забезпечуючи ефективну організацію логіки, взаємодію об'єктів і NPC, а також інтеграцію сюжетних та геймплейних

механік у межах 2D піксельної пригодницької RPG—гри. Очікуваним результатом є розробка робочого прототипу, який демонструє практичну придатність такого підходу та може слугувати методичною основою для подальших розробок у сфері 2D ігор.

1.4.3 Обґрунтування вибору інструментів

Для виконання магістерської роботи було обрано програмне забезпечення Unity, яке є одним із найбільш популярних та потужних ігрових рушіїв у світі. Його вибір можна пояснити кількома ключовими аспектами.

Кросплатформеність. Unity дозволяє створювати ігри для різних платформ, включаючи персональні комп'ютери, мобільні пристрої, консолі, а також VR/AR—пристрої та інші аудіовізуальні цифрові платформи. Така можливість робить Unity оптимальним вибором для розробників, які прагнуть охопити максимально широку аудиторію користувачів.

Зручність у використанні. Інтерфейс Unity інтуїтивно зрозумілий, а наявність великої кількості офіційної документації та навчальних матеріалів значно спрощує процес навчання та роботи з рушієм. Це дозволяє навіть початківцям ефективно розпочати розробку ігрових застосунків, а більш досвідченим розробникам – швидко реалізовувати складні механіки.

Активна спільнота користувачів. Unity підтримується великою та активною спільнотою, що забезпечує постійну підтримку, обмін знаннями, готові приклади та поради від інших розробників. Завдяки цьому вирішувати типові проблеми і отримувати допомогу стає значно простіше.

Розширюваність і підтримка плагінів. Unity дозволяє інтегрувати сторонні бібліотеки та плагіни, що відкриває можливість розширювати функціонал рушія та адаптувати його під специфічні вимоги проекту.

Наявність ресурсів. Рушій містить велику кількість як безкоштовних, так і платних ресурсів: готові моделі, ассети, текстури, анімації та навчальні

матеріали. Це дозволяє прискорити процес розробки, зменшити витрати та спростити прототипування.

Потужність графічного движка. Unity забезпечує високоякісну графіку та анімацію як для 2D, так і для 3D ігор, що дозволяє створювати вражаючі проєкти з реалістичними візуальними ефектами.

Для розробки логіки гри було обрано мову C#, яка є стандартом для Unity. C# поєднує простий синтаксис із широкими можливостями об'єктно—орієнтованого програмування, забезпечує ефективну роботу з компонентною архітектурою, підтримує реалізацію складної логіки взаємодії об'єктів та NPC, а також дозволяє швидко впроваджувати ігрові механіки. Популярність мови серед розробників і наявність великої кількості навчальних матеріалів та прикладів робить її ефективним інструментом для швидкого прототипування та подальшого масштабування ігрових проєктів.

Для написання коду використовується редактор Visual Studio Code, який поєднує легкість та швидкість роботи, підтримку численних плагінів для інтеграції з Unity і C#, а також можливість підключення систем контролю версій. Завдяки цьому забезпечується зручний та ефективний робочий процес під час розробки прототипу гри, налагодження коду та тестування логіки ігрових механік.

Отже, комбінація Unity, C# та Visual Studio Code забезпечує повний і гнучкий інструментарій для дослідження архітектури ігрового процесу та реалізації 2D піксельної пригодницької RPG—гри, що відповідає цілям та завданням магістерської роботи.

1.5 Висновки за розділом 1

У першому розділі було проведено детальний аналіз предметної області, сучасного стану ігрової індустрії та визначено основні напрямки розробки ігрових застосунків. Історичний огляд показав, що відеоігри пройшли складний шлях розвитку — від ранніх експериментальних проєктів

до глобальної індустрії, яка поєднує технологічні інновації, культурні та економічні аспекти. Класифікація ігор за жанрами, платформами та аудиторією дозволяє систематизувати розмаїття сучасних ігрових продуктів і визначити ключові принципи їх створення та комерціалізації.

Проаналізовано сучасні тенденції в розробці ігор, серед яких зростання мобільних платформ, поширення VR/AR, кросплатформеність та активне використання штучного інтелекту для генерації контенту та управління поведінкою NPC. Виявлено, що індустрія активно впливає на економіку, культуру та освіту, формуючи нові професії, інтерактивні культурні практики та інноваційні освітні методики.

У частині, присвяченій програмним засобам, проведено огляд сучасних ігрових рушіїв та порівняння їхніх технологічних можливостей. Особливу увагу приділено Unity як універсальному інструменту для 2D і 3D розробки, здатному забезпечити гнучку організацію ігрового процесу, масштабованість проєктів та інтеграцію сучасних механік. Розглянуто також особливості розробки 2D та 3D ігор, архітектурні шаблони (Game Loop, MVC, ECS), організацію логіки гри та системи взаємодії об'єктів і NPC.

Визначено мету та завдання магістерської роботи, сформульовано проблему і обґрунтовано вибір інструментів для розробки 2D піксельної пригодницької RPG—гри «Маленькі пригоди в Балабурсі». Підкреслено актуальність комплексного підходу до проєктування ігрового процесу та застосування сучасних архітектурних і технологічних рішень для ефективного створення та масштабування ігрових проєктів.

Розділ 1 забезпечує теоретичну основу для подальшої практичної реалізації проєкту, визначає ключові підходи до організації ігрової логіки та обґрунтовує вибір рушія і технологій, що дозволяє ефективно поєднати наукове дослідження і розробку функціонального прототипу гри.

2 РОЗРОБКА КОМП'ЮТЕРНИХ ІГОР

2.1 Етапи розробки гри

Процес розроблення відеоігор є базовим методологічним каркасом, що забезпечує створення якісних та захопливих інтерактивних продуктів. Традиційно він поділяється на низку послідовних етапів, кожен з яких виконує окрему функцію та слугує контрольним пунктом для оцінки поступу проєкту. У класичному вигляді життєвий цикл гри включає п'ять основних стадій: передвиробництво, виробництво, тестування, запуск та післярелізнa підтримку (рис. 2.1).

ЕТАПИ РОЗРОБКИ ІГОР

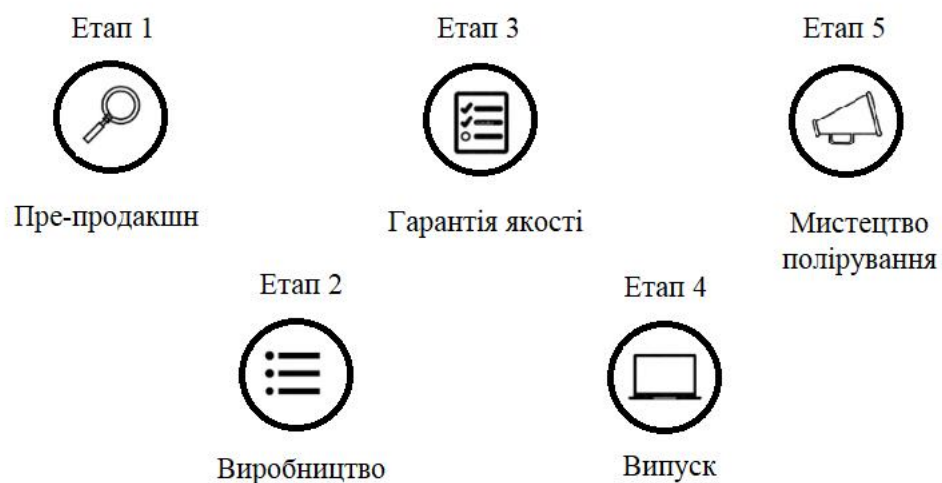


Рисунок 2.1 – Етапи розробки ігор

Першим етапом є передвиробництво, яке закладає концептуальну та організаційну основу всього циклу розроблення. На цьому етапі формуються початкові ідеї, окреслюються основні ігрові механіки, створюються попередні дизайнерські напрацювання. Ця стадія є визначальною, адже

формує напрям розвитку гри та координує подальші роботи відповідно до стратегічних цілей [8].

Після завершення концептуального планування проєкт переходить до етапу виробництва. Це найбільш масштабна та ресурсозатратна фаза, у межах якої дизайнери, програмісти, художники та інші спеціалісти спільно працюють над втіленням ідей у реальні ігрові елементи. Створюються моделі, анімації, реалізуються механіки, формується ігровий світ [8].

Наступною ключовою стадією є тестування (quality assurance). Незалежно від складності гри, вона повинна пройти системну перевірку. Тестувальники досліджують продукт на наявність технічних помилок, проблем продуктивності та недоліків у взаємодії гравця з грою. Крім виявлення помилок, ця фаза спрямована на полірування ігрового процесу, візуальних компонентів та забезпечення відповідності встановленим стандартам якості [8].

Після оптимізації та усунення критичних недоліків гра переходить до етапу запуску. Це момент, коли продукт стає доступним широкій аудиторії, і гравці вперше взаємодіють із готовим віртуальним середовищем.

Втім, робота над грою не завершується з її релізом. Заключний етап — післярелізна підтримка — забезпечує стабільність та актуальність продукту. До цього належать регулярні оновлення, виправлення нових помилок, додавання контенту й подальша оптимізація [8].

Сукупність цих етапів формує багатогранний і складний процес, який перетворює початкову ідею на повноцінний інтерактивний продукт. Значення кожної стадії важко переоцінити, оскільки саме їхня взаємодія визначає підсумкову якість ігрового проєкту.

Від початкового задуму на етапі передвиробництва до детального тестування та післярелізного вдосконалення — кожен елемент циклу розроблення відіграє свою ключову роль. Завдяки системному підходу вдається створювати ігри, які поєднують високу якість, стабільність та цікавість для користувачів [8].

2.2 Пре—продакшн

Розроблення відеогри розпочинається з комплексного планування, яке визначає хід усіх наступних робіт. Старт цього процесу вимагає ґрунтовного збору інформації про майбутній продукт, включно з технічними параметрами, творчими концепціями та потенційними обмеженнями.

Проектні менеджери та продакт—оуни відіграють ключову роль у початкових етапах. Саме вони визначають основні цілі гри та формують концептуальну базу, на якій будується увесь подальший процес розроблення:

- аналіз ринку та конкурентів. Важливо дослідити тенденції ігрової індустрії: які жанри є популярними, які моделі залучення аудиторії працюють найкраще, як позиціонуються схожі проекти. Це дозволяє правильно зорієнтувати майбутню гру на ринку;
- визначення цільової аудиторії. Одним із ключових завдань є встановлення категорії гравців, для яких створюватиметься продукт. Розуміння їхніх уподобань допомагає обрати відповідні механіки, стиль подачі та рівень складності;
- вибір платформи. Рішення щодо платформи залежить як від її популярності серед користувачів, так і від технічних можливостей команди. Платформа визначає специфіку оптимізації, структуру проекту та частково впливає на модель монетизації;
- розподіл ресурсів. Важливо визначити кадрові потреби, інструменти та обладнання. На цьому етапі складається попередній графік робіт, що забезпечує своєчасне виконання ключових задач;
- розроблення концепт—арту. Художники—концептуалісти створюють візуальну основу гри: ескізи персонажів, локацій, об'єктів та інших елементів, які формують стиль і атмосферу майбутнього проекту;

- вибір моделі монетизації. Гра може бути преміальною, безкоштовною з внутрішніми покупками, рекламною або комбінованою. Вибір моделі визначає дизайн прогресії та способи взаємодії з користувачем;
- аналіз ризиків. Недооцінений, але надзвичайно важливий крок. Команда прогнозує потенційні труднощі — технічні, фінансові, організаційні — і розробляє шляхи їхнього мінімізації;
- маркетингова стратегія. Ближче до завершення виробництва формується стратегія просування. Ефективна кампанія дозволяє збільшити впізнаваність проєкту та залучити ширшу аудиторію до моменту релізу [8].

Після завершення аналітики та стратегічного планування створюється Game Design Document (GDD). Це комплексний документ, що детально описує ігрову концепцію, механіки, сюжетні лінії, типи рівнів, архітектуру світу та інші аспекти. GDD виступає центральним орієнтиром для всієї команди.

Процес розроблення ігор — це взаємодія численних складових, що в сукупності формують якісний кінцевий продукт. Від початкових брейнштормів до ретельної деталізації у GDD — кожен етап відіграє свою важливу роль [8].

Крім технічної частини, розроблення гри має виразний творчий вимір. Початкова ідея — чи то сюжетна концепція, чи історична подія, чи оригінальна механіка — поступово трансформується завдяки ескізам, прототипам і сценарним напрацюванням [8].

Важливим компонентом є й механізм зворотного зв'язку. Тестові або бета—версії дозволяють зібрати думки користувачів та внести необхідні покращення [8].

Не менш вагомою є командна співпраця. Програмісти, дизайнери, художники, звукорежисери, сценаристи та маркетингологи разом створюють збалансований та якісний продукт.

Важливу роль відіграє також культурна та соціальна відповідність. Деякі ігри відтворюють реальні культурні явища чи історичні події, тому потребують глибокого дослідження або консультацій з експертами.

Емоційна залученість — одна з основ успішних ігор. Розробники прагнуть створити сцени та механіки, що викликають у гравця яскраві переживання — від захоплення до емпатії [8].

Швидкий розвиток технологій також суттєво впливає на процес. VR, AR, штучний інтелект, процедурна генерація відкривають нові можливості та розширюють межі індустрії [8].

Економічний вплив ігор виходить за рамки продажів. Успішні проекти створюють мерч, екранізації, колаборації та навіть туристичні маршрути. Передбачення таких можливостей може підсилити довгостроковий потенціал гри [8].

Підсумовуючи, процес створення ігор — це складна взаємодія творчості, технологій і стратегічного планування. Хоча користувач бачить лише фінальний продукт, за ним стоїть багатомісячна або багаторічна кропітка робота команди, яка формує один із найдинамічніших секторів сучасної індустрії розваг.

2.3 Виробництво

Занурюючись у сферу розроблення відеоігор, швидко стає зрозуміло, що етап продакшену є не лише серцем, а й опорою всього процесу створення гри. Це найдовша та найбільш ресурсозатратна фаза циклу розроблення, насичена різноманітними активностями, інноваціями та поділена на низку менших, проте не менш важливих підетапів [8].

Підетап 1. Прототипування: коли ідея вперше стикається з реальністю. На початку виробничої стадії центральне місце займає створення прототипів гри. Це своєрідна «репетиція» перед основним виробництвом, що дозволяє

побачити, як задумана механіка працює на практиці, та задає подальший вектор розроблення.

Для підкреслення важливості цього етапу часто наводять приклад шведського військового корабля «Васа» XVII століття. Попри вражаюче майстерність будівництва та значні вкладені ресурси, судно затонуло під час першого ж виходу в море — наслідок прорахунків у конструкції. Цей історичний епізод демонструє: без вчасного тестування навіть найамбітніші ідеї можуть зазнати невдачі. Так само і в ігровій індустрії — прототипування є критично важливим інструментом перевірки життєздатності концепцій [8].

Підетап 2. Створення візуального контенту: формування художнього світу гри. Кожна гра є складною мозаїкою елементів — персонажів, предметів, оточення, ефектів та інших активів, які формують цілісний ігровий всесвіт. Ступінь складності цього підетапу залежить від того, чи створюється гра у 2D або 3D форматі [8].

2D—ігри. Розроблення 2D—проектів є дещо простішим технічно, але не менш творчим. Візуальні активи проходять кілька стадій: ескізування, промальовування, колоризація та анімація. Простота 2D—графіки дозволяє швидше реалізовувати задум, проте якість та художній рівень залишаються надзвичайно важливими.

3D—ігри. Створення 3D—проектів — це багаторівневий процес, що потребує глибокої технічної підготовки та спеціалізованих інструментів. Він включає:

- полігональне моделювання — створення базової форми об'єктів;
- скульптинг — деталізацію та формування складних структур;
- текстурування — накладання матеріалів і текстур, що визначають зовнішній вигляд моделей;
- риггінг і анімацію — створення «скелета» та рухів персонажів.

У результаті формується повноцінний тривимірний світ, здатний забезпечити високу глибину занурення.

Виробництво візуального контенту — це поєднання мистецтва, техніки та професійної майстерності, яке перетворює концепцію в реальний художній простір гри [8].

Підетап 3. Розроблення рівнів гри. Опрацювання рівнів гри нагадує написання розділів великого літературного твору. Подібно до письменника, який продумує кожну деталь сюжету, левел—дизайнер вибудовує логіку, структуру та темп проходження гри.

Цей підетап є невід'ємною частиною загального процесу створення гри, який визначає розвиток подій, виклики для гравця, послідовність механік і загальний ритм ігрового досвіду [8]. На кожному етапі циклу розроблення історія гри поступово розгортається, відкриваючи нові простори та завдання.

Підетап 4. Проектування логіки гри: більше, ніж просто прогресія. У межах продакшену особливе місце займає створення логічної структури рівнів. Спеціалісти формують прогресію, баланс складності та динаміку ігрового процесу відповідно до жанрових вимог. Це може бути:

- лінійне проходження, де кожен розділ поступово ускладнюється;
- відкритий світ, де гравець сам визначає послідовність подій;
- процедурно згенеровані рівні, характерні, зокрема, для endless runner—ів [8].

Логіка рівнів тісно пов'язана з іншими аспектами розроблення — сюжетною побудовою, механіками, візуальним стилем та темпом гри. Усі ці елементи формують цілісний досвід, що утримує увагу та інтерес гравця.

Підетап 5. Поза межами геометрії рівнів: формування «душі» гри. Робота над рівнями не обмежується лише їхньою структурою. Левел—дизайн також включає:

- побудову ігрової економіки, яка визначає цінність ресурсів, нагороди й ризику;
- розвиток сюжету, інтеракції та сюжетних поворотів;

- створення підказок і тригерів, що допомагають гравцеві освоювати нові механіки;
- плавні переходи між локаціями, які підтримують безперервність ігрової оповіді [8].

Саме завдяки цим елементам гра здобуває емоційність, драматургію та унікальний характер.

Підетап 6. Звукове оформлення: аудіовимір ігрової реальності. Занурення у гру значною мірою залежить від звуку. Музика, шумові ефекти та озвучення персонажів формують емоційний фундамент геймплею.

Звукові ефекти інформують гравця про успіхи, небезпеки чи зміни у середовищі.

Природні звуки — кроки, вітер, вода — забезпечують автентичність.

Озвучення персонажів надає героям характер і глибину.

Це складний творчо—технічний процес, який вимагає координації між звукорежисерами, композиторами та програмістами [8].

Підетап 7. Кодинг: технічний каркас гри. На завершення виробничої фази починається програмування — етап, де всі попередні напрацювання поєднуються в єдину систему [8].

Основні завдання розробників включають:

- інтеграцію візуальних, звукових та ігрових елементів;
- реалізацію механік та систем;
- забезпечення стабільності та відсутності конфліктів;
- оптимізацію продуктивності [8].

Попри використання сучасних ігрових рушіїв, цей процес залишається надзвичайно складним, адже кожен фрагмент коду має точно відповідати задуму дизайнерів і логіці гри в цілому.

У підсумку етап продакшену — це гармонійне поєднання творчості та технологій. Саме тут ідеї набувають форми, а окремі фрагменти перетворюються на цілісний ігровий всесвіт. Кожен крок, кожен звук, кожен рядок коду — частина великої праці, що формує майбутній ігровий досвід.

2.4 Забезпечення якості

Забезпечення плавного та захопливого ігрового досвіду є одним із найважливіших завдань. У центрі процесу розробки гри знаходиться ключовий етап — ретельне тестування. Це не лише «пошук дрібних помилок»; це комплексна перевірка, спрямована на те, щоб шлях гравця був безперервним, захопливим та комфортним. Одна—єдина помилка чи збій можуть миттєво вибити гравця з ігрового світу, знецінюючи довгі години роботи, вкладені в різні етапи розробки. Саме тому в межах циклу створення гри цьому етапу приділяється така велика увага.

Варто усвідомити: етапи розробки гри нагадують розділи захопливого роману — кожен має власне значення і вимагає особливого підходу. І подібно до фінального редагування перед виданням книги, гра також повинна пройти ретельну перевірку. Це гарантує, що сюжет подається логічно, виклики утримують увагу, а пригоди залишаються незабутніми.

Процес створення відеоігор поділяє тестування на функціональне та нефункціональне.

Функціональні види тестування:

- тестування інтероперабельності. Наскільки добре гра працює на різних платформах та системах;
- регресійне тестування. Чи не порушують нові функції роботу наявних;
- Smoke—тестування. Попередня перевірка, що дозволяє виявити найбільш критичні проблеми перед детальнішим аналізом;
- локалізаційне тестування. Чи відповідає гра культурним та мовним особливостям різних регіонів;
- тестування контролю доступу. Захист від несанкціонованих користувачів та потенційних атак;
- тестування користувацького прийняття (UAT). Дозволяє оцінити, чи відповідає гра очікуванням цільової аудиторії [8].

Нефункціональні види тестування:

- тестування продуктивності. Наскільки плавно працює гра та чи забезпечує оптимальний ігровий досвід;
- тестування навантаження. Як гра поводить себе за умов великої кількості користувачів або інтенсивних операцій;
- стрес—тестування. Перевірка меж можливостей гри — коли та за яких умов вона починає давати збої;
- тестування стабільності. Аналіз тривалої роботи гри без перезавантаження;
- Volume—тестування. Перевірка роботи гри за умов великої кількості даних;
- тестування зручності використання (usability). Наскільки інтуїтивною та зрозумілою є гра для користувача;
- тестування на відповідність. Переконавання, що гра відповідає галузевим стандартам та нормам;
- тестування інсталяції. Чи встановлюється гра без помилок на різних платформах;
- тестування конфігурації. Аналіз роботи гри на різних системних конфігураціях;
- тестування відновлення після збоїв. Перевірка здатності гри відновлюватися після аварійних ситуацій [8].

У межах циклу створення гри кожен із цих видів тестування гарантує, що коли гравець зануриться у віртуальний світ, його зустріне досвід, який не лише позбавлений помилок, але й по—справжньому захоплює. Етапи розробки, а особливо тестування, демонструють прагнення індустрії забезпечити найвищу якість ігрового продукту.

Заглиблюючись у складний світ створення ігор, стає очевидно, що тестування — невід’ємна складова процесу розробки. Щоб по—справжньому зрозуміти, як створюється гра, яка резонує з користувачами, необхідно детально розглянути різні типи тестування, кожен із яких має свої

особливості й завдання. Завдяки цьому цикл розробки значно вдосконалюється, оптимізуючи досвід гравця з самого початку і до фінальних титрів.

Етапи розробки включають низку ретельних процесів, що послідовно доповнюють один одного. Одним із провідних серед них є тестування, яке перевіряє кожен аспект роботи гри та гарантує її безперебійну функціональність. Для тих, хто прагне глибше зрозуміти внутрішні механізми створення гри, доступні численні ресурси, де можна дізнатися про основні типи тестування, їх призначення та поради для ефективної організації цього процесу [8].

Кожен тип тестування фокусується на окремому аспекті гри. Наприклад, одні перевіряють функціональність та коректність роботи всіх механік, тоді як інші оцінюють продуктивність на різних пристроях, забезпечуючи сумісність і плавний геймплей. Цей багаторівневий підхід дозволяє своєчасно виявити проблеми та усунути їх ще до виходу гри, роблячи процес розробки значно ефективнішим.

2.5 Випуск

Кожен етап процесу розробки гри має важливе значення, однак особливої уваги заслуговує етап запуску. На цьому передфінальному етапі створення ігрового продукту відчувається підвищений рівень очікування як з боку команди розробників, так і з боку потенційних користувачів. Водночас фахівці індустрії чудово усвідомлюють, що вихід гри у публічний доступ не є завершенням процесу її розробки. На момент релізу в проєкті можуть залишатися окремі недоопрацювання або неочевидні технічні недоліки [8].

Навіть ретельно продумані та реалізовані ігрові продукти здатні містити приховані помилки, які проявляються лише під час взаємодії з великою кількістю реальних користувачів. Після запуску гри команда розробників продовжує активну роботу, спрямовану на стабілізацію і

покращення ігрового процесу. Паралельно починає надходити первинний зворотний зв'язок від гравців, який є цінним джерелом інформації для подальшої оптимізації механік, усунення недоліків і вдосконалення загальної якості продукту [8].

2.6 Пост—продакшн

Постпродакшн етап циклу розробки комп'ютерної гри є завершальною, але водночас тривалою фазою, що продовжує розвиток проєкту після його офіційного релізу. Після того як гра стає доступною широкому колу користувачів, робота над нею не припиняється. Навпаки, процес удосконалення переходить у нову площину, спрямовану на забезпечення стабільності, оптимізації та довготривалої підтримки програмного продукту.

Основною метою цього етапу є підвищення продуктивності та надійності гри. Оскільки платформи, операційні системи та апаратне забезпечення з часом оновлюються, розробники зобов'язані оперативно адаптувати свій продукт до нових умов. Регулярні оновлення стають необхідною складовою життєвого циклу гри, забезпечуючи не лише виправлення технічних недоліків, а й підвищення якості ігрового досвіду [8].

Важливою частиною пострелізної роботи є аналіз відгуків реальних користувачів. Після запуску гра отримує значну кількість зворотного зв'язку, який слугує цінним джерелом інформації для подальшого вдосконалення функціональності. Цей етап включає систематичне виявлення та усунення помилок, оптимізацію механік, виправлення проблем продуктивності та поліпшення інтерфейсних рішень [8].

Крім технічної підтримки, постпродакшн етап передбачає розширення ігрового контенту. Розробники регулярно додають нові ігрові механіки, розширення, сюжетні елементи та інші функціональні компоненти, орієнтуючись на потреби та інтереси цільової аудиторії. Такий підхід сприяє

підтриманню активності спільноти, продовжує життєвий цикл проєкту та забезпечує його конкурентоспроможність на ринку [8].

Окреме місце в постпродакшн фазі займає контроль якості. Кожне оновлення проходить повний цикл тестування, що дозволяє гарантувати стабільність нових версій та уникнути появи критичних помилок. Отже, якість гри підтримується на високому рівні протягом усього періоду її експлуатації.

Узагальнюючи, постпродакшн етап є невід'ємною складовою життєвого циклу комп'ютерної гри. Він забезпечує адаптацію продукту до змін технологічного середовища, підвищення його надійності, розвиток ігрового контенту та підтримку взаємодії з аудиторією. Саме завдяки цій фазі гра здатна зберігати актуальність, відповідати очікуванням користувачів і залишатися конкурентною протягом тривалого часу.

2.7 Висновки за розділом 2

Аналіз завершальних етапів життєвого циклу створення відеогри засвідчує, що реліз продукту не є фінальною точкою розробки, а лише переходом до нової, не менш важливої фази. Етап запуску супроводжується підвищеною увагою до якості, стабільності та відповідності гри очікуванням цільової аудиторії. Попри багаторівневі попередні перевірки, саме взаємодія реальних користувачів дозволяє виявити приховані недоліки, зібрати цінний зворотний зв'язок і визначити напрям подальшої оптимізації.

Подальший пострелізний супровід є невід'ємною частиною сучасного процесу розробки відеоігор. У поствиробничій фазі команда зосереджується на підвищенні продуктивності, усуненні технічних проблем та забезпеченні стабільності роботи гри в умовах постійних змін апаратних платформ і програмних середовищ. Регулярні оновлення не лише підтримують технічну актуальність проєкту, але й сприяють його розвитку шляхом розширення

контенту, впровадження нових механік та вдосконалення користувацького досвіду.

Життєвий цикл відеогри є безперервним та динамічним процесом, що триває далеко за межами її офіційного релізу. Ефективність цього циклу забезпечується постійною взаємодією між розробниками та гравцями, систематичним аналізом отриманого зворотного зв'язку та послідовним удосконаленням продукту. Сукупність цих чинників свідчить про високий рівень складності та відповідальності, притаманний сучасній ігровій індустрії, а також підкреслює важливість комплексного підходу на кожному етапі створення відеоігор.

3 ПРОЄКТУВАННЯ КОМП'ЮТЕРНОГО ДОДАТКУ

3.1 Визначення структури та функціоналу

Діаграма прецедентів (Use Case diagram) (рис. 3.1) відображає функціональні вимоги до системи з точки зору її користувачів або інших зовнішніх «акторів». Вона демонструє, які сценарії (use—case'и, або варіанти використання) доступні системі і які ролі виконують актори, що з ними взаємодіють. Зазвичай прецеденти зображуються як еліпси або овали, а актори — у вигляді схематичних фігур (наприклад, «паличкових»). Така діаграма допомагає візуалізувати, як кінцеві користувачі або зовнішні системи можуть взаємодіяти з вашим додатком [12].

У рамках розроблюваного комп'ютерного додатка діаграма прецедентів відображає перелік можливих дій, які може ініціювати користувач, а також реакцію системи на ці дії.

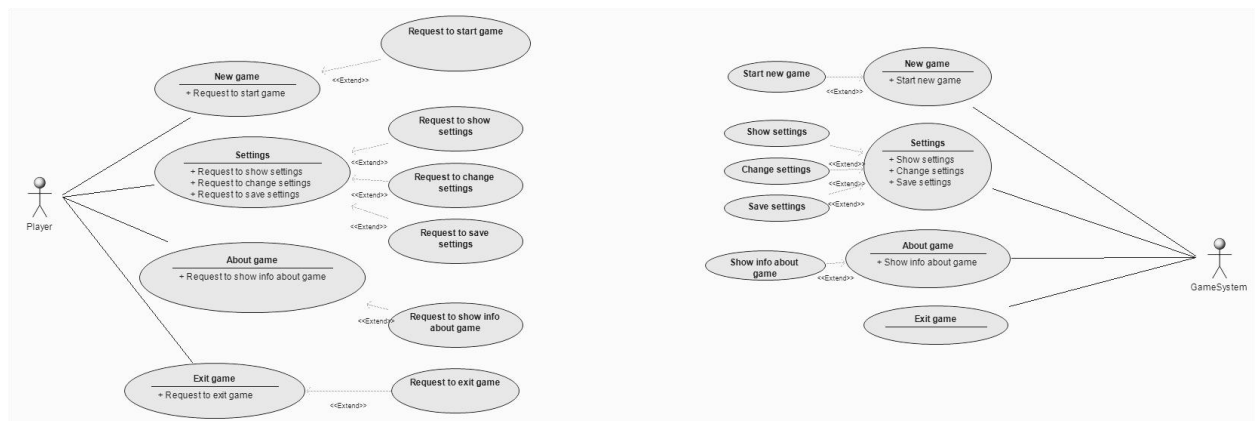


Рисунок 3.1 – Діаграма прецедентів (Use Case)

Діаграма діяльності (Activity diagram) (рис. 3.2) використовується для моделювання динамічної поведінки системи, відображаючи послідовність дій, переходи між ними та логіку виконання процесів. Кожен стан чи дія представлена окремим вузлом, а переходи демонструють потік управління. Така діаграма дозволяє проілюструвати повну послідовність процесу — від запуску застосунку до його завершення [13].

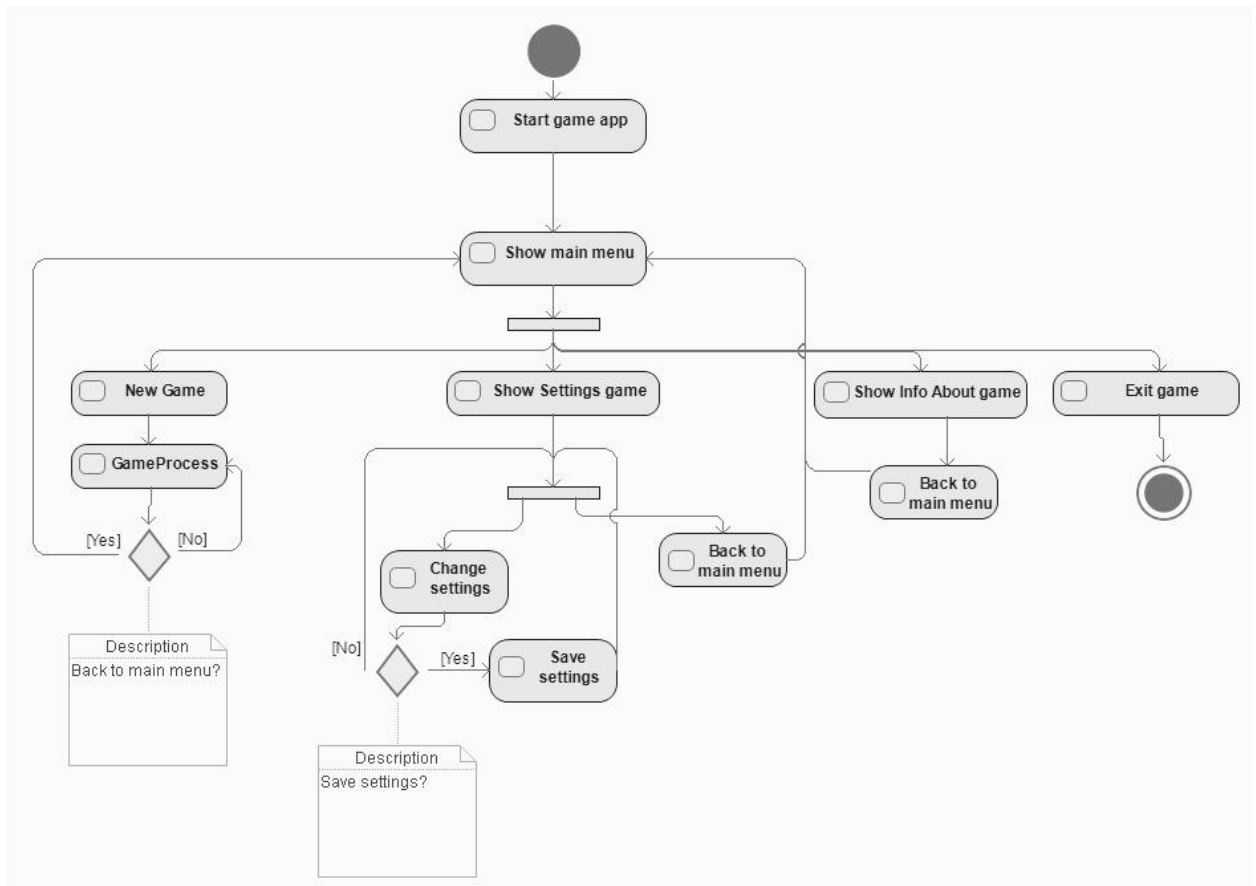


Рисунок 3.2 – Діаграма діяльності (Activity diagram)

Діаграма класів (Class diagram) використовується для відображення статичної структури програмної системи. Вона демонструє класи, їхні атрибути, методи та взаємозв'язки (агрегацію, композицію, наслідування тощо). Побудова діаграми класів є ключовим етапом об'єктно—орієнтованого проєктування, оскільки дозволяє сформувати чітку архітектурну модель системи [14].

У межах комп'ютерного додатка було сформовано низку діаграм класів, що відображають структуру таких підсистем:

- Dialogue Manager;
- Game Manager;
- Input Manager;
- UI Manager.

Кожен із менеджерів відповідає за окремий аспект роботи додатка. Наприклад, Dialogue Manager забезпечує запуск і завершення діалогових

взаємодій, а також виконання допоміжних дій до або після відтворення діалогу.

Dialogue Manager — відповідає за організацію та контроль ігрових діалогів. Він забезпечує запуск або зупинку діалогів, а також може виконувати додаткові дії перед початком або після завершення діалогу, що дозволяє гнучко управляти сценаріями взаємодії з гравцем (рис. 3.3).

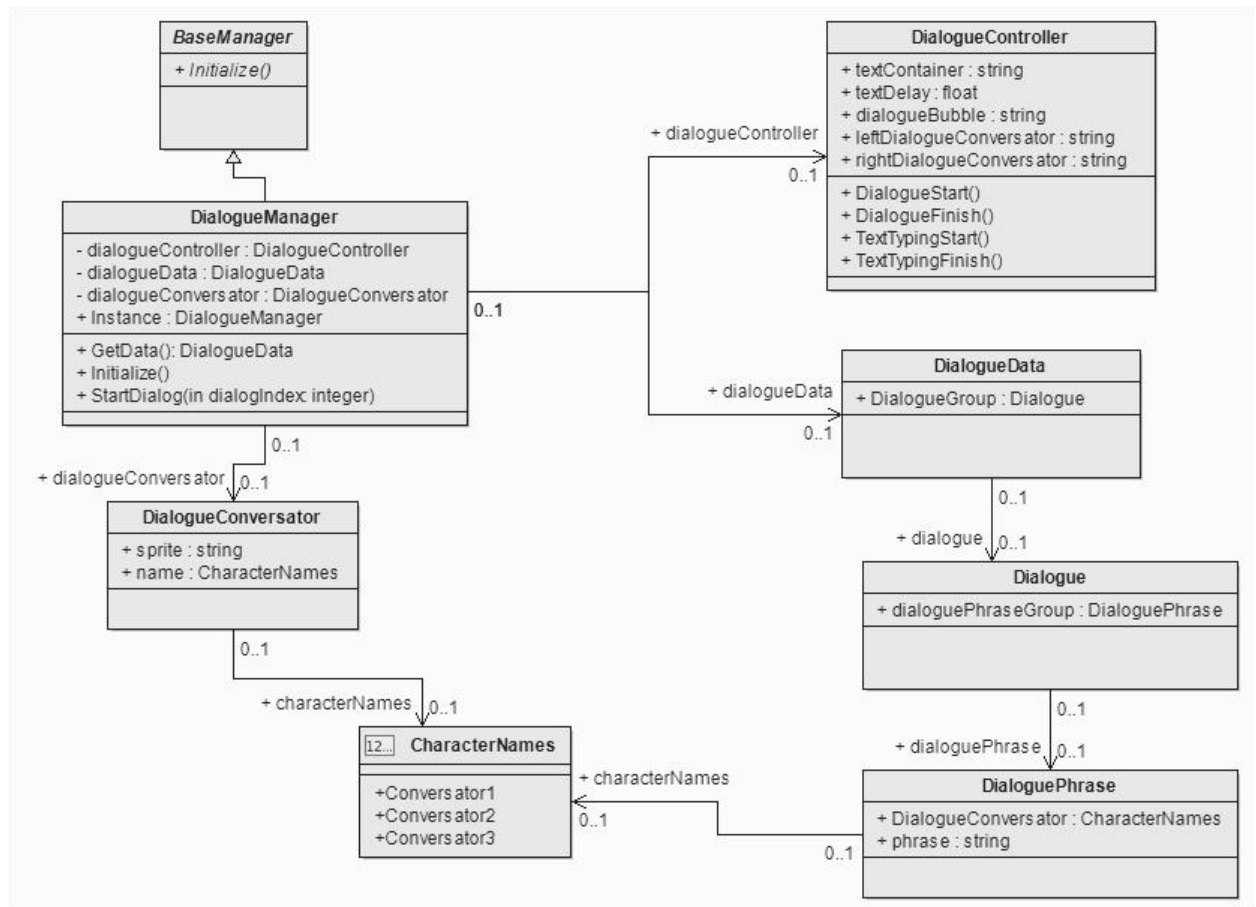


Рисунок 3.3 – Діаграма класів (Dialogue Manager)

Game Manager — координує основний ігровий процес та управляє станами гри. Цей менеджер є центральним елементом системи і виступає своєрідною точкою входу в програму, контролюючи логіку запуску, паузи та завершення гри (рис. 3.4).

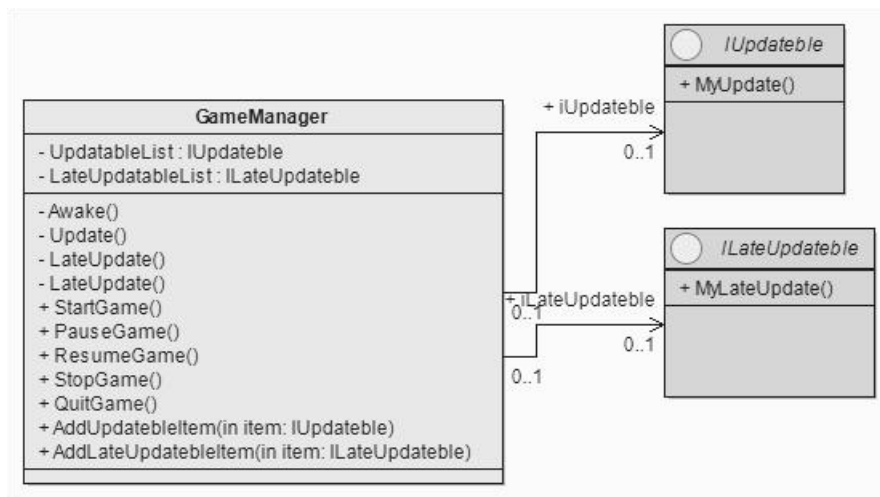


Рисунок 3.4 – Діаграма класів (Game Manager)

Input Manager — відповідає за обробку вводу гравця. У його функціонал входить налаштування дій на певні клавіші чи контролери, що дозволяє гравцю взаємодіяти з грою у зручний та передбачуваний спосіб (рис. 3.5).

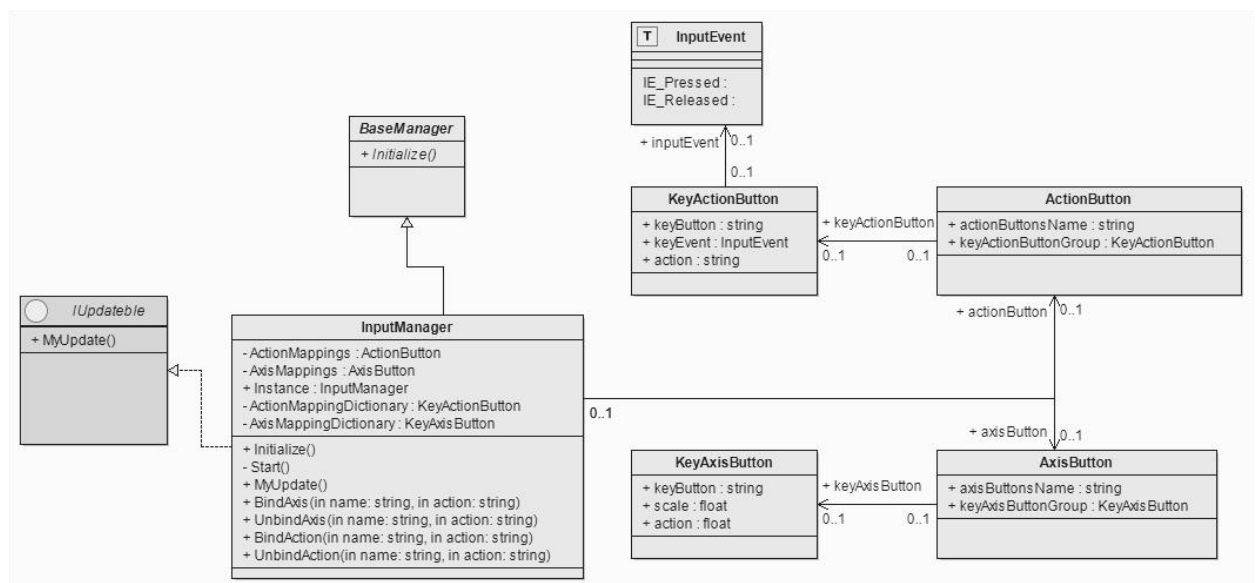


Рисунок 3.5 – Діаграма класів (Input Manager)

UI Manager — керує відображенням ігрових інтерфейсів. Він забезпечує відкриття, закриття або повернення до попереднього вікна, що дозволяє підтримувати послідовність та зручність навігації у меню та інших екранах гри (рис. 3.6).

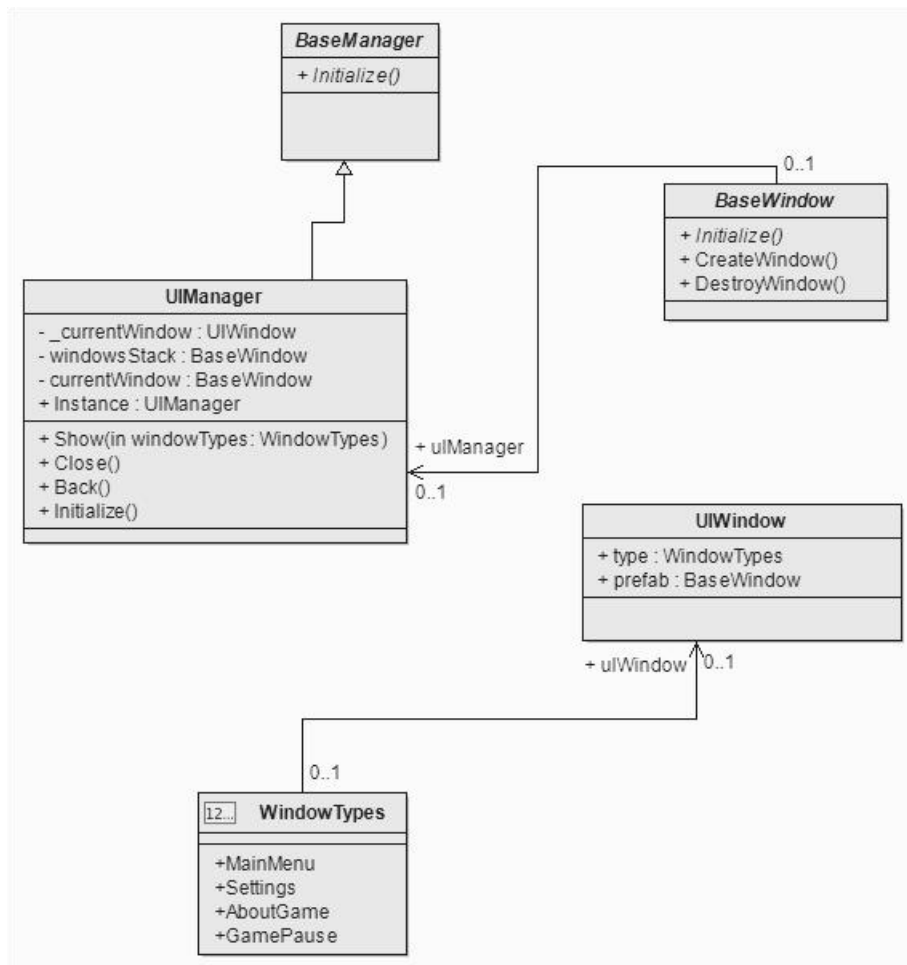


Рисунок 3.6 – Діаграма класів (UI Manager)

Діаграма послідовності (Sequence Diagram) відображає взаємодії між об'єктами системи у хронологічному порядку (рис. 3.7). Вона дозволяє наочно продемонструвати, які об'єкти беруть участь у виконанні певного процесу та у якій послідовності надходять повідомлення або виклики методів.

У даному випадку діаграма ілюструє життєвий цикл ігрового додатку: від запуску програми до основних дій користувача. Після старту додатку відображається головне меню, де користувач має можливість змінити налаштування, ознайомитися з інформацією про програму або розпочати процес гри. Діаграма демонструє, як взаємодіють об'єкти системи під час цих дій та як реалізується логічна послідовність подій.

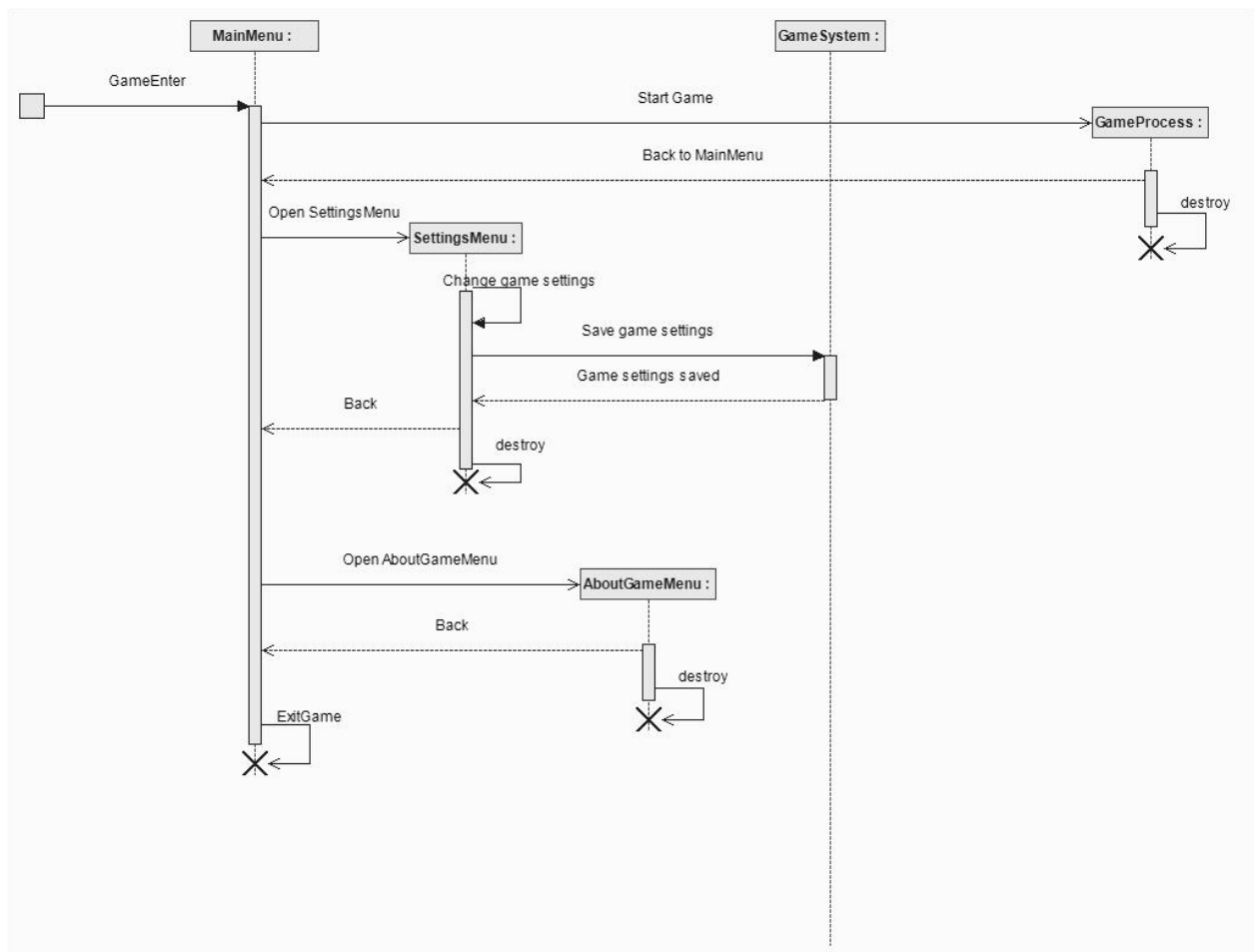


Рисунок 3.7 – Діаграма послідовності (Sequence diagram)

3.2 Визначення концепції

Гра “Маленькі пригоди в Балабурсі” відбувається у фантазійному середовищі, на території, відомій як «Смарагдовий ліс». Створена атмосфера сприяє зануренню та формує спокійний, водночас захопливий ігровий простір. Особлива увага приділяється темам відповідальності та наслідків прийнятих рішень, що реалізується через моральні дилеми, з якими стикається гравець під час проходження.

Сюжетна лінія розпочинається з викрадення головного персонажа надприродними істотами та зараження паразитом, який загрожує перетворенням на одного з них. Основним завданням персонажа є пошук способу позбавлення від паразита. У ході проходження Смарагдового лісу

користувач зустрічає різноманітні небезпеки та виклики, що формують сюжетну динаміку.

Ігровий процес ґрунтується на покроковій бойовій системі, адаптованій з правил Dungeons & Dragons (5—ї редакції). Такий підхід забезпечує стратегічну глибину боїв та розширює можливості тактичного планування.

Система діалогів реалізована у вигляді розгалуженої структури, де вибір користувача безпосередньо впливає на розвиток подій, взаємодію з іншими персонажами та стан ігрового світу (рис. 3.8). Це дозволяє формувати індивідуальні ігрові сценарії, підвищуючи інтерактивність та рівень залученості користувача.;



Рисунок 3.8 – Система діалогів з ігровими персонажами

Графічне оформлення гри “Маленькі пригоди в Балабурсі” виконане у стилі піксельної графіки, що надає проекту унікального візуального шарму та

викликає асоціації з класичними комп'ютерними іграми. Такий підхід підкреслює ретро—естетику та сприяє емоційному зануренню користувача у ігровий світ.

Інтерактивність ігрового середовища реалізована на високому рівні: користувач може взаємодіяти з різними об'єктами оточення, що впливає на перебіг подій та розвиток сюжетних ліній. Це дозволяє створити більш динамічний та гнучкий ігровий процес, де рішення та дії користувача безпосередньо формують розвиток світу.

Вплив системи Dungeons & Dragons проявляється у побудові сюжету та атмосфері гри. “Маленькі пригоди в Балабурсі” зберігає дух класичних D&D, пропонуючи користувачу зануритися в епічну фентезійну пригоду та брати участь у подіях, що поєднують рольові механіки та стратегічне планування.

3.3 Проєктування гри “Маленькі пригоди в Балабурсі”

Проєктування комп'ютерної гри є складним та багатогранним процесом, що охоплює концептуальну розробку, дизайн, програмування, тестування та низку інших аспектів. Метою цього підрозділу є надання огляду основних етапів і методів проєктування гри, а також розгляд ключових компонентів системи.

Концептуальний етап включає визначення основної ідеї гри, її жанру, ігрових механік та загального напрямку. На цьому етапі також визначається цільова аудиторія: вікова група користувачів, інтереси, очікування та потреби потенційних гравців.

Сеттинг та сюжетна лінія передбачають формування ігрового світу, його атмосфери, естетики та ключових елементів оточення. Сюжетна частина розробляється шляхом визначення основної історії, ключових подій, персонажів та конфліктів, що забезпечують інтерактивність та динаміку ігрового процесу.

Відкриття гри здійснюється через головне меню (рис. 3.9), яке слугує першим інтерфейсним елементом та відображає основні концептуальні положення проєкту.



Рисунок 3.9 – Ескіз головного меню гри

На початковому етапі створення сюжету здійснювалося дослідження популярної фантастичної літератури, комп’ютерних ігор та кінофільмів. Це дозволило визначити цікаві ідеї та концепції, які могли б слугувати основою для розробки захопливого сюжету гри.

На етапі формування ігрового середовища було визначено основні просторові та сюжетні елементи, що задають структуру світу гри. Світ “Маленькі пригоди в Балабурсі” представлено у вигляді центрального міста та прилеглих територій, серед яких ліси, печери та інші ключові локації. Кожна область має власну ідентичність, взаємодію з персонажами та завдання, що сприяють зануренню користувача у процес гри.

Для візуалізації просторової організації та логіки проходження гри було створено ескіз першого рівня (рис. 3.10), який демонструє основні

локації, маршрути пересування персонажа та розташування важливих об'єктів і завдань. Такий підхід дозволяє на ранньому етапі оцінити ігрову динаміку та планування першого рівня, забезпечуючи основу для подальшого детального проектування.

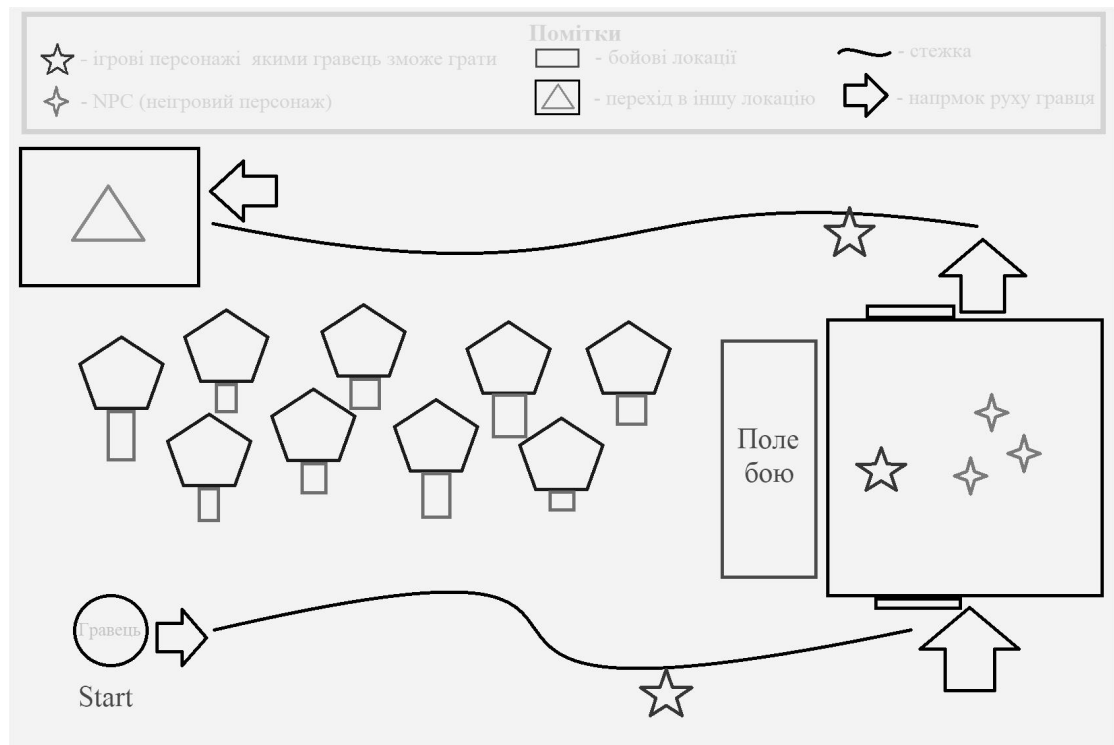


Рисунок 3.10 – Ескіз ігрової карти першого рівня

Процес розробки дизайну персонажа є комплексним і творчим етапом проектування гри, що вимагає врахування ряду концептуальних та функціональних аспектів. На початковому етапі визначається концепція персонажа: його роль у грі, особистісні характеристики, мотивації та взаємодія зі світом гри.

При створенні дизайну персонажів (рис. 3.11) враховувалися такі параметри: вік, стать, раса (з огляду на фентезійний контекст), стиль одягу, характер та поведінка персонажа, його роль у сюжеті, а також специфічні характеристики, включаючи історію та мотивацію. Особливу увагу приділено відповідності зовнішнього вигляду особистості персонажа, що дозволяє гравцеві одразу сприймати характер та роль героя. Крім того,

озброєння персонажів було підбрано відповідно до їхніх ігрових класів, що забезпечує логічність та узгодженість механік бою з візуальним оформленням.



Рисунок 3.11 – Ескізи персонажів

Особливу увагу під час проєктування приділено розробці сюжетної лінії гри. Концепція сюжету була частково заснована на механіках ігор у жанрі Dungeons & Dragons (D&D), де персонажі мають різні раси та класи, що впливає на їхні можливості та взаємодію у світі гри.

На початковому етапі сюжет формувався з урахуванням конфліктної складової та створення напруження, що дозволяє підтримувати зацікавленість користувача протягом проходження. Сюжет починається з передісторії, яка задає мотивацію головного персонажа та контекст для подальших подій. Для подальшого розвитку історії був створений

приблизний сценарій (рис. 3.12), що включає опрацьовані діалоги та позначки щодо ключових подій у текстовій і графічній формах.

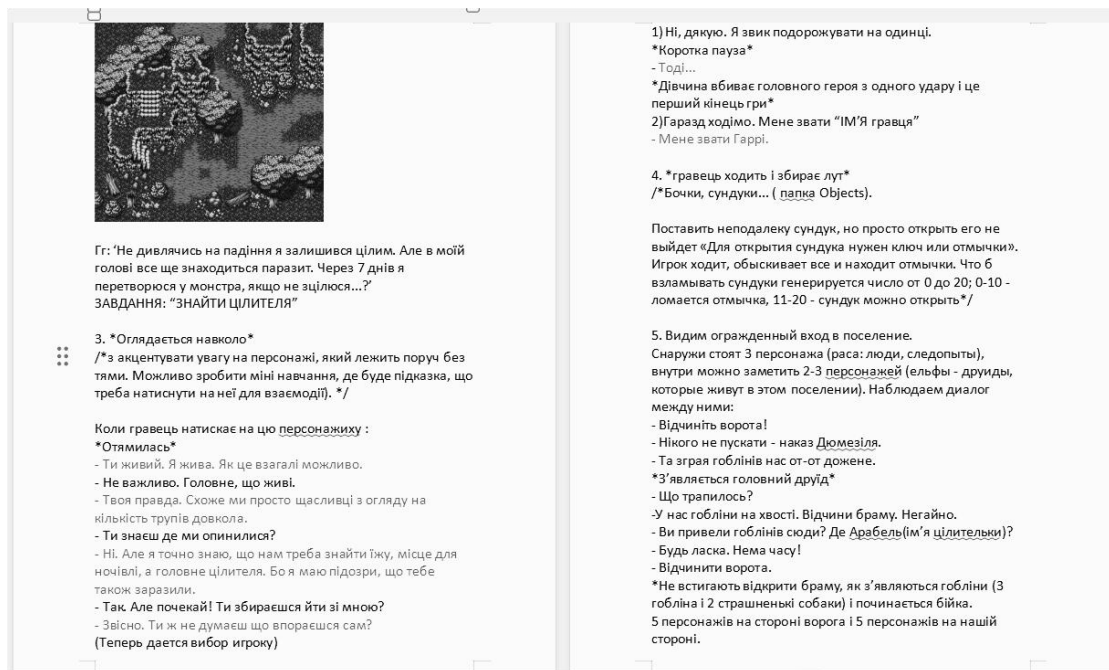


Рисунок 3.12 – Сценарна частина

3.4 Проектування першого ігрового рівня

Проектування першого рівня гри “Маленькі пригоди в Балабурсі” є багатоступеневим процесом, що охоплює кілька взаємопов’язаних етапів: концептуальний, ігрових механік, дизайну, сюжетний та технічний. Кожен етап має власні завдання та мету, спрямовані на створення цілісного ігрового досвіду.

Концептуальний етап передбачав визначення загального задуму рівня, включаючи локацію, атмосферу та сюжетні цілі. Початковий простір розташований у лісі поблизу поселення та задає напружений і загадковий тон подій. На цьому етапі визначалися основні сюжетні арки: головний персонаж дізнається про походження паразита, який його уражує, зустрічає ключових NPC та отримує завдання — знайти цілителя, здатного знищити паразита.

Додатковою метою є ознайомлення користувача з базовими механіками гри та взаємодією з персонажами.

Етап розробки ігрових механік включав інтеграцію навчальних елементів і взаємодії. Навчальні елементи реалізовані через контрольовані зустрічі з низькорівневими ворогами, що демонструють бойову систему. Взаємодія здійснюється через діалоги з персонажами, які знайомлять користувача з розгалуженою системою діалогів та можливістю формувати команду.

Дизайнерський етап передбачав розробку візуальних і звукових компонентів рівня. Лісова місцевість наповнена рослинністю, а персонажі й об'єкти підкреслюють тематику природи. Рівень містить кілька зон, кожна з яких має власні особливості та виклики. Музичний супровід адаптований до контексту подій — спокійні фонові теми для дослідження та інтенсивні бойові мотиви для сутичок.

Сюжетний етап передбачав створення текстового вступу, який пояснює передісторію зараження паразитом, та розробку зустрічей із ключовими NPC. Діалоги з персонажами дозволяють приймати рішення щодо формування команди та впливають на подальший розвиток сюжету. Бойові зіткнення включають низькорівневих ворогів для ознайомлення з механікою бою. Завершення рівня містить кульмінаційні події — рятування цілительки та поселення, зцілення головного персонажа і членів його команди, а також фінальну сцену подяки NPC.

Технічний етап включав вибір та налаштування ігрового двигуна Unity, програмування механік, фізики та інших систем, а також проведення тестування. Було здійснено альфа—тестування для перевірки основних механік, виявлення багів та балансування геймплею. У подальшому передбачено бета—тестування із залученням ширшої аудиторії для виявлення менш очевидних проблем і отримання зворотного зв'язку.

3.5 Висновки за розділом 3

У розділі 3 було виконано системне проєктування комп'ютерного додатку “Маленькі пригоди в Балабурсі”. Було визначено структуру та функціонал гри, опрацьовано взаємодію об'єктів за допомогою UML-діаграм, що дозволило формалізувати основні процеси та забезпечити логічну організацію системи.

Особливу увагу приділено розробці дизайну персонажів, що включає визначення раси, класу, зовнішнього вигляду, характеру та мотивації, що дозволяє забезпечити відповідність між сюжетом, механіками та візуальним оформленням.

Було створено сюжетну лінію на основі механік Dungeons & Dragons, що забезпечує інтригуючий конфлікт і мотивує користувача до проходження гри. На прикладі першого рівня продемонстровано послідовну реалізацію концептуальних, механічних, дизайнерських та сюжетних елементів, що включають навчальні завдання, взаємодію з NPC, бойові зіткнення та інтерактивні елементи.

На технічному етапі проведено налаштування ігрового двигуна Unity, реалізацію ігрових механік та тестування, що дозволило виявити та усунути технічні недоліки і забезпечити збалансований геймплей.

Цей розділ підтверджує, що комплексний підхід до проєктування гри дозволяє створити функціональний, інтерактивний та сюжетно завершений ігровий продукт.

4 РОЗРОБКА КОМП'ЮТЕРНОГО ДОДАТКУ “МАЛЕНЬКІ ПРИГОДИ В БАЛАБУРСІ”

4.1 Розробка сюжету гри

На початковому етапі розробки сюжету було проведено аналіз фантастичної літератури, комп'ютерних ігор та кінематографічних творів з метою пошуку цікавих ідей та концепцій, які могли б стати основою для захоплюючого ігрового сюжету.

Далі був створений уявний світ, у якому відбуваються події гри. Світ проєкту, названий “Маленькі пригоди в Балабурсі”, включає поселення Балабурс та його околиці — ліси, печери та інші локації. Він характеризується живою та різноманітною атмосферою, наповнений унікальними персонажами та завданнями. Було розроблено карту світу, визначено основні фракції, ключові локації, а також окреслено історію та культурні особливості.

При запуску гри на початковому екрані відображається 2D—ілюстрація лісу, виконана в піксельному стилі, що відповідає загальній графічній концепції гри. Назва проєкту “Маленькі пригоди в Балабурсі” розташована у верхній частині екрана, а в лівому нижньому куті — три кнопки для взаємодії з грою (рис. 4.1). Такий стиль початкового екрану створює єдину візуальну атмосферу та підкреслює відповідність інтерфейсу загальному оформленню гри.



Рисунок 4.1 – Головний екран гри

На початку ігрового процесу користувачу демонструється сюжетна передмова, яка слугує вступом до основної історії та визначає мотивацію подальших дій персонажа. Ця сцена подається не лише у вигляді візуального ряду, а й озвучується закадровим оповідачем, що підсилює емоційне занурення гравця та створює чітке розуміння контексту подій (рис. 4.2).

У сцені вступу розкривається інформація про інопланетних істот, які періодично викрадають людей з метою їх подальшої трансформації. Головний персонаж стає однією з жертв такого викрадення та приходить до тями у незнайомому органічному середовищі на борту інопланетного корабля. Оповідь супроводжує опис його фізичних відчуттів та емоційного стану: герой відчуває сильний біль та дезорієнтацію після контакту з паразитичним організмом, який використовують прибульці. Додаткового напруження додають звуки боротьби та поштовх, що вказують на можливу аварійну ситуацію на кораблі.

Ця вступна сцена формує загальну атмосферу проєкту, занурює користувача у художній світ та слугує основою для розуміння ключових подій і мотивів подальшої гри.



Рисунок 4.2 – Сцена вступу (передмова)

Після сцени вступу гравцю пропонується ввести ім'я персонажа, за якого він буде проходити гру, або скористатися варіантом, заданим за замовчуванням (рис. 4.3). Ім'я може бути як власним, так і довільно обраним.

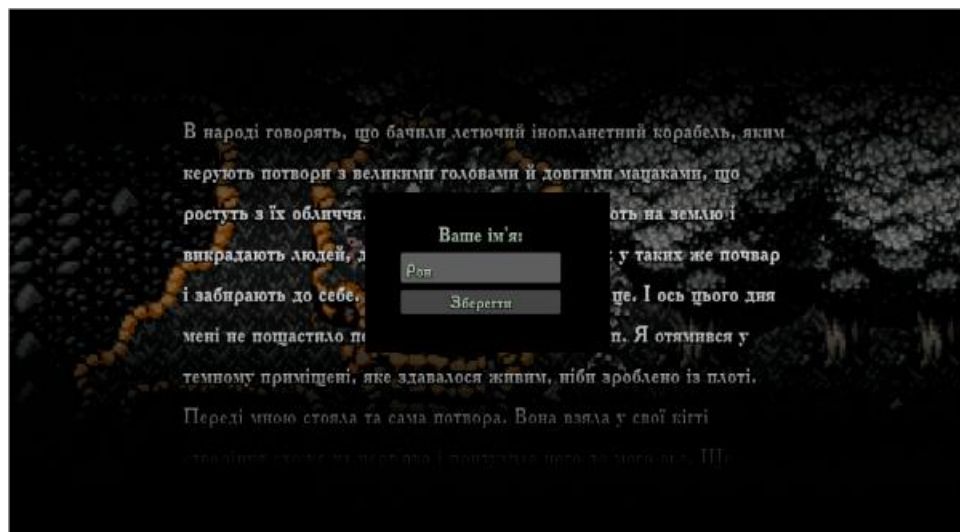


Рисунок 4.3 – Вікно введення імені ігрового персонажа

Одним із ключових елементів будь-якого ігрового проєкту є система персонажів. У рамках розробки було створено низку унікальних образів, з якими гравець взаємодіє впродовж проходження. Кожен із них має власну історію, мотивацію та характерні риси, що розкриваються у процесі гри. Окремі персонажі можуть приєднуватися до команди, надаючи гравцеві

можливість керувати їхніми діями. Загалом у грі реалізовано трьох таких союзників.

Першим потенційним напарником є Гаррі — представник вікінгського клану. Так само як і головного героя, його було викрадено інопланетними істотами, які імплантували в мозок паразитичний організм. Гаррі вдалося вижити після аварії корабля, і він вирізняється відвагою, рішучістю та майстерним володінням сокирою.

Другий персонаж, Артуро, — мешканець поселення Балабурс, що виконує обов'язки охоронця на одному з входів до селища. Дізнавшись про намір головного героя врятувати місцеву цілительку та захистити населення від небезпеки, Артуро добровільно пропонує свою допомогу. Хоча в демоверсії гри його характер розкрито лише частково, він постає як доброзичливий, відповідальний та турботливий захисник свого народу.

Третім союзником є Гусаріон — невеликий гусак з яскраво вираженим своєрідним характером. Він демонструє риси хитрості, самовпевненості та схильності до зухвалої поведінки. Його моральні принципи можуть бути неоднозначними, що робить персонажа потенційним антигероєм або союзником зі змінною вірністю.

Усі перелічені персонажі відіграють важливу роль у розвитку сюжетної лінії та взаємодії з головним героєм. Вони мають схожу долю — кожен із них став жертвою викрадення та носієм паразитичного організму — що об'єднує їх у спільній меті та мотивує допомагати у проходженні гри.

Загальна кількість союзників у грі становить три персонажі. Така кількість була обрана не випадково: формат команди з трьох компаньйонів є типовим для багатьох ігор, орієнтованих на групову взаємодію, оскільки забезпечує оптимальний баланс між різноманітністю ігрових ролей та зручністю керування. Занадто велика кількість союзників ускладнила б навігацію, тактичне планування та контроль за діями персонажів, тоді як менша — зменшила б варіативність бойових та сюжетних ситуацій. Три

активних компаньйони дають можливість реалізувати командну динаміку без перевантаження гравця надмірними механіками.

На основі зібраних ідей та концепцій був сформований основний наратив, який включає центральну сюжетну лінію, додаткові завдання (рис. 4.4) та низку виборів (рис. 4.5), що впливають на подальший розвиток подій. Основною метою було створення захопливого сюжету, здатного утримувати увагу гравця та забезпечувати цілісне сприйняття ігрового світу.

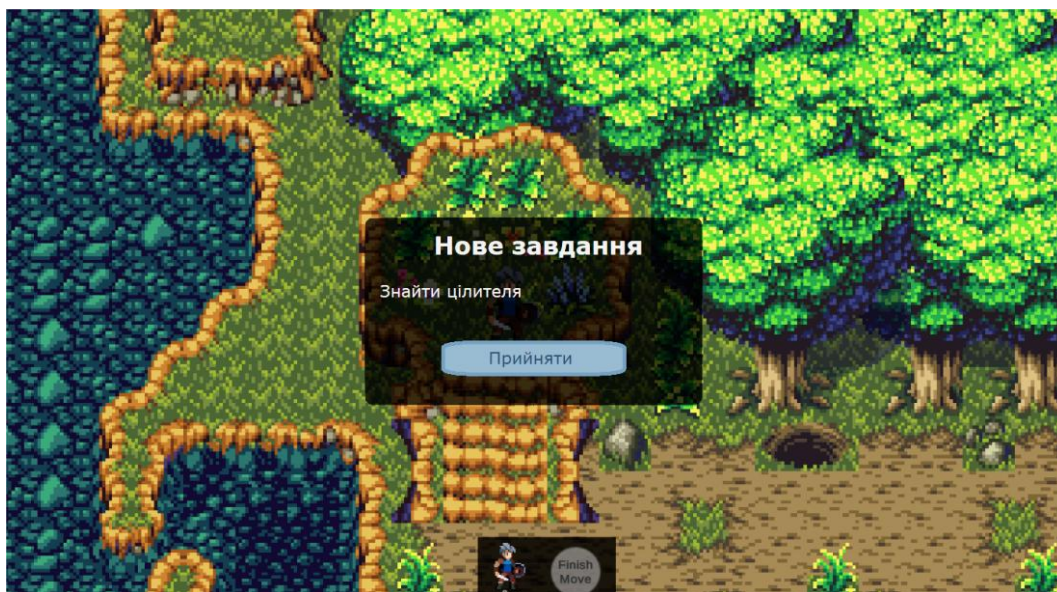


Рисунок 4.4 – Фрагмент реалізації квестової механіки

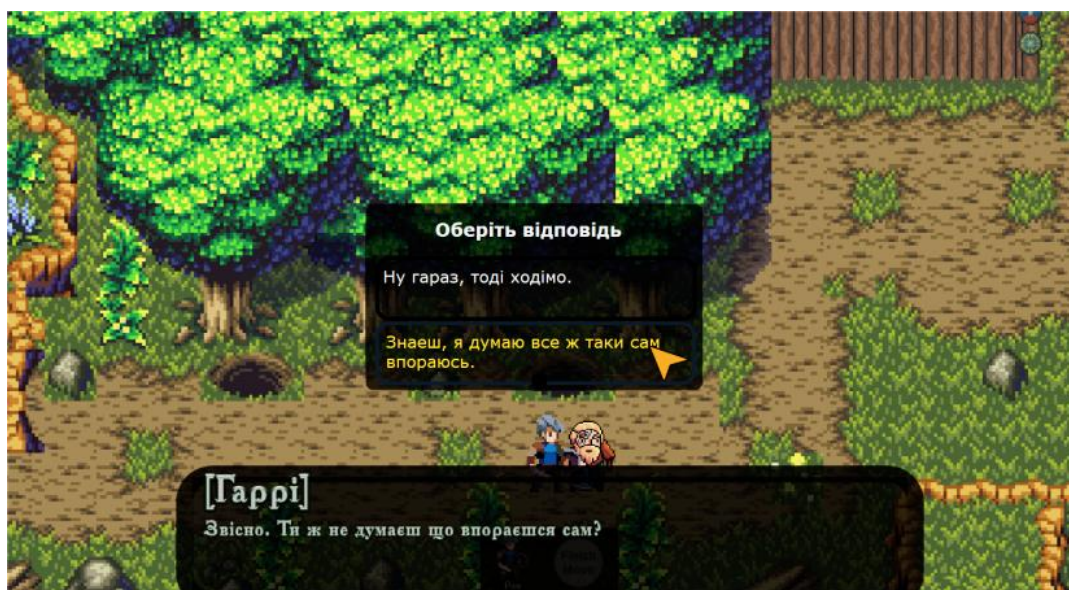


Рисунок 4.5 – Фрагмент механіки вибору з впливом на сюжет

Отримавши основне завдання та сформувавши команду союзників, гравець стикається з ключовим сюжетним рішенням: зосередитися виключно на власному порятунку чи взяти на себе відповідальність за допомогу іншим персонажам, які також постраждали від дій ворога. Такий вибір задає моральний та тактичний вектор подальшого проходження, адже кожне прийняте рішення впливає на стосунки з компаньйонами, доступність подій і навіть перебіг сюжетної лінії. Надалі на гравця чекають численні випробування, що поєднують елементи небезпеки, дослідження та взаємодії з оточенням, тому прийняте на початку рішення формує індивідуальний стиль гри та визначає можливі наслідки для розвитку історії.

4.2 Розробка дизайну ігрових локацій

У цьому підрозділі описується процес створення дизайну ігрового середовища, що є однією з ключових складових якісного ігрового проєкту. Розробка локацій (рис. 4.6) охоплює формування простору, у якому відбувається взаємодія гравця з віртуальним світом, а також визначення його елементів — об'єктів, рельєфу, архітектурних форм та територіальних зон.

Основні етапи роботи над локацією включали:

- формування загальної концепції середовища. Відповідно до жанрових особливостей подібних ігор, базовою темою було обрано “ліс”, оскільки в такому середовищі природні об'єкти та мешканці можуть виступати потенційною загрозою для персонажа;
- проєктування структури локацій. Для візуальної та змістової різноманітності були визначені дві ключові стилістичні зони — «лісова» та «поселення», які задають атмосферу ігрового світу;
- створення 2D—спрайтів та графічних елементів. Усі об'єкти локацій були промальовані у спеціалізованому редакторі PixelArt, що

дозволило забезпечити єдину стилістику та узгодженість графічних матеріалів;

– розміщення та інтеграція об'єктів у середовище. Фінальне компонування локацій виконувалося в кроссплатформовому рушії Unity, який забезпечив можливість точного розташування елементів і налаштування їх взаємодії в межах ігрового простору.



Рисунок 4.6 – Проєктування ігрового простору

4.3 Розробка ігрових персонажів та їх спрайтів

У цьому підрозділі описується процес створення ігрових спрайтів, що є невід'ємною частиною будь—якого комп'ютерного проєкту. Спрайти відтворюють персонажів і об'єкти ігрового середовища, формуючи основний візуальний досвід користувача.

Вони повинні відповідати загальній стилістиці та атмосфері гри, забезпечуючи узгодженість графічного оформлення. Для цього кожен

персонаж не лише має мати унікальний вигляд, але й володіє індивідуальним озброєнням або інструментом, що підкреслює його характер і роль у грі.

Створення спрайтів виконувалося за допомогою графічного редактора PixelArt, що дозволило досягти бажаного піксельного стилю та забезпечити єдність візуальної композиції (рис. 4.7).

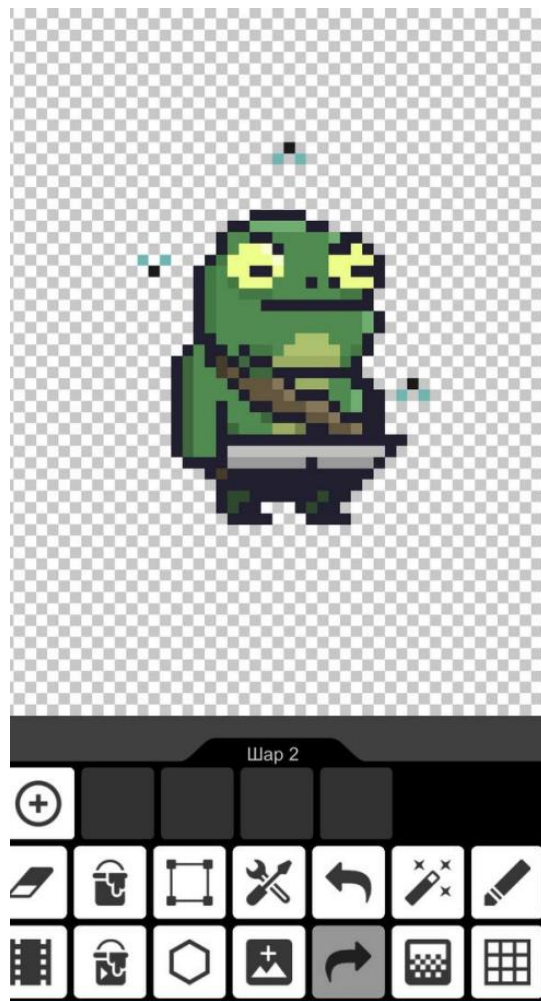


Рисунок 4.7 – Робота в програмі PixelArt

Першим спрайтом, з яким знайомиться гравець, є образ головного персонажа, за якого він керує в грі. Для передачі рис відваги, рішучості та готовності допомагати іншим герой було зображено у вигляді лицаря. Візуальне оформлення включає традиційні атрибути цього архетипу — великий меч та броню, що підкреслюють бойові здібності та статус персонажа у грі (рис. 4.8).



Рисунок 4.8 – Спрайт головного персонажа

У грі передбачено також персонажів—союзників, які допомагають гравцю протягом проходження. Кожен із них має власний характер та індивідуальний зовнішній вигляд, що відображає їхню роль у грі.

Кожен союзник має власну передісторію та мотивацію, з якою гравець може ознайомитися лише після приєднання персонажа до своєї ігрової команди. Такий підхід дозволяє поступово розкривати характери компаньйонів і створює додаткову мотивацію для взаємодії з ними, формуючи відчуття командної динаміки та глибини сюжету.

Зокрема, Артуро є корінним жителем поселення Балабурс. Його дизайн підкреслює характерні риси — невеликий зріст та зброю у вигляді лука, що відповідає його ролі охоронця та забезпечує впізнаваність персонажа серед інших (рис. 4.9).



Рисунок 4.9 – Спрайт ігрового персонажа

Гусаріон є персонажем із протиріччями у характері. Його милий та привабливий зовнішній вигляд свідомо контрастує з його непередбачуваною та зухвалою поведінкою, що підкреслює неоднозначність його моральних якостей та створює цікаву динаміку під час взаємодії з гравцем (рис. 4.10).

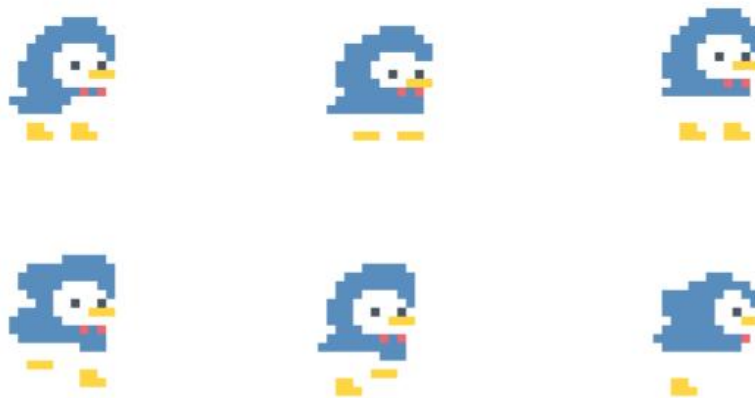


Рисунок 4.10 – Спрайт ігрового персонажа

Для втілення антагоністичних персонажів у грі було обрано образи жаби та гобліна. Ці персонажі отримали об'єднану назву «жабліни» і виконують роль ворогів, з якими гравець взаємодіє під час проходження (рис. 4.11).



Рисунок 4.11 – Спрайт антагонічного персонажа

4.4 Програмна реалізація комп'ютерного додатку

У цьому підрозділі розглядається функціонування основних менеджерів, реалізованих у проєкті Unity, а також їхні ключові методи, події та взаємодія зі скриптами гри. Менеджери забезпечують централізоване керування анімаціями, аудіо, даними, діалогами, ввідними подіями та логікою виконання завдань.

`AnimatorManager` – менеджер анімацій, що відповідає за керування аніматорами об'єктів та персонажів. Основні функціональні можливості включають:

- доступ до аніматорів за іменем, що визначається у переліку `enum`;
- створення структури об'єктів для катсцен, що забезпечує правильну організацію елементів анімації;
- створення нових анімацій, яке передбачає тимчасове відключення аніматора, додавання об'єкта до списку та призначення йому імені.

До ключових методів належать:

- `Play(enumAnimatorName)` – запуск анімації за вказаним іменем;
- `Pause(enumAnimatorName)` – призупинення анімації;

- `Stop(enumAnimatorName)` – зупинка анімації.

Менеджер також забезпечує роботу подій:

- `OnAnimationStarted` – викликається на початку анімації;
- `OnAnimationFinished` – спрацьовує після завершення анімації.

Для коректного відтворення анімацій використовується скрипт `AnimationBehaviour`, який підключається до об'єкта анімації та реалізує методи, що реагують на початок, завершення або активне виконання анімації.

`AudioManager` – менеджер аудіо, що відповідає за створення, відтворення, призупинення, зупинку та видалення звукових ефектів і музики.

Процес додавання аудіо включає:

- створення префабу `AudioSource`;
- додавання цього джерела до словника аудіо джерел;
- призначення аудіо кліпу (`AudioClip`).

Основні методи `AudioManager` забезпечують комплексне керування звуковими елементами гри:

- `Create (AudioIndex, isLoop, Vector3Pos)` – створює інстанцію звукового або музичного префабу за заданим індексом зі списку, розміщує її у визначеній точці на сцені та додає до словника `Dictionary<IDSound, Sound>`. Це дозволяє надалі керувати програванням, паузою, зупинкою або видаленням аудіо за його ID;

- `CreateAndPlay (AudioIndex, isLoop, Vector3Pos)` – аналогічно створює інстанцію префабу та додає її до словника, але одразу починає відтворення;

- `PlayOneShot (AudioIndex, Vector3Pos)` – створює інстанцію звуку або музики за індексом, відтворює її у зазначеній точці та автоматично видаляє після завершення відтворення;

- `Play (IDAudio, IDAudioSource, isLoop)` – призначає `AudioClip` зі словника певному аудіо джерелу, визначеному за ID, і запускає його відтворення;

- Play (IDAudio, AudioSource, isLoop) – аналогічно призначає AudioClip конкретному аудіо джерелу на сцені та запускає програвання;
- Pause (IDAudio) – тимчасово зупиняє відтворення звуку через аудіо джерело, знайдене за його ID у словнику;
- Pause (AudioSource) – ставить на паузу AudioClip через конкретне аудіо джерело на сцені;
- Resume (IDAudio) – відновлює відтворення звуку через аудіо джерело, визначене за ID у словнику;
- Resume (AudioSource) – відновлює відтворення AudioClip через конкретне аудіо джерело на сцені;
- Stop (IDAudio) – зупиняє відтворення аудіо за його ID у словнику;
- Stop (AudioSource) – зупиняє відтворення через зазначене аудіо джерело на сцені;
- Delete (IDAudio) – видаляє інстанцію аудіо джерела з словника та зі сцени за його ID;
- Delete (AudioSource) – видаляє об'єкт із компонентом AudioSource безпосередньо зі сцени.

Такий підхід забезпечує централізоване та гнучке керування аудіо в проєкті, дозволяючи ефективно організувати як фон, так і звукові ефекти гри.

DataManager – це менеджер для роботи з даними, який забезпечує функціонал збереження, завантаження та видалення інформації у проєкті. Основні методи включають:

- LoadData<T>(DataType dataType) – метод для завантаження збережених даних певного типу T з реєстру за вказаним ідентифікатором dataType;
- ResetData(DataType dataType) – метод, який дозволяє видалити збережені дані конкретного типу dataType;
- ResetAllData() – метод для повного очищення всіх збережених даних у системі.

Такий підхід дозволяє централізовано керувати даними, забезпечуючи безпечне збереження та відновлення інформації, а також легке видалення при необхідності.

Типи даних у проєкті визначаються за допомогою перерахування `DataType`. Наприклад, для збереження налаштувань гри застосовується скрипт `Settings`, який працює через цей механізм.

`DialogueManager` – це менеджер, відповідальний за організацію та відтворення діалогів між персонажами гри. Дані про діалоги зберігаються у класі `DialogueData`, який містить:

- репліки персонажів;
- інформацію про всіх учасників діалогу, включаючи їхні імена та спрайти.

`CharacterNames` – перерахування, яке містить імена всіх персонажів, що беруть участь у діалогах.

`DialogueController` – контролер, який відповідає за керування послідовністю та логікою діалогів у грі.

`FTextContainer` – елемент, у якому відображається текст діалогу під час його відтворення.

`FTextDelay` – параметр, що визначає затримку друкування окремих літер тексту, створюючи ефект поступового появи реплік.

`FDialogueTapPlace` – область екрану, на яку гравець може натиснути, щоб пропустити або прискорити відтворення тексту та перейти до наступної репліки.

`FLeftDialogueConversator` та `FRightDialogueConversator` – елементи, що відображають спрайти персонажів ліворуч і праворуч відповідно під час діалогу.

`FDialogueBubble` – область, що контролює відображення тексту та перевіряє, чи поміщається він у виділений простір. Якщо розмір шрифту зменшується, друкування тимчасово зупиняється і продовжується після натискання на `FDialogueTapPlace`.

Події, які обробляються DialogueManager:

- OnWriteTextStarted – викликається при початку друку тексту;
- OnWriteTextFinished – викликається після завершення друку тексту;
- OnDialogueStarted – спрацьовує при запуску діалогу;
- OnDialogueFinished – спрацьовує при завершенні діалогу.

GameManager – головний об'єкт управління, що служить точкою входу у програму. Завдяки параметру DontDestroyOnLoad він не видаляється під час завантаження нових сцен, забезпечуючи безперервне виконання всіх глобальних механізмів гри.

Цей об'єкт відповідальний за збереження списку всіх елементів, які повинні виконуватися у кожному кадрі гри. Для того, щоб скрипт виконувався щокадрово, він повинен реалізовувати інтерфейси IUpdatable або ILateUpdatable. У методі Awake відповідного скрипта слід додати його до GameManager за допомогою методів GameManager.AddUpdatableItem(IUpdatable item) або GameManager.AddLateUpdatableItem(ILateUpdatable item).

Основні події та методи GameManager включають:

- OnGamePaused – подія, що викликається при постановці гри на паузу;
- OnGameResumed – подія, що спрацьовує при знятті гри з паузи;
- StartGame() – статичний метод, який ініціює запуск гри;
- StopGame() – статичний метод, що відповідає за зупинку гри;
- QuitGame() – статичний метод, який використовується для виходу з гри.

InputManager – спрощена версія менеджера введення Unity, що відповідає за обробку дій користувача.

ActionMappings – структура, яка визначає кнопки та дії, що виконуються при їх натисканні або відпусканні.

oKeyActionButton:

- `keyEvent` – подія кнопки (натискання або відпускання);
- `keyButton` – кнопка, на яку поширюється подія;
- `action` – дія, яка виконується після взаємодії з кнопкою.

`oActionButton`:

- `actionButtonsName` – перерахування (enum), що визначає назву групи кнопок—дій (наприклад, "Interact");
- `keyActionButtonGroup` – список кнопок (`KeyActionButton`), на які підписуються події.

`AxisMappings` – перелік кнопок, які виконують прив'язану до них дію при натисканні та утриманні.

Змінні у `AxisMappings` працюють аналогічно кнопкам у `ActionMappings`, проте включають додатковий параметр `Scale`. Значення цієї змінної передається методу, підписаному на `Action<float>`, і дозволяє регулювати інтенсивність або напрямок дії.

Наприклад, для списку кнопок `MoveForward` можна додати клавіші `W` та `S` для руху вперед і назад. Значення `Scale` для `W` встановлено рівним 1, а для `S` – -1 . Метод, підписаний на дію цих кнопок, має вигляд `MoveForward(float scaleAmount)`, де параметр `scaleAmount` приймає значення 1 при утримуванні `W` та -1 при утримуванні `S`, забезпечуючи коректне керування рухом персонажа.

Методи, класи і подія:

- `BindAction(ActionButtonName name, Action action)` – метод, який підписує конкретну дію `action` на кнопку або набір кнопок з назвою `name`;
- `UnbindAction(ActionButtonName name, Action action)` – метод, який відписує дію `action` від кнопки або набору кнопок з назвою `name`.

`TaskManager` є центральним класом для управління завданнями в грі. Він зберігає список усіх активних завдань, точки їх спавну, а також відповідає за відображення інтерфейсного елементу `TaskPopup`, який дозволяє гравцеві прийняти або відхилити завдання.

TaskBase – базовий клас для створення нових завдань. Для додавання нового завдання необхідно успадкуватися від цього класу та реалізувати його методи. Клас забезпечує основний каркас, на основі якого можна створювати завдання різної складності та поведінки.

Подія OnDialogueFinished спрацьовує після завершення діалогу між персонажами, що дозволяє системі завдань та ігровій логіці продовжити свою роботу, наприклад, активувати наступне завдання або змінити стан персонажів.

GameManager є головним об'єктом гри та точкою входу в програму. Завдяки механізму DontDestroyOnLoad, він зберігається протягом усієї сесії гри.

GameManager також відповідає за виконання скриптів кожен кадр. Будь—який скрипт, який потребує регулярного оновлення, повинен реалізовувати інтерфейси IUpdatable або ILateUpdatable. У методі Awake відповідного скрипта потрібно додати його до GameManager за допомогою методів `AddUpdatableItem(IUpdatable item)` або `AddLateUpdatableItem(ILateUpdatable item)`.

Основні події та методи GameManager включають:

- `OnGamePaused` – викликається при постановці гри на паузу;
- `OnGameResumed` – викликається при відновленні гри з паузи;
- `StartGame()` – статичний метод для запуску гри;
- `StopGame()` – статичний метод для зупинки гри;
- `QuitGame()` – статичний метод для виходу з гри.

Ця структура забезпечує централізоване управління як завданнями, так і оновленням усіх активних компонентів гри, що є ключовою частиною організації ігрового процесу.

InputManager є спрощеним менеджером вводу, аналогічним Unreal Engine, який відповідає за обробку натискань і утримання кнопок гравця. Перелік кнопок, що виконують певну дію при натисканні або відпусканні, зберігається у ActionMappings. Кожен елемент цього списку містить

параметри кнопки: подію натискання чи відпускання (`keyEvent`), саму кнопку (`keyButton`) та дію (`action`), яку потрібно виконати при взаємодії з цією кнопкою. Додатково існує `oActionButton`, що групує кнопки за назвою дії (`actionButtonsName`) та зберігає список кнопок (`keyActionButtonGroup`), підписаних на відповідні події.

Для кнопок, які виконують дію при утриманні, використовується `AxisMappings`. Змінні у цьому списку працюють аналогічно `ActionMappings`, але мають додаткову змінну `Scale`, яка передає значення параметру методу, підписаного на `Action<float>`. Наприклад, створено список кнопок під назвою `MoveForward`, у якому визначені клавіші для руху вперед і назад: `W` та `S`. Значення змінної `Scale` встановлено відповідно `1` та `-1`. Метод, підписаний на `action` цих кнопок, має вигляд `MoveForward(float scaleAmount)`, де `scaleAmount` приймає значення `1` при утриманні клавіші `W` та `-1` при утриманні клавіші `S`.

Такий підхід дозволяє централізовано керувати всіма елементами управління, забезпечуючи можливість гнучкої прив'язки дій до кнопок та їх масштабування залежно від потреб ігрового процесу.

Методи `BindAction(ActionButtonName name, Action action)` та `UnbindAction(ActionButtonName name, Action action)` забезпечують прив'язку дій до кнопок та можливість відписки від них у будь-який момент. Перший метод підписує дію `action` на всі кнопки, що належать до групи під назвою `name`, тоді як другий метод дозволяє відписати цю дію від відповідних кнопок.

`TaskManager` є основним класом для управління завданнями в грі. Він зберігає список усіх активних завдань, точки їх спавну та відповідає за відображення інтерфейсного елементу `TaskPopup`, який дозволяє гравцеві приймати або відхиляти завдання.

`TaskBase` є базовим класом для створення нових завдань. Для додавання нового завдання необхідно успадкуватися від цього класу та реалізувати його методи. Завдяки цьому створюється основний каркас, на

основі якого можна формувати завдання різної складності та поведінки в ігровому процесі.

4.5 Висновки за розділом 4

Перш ніж розпочати активну розробку гри, було визначено загальну концепцію та структуру історії гри. Це включало створення фантазійного світу, де гравець міг би відчувати себе частиною епічного пригодницького шляху. Було опрацьовано головну сюжетну лінію, квести та взаємодію з персонажами. Сюжет гри включав елементи пригод та боротьби зі злом.

Під час розробки ігрового додатку також було детально опрацьовано систему менеджерів, що забезпечують функціонування різних аспектів гри. Це включало керування анімаціями, звуком, даними, діалогами, введенням гравця, а також управління завданнями та кадрами гри. Організація роботи менеджерів дозволяє централізовано контролювати різні процеси, забезпечувати взаємодію між об'єктами та створювати інтерактивне середовище, що є ключовим для стабільності, узгодженості ігрового процесу та зручності подальшого розвитку гри.

ВИСНОВКИ

У ході виконання магістерської дипломної роботи було проведено комплексне дослідження процесу розробки комп'ютерних ігрових додатків, що включало аналіз професійних ролей, технологій та методологій, застосовуваних у сучасній ігровій індустрії. Зокрема, визначено ключові категорії фахівців у розробці ігор: програмісти, художники, геймдизайнери та звукорежисери, а також уточнено їхні основні функціональні обов'язки, що забезпечують інтегровану роботу над створенням ігрового продукту.

Було систематизовано та описано етапи розробки комп'ютерних ігор, що включають формування концепції, прототипування, створення вертикального зрізу, виробництво, тестування та підготовку до релізу. Така поетапна структура дозволяє оптимізувати процес розробки та забезпечує ефективне управління ресурсами та часовими витратами на кожному етапі.

В межах практичної частини дослідження було розроблено комп'ютерну гру у жанрі 2D—пригоди з елементами RPG, використовуючи сучасні технологічні інструменти, зокрема Unity, Visual Studio та мову програмування C#. Розробка включала створення системи діалогів, набору ігрових завдань, 2D—графіки, анімацій персонажів та динамічних бойових механік. Для моделювання діалогів та архітектури проекту застосовувалися додаткові інструменти, такі як draw.io та Modelio, що забезпечило гнучку структуру програмного продукту.

Тестування гри показало відповідність розробленого продукту поставленим технічним завданням, коректну функціональність та завершеність ігрового процесу.

Проведене дослідження та практична реалізація демонструють ефективність комплексного підходу до розробки комп'ютерних ігор на основі сучасних технологій. Отримані результати підтверджують доцільність застосування методів системного проектування, інтердисциплінарної співпраці фахівців та сучасних програмних платформ у створенні якісних

ігрових продуктів. Ці висновки можуть бути використані для подальших наукових досліджень у галузі розробки ігор та оптимізації процесів створення програмних продуктів у сфері інтерактивних медіа.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Kent S. The Ultimate History of Video Games / S. Kent. — Нью—Йорк : Three Rivers Press, 2001. — 624 с.
2. Donovan T. Replay: The History of Video Games / T. Donovan. — Лондон : Yellow Ant, 2010. — 480 с.
3. The Routledge Companion to Video Game Studies / ed. : M. J. P. Wolf. — Нью—Йорк : Routledge, 2023. — 544 с.
4. Rogers S. Level Up! The Guide to Great Video Game Design / S. Rogers. — Вілей, 2014. — 520 с.
5. Wolf M. J. P. Video Game Studies Reader / M. J. P. Wolf. — New York : Routledge, 2015. — 600 p.
6. Milgram P. A Taxonomy of Mixed Reality Visual Displays / P. Milgram, F. Kishino // IEICE Transactions on Information Systems. — 1994. — Vol. E77—D. — P. 1321–1329.
7. Yannakakis G. Artificial Intelligence and Games / G. Yannakakis, J. Togelius. — Springer, 2018. — 310 p.
8. Five Key Game Development Stages: A Look Behind The Scenes [Electronic resource] – Access mode : <https://game—ace.com/blog/game—development—stages/>
9. Alves N. M. A Survey on Game Engines for Serious Games Development / N. M. Alves, R. Roque. — Y: Proceedings of the 8th International Conference on the Foundations of Digital Games (FDG), 2016. — 12 с.
10. Gregory J. Game Engine Architecture / J. Gregory. — 3-тє вид. — Бока—Ратон : CRC Press, 2018. — 1248 с.
11. Hollmann T. Development of an Entity Component System Architecture for Game Engines / T. Hollmann. — Abschlussarbeit, Universität Koblenz—Landau, 2018. — 72 с.
12. Unified Modeling Language (UML). Діаграми UML

[Електронний ресурс] // Вінницький національний технічний університет.
— Режим доступу : <https://iq.vntu.edu.ua/repository/getfile.php/8075.pdf>

13. Діаграми UML: загальний опис та приклади [Електронний ресурс] // Evergreens: статті про IT та розробку. — Режим доступу :
<https://evergreens.com.ua/ua/articles/uml—diagrams.html>

14. Навіщо застосовується діаграма класів [Електронний ресурс] // NATINC — Інформаційні технології. — Режим доступу :
<https://natinc.jek.v.ua/articles/navishho—zastosovuetsja—diagrama—klasiv.html>

ДОДАТОК А
Текст програми

Level1EntryPoint.cs — скрипт, що виконує роль точки входу для першого ігрового рівня. Він ініціалізує ключові компоненти сцени, налаштовує початкові параметри ігрового середовища та забезпечує послідовний запуск основних систем гри. Даний компонент відповідає за підготовку рівня до взаємодії з гравцем

```
using System.Collections;
    using Core.InputManagerLogic;
    using Core.GameManagerLogic;
    using Core.DialogueManagerLogic;
    using Core.TaskManagerLogic.Core;
    using UnityEngine;
    using UnityEngine.SceneManagement;

    public class Level1EntryPoint : MonoBehaviour
    {
        [SerializeField] private TaskManager taskManager;
        [SerializeField] private GameLogic.FightLogic fightLogic;
        [SerializeField] private GameLogic.HUDController HUD;

        //Player who attacking now
        [HideInInspector] public GameLogic.Actors.PlayerActor
CurrentPlayerActor;
        //Enemy who attacking now
        [HideInInspector] public GameLogic.Actors.EnemyActor CurrentEnemyActor;

        [HideInInspector] public GameLogic.Actors.PlayerActor
TempCurrentPlayerActor;
        [HideInInspector] public GameLogic.Actors.EnemyActor
TempCurrentEnemyActor;

        //Player who has been attacked by enemy
        [HideInInspector] public GameLogic.Actors.PlayerActor
SelectedPlayerActor;
        //Enemy who has been attacked by player
        [HideInInspector] public GameLogic.Actors.EnemyActor SelectedEnemyActor;

        public int movesCount = 0;
        [HideInInspector] public bool CanAttackEnemy = false;

        public static Level1EntryPoint Instance;
```

```

void Start()
{
    if (Instance == null) Instance = this;
    Core.AudioManager.AudioManager.FInstance.CreateAndPlay(3, true,
Vector3.zero);

    //GameManager.FInstance.AddUpdatableItem(this);
    CurrentPlayerActor =
GameLogic.Actors.ActorTypes.MainPlayerActor.Instance;
    Debug.LogWarning($"ITS MEEEE
{Level1EntryPoint.Instance.CurrentPlayerActor.gameObject.name}");
    HUD.Initialize();
    HUD.PlayerHealthUpdate();
    taskManager.Initialize();

    //CurrentActor.UnlockMove();
    //TaskManager.FInstance.SpawnTask();
    //TaskManager.FInstance.SetCurrentTask(0);
    //TaskManager.FInstance.StartTask();
    //Cursor.lockState = CursorLockMode.Locked;
    InputManager.FInstance.BindAction(ActionButtonName.GamePause,
        GameManager.FInstance.PauseResumeGame);
    //DialogueManager.FInstance.StartDialog(0, DialogueStart,
DialogueEnd);

    StartCoroutine(MyUpdate());
}

private void DialogueStart()
{
    Debug.Log("DialStart");
}
private void DialogueEnd()
{
    Debug.Log("DialEnd");
}
void OnDestroy()
{
    InputManager.FInstance.UnbindAction(ActionButtonName.GamePause,
        GameManager.FInstance.PauseResumeGame);
    //GameManager.FInstance.RemoveUpdatableItem(this);
}

public IEnumerator MyUpdate()
{

```

```

while (true)
{
    Debug.Log("ENEMY NULL");
    if (SelectedEnemyActor != null)
    {
        var moveDirection =
SelectedEnemyActor.transform.position - CurrentPlayerActor.transform.position;
        Debug.Log("ENEMY NOT NULL");
        var distance =
Vector2.Distance(CurrentPlayerActor.transform.position,
SelectedEnemyActor.transform.position);

        if (distance > 1.3f)
        {
            Debug.Log("PLAYER MOVE");
            CurrentPlayerActor.anim.SetBool("Run", true);
            CurrentPlayerActor.transform.position =
Vector2.MoveTowards(CurrentPlayerActor.transform.position,
SelectedEnemyActor.transform.position, CurrentPlayerActor.moveSpeed *
Time.deltaTime);

            if (moveDirection.x > 0)
CurrentPlayerActor.characterSprite.flipX = CurrentPlayerActor.IsInvertedSpriteFlip;
            if (moveDirection.x < 0)
CurrentPlayerActor.characterSprite.flipX = !CurrentPlayerActor.IsInvertedSpriteFlip;
        }
        else
        {
            Debug.Log("PLAYER ATTACK");
            CurrentPlayerActor.anim.SetBool("Run", false);
            CurrentPlayerActor.anim.SetTrigger("Action");
            //yield return new WaitForSeconds(.3f);

            CurrentPlayerActor.PerformAction(SelectedEnemyActor);
            TempCurrentEnemyActor = SelectedEnemyActor;
            SelectedEnemyActor = null;
            TempCurrentPlayerActor = CurrentPlayerActor;
            CurrentPlayerActor = null;
        }
    }
    else if (SelectedPlayerActor != null)
    {
        Debug.Log("PLAYER NOT NULL");
        var moveDirection =
SelectedPlayerActor.transform.position - CurrentEnemyActor.transform.position;

```

```

        var distance =
Vector2.Distance(CurrentEnemyActor.transform.position,
SelectedPlayerActor.transform.position);

        if (distance > 2f)
        {
            Debug.Log("ENEMY MOVE");
            CurrentEnemyActor.anim.SetBool("Run", true);
            CurrentEnemyActor.transform.position =
Vector2.MoveTowards(CurrentEnemyActor.transform.position,
SelectedPlayerActor.transform.position, CurrentEnemyActor.moveSpeed *
Time.deltaTime);

            if (moveDirection.x > 0)
CurrentEnemyActor.characterSprite.flipX = CurrentEnemyActor.IsInvertedSpriteFlip;
            if (moveDirection.x < 0)
CurrentEnemyActor.characterSprite.flipX = !CurrentEnemyActor.IsInvertedSpriteFlip;
        }
        else
        {
            if (moveDirection.x > 0)
CurrentEnemyActor.characterSprite.flipX = CurrentEnemyActor.IsInvertedSpriteFlip;
            if (moveDirection.x < 0)
CurrentEnemyActor.characterSprite.flipX = !CurrentEnemyActor.IsInvertedSpriteFlip;
            Debug.Log("ENEMY ATTACK");
            CurrentEnemyActor.anim.SetBool("Run", false);
            CurrentEnemyActor.anim.SetTrigger("Action");
            //yield return new WaitForSeconds(.3f);

            CurrentEnemyActor.PerformAction(SelectedPlayerActor);
            SelectedPlayerActor = null;
            CurrentEnemyActor = null;
            fightLogic.SelectPlayer();
        }
    }
    yield return null;
}
}
}

```

CameraController.cs — скрипт відповідає за керування позиціюванням та рухом камери під час гри. Він забезпечує динамічне відстеження гравця або інших об'єктів сцени, створюючи плавне й інформативне візуальне

відображення ігрового процесу. Контролер може містити алгоритми слідування, обмеження руху камери та налаштування кута огляду.

```
using System;
using UnityEngine;
using Core.GameManagerLogic;

namespace GameLogic
{
    class CameraController : MonoBehaviour, IUpdateble
    {
        private Vector3 newPos = Vector3.zero;
        public bool IsCameraFollowsPlayer;
        public static CameraController Instance;
        private void Awake()
        {
            if (Instance == null) Instance = this;
            IsCameraFollowsPlayer = true;
            GameManager.FInstance.AddUpdatableItem(this);
        }
        private void OnDestroy()
        {
            GameManager.FInstance.RemoveUpdatableItem(this);
        }

        public void MyUpdate()
        {
            Debug.Log(IsCameraFollowsPlayer);
            if (!IsCameraFollowsPlayer) return;
            newPos =
Actors.ActorTypes.MainPlayerActor.Instance.transform.position;
            newPos.z = -10;
            newPos.x = Mathf.Clamp(newPos.x, -7, 13);
            newPos.y = Mathf.Clamp(newPos.y, -1, 16);
            transform.position = newPos;
        }
    }
}
```

HUDController.cs — компонент, що керує елементами інтерфейсу користувача (HUD). Він оновлює інформацію, пов'язану зі станом персонажа, ігровими ресурсами або подіями, забезпечуючи гравця актуальними

візуальними даними. Скрипт також може відповідати за активацію або приховування окремих UI—блоків залежно від ситуації у грі.

```
using System;
using System.Collections.Generic;
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

namespace GameLogic
{
    public class HUDController:MonoBehaviour
    {
        [SerializeField] private Slider playerHealthSlider;
        [SerializeField] private FightLogic fightLogic;
        public Button endMoveButton;
        public TMPro.TextMeshProUGUI playerName;
        public Image playerIcon;
        [SerializeField] private UI.HUDTeamPlayer HUDTeamPlayerPrefab;
        [SerializeField] private Transform HUDTeamPlayerGroup;
        public List<UI.HUDTeamPlayer> HUDTeamPlayers = new
List<UI.HUDTeamPlayer>();
        public static HUDController Instance;
        public void SetPlayerName(string name) => playerName.text = name;

        public void Initialize()
        {
            if (Instance == null) Instance = this;
            endMoveButton.onClick.AddListener(EndMove);
        }

        private void EndMove()
        {
            Level1EntryPoint.Instance.movesCount = 0;
            Level1EntryPoint.Instance.CurrentEnemyActor =
Level1EntryPoint.Instance.TempCurrentEnemyActor;
            Level1EntryPoint.Instance.SelectedPlayerActor =
Level1EntryPoint.Instance.TempCurrentPlayerActor;

            if (fightLogic.IsAllEnemiesDead())
            {
                fightLogic.OnEnemiesDead?.Invoke();
                Debug.LogWarning("ALL ENEMIES DEAD");
                return;
            }
        }
    }
}
```

```

        }
        //fightLogic.SelectPlayer();
    }

    public void PlayerHealthUpdate()
    {
        playerHealthSlider.value =
Level1EntryPoint.Instance.CurrentPlayerActor.health /
Level1EntryPoint.Instance.CurrentPlayerActor.maxHealth;
        Debug.LogWarning(playerHealthSlider.value);
    }
    public void PlayerInfoUpdate()
    {
        playerName.text =
Level1EntryPoint.Instance.CurrentPlayerActor.info.actorName;
        playerIcon.sprite =
Level1EntryPoint.Instance.CurrentPlayerActor.info.actorIcon;
        Debug.LogWarning(playerHealthSlider.value);
    }

    public void AddTeamUserToHUD(Sprite userSprite, string userName)
    {
        var hudTeamPlayer = Instantiate(HUDTeamPlayerPrefab,
HUDTeamPlayerGroup);

        hudTeamPlayer.SetInfo(userSprite, userName);
        HUDTeamPlayers.Add(hudTeamPlayer);
    }
}
}

```

ObjectClick.cs — скрипт, що реалізує систему взаємодії користувача з об'єктами сцени. Він обробляє події натискання мишею, визначає обраний об'єкт і виконує відповідні дії залежно від його типу. Даний компонент є важливим для реалізації інтерактивності і дозволяє гравцю взаємодіяти з ігровими елементами.

```

using System;
using UnityEngine;
using Core.GameManagerLogic;

namespace GameLogic

```

```

{
    public class ObjectClick : MonoBehaviour
    {
        private void Start()
        {
            //GameManager.FInstance.AddUpdatableItem(this);
        }
        private void OnDestroy()
        {
            //GameManager.FInstance.RemoveUpdatableItem(this);
        }

        public void MyUpdate()
        {
            if (Input.GetMouseButtonDown(0))
            {
                Vector2 ray =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
                RaycastHit2D hit= Physics2D.Raycast(ray,
Vector2.zero);

                if(hit.collider.GetComponent<Actors.EnemyActor>())
                {
                    Debug.Log("Enemy");
                }
            }
        }
    }
}

```

PlayerNameBlock.cs — скрипт, що відповідає за відображення та оновлення імені гравця в інтерфейсі. Він керує текстовими полями, а також забезпечує коректне відображення даних під час зміни стану гравця або переходу між сценами.

```

using System;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

namespace GameLogic
{

```

```

public class PlayerNameBlock:MonoBehaviour
{
    [SerializeField] private TMP_InputField nameInputField;
    [SerializeField] private Button saveNameButton;

    public Action OnNameSaved;

    private void Awake()
    {
        nameInputField.text = "Рон";
        OnNameSaved += () =>
        HUDController.Instance.SetPlayerName(nameInputField.text);
        OnNameSaved += () => gameObject.SetActive(false);
        OnNameSaved += () =>
        HUDController.Instance.gameObject.SetActive(true);
        saveNameButton.onClick.AddListener(OnNameSaved.Invoke);
        Actors.ActorTypes.MainPlayerActor.Instance.info.actorName
        = nameInputField.text;
        HUDController.Instance.PlayerInfoUpdate();
    }
}

```

StartGameCredits.cs — компонент, що відповідає за відтворення початкових титрів під час запуску гри. Він керує рухом текстових елементів, їх відображенням та переходами між екранними станами. Скрипт може включати керування анімацією, швидкістю прокрутки та взаємодією з іншими UI—компонентами.

```

using System;
using Core.TaskManagerLogic.Core;
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

namespace GameLogic
{
    public class StartGameCredits : VerticalMover
    {
        public bool IsStartGameCredits;
    }
}

```

```

[SerializeField] private BlackoutScreenController
blackoutScreenController;
//[SerializeField] private Image blackBackground;
//[SerializeField] private float blackoutSpeed;
[SerializeField] private GameObject creditImage;
[SerializeField] private PlayerNameBlock playerNameBlock;

//private Vector4 newColor;

private void Awake()
{
    if (IsStartGameCredits)
    {
        blackoutScreenController.OnScreenBlack +=
OnScreenBlackoutedStartGame;
        blackoutScreenController.OnScreenWhite +=
OnScreenWhiteoutedStartGame;
    }
    else
    {
        blackoutScreenController.OnScreenBlack +=
OnScreenBlackoutedEndGame;
    }
}

public override void FinishedMove()
{
    blackoutScreenController.newColor.w =
blackoutScreenController.blackBackground.color.a;
    blackoutScreenController.StartBlackout();
    //StartCoroutine(BlackoutScene());
    playerNameBlock.OnNameSaved +=
blackoutScreenController.StartWhiteout;
    //Destroy(transform.parent.gameObject);
}

private void OnScreenBlackoutedStartGame()
{
    creditImage.SetActive(false);
    playerNameBlock.gameObject.SetActive(true);
    Cursor.lockState = CursorLockMode.None;
    TaskManager.FInstance.SpawnTask();
    TaskManager.FInstance.SetCurrentTask(0);
}

```

```

        Actors.ActorTypes.MainPlayerActor.Instance.anim.SetTrigger("Idle");
        blackoutScreenController.OnScreenBlack -=
OnScreenBlackoutedStartGame;
    }
    private void OnScreenBlackoutedEndGame()
    {
        Core.AudioManager.AudioManager.FInstance.DeleteCurrent();
        blackoutScreenController.OnScreenBlack -=
OnScreenBlackoutedEndGame;
        UnityEngine.SceneManagement.SceneManager.LoadSceneAsync(0);
    }

    private void OnScreenWhiteoutedStartGame()
    {
        TaskManager.FInstance.CanShowPopup = false;
        TaskManager.FInstance.PreviewTask();

        Core.GameManagerLogic.GameManager.FInstance.RemoveUpdatableItem(this);
        blackoutScreenController.OnScreenWhite -=
OnScreenWhiteoutedStartGame;
        //Destroy(transform.parent.gameObject);
    }

    //public IEnumerator BlackoutScene()
    //{
    //    while (newColor.w < 1)
    //    {
    //        yield return new WaitForSeconds(0.1f);
    //        newColor.w += blackoutSpeed;
    //        blackBackground.color = newColor;
    //    }
    //    creditImage.SetActive(false);
    //    playerNameBlock.gameObject.SetActive(true);
    //    Cursor.lockState = CursorLockMode.None;
    //    TaskManager.FInstance.SpawnTask();
    //    TaskManager.FInstance.SetCurrentTask(0);
    //}

    Actors.ActorTypes.MainPlayerActor.Instance.anim.SetTrigger("Idle");
    //}

    //private IEnumerator WhiteoutScene()
    //{
    //    while (newColor.w > 0)

```

```

        //      {
        //          yield return new WaitForSeconds(0.1f);
        //          newColor.w -= blackoutSpeed;
        //          blackBackground.color = newColor;
        //      }
        //      TaskManager.FInstance.CanShowPopup = false;
        //      TaskManager.FInstance.PreviewTask();
        //      Destroy(transform.parent.gameObject);
        //}
    }
}

```

BlackoutScreenController.cs — скрипт, що реалізує затемнення екрана для створення плавних переходів між сценами або ігровими станами. Він керує прозорістю графічного елемента, забезпечуючи появу або зникнення ефекту чорного фону, що покращує візуальне сприйняття зміни подій у грі.

```

using System;
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class BlackoutScreenController : MonoBehaviour
{
    public Image blackBackground;
    public float blackoutSpeed;

    public Action OnScreenBlack;
    public Action OnScreenWhite;

    public Vector4 newColor;

    public void StartBlackout() => StartCoroutine(BlackoutScene());
    public void StartWhiteout() => StartCoroutine(WhiteoutScene());

    private IEnumerator BlackoutScene()
    {
        while (newColor.w < 1)
        {
            yield return new WaitForSeconds(0.1f);
            newColor.w += blackoutSpeed;
        }
    }
}

```

```

        blackBackground.color = newColor;
    }
    OnScreenBlack?.Invoke();
}

private IEnumerator WhiteoutScene()
{
    while (newColor.w > 0)
    {
        yield return new WaitForSeconds(0.1f);
        newColor.w -= blackoutSpeed;
        blackBackground.color = newColor;
    }

    OnScreenWhite?.Invoke();
}
}

```

StartFightTrigger.cs — компонент, що є тригером, що активує початок бойової сцени. Він відстежує вхід гравця в певну область та запускає логіку бою або відповідні анімації. Скрипт використовується для переходу від мирного режиму до бойового режиму гри.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class StartFightTrigger : MonoBehaviour
{
    [SerializeField] private BlackoutScreenController
blackoutScreenController;
    [SerializeField] private GameObject mainMap;
    [SerializeField] private GameObject fightMap;
    private void OnTriggerEnter2D()
    {
        blackoutScreenController.OnScreenBlack += OnBlackouted;
        blackoutScreenController.OnScreenWhite += OnWhiteouted;
        blackoutScreenController.StartBlackout();
        GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.LockMove();
    }
}

```

```

        private void OnBlackouted()
        {
            mainMap.SetActive(false);
            fightMap.SetActive(true);

            Core.GameManagerLogic.GameManager.FInstance.RemoveUpdatableItem(GameLogic.CameraController.Instance);
            GameLogic.CameraController.Instance.transform.position = new
            Vector3(53, 13, -10);
            blackoutScreenController.StartWhiteout();
        }
        private void OnWhiteouted()
        {
            Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(7, null, ()
            => Level1EntryPoint.Instance.CanAttackEnemy = true);

            blackoutScreenController.OnScreenBlack -= OnBlackouted;
            blackoutScreenController.OnScreenWhite -= OnWhiteouted;

            //blackoutScreenController.StartWhiteout();
        }
    }
}

```

FightLogic.cs — скрипт, що містить основну логіку бойової системи. Він визначає поведінку супротивників, обробляє атаки, вираховує отриману шкоду та контролює послідовність бойових дій. Даний компонент є ключовим для реалізації механіки бою та взаємодії між персонажами.

```

using System;
using System.Collections.Generic;
using System.Collections;
using UnityEngine;

namespace GameLogic
{
    class FightLogic : MonoBehaviour
    {
        [SerializeField] private Actors.EnemyActor EnemyPrefab;
        [SerializeField] private int EnemiesOnScene = 3;
    }
}

```

```

[SerializeField] private List<Transform> EnemiesTransforms = new
List<Transform>();
[SerializeField] private List<Vector3> PlayersPositions = new
List<Vector3>();
[SerializeField] private Transform[] VillagePlayerPositions;
[SerializeField] private Transform VillageHealerPosition;
[SerializeField] private BlackoutScreenController blackoutScreen;
[SerializeField] private GameObject cage;
[SerializeField] private GameObject endCredits;
[SerializeField] private GameObject endScene;
[SerializeField] private GameObject fightScene;
[SerializeField] private GameObject mainMapScene;
[SerializeField] private Actors.ActorTypes.HealerActor
healerActor;

[SerializeField] private GameObject magic;
[SerializeField] private Vector3[] playerMoveToCage;
[SerializeField] private Animator arabelMagicOutCage;

public List<Actors.PlayerActor> players = new
List<Actors.PlayerActor>();
private List<Actors.EnemyActor> enemies = new
List<Actors.EnemyActor>();

private int currentPlayerID = -1;
private int currentEnemyID = -1;

public Action OnPlayerSelected;
public Action OnEnemySelected;

public Action OnPlayersDead;
public Action OnEnemiesDead;

public static FightLogic Instance;

private void Start()
{
    if (Instance == null) Instance = this;
    Core.AudioManager.AudioManager.FInstance.DeleteCurrent();
    Core.AudioManager.AudioManager.FInstance.CreateAndPlay(0,
true, Vector3.zero);

    HUDController.Instance.endMoveButton.interactable = true;
    foreach (var hudPlayer in HUDController.Instance.HUDTeamPlayers)
    {

```

```

        hudPlayer.SetMeInactive();
    }

    Actors.ActorTypes.MainPlayerActor.Instance.IsMyTeamFollowsMe = false;
    players.Add(Actors.ActorTypes.MainPlayerActor.Instance);
    foreach (var teamPlayer in
Actors.ActorTypes.MainPlayerActor.Instance.myTeam.Values)
    {
        teamPlayer.anim.SetBool("Run", false);
        players.Add(teamPlayer);
    }

    for (int i = 0; i < players.Count; i++)
    {
        players[i].transform.position = PlayersPositions[i];
    }

    for (int i = 0; i < EnemiesOnScene; i++)
    {
        var enemyObject = Instantiate(EnemyPrefab,
EnemiesTransforms[i].position, Quaternion.identity);
        enemyObject.IsHealthDisplay = true;
        enemyObject.healthText.text =
    $"{enemyObject.maxHealth}";
        //enemyObject.transform.localScale =
    EnemiesTransforms[i].localScale;
        enemies.Add(enemyObject);
    }
    SelectPlayer();
}

private bool IsAllPlayersDead()
{
    var playerID = 0;
    var deadPlayersCount = 0;

    while (playerID < players.Count)
    {
        if (players[playerID].health <= 0)
            deadPlayersCount++;
        playerID++;
    }
    if (deadPlayersCount == players.Count) return true;
}

```

```

        return false;
    }

    private IEnumerator PlayerEndMoveToCage()
    {
        arabelMagicOutCage.SetTrigger("OutFromCage");
        yield return new WaitForSeconds(2);

        Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(8, null,
FightEnd);

        Actors.ActorTypes.MainPlayerActor.Instance.OnPathFinished
-= ()=>StartCoroutine(PlayerEndMoveToCage());
    }

    public void SelectPlayer()
    {
        if (IsAllEnemiesDead())
        {
            Debug.LogWarning("ENEMIES DEAD");

            Core.AudioManager.AudioManager.FInstance.DeleteCurrent();

            Core.AudioManager.AudioManager.FInstance.CreateAndPlay(1, true, Vector3.zero);
            HUDController.Instance.gameObject.SetActive(false);
            //cage.SetActive(false);

            Actors.ActorTypes.MainPlayerActor.Instance.LockMove();

            StartCoroutine(Actors.ActorTypes.MainPlayerActor.Instance.MovePath(playerMove
ToCage));

            Actors.ActorTypes.MainPlayerActor.Instance.OnPathFinished +=
()=>StartCoroutine(PlayerEndMoveToCage());

        }

        if (currentPlayerID > 0)
        {
            HUDController.Instance.HUDTeamPlayers[currentPlayerID-1].SetMeInactive();

        }

        if (IsAllPlayersDead()) {

```

```

        OnPlayersDead?.Invoke();
        Debug.LogWarning("ALL PLAYERS DEAD");
        return;
    }

    do
    {
        currentPlayerID++;
        Debug.Log(currentPlayerID);

        var isCurrentPlayerOutOfRange = currentPlayerID >=
players.Count;

        if (isCurrentPlayerOutOfRange) currentPlayerID = 0;
        break;
    }
    while (players[currentPlayerID].health > 0);

    Level1EntryPoint.Instance.CurrentPlayerActor =
players[currentPlayerID];
    Debug.LogWarning($"ITS MEEEE
{Level1EntryPoint.Instance.CurrentPlayerActor.name}");
    if
(Level1EntryPoint.Instance.CurrentPlayerActor.gameObject.name!="Player")

        HUDController.Instance.HUDTeamPlayers[currentPlayerID-1].SetMeActive();
        OnPlayerSelected?.Invoke();
        HUDController.Instance.PlayerHealthUpdate();
        HUDController.Instance.PlayerInfoUpdate();
        Level1EntryPoint.Instance.CurrentPlayerActor.OnDied +=
RemovePlayer;

        Debug.Log("SELECTED PLAYER");
    }

    private void FightEnd()
    {
        blackoutScreen.StartBlackout();
        blackoutScreen.OnScreenBlack += ReturnToTheVillage;
    }

    private void ReturnToTheVillage()
    {
        //fightScene.SetActive(false);

```

```

Core.GameManagerLogic.GameManager.FInstance.AddUpdatableItem(CameraController.
Instance);

        mainMapScene.SetActive(true);
        endScene.SetActive(true);
        for (int i = 0; i < players.Count; i++)
        {
            players[i].transform.position =
VillagePlayerPositions[i].position;
        }
        healerActor.gameObject.SetActive(true);
        //HealerTransform.position =
VillageHealerPosition.position;
        blackoutScreen.StartWhiteout();
        blackoutScreen.OnScreenWhite += StartHealerVillageDialogue;
        blackoutScreen.OnScreenBlack -= ReturnToTheVillage;
    }
    private void StartHealerVillageDialogue()
    {

        Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(9, null,
        ()=>StartCoroutine(StartHealing()));
        blackoutScreen.OnScreenWhite -= StartHealerVillageDialogue;
    }

    private IEnumerator StartHealing()
    {

        healerActor.PerformAction(Actors.ActorTypes.MainPlayerActor.Instance);
        magic.SetActive(true);
        yield return new WaitForSeconds(2);

        Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(10, null,
        StartEndCredits);
    }

    private void StartEndCredits()
    {
        endCredits.SetActive(true);
    }

    public void RemovePlayer()
    {

```

```

        players.RemoveAt(currentPlayerID);
    }

    public bool IsAllEnemiesDead()
    {
        var enemyID = 0;
        var deadEnemiesCount = 0;

        while (enemyID < enemies.Count)
        {
            if (enemies[enemyID].health <= 0)
                deadEnemiesCount++;
            enemyID++;
        }
        if (deadEnemiesCount == enemies.Count) return true;
        return false;
    }

    public void SelectEnemy()
    {
        if (IsAllEnemiesDead()) {
            OnEnemiesDead?.Invoke();
            Debug.LogWarning("ALL ENEMIES DEAD");
            return;
        }

        do
        {
            currentEnemyID++;

            var isCurrentEnemyOutOfRange = currentEnemyID >=
enemies.Count;

            if (isCurrentEnemyOutOfRange) currentEnemyID = 0;
        }
        while (enemies[currentEnemyID].health <= 0);
        Level1EntryPoint.Instance.CurrentEnemyActor =
enemies[currentEnemyID];
        OnEnemySelected?.Invoke();
        Debug.Log("SELECTED ENEMY");
    }
}
}

```

VillageTrigger.cs — скрипт, що відповідає за активацію подій при вході гравця в зону села або іншу локацію. Він запускає діалоги, сюжетні завдання або зміни оточення. Компонент служить інструментом для керування переходами між сюжетними етапами.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class VillageTrigger : MonoBehaviour
{
    [SerializeField] private Vector3[] playerIntoVillagePath;
    [SerializeField] private Vector3[] playerToBoarPath;
    [SerializeField] private Vector3[] druidToGatesPath;
    [SerializeField] private Vector3[] enemyMovePathForward1;
    [SerializeField] private Vector3[] enemyMovePathForward2;
    [SerializeField] private Vector3[] enemyMovePathEscaping;
    [SerializeField] private Vector3[] other1MovePathHiding;
    [SerializeField] private Vector3[] other2MovePathHiding;
    [SerializeField] private Vector3[] ranger1MovePath;
    [SerializeField] private Vector3[] ranger2MovePath;
    [SerializeField] private Vector3[] ranger2MovePathBack;

    [SerializeField] private GameLogic.Actors.CharacterActor mainDruid;
    [SerializeField] private GameLogic.Actors.ActorTypes.KnightActor
rangerPrefab;
    [SerializeField] private GameLogic.Actors.CharacterActor other1Prefab;
    [SerializeField] private GameLogic.Actors.CharacterActor other2Prefab;
    [SerializeField] private GameLogic.Actors.EnemyActor enemyPrefab1;
    [SerializeField] private GameLogic.Actors.EnemyActor enemyPrefab2;

    [SerializeField] private Vector3 rangerAppearPoint1;
    [SerializeField] private Vector3 rangerAppearPoint2;

    [SerializeField] private Vector3 otherAppearPoint1;
    [SerializeField] private Vector3 otherAppearPoint2;

    [SerializeField] private Vector3 enemyAppearPoint1;
    [SerializeField] private Vector3 enemyAppearPoint2;

    [SerializeField] private Vector3 cavePoint;
```

```

private Transform targetCameraMovePoint;
private Transform mainCamera;
public float mainCameraMoveSpeed;
private Transform gates;
public Vector3 gatesEndPoint;
private GameLogic.Actors.CharacterActor mainDruidObject;
private bool IsEnemyEscaping;

private GameLogic.Actors.ActorTypes.KnightActor ranger1;
private GameLogic.Actors.ActorTypes.KnightActor ranger2;

private GameLogic.Actors.CharacterActor other1;
private GameLogic.Actors.CharacterActor other2;

private GameLogic.Actors.EnemyActor enemy1;

private int newOrderInLayer = 0;
private void Awake()
{
    gates = GameObject.FindGameObjectWithTag("Gates").transform;
    targetCameraMovePoint =
GameObject.FindGameObjectWithTag("TargetCameraMovePoint").transform;
    mainCamera = Camera.main.transform;
    ranger1 = Instantiate(rangerPrefab, rangerAppearPoint1,
Quaternion.identity);
    ranger2 = Instantiate(rangerPrefab, rangerAppearPoint2,
Quaternion.identity);
    ranger2.transform.GetChild(0).GetComponent<SpriteRenderer>().flipX =
true;

    other1 = Instantiate(other1Prefab, otherAppearPoint1,
Quaternion.identity);
    other2 = Instantiate(other2Prefab, otherAppearPoint2,
Quaternion.identity);
    other1.IsInvertedSpriteFlip = true;
    other2.IsInvertedSpriteFlip = true;
}

private void OnTriggerEnter2D(Collider2D collision)
{
    if
(collision.GetComponent<GameLogic.Actors.ActorTypes.MainPlayerActor>())
{

```

```

        //GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.LockMove();
        StartCoroutine(CutSceneVillage());
        GetComponent<BoxCollider2D>().enabled = false;
        Debug.Log("GGGG");
    }
}

private IEnumerator CutSceneVillage()
{
    GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.LockMove();
    GameLogic.CameraController.Instance.IsCameraFollowsPlayer = false;
    var newPos = targetCameraMovePoint.position;
    newPos.z = mainCamera.position.z;
    GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.OnPathFinished
+= PlayerEndEnterToVillage;

    StartCoroutine(GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.MovePath(playerI
ntoVillagePath));
    while (targetCameraMovePoint.position.y != mainCamera.position.y)
    {
        mainCamera.position = Vector3.MoveTowards(mainCamera.position,
newPos, mainCameraMoveSpeed * Time.deltaTime);
        yield return null;
    }
    Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(2,
null, MainDruidAppear);
}

private void PlayerEndEnterToVillage()
{
    GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.characterSprite.flipX = false;
    GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.OnPathFinished -
= PlayerEndEnterToVillage;
}

private void MainDruidAppear()
{
    mainDruidObject = Instantiate(mainDruid);
    StartCoroutine(mainDruidObject.MovePath(druidToGatesPath));
    mainDruidObject.OnPathFinished += MainDruidEndPathToGates;
}

```

```

private void MainDruidEndPathToGates()
{
    mainDruidObject.OnPathFinished -= MainDruidEndPathToGates;
    Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(3,
null, () => StartCoroutine(OpenGates()));
}
private IEnumerator OpenGates()
{
    newOrderInLayer = 2;
    while (gates.position != gatesEndPoint)
    {
        gates.position = Vector3.MoveTowards(gates.position,
gatesEndPoint, Time.deltaTime);
        yield return null;
    }
    StartCoroutine(ranger1.MovePath(ranger1MovePath));
    ranger1.OnPathFinished += () =>
    {
        ranger1.SetOrderInLayer(newOrderInLayer);
        ranger1.FlipX();
    };

    StartCoroutine(ranger2.MovePath(ranger2MovePath));
    ranger2.OnPathFinished += () =>
    {
        ranger2.SetOrderInLayer(newOrderInLayer);
        ranger2.FlipX();
    };
    yield return new WaitForSeconds(1.5f);
    StartCoroutine(EnemiesAppear());
}
private IEnumerator EnemiesAppear()
{
    Core.AudioManager.AudioManager.FInstance.DeleteCurrent();
    Core.AudioManager.AudioManager.FInstance.CreateAndPlay(0, true,
Vector3.zero);

    enemy1 = Instantiate(enemyPrefab1, enemyAppearPoint1,
Quaternion.identity);
    var enemy2 = Instantiate(enemyPrefab2, enemyAppearPoint2,
Quaternion.identity);
    enemy1.maxHealth = 15;
    enemy2.maxHealth = 100;
    enemy1.IsInvertedSpriteFlip = true;
}

```

```

enemy2.IsInvertedSpriteFlip = false;

StartCoroutine(enemy1.MovePath(enemyMovePathForward1));
yield return new WaitForSeconds(1);
StartCoroutine(other1.MovePath(other1MovePathHiding));

StartCoroutine(enemy2.MovePath(enemyMovePathForward2));
yield return new WaitForSeconds(1);
StartCoroutine(other2.MovePath(other2MovePathHiding));

enemy1.OnPathFinished += () =>
{
    StartCoroutine(StartCharacterFire(ranger1, enemy1));

StartCoroutine(GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.MovePath(playerT
oBoarPath));

GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.OnPathFinished +=
    () =>StartCoroutine(PlayerFightTheBoar());

};
enemy2.OnPathFinished += () =>
StartCoroutine(StartCharacterFire(ranger2, enemy2));
enemy1.OnDied += () =>
{
    StopAllCoroutines();
    StartCoroutine(enemy2.MovePath(enemyMovePathEscaping));
    enemy2.OnPathFinished -= () =>
StartCoroutine(StartCharacterFire(ranger2, enemy2));
    enemy2.OnPathFinished += () =>
    {
        enemy2.health = 0;
        StartCoroutine(MoveCameraToVillage());
        Destroy(enemy2.gameObject);
    };
    StartCoroutine(MoveCameraToCave());
};
}

private IEnumerator PlayerFightTheBoar()
{
    GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.FreezePosition();

    while (enemy1.health > 0)

```

```

    {

GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.PerformAction(enemy1);
        Debug.LogWarning($"ENEMY {enemy1.health}");
        yield return new WaitForSeconds(1);
    }
    GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.OnPathFinished -
=
        () => StartCoroutine(PlayerFightTheBoar());
    }

    private IEnumerator
StartCharacterFire(GameLogic.Actors.ActorTypes.KnightActor actor,
GameLogic.Actors.CharacterActor enemyActor)
    {
        Debug.LogWarning("START FIRE");
        while (enemyActor.health > 0)
        {
            enemyActor.anim.SetTrigger("Action");
            actor.PerformAction(enemyActor);
            yield return new WaitForSeconds(1);
        }

GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.UnfreezePosition();
        Debug.LogWarning("STOP FIRE");
    }

    private IEnumerator MoveCameraToCave()
    {
        var newPos = cavePoint;
        newPos.z = mainCamera.position.z;
        while (cavePoint.y != mainCamera.position.y)
        {
            mainCamera.position = Vector3.MoveTowards(mainCamera.position,
newPos, mainCameraMoveSpeed * Time.deltaTime);
            yield return null;
        }
    }

    private IEnumerator MoveCameraToVillage()
    {
        var newPos = targetCameraMovePoint.position;
        newPos.z = mainCamera.position.z;

        while (targetCameraMovePoint.position.y != mainCamera.position.y)

```

```

        {
            mainCamera.position = Vector3.MoveTowards(mainCamera.position,
newPos, mainCameraMoveSpeed * Time.deltaTime);
            yield return null;
        }
        Core.AudioManager.AudioManager.FInstance.DeleteCurrent();
        Core.AudioManager.AudioManager.FInstance.CreateAndPlay(3, true,
Vector3.zero);
        Debug.LogWarning("REVERSE");
        newOrderInLayer = 3;
        Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(4,
null, RangerTalk);
    }

    private void RangerTalk()
    {
        StartCoroutine(ranger2.MovePath(ranger2MovePathBack));
        ranger2.OnPathFinished += () =>
Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(5, null,
ContinueGame);
    }

    private void ContinueGame()
    {
        GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.UnlockMove();
        GameLogic.CameraController.Instance.IsCameraFollowsPlayer = true;

        GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.AddToMyTeam(GameLogic.Actors.Ac
torType.Knight, ranger2);
    }
}

```

GhooseTrigger.cs — компонент взаємодії з одним із компаньйонів гравця — персонажем Ghoose. Скрипт визначає поведінку компаньйона під час наближення гравця, активує діалоги, анімації чи інші події. Він використовується для розширення системи супутників та їх участі в ігровому процесі.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

```

```

public class GooseTrigger : MonoBehaviour
{
    [SerializeField] private Transform bush;
    [SerializeField] private Vector3 bushNewPos;
    [SerializeField] private float bushMoveSpeed;
    [SerializeField] private GameLogic.Actors.ActorTypes.GooseActor
goosePrefab;

    private GameLogic.Actors.ActorTypes.GooseActor goosePlayer;
    private Vector3 startBushPos;
    private int bushMoveCircleTimes = 2;

    private void Awake()
    {
        startBushPos = bush.position;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if
(collision.GetComponent<GameLogic.Actors.ActorTypes.MainPlayerActor>())
        {
            GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.LockMove();
            StartCoroutine(BushMove());
            GetComponent<BoxCollider2D>().enabled = false;
        }
    }

    private IEnumerator BushMove()
    {
        for (int i = 0; i < bushMoveCircleTimes; i++)
        {
            while (bush.position != bushNewPos)
            {
                bush.position = Vector3.MoveTowards(bush.position,
bushNewPos, bushMoveSpeed * Time.deltaTime);
                yield return null;
            }

            while (bush.position != startBushPos)
            {
                bush.position = Vector3.MoveTowards(bush.position,
startBushPos, bushMoveSpeed * Time.deltaTime);
                yield return null;
            }
        }
    }
}

```

```
        }  
    }  
    goosePlayer = Instantiate(goosePrefab);  
    Core.DialogueManagerLogic.DialogueManager.FInstance.StartDialog(6,  
null, GooseDialogEnd);  
    }  
  
    private void GooseDialogEnd()  
    {  
  
GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.AddToMyTeam(GameLogic.Actors.Ac  
torType.Goose, goosePlayer);  
        GameLogic.Actors.ActorTypes.MainPlayerActor.Instance.UnlockMove();  
    }  
}
```

ДОДАТОК Б
Слайди презентації



Рисунок Б.1 – Слайд 1

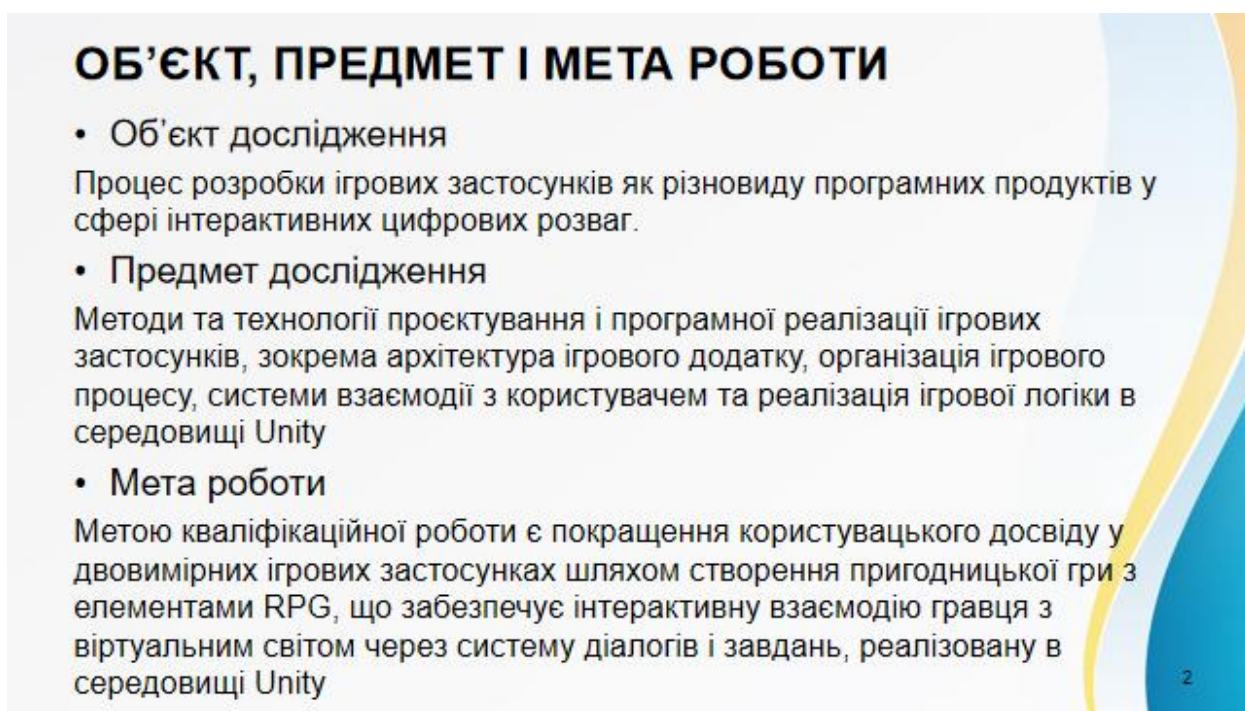
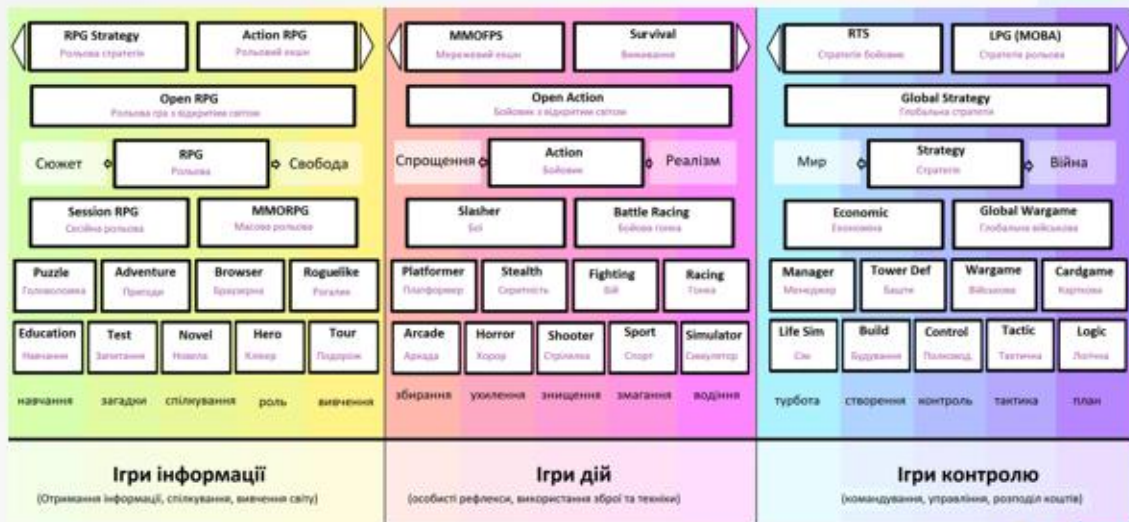


Рисунок Б.2 – Слайд 2

Класифікація ігор по жанрам



3

Рисунок Б.3 – Слайд 3

Сучасні популярні ігрові рушії

Рушій	Типи ігор	Платформи	Мова/підхід	Переваги
Unity	2D, 3D	PC, мобільні (Android, iOS), Web, консолі	C#, компонентна архітектура	Кросплатформеність, велика спільнота, багато навчальних матеріалів, популярний серед інді-розробників і мобільних ігор
Unreal Engine	Переважно 3D	PC, консолі, мобільні	C++, система Blueprints	Високоякісна графіка, фотореалізм, швидке прототипування без коду
Godot	2D, 3D	PC, мобільні, Web	GDScript, C#, C++	Відкритий код, повністю безкоштовний, легкий, зручний для навчальних проєктів та невеликих ігор
GameMaker Studio	2D	PC, мобільні, Web	GML, візуальне програмування	Швидка розробка 2D-ігор, простий у використанні, підходить для початківців
Construct	2D	Web, мобільні	Повністю візуальне програмування	Не потребує знань програмування, дуже швидке створення прототипів

4

Рисунок Б.4 – Слайд 4

Розробка ігор: 2D vs 3D

Характеристика	2D	3D
Простір	Двовимірний	Тривимірний
Об'єкти	Спрайти, тайл-сети, шари	Моделі, текстури, освітлення
Взаємодія	Колізії, анімації, логіка	Колізії в 3D, анімація скелетів, фізика
Ресурсоемність	Низька	Висока
Платформи	Мобільні, веб	ПК, консолі, VR
Інструменти	Unity, Godot	Unity, Unreal

Рисунок Б.5 – Слайд 5

Вибір програмного забезпечення



Рисунок Б.6 – Слайд 6

Вибір мови програмування



Рисунок Б.7 – Слайд 7

ЕТАПИ РОЗРОБКИ ІГОР



Рисунок Б.8 – Слайд 8

Створення UML- діаграм



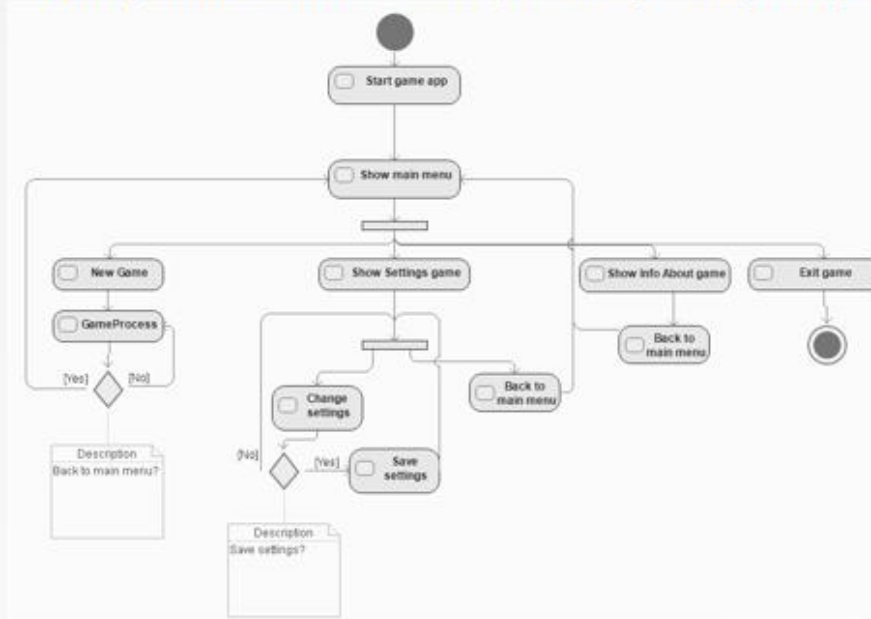
Рисунок Б.9 – Слайд 9

Діаграма прецедентів (Use Case diagram)



Рисунок Б.10 – Слайд 10

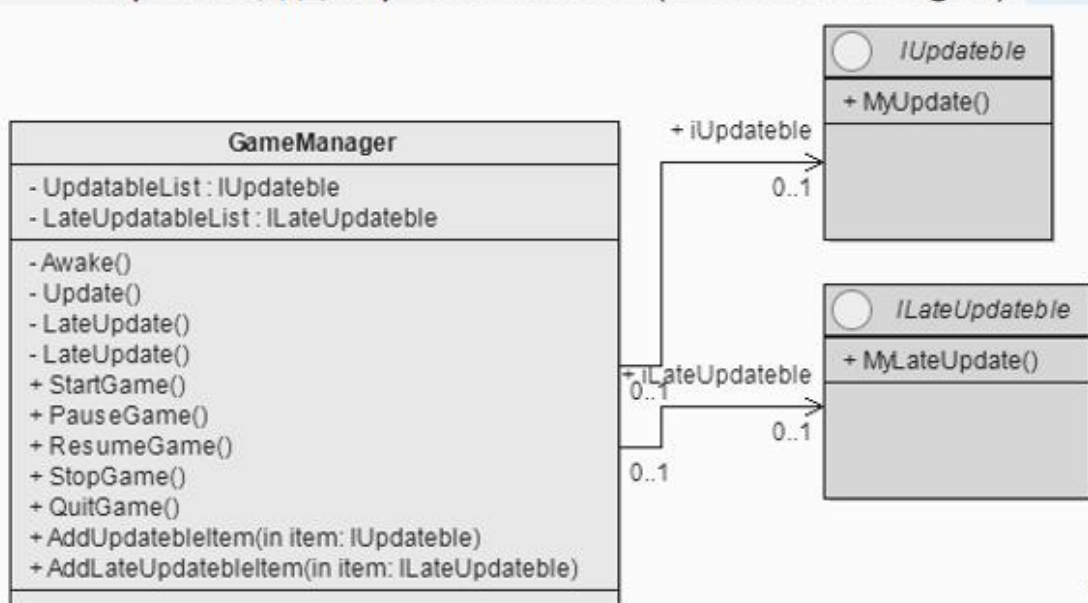
Діаграма діяльності (Activity diagram)



11

Рисунок Б.11 – Слайд 11

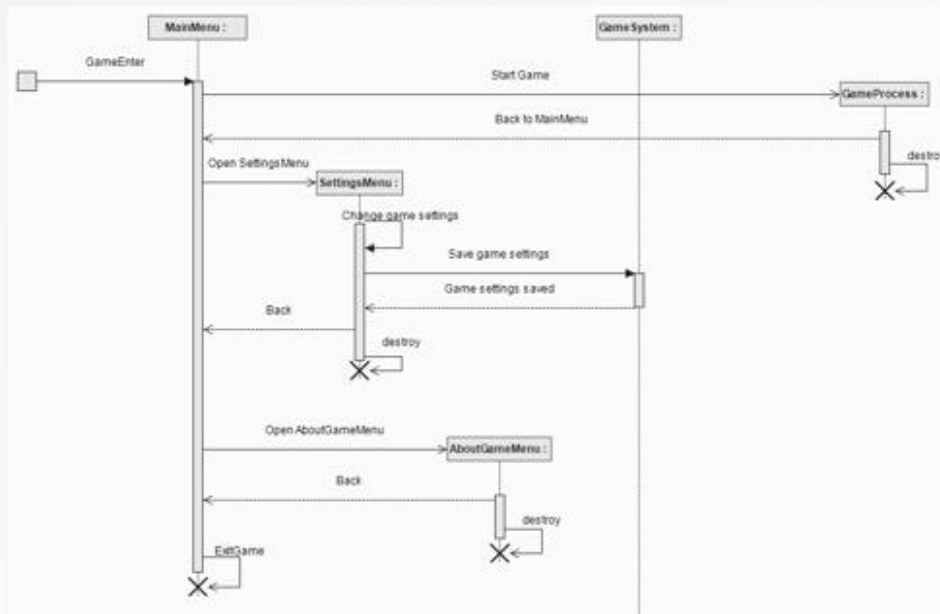
Приклад діаграми класів (Game Manager)



12

Рисунок Б.12 – Слайд 12

Діаграма послідовності (Sequence diagram)



13

Рисунок Б.13 – Слайд 13

Визначення концепції



Рисунок Б.14 – Слайд 14

Розробка дизайну ігрових локацій

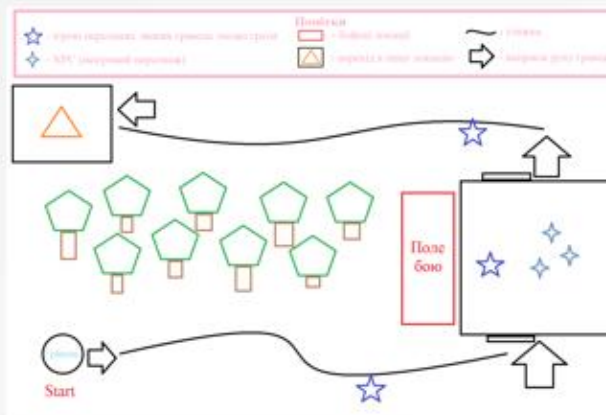


Рисунок Б.15 – Слайд 15

Менеджери

Назва менеджера	Призначення
AnimatorManager	керування запуском, паузою та зупинкою анімацій і реакцією на їх події
AudioManager	відтворення та керування музикою і звуковими ефектами гри
DataManager	збереження, завантаження та очищення ігрових даних і налаштувань
DialogueManager	управління діалогами, відображенням реплік і персонажів
GameManager	глобальне керування станом гри та переходами між сценами.
InputManager	обробка введення гравця (кнопки, утримання, осі руху)
TaskManager	керування активними завданнями та їх логікою виконання
TaskBase	базовий клас для створення та розширення ігрових завдань

Рисунок Б.16 – Слайд 16

Трејлер ігрового застосунку



17

Рисунок Б.17 – Слайд 17

Висновки

У ході роботи було досліджено процес розробки ігрових застосунків, проаналізовано ключові професійні ролі та етапи створення ігор. У практичній частині розроблено 2D-гру з елементами RPG, включно з діалоговою системою, графікою, анімаціями та бойовими механіками. Проведене тестування підтвердило відповідність гри технічному завданню. Робота демонструє ефективність комплексного підходу та сучасних технологій у створенні якісних ігрових продуктів.

18

Рисунок Б.18 – Слайд 18

ДОДАТОК В**Відео-трейлер гри**

<https://youtu.be/DBjUyeExBhY?si=rIUz—By6ZDuT6ljU>