**Міністерство освіти і науки України**
**Тернопільський національний технічний університет імені Івана Пулюя**

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету )
Кафедра програмної інженерії
(повна назва кафедри)

# КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Розробка 3D-гри жанру RPG на Unity

Виконав(ла): студент(ка) VI курсу, групи СПм-61
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

_____ Ткачук Р. С.
(підпис) (прізвище та ініціали)

Керівник _____ Бойко І. В.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Стоянов Ю. М.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Петрик М. Р.
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
**Тернопільський національний технічний університет імені Івана Пулюя**

Факультет Комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)

Кафедра Програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ _____
(підпис) (прізвище та ініціали)
« » 20___ р.

# З А В Д А Н Н Я
## НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Ткачуку Роману Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Розробка 3D-гри жанру RPG на Unity»

Керівник роботи доктор фіз.-мат. наук, професор кафедри ПІ Бойко Ігор Володимирович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 20___ року № _____

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи Прототип багатокористувацької 3D-гри жанру RPG на Unity, структурована кваліфікаційна робота магістра.

4. Зміст роботи (перелік питань, які потрібно розробити)
Огляд предметної області та технологій системи, формування вимог до системи, проектування проєкту, опис етапу розробки гри, опис проведення тестування, впровадження та подальшої підтримки розділ охорони праці, розділ охорони праці та безпеки життєдіяльності в надзвичайних ситуаціях.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Діаграма варіантів використання, діаграма класів, дві діаграми послідовностей, візуальне відображення зв'язків між анімаціями, плани рівнів гри, знімки екрану при проведенні тестування розробленого функціоналу та контенту гри.

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

7. Дата видачі завдання _____

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Аналіз предметної області та формування теми роботи | | |
| | | | |
| 2 | Визначення мети, задач та об’єкта дослідження | | |
| | | | |
| 3 | Формування функціональних і нефункціональних вимог | | |
| | | | |
| 4 | Проєктування архітектури програмного забезпечення | | |
| | | | |
| 5 | Розробка UML-діаграм (варіанти використання, класи) | | |
| | | | |
| 6 | Реалізація основних ігрових механік у Unity | | |
| | | | |
| 7 | Розробка інтерфейсу користувача та взаємодій | | |
| | | | |
| 8 | Тестування програмного продукту та усунення помилок | | |
| | | | |
| 9 | Оформлення розділу з охорони праці та БЖД | | |
| | | | |
| 10 | Підготовка та оформлення магістерської роботи | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Студент _____ Ткачук Роман Сергійович
(підпис) (прізвище та ініціали)

Керівник роботи _____ Бойко Ігор Володимирович
(підпис) (прізвище та ініціали)

# АНОТАЦІЯ

Магістерська робота виконана студентом Ткачуком Романом Сергійовичом кафедри програмної інженерії, факультету комп'ютерно-інформаційних систем та програмної інженерії.

Тема магістерської роботи: «Розробка 3D-гри жанру RPG на Unity» присвячена розробці програмного забезпечення гри «Adventure Guild». Робота викладена на 83 сторінках, містить 9 рисунків, таблиць немає, 4 додатки та 38 джерел у списку використаної літератури.

У магістерській роботі розглянуто процес проєктування та розробки ігрового програмного забезпечення у жанрі пригодницької рольової гри. Проаналізовано предметну область, сформовано словник основних термінів, а також визначено функціональні та нефункціональні вимоги до програмного продукту. Особливу увагу приділено моделюванню взаємодії користувача з системою за допомогою діаграм варіантів використання та визначенню архітектурних рішень.

У процесі проєктування обґрунтовано вибір компонентної архітектури, що дозволяє забезпечити модульність, масштабованість та зручність подальшого розвитку програмного забезпечення. Розглянуто ключові проєктні рішення, а також підхід до організації процесу розробки з використанням гнучкої методології Scrum.

У розділі, присвяченому розробці програмного забезпечення, описано структуру проєкту, реалізацію основного функціоналу гри, створення компонентів та класів системи, а також розробку графічної складової, включаючи анімації персонажів і створення ігрових рівнів. Реалізація гри здійснювалася з використанням ігрового рушія Unity та мови програмування C#.

Практичне значення роботи полягає у створенні працездатного прототипу ігрового програмного забезпечення, який може бути використаний як основа для подальшого розвитку, розширення функціональності та впровадження нових ігрових механік. Результати роботи можуть бути корисними для студентів та

розробників, які займаються створенням ігор з використанням компонентного підходу та сучасних інструментів ігрової індустрії.

Апробація результатів магістерської роботи здійснювалася у формі написання та подання тез доповіді.

Ключові слова: ІГРОВЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, UNITY, КОМПОНЕНТНА АРХІТЕКТУРА, UML, SCRUM, ADVENTURE GUILD.

# ABSTRACT

The master's thesis was completed by Roman Serhiiovych Tkachuk, a student of the Department of Software Engineering, Faculty of Computer and Information Systems and Software Engineering. The topic of the master's thesis, "Development of a 3D RPG Game Using Unity", is devoted to the development of game software «Adventure Guild». The thesis consists of 83 pages, contains 9 figures, no tables, 4 appendices, and 38 references in the list of sources.

The thesis examines the process of designing and developing game software in the genre of an adventure role-playing game. The subject area is analyzed, a glossary of key terms is formed, and functional and non-functional requirements for the software product are defined.

During the design process, the choice of a component-based architecture is substantiated, which ensures modularity, scalability, and convenience of further software development. Key design decisions are considered, as well as the approach to organizing the development process using the Scrum agile methodology.

The section devoted to software development describes the project structure, implementation of the main game functionality, development of system components and classes, as well as the creation of graphical content, including character animations and game level design. The game was implemented using the Unity game engine and the C# programming language.

The practical significance of the work lies in the development of a functional prototype of game software that can be used as a basis for further development, functional expansion, and implementation of new game mechanics. The results of the thesis may be useful for students and developers involved in game development using a component-based approach and modern tools of the game industry.

The results of the master's thesis were validated through the preparation and submission of conference theses.



Keywords: GAME SOFTWARE, UNITY, COMPONENT-BASED ARCHITECTURE, UML, SCRUM, ADVENTURE GUILD.

**ЗМІСТ**

# ВСТУП

Стрімкий розвиток ігрової індустрії та збільшення популярності інтерактивних 3D-середовищ зумовлюють потребу у створенні високоякісних програмних продуктів, що поєднують технічну складність, оптимальність архітектури та зручність взаємодії користувача з програмою. Жанр RPG (role-playing game) є одним із найбільш розповсюджених напрямів сучасних 3D-ігор, що вирізняється розвиненою ігровою логікою, системами прогресу персонажа, складними сценаріями взаємодії та вимогами до гнучкої архітектури програмного забезпечення. Тому розробка 3D-гри цього жанру із застосуванням сучасних інструментів та інженерних методів є актуальною задачею, що поєднує теоретичні та практичні аспекти програмної інженерії.

Метою даної кваліфікаційної роботи є розробка та дослідження прототипу багатокористувацької 3D-гри «Adventure Guild», жанру RPG на базі ігрового рушія Unity, що включає побудову ігрової архітектури, реалізацію ключових механік, створення інтерфейсу користувача, системи взаємодій, бойової логіки та управління станами гри.

Для досягнення поставленої мети необхідно вирішити такі задачі:

– проаналізувати предметну область та сучасні підходи до розробки ігор жанру RPG;
– визначити та обґрунтувати вибір технологій і засобів реалізації;
– сформувати вимоги до функціоналу та структури 3D-гри;
– спроєктувати архітектуру системи з використанням UML-діаграм та принципів об’єктно-орієнтованого проєктування;
– реалізувати основні модулі гри, включно з рухом персонажа, системою бою, взаємодіями з об’єктами середовища, інтерфейсом та переходами між сценами;
– провести тестування програмного продукту та оцінити його відповідність вимогам.

Об’єктом дослідження є процес розробки інтерактивних 3D-ігор на основі

сучасних ігрових рушіїв.

Предметом дослідження є методи, моделі та алгоритми створення 3D-гри жанру RPG на платформі Unity, а також програмні засоби реалізації її ігрових механік.

Апробація результатів роботи відбувалася у формі підготовки тез доповіді за матеріалами дослідження та розробки програмного продукту. Представлені результати можуть бути використані як основа для подальшого вдосконалення ігрових систем, розширення функціоналу та впровадження нових механік у межах проєкту.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розділ «Аналіз предметної області та формування вимог» спрямований на дослідження основних аспектів проблемної області, у межах якої здійснюється розробка програмного забезпечення. Аналіз предметної області є необхідним етапом проєктування, оскільки дозволяє визначити ключові характеристики майбутнього програмного продукту, виявити його функціональні можливості, обмеження та особливості використання. Для програмних продуктів ігрового спрямування такий аналіз має особливе значення, оскільки поєднує технічні аспекти реалізації з вимогами до ігрового процесу, взаємодії користувача з системою та візуального представлення ігрового світу.

У межах даного розділу розглядаються основні поняття та особливості жанру рольових ігор, структура ігрового середовища, типові механіки та сценарії взаємодії гравця з об’єктами гри. На основі проведеного аналізу формується перелік функціональних і нефункціональних вимог до програмного забезпечення, які визначають напрям подальшого проєктування та реалізації системи. Для більш детального розуміння контексту розробки на початковому етапі подається опис предметної області, у якому окреслюється загальна концепція гри, її основні елементи та принципи функціонування.

## 1.1 Опис предметної області

Предметна область даної кваліфікаційної роботи охоплює процес розробки прототипу тривимірної комп’ютерної гри жанру RPG у фентезійному сетингу з використанням ігрового рушія Unity [1-3]. Програмний продукт має назву «Adventure Guild» та концептуально орієнтований на багатокористувацьку взаємодію, де гравці виконують ролі різних персонажів і спільно беруть участь у виконанні ігрових завдань. Водночас у межах даної роботи розглядається початковий етап реалізації — однокористувацький прототип гри.

На поточному етапі у прототипі реалізовано одного ігрового персонажа — лицаря, який є центральним елементом ігрової системи. Гравець керує персонажем у межах окремих ігрових сцен, взаємодіє з об’єктами середовища та бере участь у бойових зіткненнях. Реалізовані механіки дозволяють продемонструвати базові принципи жанру RPG, включно з рухом, анімацією, взаємодією та управлінням станами персонажа.

Предметна область також включає організацію ігрового світу, логіку взаємодії між об’єктами, обробку подій, колізій і тригерів, а також керування переходами між сценами. Розробка здійснюється з використанням компонентного підходу Unity, де функціональність об’єктів визначається сукупністю компонентів та скриптів, написаних мовою програмування C#. Для реалізації динамічного ігрового процесу застосовуються механізми оновлення кадрів, корутини та система анімацій.

Таким чином, предметна область поєднує програмну інженерію, геймдизайн та інтерактивні технології і спрямована на створення функціонального прототипу 3D RPG-гри, який відображає загальну концепцію багатокористувацького проєкту «Adventure Guild» та слугує основою для подальшого розвитку ігрової системи.

## 1.2 Аналіз вимог

На етапі аналізу вимог визначаються функціональні та нефункціональні характеристики програмної системи, які формують основу для проєктування архітектури та подальшого конструювання. Для гри «Andventure Guild» вимоги були сформовані з урахуванням особливостей жанру, очікувань користувачів та технічних можливостей обраного технологічного стеку — Unity та C#.

### 1.2.1 Функціональні вимоги

Функціональні вимоги визначають перелік можливостей та поведінку програмного продукту, які мають бути реалізовані у межах прототипу 3D-гри жанру RPG «Adventure Guild». Вони формуються на основі аналізу предметної області, жанрових особливостей RPG-ігор та обраної концепції проєкту. Реалізація наведених вимог забезпечує коректну роботу гри, цілісність ігрового процесу та взаємодію користувача з ігровим середовищем. Основні функціональні вимоги до даної гри ключають:

1. Вимоги до ігрового середовища. Програмний продукт повинен забезпечувати наявність ігрових сцен, що представляють окремі частини ігрового світу. Система має підтримувати завантаження, ініціалізацію та коректне перемикання між сценами під час виконання гри. Кожна сцена повинна містити об’єкти оточення, джерела освітлення, колайдери та інші елементи, необхідні для взаємодії з ігровим персонажем.
2. Вимоги до керування ігровим персонажем. Гра повинна надавати користувачеві можливість керування ігровим персонажем у реальному часі за допомогою стандартних засобів вводу. Система має забезпечувати рух персонажа у тривимірному просторі, обертання, виконання дій та коректну реакцію на команди гравця. Керування повинно бути інтуїтивно зрозумілим і забезпечувати плавність рухів та анімацій.
3. Вимоги до бойової системи. Прототип гри повинен містити базову бойову систему, що дозволяє ігровому персонажу взаємодіяти з ворогами або іншими активними об’єктами середовища. Бойова логіка має включати виконання атак, їх комбінацій, блокування атак, визначення результатів взаємодії та зміну станів персонажа. Реалізація бойової системи повинна відповідати жанру RPG та забезпечувати наочність і передбачуваність ігрових дій.
4. Вимоги до системи анімацій. Система повинна забезпечувати

відтворення анімацій руху, бойових дій та інших станів персонажа. Анімації мають коректно синхронізуватися з ігровою логікою та перемикатися залежно від поточного стану персонажа. Для керування анімаціями необхідно використовувати відповідні засоби ігрового рушія Unity.

5. Вимоги до взаємодії з об’єктами середовища. Гра повинна підтримувати взаємодію ігрового персонажа з об’єктами ігрового світу. Система має обробляти події зіткнення та активації тригерів, що дозволяє реалізувати базові сценарії взаємодії, такі як активація механізмів або реакція на входження в певні зони.
6. Вимоги до інтерфейсу користувача. Програмний продукт повинен містити базовий інтерфейс користувача, який відображає ключову інформацію про стан гри. Інтерфейс має бути зрозумілим, мінімалістичним та не перевантажувати ігровий простір. Елементи інтерфейсу повинні оновлюватися відповідно до змін ігрового стану.
7. Вимоги до системи керування станами гри. Система повинна підтримувати основні стани гри, такі як активний ігровий процес, пауза та завершення сцени. Перехід між станами має здійснюватися коректно та без порушення цілісності ігрових даних.
8. Вимоги до стабільності та коректності роботи. Прототип гри повинен коректно обробляти помилки виконання, уникати критичних збоїв та забезпечувати стабільну роботу в межах визначених сценаріїв використання. Логіка гри має бути структурованою та підтримуваною для подальшого розвитку проєкту.

### 1.2.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики програмного продукту та обмеження, яким має відповідати прототип 3D-гри жанру RPG «Adventure Guild». Дані вимоги не описують конкретні функції системи, проте

безпосередньо впливають на зручність використання, стабільність, продуктивність та можливість подальшого розвитку програмного продукту:

1. Вимоги до продуктивності. Прототип гри повинен забезпечувати стабільну роботу в режимі реального часу на персональних комп’ютерах середнього рівня продуктивності. Частота оновлення кадрів має бути достатньою для комфортного ігрового процесу без помітних затримок або ривків анімації. Обробка ігрової логіки, фізики та анімацій повинна здійснюватися ефективно з мінімальним навантаженням на процесор та графічну підсистему.
2. Вимоги до надійності та стабільності. Система повинна працювати стабільно протягом тривалого часу без аварійних завершень або втрати ігрового стану. Прототип має коректно обробляти стандартні сценарії використання та некритичні помилки, не порушуючи цілісність ігрового процесу.
3. Вимоги до зручності використання. Інтерфейс користувача та система керування повинні бути інтуїтивно зрозумілими та доступними для користувача без необхідності попереднього навчання. Взаємодія з ігровим світом має здійснюватися логічно та послідовно, забезпечуючи комфортне сприйняття ігрового процесу.
4. Вимоги до масштабованості. Архітектура прототипу повинна бути спроєктована з урахуванням можливості подальшого розширення функціоналу гри. Структура коду та організація ігрових модулів мають дозволяти додавання нових механік, персонажів, ігрових режимів та мережевих компонентів без суттєвих змін існуючої логіки.
5. Вимоги до підтримуваності та супроводжуваності. Код програмного продукту повинен бути структурованим, читабельним та задокументованим. Це забезпечує зручність внесення змін, виправлення помилок та подальшого супроводу проєкту. Назви класів, методів і змінних мають відповідати їхньому призначенню та прийнятим стандартам мови програмування C#.

6. Вимоги до сумісності. Прототип гри повинен коректно працювати в середовищі операційної системи Windows із використанням актуальної версії ігрового рушія Unity. Система має бути сумісною зі стандартними пристроями введення, такими як клавіатура та миша.
7. Вимоги до безпеки. Програмний продукт повинен забезпечувати коректну обробку даних ігрового процесу та уникати несанкціонованого доступу до внутрішніх ресурсів. Хоча прототип не передбачає зберігання конфіденційних даних користувачів, логіка обробки інформації має бути захищеною від некоректного використання.
8. Вимоги до якості програмного коду. Код гри має відповідати принципам об’єктно-орієнтованого програмування, зокрема принципам модульності, повторного використання та розмежування відповідальності. Це дозволяє підвищити загальну якість програмного продукту та спростити процес подальшої розробки.

### 1.2.3 Обмеження

Для точного моделювання вимоги також важливо врахувати технічні та організаційні обмеження:

- використання ігрового рушія Unity як основної платформи розробки;
- використання мови програмування C#;
- обмеження продуктивності на слабких системах;
- необхідність дотримання часових рамок навчального проєкту;
- обмежений набір інструментів та активів, що використовуються в розробці.

## 1.3 Технології системи

Розробка прототипу 3D фентезі гри жанру RPG «Adventure Guild» базується на поєднанні сучасних програмних технологій та інструментів, які забезпечують реалізацію інтерактивного ігрового середовища, ігрової логіки та візуальних компонентів. Обраний набір технологій дозволяє створити гнучку, розширювану та підтримувану архітектуру програмного продукту.

Основною технологічною платформою для розробки гри є ігровий рушій Unity [1-3], який є одним із найбільш поширених інструментів для створення інтерактивних 2D та 3D застосунків. Unity широко використовується не лише в ігровій індустрії, але й у сфері навчальних симуляцій, інженерного моделювання, архітектурної візуалізації, а також у проєктах, пов’язаних із віртуальною та доповненою реальністю. Така універсальність робить Unity доцільним вибором для реалізації прототипу RPG-гри в межах магістерської роботи.

Ігровий рушій Unity надає широкий набір інструментів для створення тривимірних ігрових середовищ та реалізації складної ігрової логіки.

Серед ключових особливостей двигуна можна виділити:

1. Потужний редактор сцен, який дозволяє створювати тривимірні середовища, розміщувати моделі, налаштовувати освітлення, камери, анімації, фізику та логіку взаємодії об’єктів.
2. Фізичний рушій, що забезпечує реалістичну симуляцію руху, зіткнень, гравітації та поведінки твердих тіл. Це робить можливим створення складних механізмів і поведінкових сценаріїв, які використовуються як у іграх, так і у технічних симуляторах.
3. Компонентна архітектура, завдяки якій кожен об’єкт сцени може бути доповнений окремими функціональними модулями (компонентами), що забезпечують гнучкість та масштабованість розробки.
4. Мова програмування C# [4,5], що використовується для написання сценаріїв поведінки (скриптів), реалізації ігрової логіки, взаємодії між

об’єктами, керування анімацією, штучним інтелектом, інтерфейсом користувача та іншими елементами проєкту.

5. Вбудовані інструменти анімації (Animator, Animation Controller [6]), які дозволяють створювати плавні переходи між станами персонажа, налаштовувати поведінкові дерева та складну систему рухових анімацій.
6. Система UI, що надає можливість створювати інтерактивні меню, індикатори станів, інвентар, діалогові вікна та інші елементи інтерфейсу, необхідні для RPG-проєкту.
7. Підтримка шейдерів та матеріалів, які забезпечують реалістичне відображення поверхонь, роботу зі світлом та ефектами постобробки.
8. Пакетні інструменти Unity Package Manager, які дозволяють підключати додаткові модулі, зокрема системи навігації [7], штучного інтелекту, інструменти візуального скриптингу, бібліотеки для VR/AR та ін.
9. Підтримка кросплатформенності, що забезпечує можливість збірки проєкту для Windows, Linux, Android, iOS, WebGL та інших платформ без необхідності значних змін у коді.

Для реалізації поведінки об’єктів та ігрової логіки використовується мова програмування C#, яка є основною мовою скриптингу в Unity. Завдяки підтримці об’єктно-орієнтованого програмування, C# дозволяє структурувати логіку гри у вигляді окремих класів і компонентів із чітко визначеною відповідальністю. Це сприяє підвищенню читабельності коду, його повторному використанню та спрощенню процесу супроводу. Крім того, мова підтримує сучасні програмні парадигми, зокрема подієву модель та принципи SOLID, що є важливим для побудови масштабованих ігрових систем.

Архітектура гри «Adventure Guild» ґрунтується на компонентному підході, який є базовим принципом організації об’єктів у Unity. Кожен ігровий об’єкт формується як набір незалежних компонентів, що відповідають за окремі аспекти його поведінки. Наприклад, ігровий персонаж може містити компоненти руху, обробки зіткнень, здоров’я, анімації та бойової логіки. Такий підхід забезпечує гнучкість розробки, дозволяє легко змінювати або доповнювати функціонал та

зменшує складність підтримки коду.

Для створення інтерфейсу користувача використовується вбудована система UI Unity, яка дозволяє реалізувати елементи відображення стану гри, меню та інформаційні індикатори. Графічне оформлення ігрового середовища забезпечується за рахунок використання матеріалів, шейдерів та системи освітлення, що сприяє формуванню цілісної візуальної атмосфери фентезійного світу.

У процесі розробки також застосовуються допоміжні інструменти та технології. Для написання та налагодження програмного коду використовуються середовища розробки Visual Studio Code, які надають зручні засоби для дебагінгу та аналізу помилок. Керування версіями проєкту здійснюється за допомогою системи Git, що дозволяє відстежувати зміни в коді та підтримувати стабільність розробки. Для побудови UML-діаграм і візуалізації архітектури системи використовуються інструменти типу draw.io.

Загалом, використання ігрового рушія Unity, мови програмування C#, компонентної архітектури та сучасних інструментів розробки формує надійну технічну основу для створення прототипу 3D RPG-гри «Adventure Guild». Обрані технології забезпечують відповідність програмного продукту вимогам до якості, гнучкості та потенціалу подальшого розвитку.

У процесі розробки прототипу гри «Adventure Guild» для підготовки тривимірних моделей персонажів до використання в ігровому рушії застосовувався програмний засіб AccuRIG [11]. Даний інструмент дозволяє виконувати автоматизований ригінг моделей, створюючи скелетну структуру та налаштовуючи прив’язку вершин моделі до кісток. Використання AccuRIG значно спростило підготовчий етап інтеграції персонажів у систему анімацій Unity та забезпечило коректну роботу скелетних анімацій без необхідності ручного налаштування.

Для пошуку та використання матеріалів, текстур і допоміжних ігрових ресурсів у проєкті застосовувався сервіс Unity Asset Store [12], який є офіційною платформою розповсюдження асетів для ігрового рушія Unity. Даний ресурс

надає широкий спектр готових компонентів, графічних матеріалів, моделей та інструментів, оптимізованих для безпосереднього використання в Unity. Застосування Asset Store дозволило пришвидшити процес створення ігрового середовища та зосередитися на реалізації логіки та архітектури прототипу.

Для отримання тривимірних моделей персонажів та об’єктів ігрового середовища використовувався спеціалізований ресурс CGTrader [10], який містить бібліотеку 3D-моделей різного рівня деталізації та призначення. Моделі, отримані з даного ресурсу, проходили етапи адаптації, зокрема налаштування масштабів, матеріалів та оптимізації полігональної структури, що забезпечило їх коректне використання в ігровому рушії та відповідність вимогам до продуктивності.

Для пошуку та використання анімацій ігрових персонажів застосовувався сервіс Mixamo [9], який надає бібліотеку готових анімацій для персонажів із підтримкою скелетної структури. Інтеграція анімацій із Mixamo у поєднанні з попередньо виконаним ригінгом моделей дозволила швидко реалізувати рухи, бойові дії та інші поведінкові стани персонажа. Отримані анімації були адаптовані до системи Animator у Unity та використані для побудови анімаційних станів прототипу гри.

Застосування зовнішніх ресурсів і спеціалізованих інструментів у поєднанні з ігровим рушієм Unity забезпечило ефективний пайплайн розробки, скорочення часу створення контенту та підвищення якості візуальної складової прототипу «Adventure Guild». Такий підхід відповідає сучасним практикам ігрової індустрії та демонструє комплексне використання програмних засобів у процесі розробки ігрових систем.

У результаті проведеного аналізу предметної області було досліджено специфіку розробки 3D-ігор жанру RPG, визначено основні складові ігрових систем та особливості їх реалізації з використанням ігрового рушія Unity. Проаналізовано ключові поняття та терміни, що використовуються під час створення ігрових проєктів, сформовано словник предметної області та описано концепцію гри «Adventure Guild» як прототипу майбутнього багатокористувацького проєкту. На основі проведеного аналізу було

сформульовано функціональні та нефункціональні вимоги до програмного продукту, які визначають його структуру, поведінку та якісні характеристики. Отримані результати стали основою для подальшого етапу проєктування архітектури системи та реалізації ключових механік гри.

# 2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

На етапі проєктування програмного забезпечення здійснюється перехід від сформульованих вимог та опису предметної області до формування структурної та логічної моделі системи. Даний етап є одним із ключових у життєвому циклі розробки програмного забезпечення, оскільки саме на ньому визначаються архітектурні рішення, принципи взаємодії компонентів, структура програмних модулів та загальні підходи до реалізації функціональності. Грамотно виконане проєктування дозволяє забезпечити масштабованість, підтримуваність та ефективність програмного продукту, а також мінімізувати ризики виникнення помилок на подальших етапах розробки.

У межах цього розділу розглядаються основні проєктні рішення, прийняті під час розробки гри «Adventure Guild». Зокрема, описується архітектура системи, обґрунтовується вибір основних технологій та інструментів розробки, а також аналізуються підходи до організації взаємодії між основними компонентами програмного забезпечення. Особлива увага приділяється забезпеченню модульності системи та можливості її подальшого розширення відповідно до змін вимог або впровадження нових ігрових механік.

## 2.1 Архітектура системи та методологія розробки

У даному підрозділі розглядаються ключові проєктні рішення, прийняті під час розробки програмного забезпечення гри «Adventure Guild». Зокрема, описуються обрана архітектура системи та підхід до організації процесу розробки, які визначають структуру програмного продукту, принципи взаємодії його компонентів і послідовність виконання робіт. Розгляд цих аспектів дозволяє обґрунтувати прийняті технічні та організаційні рішення, а також показати їх вплив на масштабованість, підтримуваність і якість програмного забезпечення в цілому.

### 2.1.1 Обрана архітектура системи

Архітектура гри «Adventure Guild» реалізована з використанням компонентного підходу [16. 17], який є стандартним та широко застосовуваним у процесі розробки ігрового програмного забезпечення на платформі Unity. Даний підхід передбачає побудову системи на основі набору незалежних компонентів, кожен з яких відповідає за окремий аспект функціонування об’єктів гри. Така архітектурна модель дозволяє досягти високого рівня модульності, гнучкості та масштабованості програмного забезпечення.

Основною концепцією компонентної архітектури є представлення кожного елемента гри у вигляді ігрового об’єкта (GameObject), який виступає контейнером для набору компонентів. Ці компоненти визначають поведінку об’єкта, його фізичні характеристики, логіку взаємодії з іншими елементами системи, а також візуальні та звукові властивості. У грі «Adventure Guild» такий підхід використовується для реалізації персонажів, ворогів, об’єктів навколишнього середовища та елементів інтерфейсу користувача.

Такий підхід забезпечує:

1. Модульність – кожен компонент реалізує окрему функціональність і може бути незалежно замінений або оновлений без впливу на інші частини системи.
2. Повторне використання коду – компоненти можна додавати до різних об’єктів гри, зменшуючи дублювання логіки.
3. Гнучкість розвитку – легко інтегрувати нові механіки або змінювати існуючі без переробки всієї системи.

У даному проєкті виділено кілька основних типів компонентів:

1. Компоненти персонажів – відповідають за управління гравцем, взаємодію з інвентарем, систему бою та життєві показники.
2. Компоненти ворогів – реалізують поведінку штучного інтелекту, навігацію та реакцію на дії гравця.

3. Компоненти світу – забезпечують генерацію рівнів, фізичні взаємодії об’єктів та взаємодію з навколишнім середовищем.
4. Компоненти інтерфейсу – керують відображенням інформації для гравця, включаючи HUD, меню та панелі інвентарю.

Для ігрових персонажів застосовуються компоненти, що відповідають за керування рухом, обробку введення користувача, систему здоров’я, бойові механіки та управління інвентарем. Окрім цього, використовуються компоненти, які забезпечують анімацію персонажів, відтворення візуальних ефектів та синхронізацію звукових подій. Завдяки такій структурі кожен аспект поведінки персонажа реалізується окремо, що значно спрощує внесення змін та подальший розвиток ігрової логіки.

Компонентна архітектура дозволяє зменшити зв’язаність між частинами системи, оскільки кожен компонент реалізує чітко визначену функціональність та може бути незалежно змінений або замінений без впливу на інші елементи. Це сприяє підвищенню якості коду та полегшує процес тестування. Важливою перевагою також є можливість повторного використання компонентів, коли одна й та сама реалізація може застосовуватися для різних об’єктів гри без дублювання програмної логіки [20-25].

### 2.1.2 Методологія розробки проекту

Для організації процесу розробки програмного забезпечення у проєкті використовується гнучка методологія управління Scrum [13,14]. Згідно з даним підходом, розробка гри поділена на короткі ітерації — спринти тривалістю від одного до двох тижнів. Після завершення кожного спринту формується працездатна версія окремих функціональних компонентів або ігрових механік, що дозволяє здійснювати регулярну перевірку результатів та оперативно вносити необхідні корективи.

Регулярні огляди виконаних завдань і поступове нарощування функціональності дозволяють своєчасно виявляти проблемні місця та

мінімізувати ризики на пізніх етапах розробки. Поєднання компонентної архітектури з гнучкими підходами до управління проєктом забезпечує ефективне керування складністю системи та стабільність її роботи.

Загалом, обрана архітектурна модель дозволяє забезпечити гнучкість і масштабованість програмного забезпечення, що є особливо важливим для тривимірних ігор з великою кількістю взаємодіючих підсистем, таких як фізика, анімація, штучний інтелект та інтерфейс користувача.

## 2.2 Проектування відношень між акторами і прецедентами

Проєктування відношень між акторами та прецедентами (варіантами використання) є важливим етапом моделювання функціональної поведінки програмного забезпечення [15,16]. Діаграма варіантів використаннях дозволяє наочно представити взаємодію користувача з системою, визначити основні сценарії використання гри та встановити логічні зв’язки між окремими функціональними можливостями. У межах даного підрозділу розглядається взаємодія актора з основними прецедентами гри «Adventure Guild», а також аналізуються відношення типу include та extend, які використовуються для деталізації та повторного використання функціональності.

Діаграма варіантів використання для гри демонструє повний набір доступних гравцеві взаємодій: від навігації та бойових дій до роботи з інвентарем і здобиччю. Вона дозволяє сформувати чітке уявлення про функціональну модель гри та визначає основу для подальшого проєктування й розробки архітектури системи. Нижче, на рисунку 2.1 зображено діаграму варіантів використання сформовану відповідно до вимог проекту:

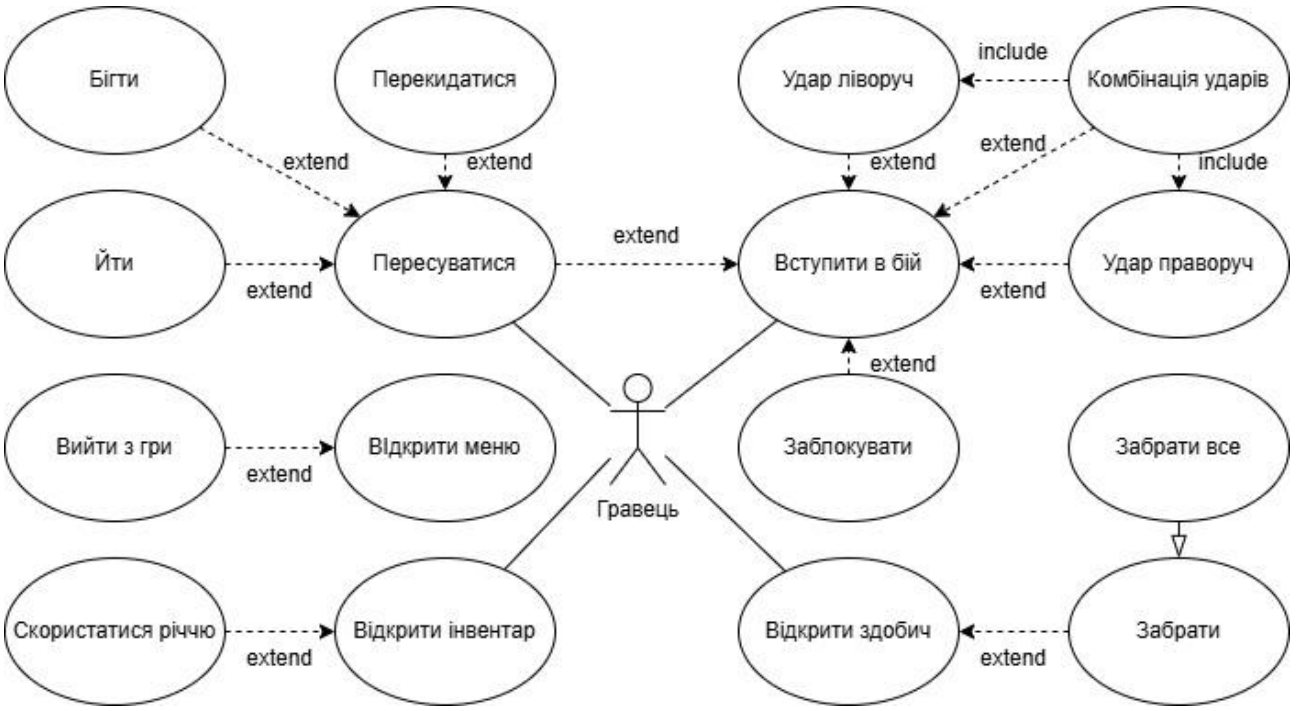


Рисунок 2.1 – Діаграм варіантів використання

Основним актором системи є гравець, який безпосередньо взаємодіє з ігровим середовищем та ініціює виконання всіх доступних варіантів використання. Основні групи функціональності:

1. Навігаційні дії гравця. Базовим варіантом є «Пересуватися». До нього за допомогою зв’язків extend доповнюються такі дії, як:
   – «Йти»;
   – «Бігти»;
   – «Перекидатися».

   Ці дії є розширенням основного сценарію пересування і виконуються за певних умов або за бажанням гравця.
2. Бойова система. Центральним бойовим варіантом є «Вступити в бій». До нього також через extend додаються окремі бойові дії:
   – «Удар ліворуч»;
   – «Удар праворуч»;
   – «Комбінація ударів»;
   – «Заблокувати».

   У свою чергу «Комбінація ударів» включає базові атаки (include), що

підкреслює залежність цього сценарію від попередніх — комбінація складається з окремих ударів.

3. Інтеракція з предметами. Гравець може відкривати здобич («Відкрити здобич») та забирати предмети («Забрати»). Сценарій «Забрати все» розширює базовий варіант отримання предметів через extend.
4. Дії з інтерфейсом. Гравцеві доступні окремі сценарії:
   - «Відкрити інвентар»;
   - «Відкрити меню»;
   - «Скористатися річчю»;
   - «Вийти з гри».

   Вони не залежать від інших варіантів використання і є самостійними діями користувача.

. На діаграмі варіантів використання гравець пов’язаний із ключовими прецедентами, які визначають основну логіку ігрового процесу. Можна виділити такі основні сценарії варіантів використання програмного продукту гравцем:

1. Одним із базових варіантів використання є пересування персонажа. Даний прецедент описує можливість переміщення гравця ігровим світом та є фундаментальним для реалізації ігрового процесу. Пересування реалізується через розширювальні прецеденти, такі як йти, бігти та перекатитися, які використовують відношення extend. Це означає, що зазначені дії є спеціалізованими формами пересування та виконуються за певних умов, наприклад, залежно від стану персонажа або дій гравця.
2. Наступним важливим варіантом використання є вступ у бій, який описує початок бойової взаємодії між гравцем і ворогами. Даний прецедент розширюється низкою бойових дій, зокрема ударом ліворуч, ударом праворуч та блокуванням. Крім того, для бойової системи використовується відношення include у прецеденті комбінація ударів, що вказує на обов’язкове виконання кількох базових атак для формування складнішої бойової дії. Такий підхід дозволяє логічно структурувати бойову механіку та уникнути дублювання функціональності.

3. Ще одним варіантом використання є відкриття меню, яке надає гравцеві доступ до основних налаштувань гри, збереження прогресу або виходу з гри. Даний прецедент може бути розширений варіантом вийти з гри, що активується за відповідного вибору користувача. Використання відношення extend у цьому випадку дозволяє відокремити основну функціональність меню від додаткових дій, які виконуються не завжди.
4. Важливу роль у грі відіграє варіант використання відкриття інвентарю, який дозволяє гравцеві переглядати зібрані предмети, керувати екіпіровкою та використовувати ресурси. Інвентар тісно пов’язаний із варіантом використання скористатися річчю, що реалізується як розширення основного прецеденту. Це означає, що використання предметів можливе лише після відкриття інвентарю та вибору відповідного елемента.
5. Окремим варіантом використання є відкриття здобичі, який описує взаємодію гравця з об’єктами, що містять нагороду, наприклад скринями або трофеями після перемоги над ворогами. Даний прецедент може бути розширений діями забрати або забрати все, що дозволяє гравцеві вибирати спосіб отримання здобутих предметів. Такий підхід забезпечує зручність керування ресурсами та гнучкість ігрового процесу.
6. До переліку основних варіантів використання також належить блокування, яке дозволяє гравцеві захищатися від атак супротивників під час бою. Цей прецедент є розширенням варіанту вступити в бій і виконується лише за наявності активної бойової ситуації. Реалізація блокування як окремого варіанту використання дозволяє чітко відокремити захисні дії від атакувальних.
7. Таким чином, діаграма варіантів використання гри «Adventure Guild» відображає ключові сценарії взаємодії гравця з системою та демонструє логічні зв’язки між окремими прецедентами. Використання відношень include та extend дозволяє структурувати функціональність гри, забезпечити повторне використання базових дій та підвищити

зрозумілість моделі. Отримана модель є основою для подальшого проєктування класів, компонентів та реалізації функціональності програмного забезпечення.

Таким чином, побудова та аналіз діаграми варіантів використання дозволили формалізувати взаємодію користувача з прототипом гри «Adventure Guild» та визначити основні сценарії використання програмного продукту. Виділені варіанти використання відображають ключові дії гравця та логіку роботи системи в межах однокористувацького режиму. Отримана модель слугує основою для подальшого проєктування архітектури системи, визначення структури класів і реалізації ігрових механік, а також забезпечує узгодженість між вимогами до системи та її програмною реалізацією.

## 2.3 Визначення класів системи

Проєктування класів є одним із ключових етапів розробки програмного продукту, оскільки визначає внутрішню структуру системи, взаємозв’язки між її складовими та розподіл відповідальності між окремими компонентами. У межах даної роботи проєктування класів спрямоване на формалізацію архітектури прототипу 3D-гри «Adventure Guild» та забезпечення логічної й зрозумілої організації програмного коду [18].

Класи системи проєктувалися з урахуванням особливостей компонентної архітектури ігрового рушія Unity, де поведінка об’єктів реалізується через скрипти, прив’язані до ігрових об’єктів сцени. Значну увагу було приділено розмежуванню логіки керування персонажем, обробки подій, взаємодії з ігровим середовищем, керування анімаціями та станами гри. Це дозволило уникнути надмірної концентрації функціональності в одному класі та забезпечити модульність системи.

Для візуалізації структури програмного продукту та взаємодії між класами була побудована діаграма класів UML [15], яка відображає основні класи системи, їхні атрибути, методи та типи зв’язків. Діаграма класів слугує наочною моделлю

програмної архітектури та є основою для подальшого опису кожного класу і його ролі в реалізації прототипу гри. Нижче, на рисунку 2.2 зображено безпосередньо діаграму класів проекту.

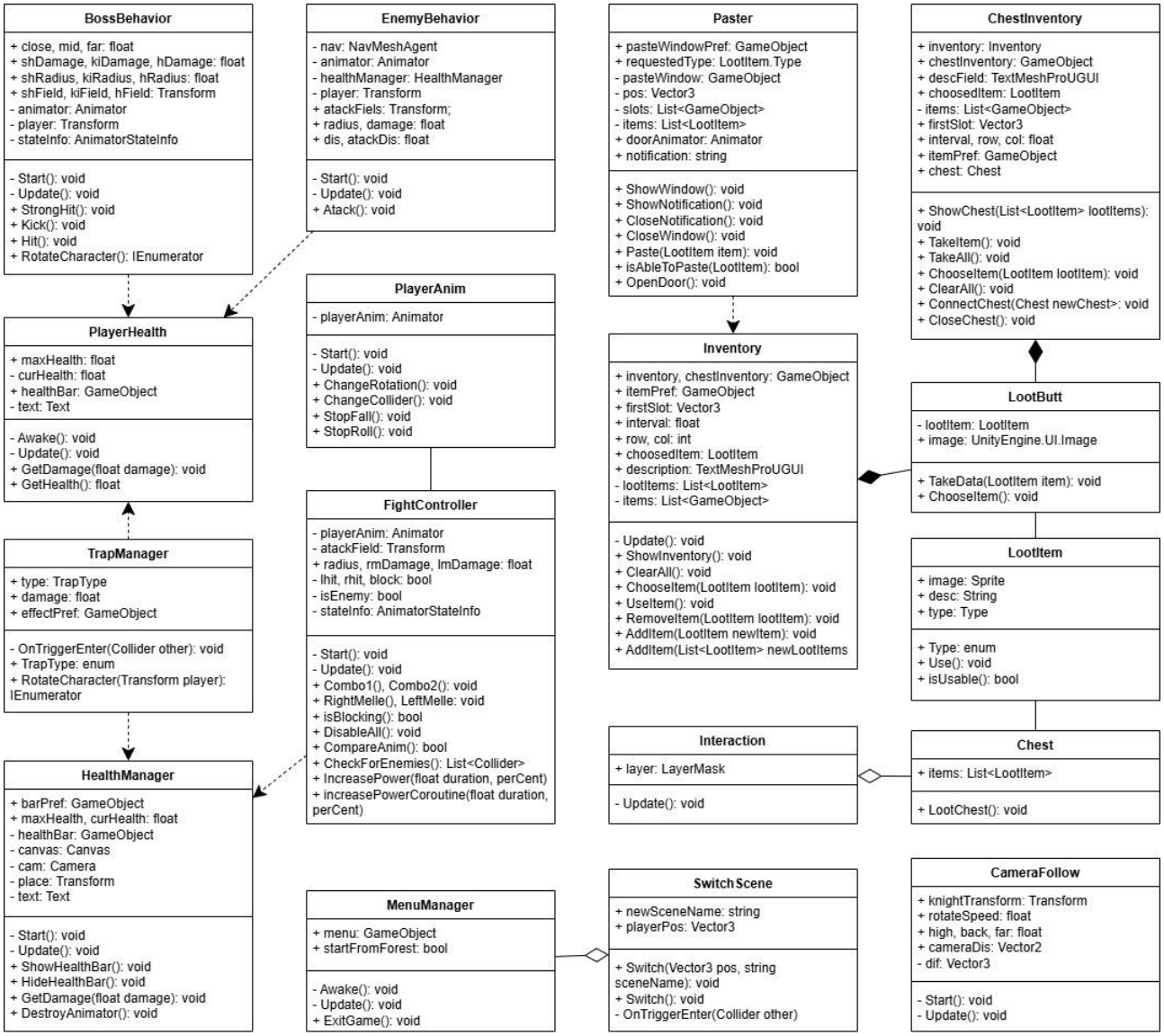


Рисунок 2.2 – Діаграма класів

Далі здійснено детальний опис деяких, найскладніших та найважливіших класів системи, їх полів, методів та зв’язків із іншими класами (див. додаток А):

1. BossBehavior. Клас описує поведінку боса як спеціального різновиду ворога з розширеним набором атак, дистанцій та анімаційних станів. Основна мета — керування логікою бою (тип атаки, момент удару, повороти до цілі), а також синхронізація дій із системою анімацій. Має

залежність від PlayerHealth: під час успішної атаки boss ініціює зменшення здоров’я гравця (через виклик отримання шкоди). Використовує Unity-підсистеми Animator, Transform, корутини (IEnumerator), тобто тісно інтегрований з механікою Game Loop.

Поля:

– close, mid, far: float — пороги дистанцій до гравця, які використовуються для вибору “сценарію” поведінки (ближній/середній/дальній бій);
– shDamage, kiDamage, hDamage: float — значення шкоди різних типів атак (умовно: “сильний удар”, “удар ногою/kick”, “hit” тощо);
– shRaduis, kiRaduis, hRadius: float — радіуси ураження для кожного типу атаки (використовуються при перевірці попадання);.
– shField, kiField, hField: Transform — розташування епіцентру зони ураження для кожного типу атаки;
– animator: Animator — посилання на Animator, що керує анімаціями боса;
– player: Transform — ціль (гравець), відносно якої обчислюються дистанції та напрям;
– stateInfo: AnimatorStateInfo — дані про поточний стан анімації (корисно для таймінгу ударів/комбо).

Методи:

– Start() — ініціалізація полів, пошук посилань (player/animator), початкові налаштування;
– Update() — “цикл поведінки”: вимір дистанції, вибір атаки, контроль повороту, запуск анімацій;
– StrongHit(), Kick(), Hit() — логіка конкретних атак (визначення попадання в радіусі/полі, застосування шкоди, запуск анімації);
– RotateCharacter(): IEnumerator — корутина [8] для повороту гравця до боса, перед анімацією його відкидання після отримання атаки.

2. FightController. Центральний клас бойової логіки гравця: керування

комбо-атаками, таймінгами, блокуванням дій, взаємодія з ворогами, а також узгодження бойових дій із анімацією. Має залежність від HealthManager (на діаграмі штрихова стрілка): часто це означає, що бойова система може взаємодіяти зі здоров'ям ворогів або UI-станами здоров'я. Працює в парі з PlayerAnim/Animator — це критично для бойової системи, бо удари мають відповідати анімаційним таймінгам.

Поля:

- playerAnim: Animator — Animator для запуску/перемикання бойових анімацій;
- attackField: Transform — точка/зона атаки гравця;
- radius, rmDamage, lmDamage: float — радіус ураження та шкода для різних типів атак;
- hit, right, block: bool — прапорці станів: який тип атаки відбувся, чи активний блок;
- isEnemy: bool — прапорець, що може позначати наявність ворога у зоні видимості;
- stateInfo: AnimatorStateInfo — інформація про поточний анімаційний стан;

Методи:

- Start(), Update() — ініціалізація та щокадрове керування бойовим станом;
- Combo(), Combo2() — логіка комбінацій (послідовні удари за таймінгом);
- RightMelle(), LeftMelle() — виконання конкретних ударів (права/ліва атака);
- isBlocking(): bool — перевірка чи активний блок (може використовуватись ворогами/логікою шкоди);
- DisableAll() — "скидання" бойових станів/прапорців (після комбо або при перериванні);
- CompareAnim(): bool — перевірка поточного анімаційного стану

(наприклад, чи можна продовжувати комбо);

- CheckForEnemies(): List<Collider> — пошук ворогів у зоні атаки (наприклад, через overlap-методи фізики);
- IncreasePower(float duration, perCent), increasePowerCoroutine(float duration, perCent) — тимчасове підсилення параметрів шкоди, реалізоване через корутину.

3. HealthManager. Узагальнений менеджер здоров'я для ворогів, що поєднує логіку значень здоров'я та UI-відображення. У діаграмі він має багато UI-полів, тому виступає як "прошарок" між даними здоров'я і відображенням. На нього посилаються EnemyBehavior, TrapManager і FightController (за діаграмою): тобто всі підсистеми, які впливають на здоров'я ворогів, можуть працювати через єдиний менеджер.

   Поля:

   - barPref: GameObject — префаб смуги здоров'я;
   - maxHealth, curHealth: float — максимальне і поточне здоров'я;
   - healthBar: GameObject — поточний екземпляр смуги здоров'я;
   - canvas: Canvas — Canvas для UI;
   - cam: Camera — камера для коректного позиціонування UI над персонажем;
   - place: Transform — точка, до якої прив'язується UI (наприклад, над головою ворога);
   - text: Text — текстовий індикатор (значення HP).

   Методи:

   - Start(), Update() — створення та позиціонування UI та регулярне оновлення позиції шкали здоров'я ворогів відносно їх положення;
   - ShowHealthBar(), HideHealthBar() — керування видимістю індикатора здоров'я;
   - GetDamage(float damage) — застосування шкоди до curHealth з подальшим оновленням інтерфейсу;
   - DestroyAnimator() — логіка "завершення" об'єкта, вимкнення

анімацій після смерті.

4. Inventory. Основний клас інвентаря: зберігає список луту, керує UI інвентаря, вибраним предметом, додаванням та видаленням предметів, та взаємодіє з кнопками і елементами відображення. Отримує дані від Paster передача предметів (наприклад, у механізм для його полагодження). Має композицію з LootBut, котрий відповідає за реалізацію клавіш у інтерфейсі. Inventory керує UI-кнопками предметів і керує їх життєвим циклом. Працює з LootItem як з моделлю даних предметів.

   Поля:

   – inventory, chestInventory: GameObject — панелі інвентаря гравця та інвентаря скрині (для режиму взаємодії);
   – itemPref: GameObject — префаб UI-елемента предмета (кнопки/іконки);
   – firstSlot: Vector3 — координати першого слоту в UI-сітці;
   – interval: float — відступ/крок між слотами;
   – row, col: int — логіка сітки (рядки/колонки), позиціонування предметів;
   – chosenItem: LootItem — вибраний предмет;
   – description: TextMeshProUGUI — опис предмета в UI;
   – loottItems: List<LootItem> — список предметів як даних (модель);
   – items: List<GameObject> — список UI-об’єктів (візуальне представлення).

   Методи:

   – Update() — контроль ввімкнення/вимкнення інвентаря, оновлення вибору тощо;
   – ShowInventory() — відображення UI та побудова сітки предметів;
   – ClearAll() — очищення UI-сітки (видалення/деактивація слотів);
   – ChooseItem(LootItem lootItem) — вибір предмета для подальших дій із ним та оновлення опису;

- UseItem() — використання вибраного предмета (виклик LootItem.Use() або подібної логіки);
- RemoveItem(LootItem lootItem) — видалення предмета з інвентаря.
- AddItem(LootItem newItem) — додавання одного предмета;
- AddItem(List<LootItem> newLootItems) — масове додавання (наприклад, після луту скрині).

Далі проведено більш загальний, але не менш важливий опис решти класів системи котрі відповідають за керування гравця, поведінку ворогів, логіку інтерфейсу, і тд.:

1. EnemyBehavior. Це базовий контролер поведінки звичайних ворогів. Він використовує компоненти навігації, анімації та управління здоров'ям (HealthManager), щоб ворог міг пересуватися, переслідувати гравця та здійснювати атаки. EnemyBehavior має зв'язок із гравцем, оскільки координує напрямок атаки та дистанцію до нього. Цей клас є фундаментальним для всієї бойової системи та налаштовує індивідуальну поведінку кожного противника.
2. PlayerHealth. Відповідає за здоров'я гравця, його відображення на UI та обробку отримання шкоди. Клас взаємодіє з системами, що завдають ушкодження (наприклад, EnemyBehavior, BossBehavior). Він лише керує станом здоров'я, не відповідаючи за іншу логіку персонажа.
3. PlayerAnim. Керує анімацією гравця, змінює стани та реагує на дії, що надходять із контролера бою або руху. Він пов'язаний із аніматором персонажа, викликає анімаційні переходи та забезпечує правильне відтворення ударів, перекатів і рухів. Отримує команди від FightController і системи руху.
4. TrapManager. Керує пастками на рівні — їх типовою поведінкою, видом ефекту та взаємодією з гравцем або ворогами. TrapManager зазвичай пов'язаний із тригерами Unity, а також взаємодіє з класами здоров'я (PlayerHealth, HealthManager) для завдавання шкоди. Також може мати

посилання на ефекти (візуальні чи звукові), виступаючи координатором пасток.

5. ChestInventory. Опрацьовує систему підбору предметів із об’єктів що це передбачають, наприклад трупи ворогів, або скрині. Містить список предметів що є у цьому об’єкті, взаємодіє з Inventory гравця поміщаючи туди предмети та Chest, котрий зберігає список предметів. Активно працює з UI для показу інтерфейсу здобичі. Має зв’язок із LooButt, оскільки той відображає список предметів, що містяться у здобичі.
6. Paster. Відповідає за систему вставлення відповідного типу предметів у контейнер. Він пов’язаний із Inventory, ChestInventory, LootItem та системами UI. Керує логікою створення вікна для вставлення, показу інвентарю та перевірки, чи може гравець вставити обраний у інвентарі предмет у контейнер.
7. LootButt. Керує кнопками здобичі у інвентарі гравця та інвентарі здобичі. Він посилається на конкретний LootItem та взаємодіє з Inventory або ChestInventory залежно від контексту. Цей клас є важливою частиною механіки підбору предметів.
8. LootItem. Описує предмети, що можуть бути в інвентарі. Містить дані про зображення, тип предмету, опис і доступні дії (наприклад, "використати"). Він не має складної логіки, але є основою всієї системи здобичі та інвентаря.
9. Chest. Являє собою список предметів що є у здобичі, котрі можна буде підібрати. Напряму взаємодіє із LootItem, котрі являють собою предмети у здобичі. Interaction взаємодіє із Chest коли відкриває вікно здочі натискаючи на здобич.
10. Interaction. Базовий клас, який керує взаємодією гравця з об’єктами світу. Використовує layer mask для перевірки взаємодій і визначає, на що гравець може навестися чи з чим може взаємодіяти. Він тісно працює з Chest та Paster та іншими об’єктами відчиняючи вікна здобичі та вставлення відповідно.

11. MenuManager. Керує інтерфейсом меню гри — відкриттям, закриттям. Також виконує важливу роль у перемиканні сцен, тим самим взаємодіючи із класом SwitchScene.
12. SwitchScene. Відповідає за перемикання сцен. Має зв’язки з об’єктом гравця (позиція), щоб при перемиканні сцени перемістити його відразу у передбачене місце, тригерами, які запускають зміну сцени. Клас використовує Unity-систему SceneManager.
13. CameraFollow. Керує слідуванням камери за гравцем, обчислює позицію зміщення, плавність руху камери та орієнтацію. Зв’язаний із об’єктом гравця, а саме його місцем розташування. Має поля для контролювання відстані камери від гравця. Є технічним компонентом, який забезпечує правильне відображення ігрової сцени.

Таким чином, проєктування класів системи прототипу гри «Adventure Guild» дозволило сформувати чітку та структуровану об’єктно-орієнтовану архітектуру, у якій кожен клас відповідає за окремий аспект ігрової логіки або взаємодії з користувачем. Виділення ключових класів із детальним описом та узагальнений розгляд допоміжних компонентів забезпечують збалансований підхід до проєктування, що поєднує глибину аналізу та цілісність архітектурної моделі. Діаграма класів відображає цілісну структуру програмного продукту та підкреслює взаємодію між логічно розділеними підсистемами. Застосування різних типів зв’язків дозволило точно визначити ступінь залежності між класами й забезпечити правильну організацію архітектури. Побудована модель демонструє, як окремі компоненти гри об’єднані в єдину злагоджену систему, що підтримує гнучкість, масштабованість і подальше розширення гри в рамках компонентної архітектури.

## 2.4 Аналіз взаємодії об’єктів за часом

Аналіз взаємодії об’єктів за часом є важливим етапом проєктування програмного забезпечення, оскільки дозволяє дослідити динамічну поведінку системи та порядок обміну повідомленнями між її компонентами під час виконання основних сценаріїв роботи. На відміну від статичних моделей, які відображають структуру системи та взаємозв’язки між класами, часовий аналіз зосереджується на послідовності виконання операцій і зміні станів об’єктів у процесі роботи програми.

У межах даного підрозділу розглядається взаємодія ключових компонентів прототипу гри «Adventure Guild» у часі з використанням діаграм послідовностей UML. Такі діаграми дозволяють наочно відобразити порядок викликів методів, умови виконання окремих дій та реакцію системи на події, ініційовані користувачем або ігровим середовищем. Особливу увагу приділено аналізу сценаріїв, що пов’язані з бойовою взаємодією, керуванням станами персонажів та обробкою наслідків ігрових подій.

Застосування діаграм послідовностей у процесі проєктування сприяє глибшому розумінню логіки функціонування системи, виявленню потенційних проблем синхронізації та забезпеченню узгодженої роботи між різними підсистемами гри, зокрема бойовою логікою, системою анімацій та механізмами керування станами об’єктів. Отримані результати аналізу використовуються для обґрунтування реалізаційних рішень, описаних у наступних підрозділах роботи

### 2.4.1 Сценарій виконання атаки

Нижче, на рисунку 2.3, подана діаграма послідовностей відображає сценарій виконання бойової атаки гравцем та подальшу обробку її результатів у прототипі гри «Adventure Guild». Діаграма ілюструє динамічну взаємодію між користувачем, контролером бойової логіки, системою анімацій та менеджером здоров’я неігрового персонажа.

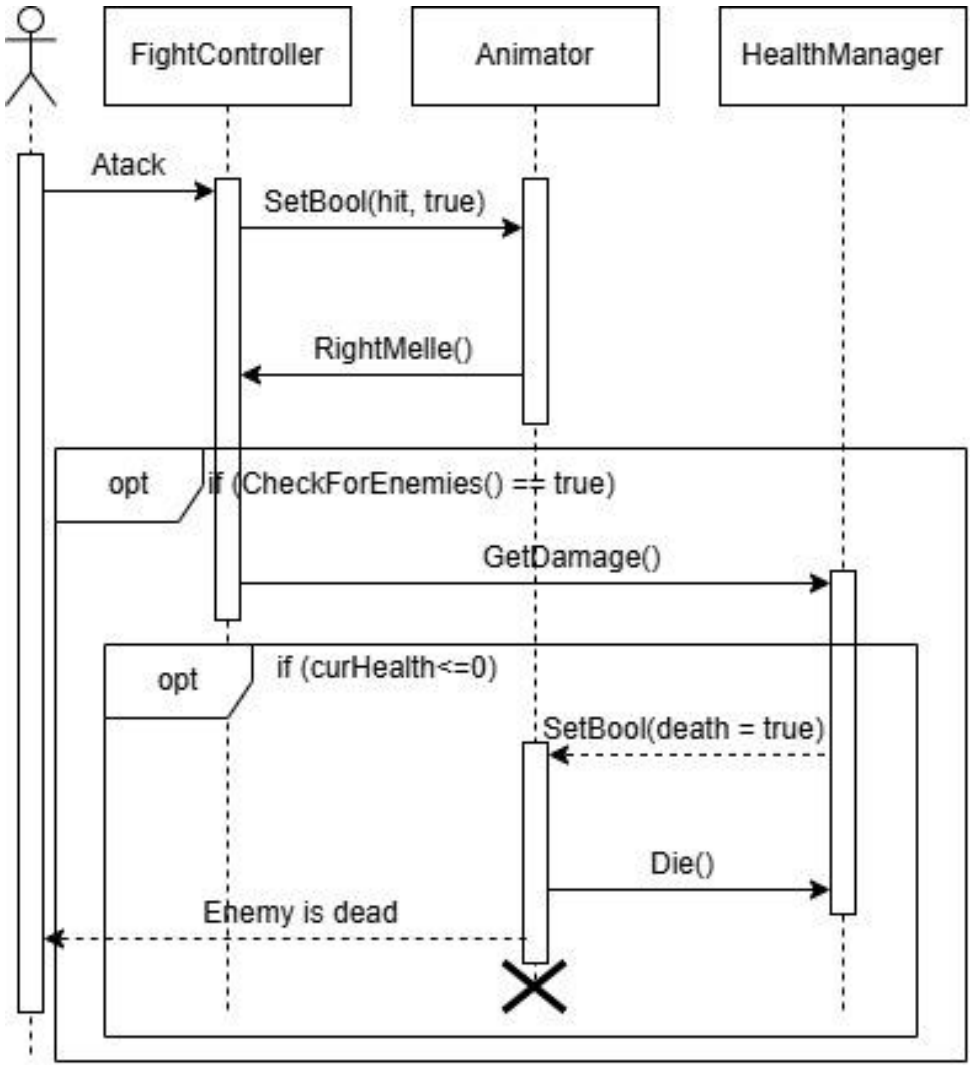


Рисунок 2.3 – Діаграма послідовностей сценарію атаки

Процес починається з ініціації атаки гравцем, після чого відповідна команда передається до класу FightController. Контролер бою запускає атаку, встановлюючи параметр hit у компоненті Animator у значення true, що призводить до відтворення анімації удару персонажа. Такий підхід забезпечує синхронізацію ігрової логіки з візуальним відображенням бойових дій.

Під час виконання анімації атаки за допомогою механізму подій анімації (Animation Events) викликається метод RightMelle() класу FightController. У межах цього методу здійснюється перевірка наявності ворогів у зоні атаки гравця шляхом виклику функції CheckForEnemies(). У разі виявлення противника виконується нанесення шкоди шляхом виклику методу GetDamage() класу HealthManager, який відповідає за зміну стану здоров’я ворога

Після застосування шкоди відбувається перевірка поточного значення здоров’я. Якщо рівень здоров’я стає меншим або дорівнює нулю, в компоненті Animator ворога встановлюється параметр death у значення true, що ініціює відтворення анімації смерті. До цієї анімації також прив’язаний виклик методу Die() класу HealthManager, який деактивує компоненти, відповідальні за поведінку “живого” ворога, та переводить ігровий об’єкт у неактивний стан.

Таким чином, представлена діаграма послідовностей демонструє повний цикл бойової взаємодії — від ініціації атаки гравцем до коректного завершення життєвого циклу ворога. Вона наочно показує узгоджену роботу бойової логіки, системи анімацій та механізму керування станом здоров’я, що забезпечує цілісність і передбачуваність ігрового процесу.

### 2.4.2 Сценарій збору предметів

Далі, на рисунку 2.4, подана діаграма послідовностей що відображає сценарій взаємодії користувача з ігровим об’єктом типу “скриня” у прототипі гри «Adventure Guild». Діаграма ілюструє процес відкриття скрині, перегляду її вмісту, вибору та отримання предмета, а також завершення взаємодії шляхом закриття інтерфейсу скрині.

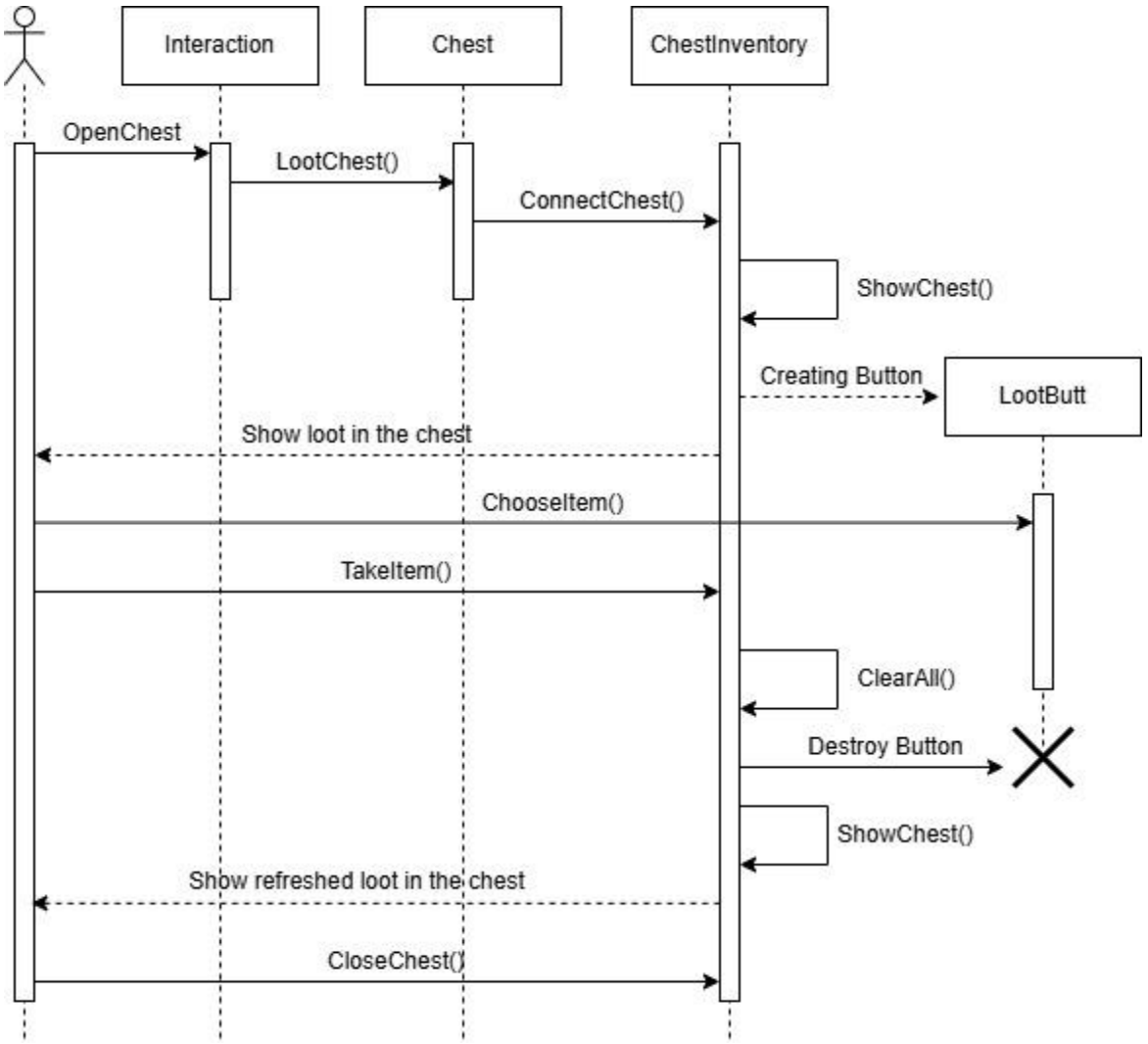


Рисунок 2.4 – Діаграма послідовностей сценарію взаємодії із скринею

Сценарій починається з ініціації дії користувачем, який надсилає команду на відкриття скрині до класу Interaction. У відповідь клас Chest, що зберігає список предметів, які знаходяться у скрині, викликає метод LootChest(). Даний метод відповідає за підготовку даних для відображення вмісту скрині та ініціює подальшу взаємодію з інтерфейсом інвентаря скрині.

На наступному етапі клас Chest викликає метод ConnectChest() класу ChestInventory, передаючи йому актуальний список предметів. Клас ChestInventory виконує підключення скрині до інтерфейсу та викликає метод ShowChest(), у результаті чого відкривається вікно інвентаря скрині. Під час відображення інтерфейсу для кожного предмета створюється відповідний елемент користувацького інтерфейсу типу LootButt, який візуально представляє одиницю

луту та забезпечує можливість взаємодії з нею.

Після відображення вмісту скрині користувач обирає потрібний предмет, натискаючи на відповідну кнопку. У результаті викликається метод ChooseItem() класу LootButt, що встановлює вибраний предмет активним для подальших дій. Далі користувач ініціює дію отримання предмета, натискаючи кнопку “Взяти”, яка викликає метод TakeItem() класу ChestInventory. У межах цього методу відбувається передача предмета до інвентаря гравця та оновлення вмісту скрині.

Після вилучення предмета інтерфейс скрині оновлюється: раніше створені елементи LootButt очищуються, а потім формується новий список кнопок, що відображає поточний вміст скрині без вилученого предмета. Таким чином забезпечується актуальне відображення стану інвентаря скрині. Завершальним етапом сценарію є виклик методу CloseChest() класу ChestInventory, який закриває інтерфейс скрині та завершує взаємодію користувача з даним ігровим об’єктом.

Представлена діаграма послідовностей демонструє узгоджену роботу механізму взаємодії, системи зберігання предметів та користувацького інтерфейсу, забезпечуючи коректну і зрозумілу для користувача реалізацію процесу роботи зі скринею в ігровому середовищі.

У результаті аналізу взаємодії об’єктів за часом за допомогою діаграм послідовностей було формалізовано динамічну поведінку ключових підсистем прототипу гри «Adventure Guild». Побудовані діаграми дозволили наочно відобразити порядок виконання операцій, обмін повідомленнями між об’єктами та зміну їхніх станів у межах основних ігрових сценаріїв, зокрема бойової взаємодії та роботи з ігровими контейнерами.

Отримані моделі підтверджують узгодженість проєктних рішень і демонструють коректну інтеграцію бойової логіки, системи анімацій, механізмів керування станами об’єктів та користувацького інтерфейсу. Діаграми послідовностей слугують важливим доповненням до статичних UML-моделей, забезпечуючи цілісне уявлення про функціонування програмного продукту та створюючи основу для реалізації і тестування відповідних компонентів системи в подальших етапах розробки.

У межах даного розділу було виконано проєктування програмного забезпечення прототипу гри «Adventure Guild», що дозволило формалізувати як статичну, так і динамічну структуру системи. За допомогою діаграми варіантів використання визначено основні сценарії взаємодії користувача з програмним продуктом, а діаграма класів відобразила архітектуру системи, її ключові компоненти та взаємозв’язки між ними. Побудовані діаграми послідовностей доповнили модель системи, наочно продемонструвавши динаміку виконання основних ігрових процесів та порядок взаємодії між об’єктами під час реалізації ключових сценаріїв. У сукупності представлені проєктні моделі забезпечують цілісне розуміння структури програмного продукту та слугують надійною основою для реалізації і тестування ігрових механік, описаних у наступних розділах роботи.

# 3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У даному розділі розглянуто процес практичної реалізації програмного забезпечення відповідно до прийнятих проєктних рішень та сформульованих вимог. Основна увага приділяється реалізації ключових ігрових механік, внутрішній логіці роботи системи, взаємодії програмних компонентів, а також створенню графічної складової та ігрового контенту. Розробка здійснювалася з використанням ігрового рушія Unity та мови програмування C#, що дозволило ефективно реалізувати компонентно-орієнтовану архітектуру та забезпечити гнучкість і масштабованість програмного продукту.

Процес розробки охоплював створення та налаштування ігрових сцен, написання скриптів поведінки об’єктів, інтеграцію анімацій, реалізацію систем бою, взаємодії, інвентарю, штучного інтелекту ворогів і користувацького інтерфейсу. Окрему увагу було приділено узгодженню логіки гри з анімаційною системою, оптимізації роботи компонентів та забезпеченню стабільної роботи прототипу. Наведені у розділі підрозділи демонструють практичну реалізацію спроєктованої системи та підтверджують працездатність запропонованих архітектурних і технічних рішень.

## 3.1 Структура проєкту

Структура проєкту гри “Adventure Guild” організована таким чином, щоб забезпечити зручність розробки, логічне групування ресурсів та легке масштабування гри. У межах рушія Unity використовуються окремі папки для моделей, анімацій, матеріалів, сцен, префабів та інших елементів, що дозволяє підтримувати чистоту проєкту й швидко орієнтуватися у всіх його частинах.

Основні директорії проєкту:

1. Animations – містить усі анімації гравця, ворогів та інших об’єктів. Тут зберігаються анімаційні кліпи та контролери (Animator Controller), що керують переходами між станами персонажів.

2. Materials – зберігає матеріали, які використовуються у грі для відображення поверхонь моделей, предметів і довкілля.
3. Models – містить 3D моделі персонажів, ворогів, предметів та елементів оточення, імпортовані з зовнішніх редакторів.
4. Packages – технічна папка Unity, що містить встановлені додаткові модулі та залежності.
5. Prefabs – включає префаби гравця, ворогів, предметів, елементів інтерфейсу та частин рівня. Префаби дозволяють швидко створювати нові екземпляри об’єктів із заданими компонентами та поведінкою.
6. Resources – містить ресурси що використовуються для відображення UI елементів, а саме: зображення, іконки, фони.
7. Scenes – містить усі сцени проєкту. Кожна з них відповідає окремому ігровому середовищу або службовій області.
8. Scripts – папка, що включає всі скрипти гри. На поточному етапі вони не розбиті на підкатегорії та містяться в одному місці, що спрощує доступ, але в майбутньому може бути розширене структуризацією за функціональними системами.
9. Settings – папка з конфігураційними файлами Unity (Input System, Render Settings, тощо).

У проєкті створено три основні сцени, кожна з яких виконує свою роль у побудові ігрового процесу:

1. Forest – перший рівень гри, стилізований під лісову локацію, де гравець стикається з гоблінами. Сцена містить геометрію місцевості, ворогів, тригери та об’єкти навколишнього середовища.
2. Factory – другий рівень із технологічною атмосферою, населеним автоматонами. Локація відрізняється стилем, складністю та типами ворогів.
3. Game – службова сцена, що містить гравця, камеру та весь інтерфейс користувача: HUD, панелі інвентарю, меню та інші UI-елементи. Ця сцена завантажується першою, а об’єкти гравця та інтерфейсу

переносяться між рівнями через DontDestroyOnLoad, забезпечуючи стабільну та безперервну роботу ігрових систем.

Така організація дозволяє легко керувати ресурсами та сценами, відокремлюючи візуальні, логічні та технічні елементи. Структура проєкту побудована так, щоб підтримувати як розвиток гри, так і її масштабування у майбутньому.

### 3.2 Реалізація основного функціоналу гри

Реалізація основного функціоналу прототипу 3D-гри жанру RPG «Adventure Guild» здійснювалася з використанням компонентно-орієнтованого підходу, який є базовою архітектурною концепцією ігрового рушія Unity. Відповідно до цього підходу, кожен елемент ігрового середовища — ігровий персонаж, вороги, предмети, інтерактивні об’єкти, механізми та елементи інтерфейсу користувача — представлений у вигляді об’єкта сцени з набором компонентів, що визначають його поведінку, фізичні властивості, анімаційні стани та взаємодію з іншими об’єктами. Така організація дозволяє забезпечити модульність, гнучкість та повторне використання програмного коду, а також спрощує процес розширення ігрової логіки.

Процес розробки програмного продукту організовувався з використанням гнучкого підходу до управління розробкою, що відповідає принципам методології Scrum. Реалізація функціоналу здійснювалася ітераційно, з поступовим додаванням та вдосконаленням окремих ігрових механік. Кожна ітерація передбачала постановку конкретних завдань, їх реалізацію, перевірку працездатності та аналіз отриманих результатів, що дозволяло своєчасно виявляти недоліки та коригувати архітектурні рішення.

На даному етапі роботи було реалізовано ключові механіки прототипу гри, зокрема керування ігровим персонажем, бойову систему, взаємодію з об’єктами ігрового світу, роботу зі скринями та інвентарем, систему анімацій, а також базові елементи інтерфейсу користувача. Для реалізації логіки використовувався

скриптовий підхід із застосуванням мови програмування C#, що забезпечило узгоджену взаємодію між компонентами системи та відповідність реалізації проєктним рішенням, описаним у попередніх розділах роботи.

У процесі реалізації прототипу гри «Adventure Guild» було розроблено низку ключових програмних компонентів, які забезпечують функціонування основних ігрових механік, взаємодію об’єктів, керування персонажем та роботу користувацького інтерфейсу. Архітектура реалізації побудована на принципах компонентно-орієнтованого підходу, що дозволяє чітко розмежувати відповідальність між окремими модулями та забезпечити узгоджену роботу всієї системи. Кожен компонент відповідає за окремий аспект ігрового процесу, а їх взаємодія формує цілісність, стабільність та передбачуваність роботи програмного продукту.

Одним із центральних елементів реалізації є система керування ігровим персонажем. Вона відповідає за обробку введення користувача, керування рухом персонажа, швидкістю пересування та переходами між різними анімаційними станами. Логіка керування інтегрована з камерою, що забезпечує коректне слідування за персонажем і підтримку відповідного ракурсу огляду. Система руху тісно пов’язана з бойовою механікою: під час виконання атак пересування персонажа може обмежуватися або модифікуватися, а дії гравця синхронізуються з відповідними анімаціями.

Система здоров’я реалізована у вигляді універсального компонента, який використовується як для гравця, так і для неігрових персонажів. Вона відповідає за зберігання та зміну поточного рівня здоров’я, обробку отриманої шкоди, а також за ініціацію подій, пов’язаних із загибеллю об’єкта. Для гравця система здоров’я передає дані до інтерфейсу користувача для оновлення індикатора HP, тоді як для ворогів вона активує анімацію смерті та виконує деактивацію відповідних компонентів після завершення життєвого циклу персонажа. Такий підхід забезпечує уніфіковану логіку обробки стану об’єктів і спрощує її подальше розширення.

Бойова система є окремим функціональним модулем, який відповідає за

обробку атак, визначення попадань та нанесення шкоди. Вона базується на перевірці наявності ворогів у зоні атаки під час виконання бойових анімацій, що реалізується за допомогою тригерних колайдерів або механізмів рейкастингу. Система враховує затримки між ударами, підтримує комбінації атак та забезпечує синхронізацію бойової логіки з анімаційною системою. Це дозволяє досягти природного та зрозумілого для користувача ігрового процесу.

Система інвентарю реалізована як окремий компонент, який відповідає за зберігання, відображення та використання ігрових предметів. Інвентар відображається через окрему панель користувацького інтерфейсу та забезпечує вибір предметів і виконання відповідних дій. Підбір предметів реалізовано через взаємодію з об’єктами типу “скриня”, які містять інформацію про доступний лут. Після отримання предмета відповідні дані передаються до інвентаря гравця, а інтерфейс оновлюється з урахуванням змін.

Важливу роль у реалізації ігрового процесу відіграє система штучного інтелекту ворогів. Поведінка неігрових персонажів базується на аналізі відстані до гравця та поточного стану взаємодії. За певних умов вороги переходять у режим переслідування, атаки або припиняють активні дії. Різні типи ворогів мають відмінні моделі поведінки: прості противники використовують базову логіку наближення й атаки, тоді як складніші вороги та боси мають розширені сценарії бою з декількома типами атак, вибір яких залежить від дистанції до гравця.

Окремим модулем реалізовано систему користувацького інтерфейсу, яка відповідає за відображення стану гри та взаємодію з користувачем. Інтерфейс відображає рівень здоров’я гравця, вміст інвентарю, меню паузи, підказки та інші інформаційні елементи. UI-система реагує на події, що відбуваються в ігровому процесі, та оновлює відображувані дані в режимі реального часу, забезпечуючи зручність та інформативність для користувача.

Усі реалізовані компоненти працюють узгоджено, формуючи цілісну ігрову систему. Взаємодія між модулями здійснюється через виклики методів, обробку подій та обмін даними, що дозволяє досягти високого рівня модульності та розширюваності. Така архітектура забезпечує стабільність прототипу гри

«Adventure Guild» та створює надійну основу для подальшого розвитку, зокрема додавання нових механік, типів ворогів або ігрових об’єктів без суттєвих змін у вже реалізованому функціоналі.

В основі реалізації бойової підсистеми прототипу гри «Adventure Guild» лежить клас FightController, який виконує роль центрального координатора бойової логіки гравця. Даний клас відповідає за обробку атак, керування таймінгами ударів і синхронізацію бойових дій з анімаційною системою. Для запуску відповідних анімацій FightController взаємодіє з компонентом PlayerAnim, а для визначення результатів удару використовує залежності від об’єктів класів EnemyBehavior або BossBehavior. Такий тип взаємодії є прикладом залежності, коли компонент тимчасово використовує функціональність інших об’єктів для виконання конкретної операції, не володіючи ними та не керуючи їх життєвим циклом.

Поведінка ворогів у грі реалізована за допомогою класів EnemyBehavior та BossBehavior, які відповідають за пересування, вибір моменту атаки та взаємодію з гравцем. Ці класи мають асоціативні зв’язки з компонентами HealthManager та Animator, що забезпечує узгоджену роботу логіки поведінки, анімацій та обробки шкоди. Поведінка ворогів залежить від позиції гравця, що дозволяє динамічно визначати напрямок руху, дистанцію атаки та поточний стан. Для босів реалізовано складнішу модель поведінки, яка включає декілька типів атак, різні зони ураження та більш детальне керування таймінгами дій, що підвищує складність ігрового процесу.

Компоненти, пов’язані з обробкою здоров’я, реалізовані через класи PlayerHealth та HealthManager. Дані класи мають композиційні зв’язки з візуальними елементами інтерфейсу, що означає їх тісну прив’язаність до життєвого циклу відповідних ігрових об’єктів. У разі знищення персонажа або ворога разом з ним деактивуються й відповідні індикатори здоров’я. Асоціативні зв’язки між HealthManager та класами поведінки ворогів забезпечують коректний обмін інформацією під час нанесення шкоди та обробки наслідків бойових взаємодій.

Система інвентаря реалізована у вигляді групи взаємопов’язаних класів Inventory, ChestInventory, LootItem та LootButt. Класи Inventory і ChestInventory мають агрегаційні зв’язки з LootItem, оскільки предмети можуть існувати незалежно від конкретного контейнера або інтерфейсу, в якому вони відображаються. Клас Paster виконує роль посередника між предметами, що знаходяться в ігровому світі, та інвентарем гравця, забезпечуючи контекстну взаємодію та передачу даних. Об’єкт типу Chest асоційований з ChestInventory, відкриваючи доступ до вмісту скрині, однак не володіє інвентарем безпосередньо, що дозволяє оновлювати інтерфейс незалежно від існування конкретної скрині.

Окремий блок системи становлять компоненти взаємодії та технічної інфраструктури гри. Клас Interaction відповідає за визначення об’єктів, з якими може взаємодіяти гравець, використовуючи інформацію про шари колізій. Він утворює слабкі залежності з такими компонентами, як Chest, TrapManager та іншими інтерактивними об’єктами, оскільки викликає їхні методи без збереження посилань як власних полів. Класи MenuManager та SwitchScene забезпечують керування інтерфейсом користувача та переходами між сценами відповідно, маючи мінімальні прямі зв’язки з ігровими об’єктами. Компонент CameraFollow асоційований з ігровим персонажем і відповідає за коректне слідування камери, забезпечуючи зручний огляд ігрового світу [1-3].

Загалом, реалізація основного функціоналу прототипу гри «Adventure Guild» демонструє узгоджене використання різних типів зв’язків між компонентами — залежностей, асоціацій, агрегацій та композицій. Такий підхід забезпечує модульність, зменшує зв’язність між підсистемами та створює гнучку архітектуру, придатну для подальшого розвитку і масштабування програмного продукту.

Таким чином, реалізація основного функціоналу прототипу гри «Adventure Guild» базується на чітко структурованій компонентно-орієнтованій архітектурі, у межах якої кожен програмний компонент виконує визначену роль та взаємодіє з іншими елементами системи через формалізовані зв’язки. Запропонований підхід забезпечує узгоджену роботу ігрових механік, стабільність виконання

програмного продукту та можливість його подальшого розширення, що є важливим для розвитку гри та впровадження нових функціональних можливостей на наступних етапах розробки.

### 3.3 Створення графічної складової та контенту гри

У процесі розробки гри “Adventure Guild” значну увагу було приділено створенню графічної складової та наповненню гри контентом. Цей етап включав підбір моделей, налаштування анімацій персонажів, створення ігрових локацій та формування цілісного візуального стилю проєкту. Для отримання якісних ресурсів використовувалися спеціалізовані платформи, а подальша інтеграція здійснювалася всередині Unity.

Під час роботи над графічною частиною було застосовано такі платформи та сервіси:

1. CGTrader[6] — інтернет-платформа для пошуку та завантаження 3D-моделей. На цьому ресурсі були знайдені моделі персонажів, ворогів, об’єктів оточення (дерева, каміння, скрині, намети, механічні елементи) та інші елементи, необхідні для побудови рівнів.
2. Mixamo[5] — сервіс з великою бібліотекою безкоштовних анімацій. На Mixamo були підібрані та завантажені анімації для:
   - персонажа-гравця (хідба, біг, атаки мечем, отримання урону, смерть);
   - гоблінів та інших ворогів (пересування, атака, смерть);
   - фінального боса-автоматона (атаки, важкі удари, смерть, пересування).
3. AccuRIG[7] — програма для швидкого ригінгу моделей, які не мають скелету. Вона використовувалась для підготовки моделей, завантажених із CGTrader, щоб потім коректно застосувати до них анімації із сайту Mixamo.

Створення графічної складової та ігрового контенту прототипу гри «Adventure Guild» здійснювалося безпосередньо в середовищі Unity Editor, який надає комплексний набір інструментів для роботи з тривимірними сценами, об’єктами та візуальними ефектами. Unity Editor дозволяє в інтерактивному режимі формувати ігрові рівні, розміщувати моделі персонажів та об’єктів оточення, налаштовувати освітлення, камери, матеріали й анімації. Завдяки візуальному інтерфейсу та тісній інтеграції з програмною логікою, процес створення контенту є наочним, керованим і зручним для поетапного доопрацювання.

Особливою перевагою Unity Editor є можливість швидкої перевірки результатів змін у режимі реального часу, що значно спрощує процес налагодження та корекції графічної складової гри. Редактор підтримує роботу з ієрархією об’єктів сцени, системою префабів, матеріалів і шейдерів, що дозволяє забезпечити узгодженість візуального стилю та оптимізувати повторне використання елементів контенту. У поєднанні з інструментами анімації та налаштування фізики Unity Editor виступає як універсальне середовище, яке об’єднує створення графічного контенту та його інтеграцію з ігровою логікою в єдиний процес розробки.

### 3.3.1 Налаштування анімацій

У межах реалізації графічної складової та ігрової логіки прототипу гри «Adventure Guild» було налаштовано систему анімацій персонажів із використанням механізму Animator Controller ігрового рушія Unity [6,9,26]. Для кожного типу персонажів, зокрема гравця, звичайних ворогів та боса, було створено окремі контролери анімацій, які відображають специфіку їхньої поведінки та ролі в ігровому процесі.

Для всіх типів персонажів логіка нанесення шкоди була синхронізована з виконанням відповідних анімацій атак. З цією метою у визначених ключових кадрах анімацій використовувалися події анімації (Animation Events), які

викликають методи бойової логіки в скриптах. Такий підхід забезпечує точну відповідність між візуальним відтворенням удару та фактичним застосуванням шкоди, що є критично важливим для формування коректної та зрозумілої бойової системи.

Найбільш складним є Animator Controller гравця, який включає набір станів, що відповідають за пересування, бойові дії, отримання шкоди, смерть та інші можливі сценарії поведінки.

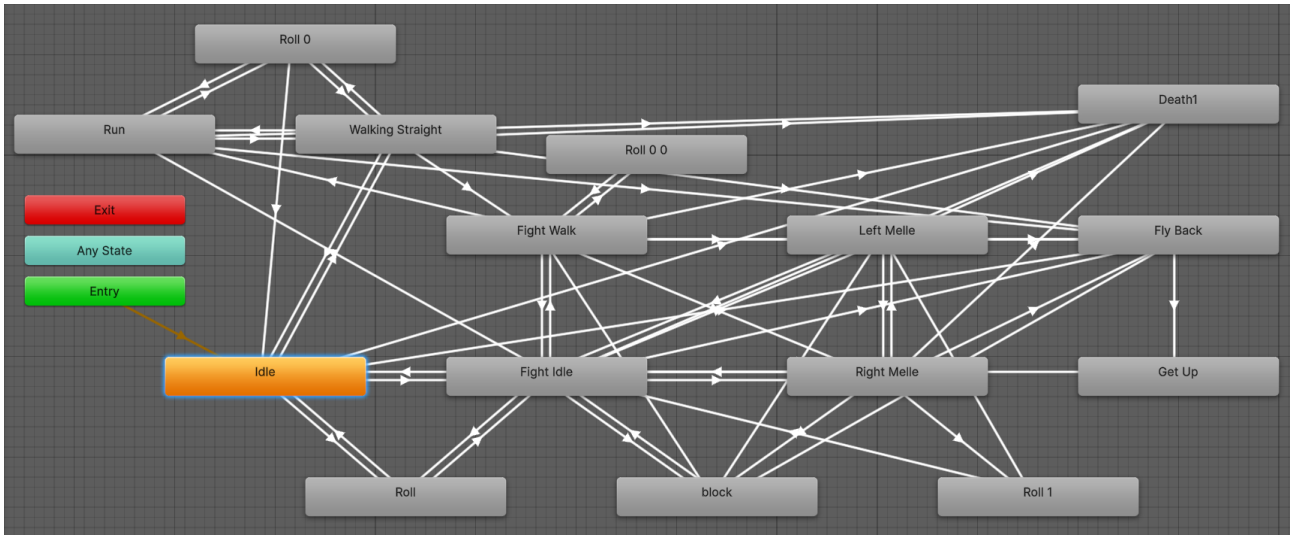


Рисунок 3.1 – Animator Controller гравця

Переходи між станами були налаштовані на основі параметрів логічного типу, що встановлюються відповідними компонентами ігрової логіки. Анімації поєднані плавними переходами, що дозволяє досягти природної поведінки персонажа та реалістичної реакції на дії користувача. Структуру взаємозв’язків і переходів між анімаційними станами гравця наведено у вікні Animator (рис. 3.1).

Для звичайних ворогів, зокрема гоблінів та автоматонів, було реалізовано спрощені контролери анімацій, які містять базові стани: стан спокою, пересування, атаки та смерті.

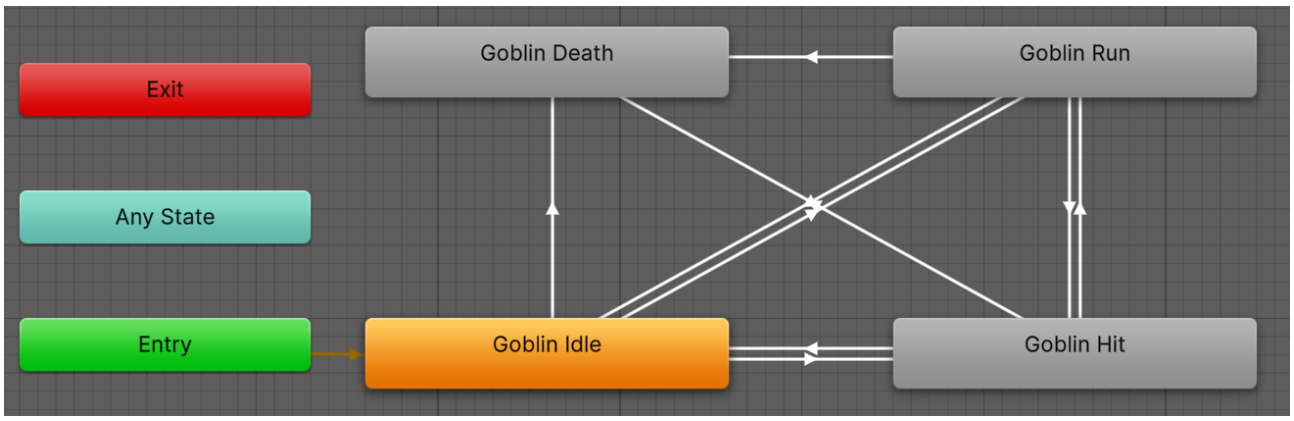


Рисунок 3.2 – Animator Controller ворогів

Пермикання між анімаційними станами здійснюється на основі логіки, реалізованої у класі EnemyBehavior, який аналізує поточний стан взаємодії з гравцем. Взаємозв'язки між основними анімаціями простих ворогів наведено на рисунку 3.2.

Для боса-автоматона була створена окрема, більш складна система анімацій, що відображає його розширений бойовий функціонал. Контролер анімацій боса включає стани спокою, пересування, різні типи атак (удари, стрибкові та заряджені атаки), проміжні стани затримки між атаками, а також стан смерті.

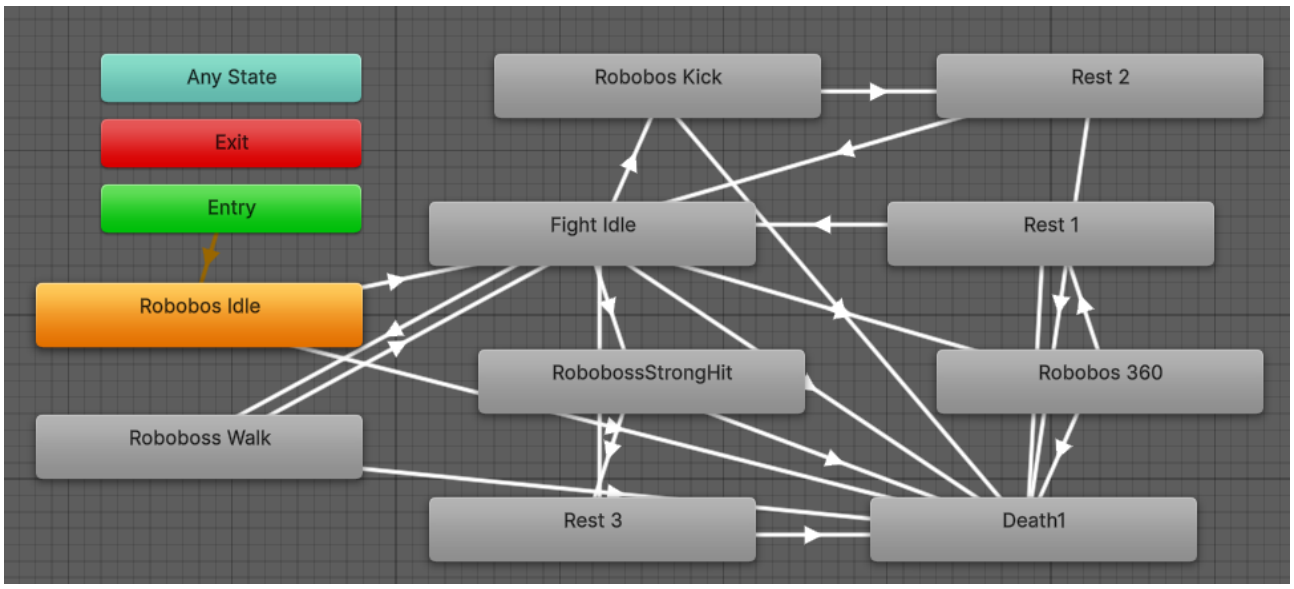


Рисунок 3.3 – Animation Controller боса-робота

Логіка вибору анімаційних станів боса залежить від відстані до гравця та поточного етапу бою, що дозволяє формувати динамічний і складний сценарій протистояння. Структуру взаємозв’язків між анімаціями, які відображають поведінку боса-робота, наведено на рисунку 3.3.

Таким чином, використання системи Animator Controller у поєднанні з подіями анімації дозволило реалізувати узгоджену та масштабовану систему анімацій, яка тісно інтегрована з бойовою логікою та поведінкою персонажів. Запропонований підхід забезпечує реалістичність ігрового процесу, підвищує його динамічність та створює основу для подальшого розвитку ігрових механік.

### 3.3.2 Створення рівнів

У межах реалізації графічної та контентної складової прототипу гри «Adventure Guild» було розроблено два повноцінні ігрові рівні — «Forest» та «Factory», кожен із яких має власну атмосферу, стилістику та набір ігрових механік. Рівні створювалися безпосередньо в середовищі Unity Editor із використанням тривимірних моделей, префабів, системи освітлення та тригерних зон, що забезпечило цілісність ігрового простору та логічну побудову ігрового процесу.

Рівень «Forest». Даний рівень є стартовою локацією гри та представляє собою невеликий лісовий масив, який слугує ознайомчим етапом для гравця. Основними елементами даного рівня є:

– дерев’яні намети гоблінського табору;
– декорації з дерев, каміння та інших природних об’єктів;
– скриня з корисними предметами (зілля лікування, зілля сили);
– кілька гоблінів, що перебувають у межах табору;
– вихід до наступного рівня — прохід у підземний завод Factory, який реалізовано за допомогою тригерної зони, розташованої біля входу.

Даний рівень не є надто складним за структурою та складністю, оскільки його основне призначення — поступове ознайомлення гравця з базовими

механіками гри, зокрема системою пересування, бойовою логікою, взаємодією з об’єктами та роботою інвентаря. Межі локації обмежені природними перешкодами у вигляді високих гір, що унеможливлює вихід гравця за межі сцени та сприяє збереженню ігрового балансу. Далі, на рисунку 3.4, зображено безпосередньо сам план рівня «Forest».

Рисунок 3.4 – Рівень «Forest»

Рівень «Factory». Другим ігровим рівнем є «Factory» — підземний механізований комплекс, виконаний у стилі фентезійного стімпанку з

використанням кам’яних стін, металевих конструкцій та елементів застарілої технології. План даного рівня наведено на рисунку 3.5.

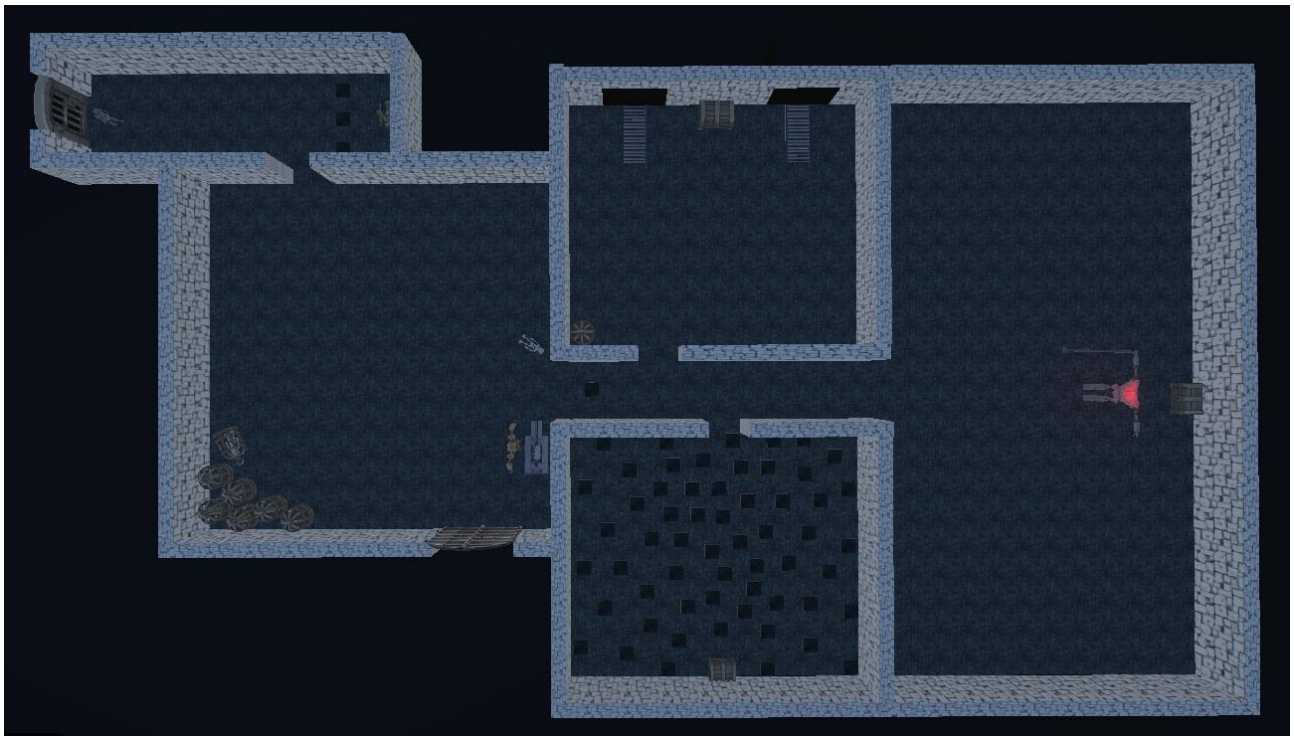

Рисунок 3.5 – План підземелля-фабрики автоматонів

Для формування відповідної атмосфери на рівні використано темний туман, який обмежує огляд місцевості та підкреслює замкнений характер підземелля. Оточення доповнене клітками з гоблінами, над якими проводяться експерименти, а також скелетами шукачів пригод, з яких гравець може отримати корисні предмети.

Початкове приміщення рівня містить скриню із зіллям лікування та гобліна, який після появи гравця починає рух у його напрямку, однак активує пастку. Такий сценарій дозволяє продемонструвати небезпеку підземелля та ознайомити гравця з механікою пасток. У наступній зоні розташований автоматон — фентезійний робот, що намагається полагодити важіль, який відкриває доступ до подальших приміщень.

Основною задачею гравця на даному рівні є збирання трьох частин зламаного механізму, необхідних для відкриття масивних металевих дверей.

Частини розміщені у трьох тематичних кімнатах:

- кімната з пастками — приміщення з натискними плитками-пастками, які після активації вибухають, завдаючи шкоди гравцеві та відкидаючи його на певну відстань. У кінці кімнати розташована скриня з однією з частин механізму та додатковими предметами. Додаткову складність створює можливість ланцюгового спрацювання пасток;
- кімната з конвеєрами — приміщення з двома активними конвеєрними лініями. Після активації тригера біля скрині двері зачиняються, а з конвеєрів по черзі з'являються шість автоматонів. Вихід із кімнати відкривається лише після їх знищення;
- кімната з босом-автоматоном — найбільша зала рівня, у якій відбувається бій із масивним бойовим роботом. Після перемоги над босом стає доступною скриня з третьою частиною механізму.

Після збирання всіх елементів механізму гравець може активувати важіль, відчинити головні двері заводу та завершити проходження рівня. На поточному етапі прототипу завершення рівня повертає гравця до локації «Forest», однак у подальшому планується розширення гри новими рівнями та розвиток сюжетної лінії.

Перехід між ігровими рівнями реалізовано із використанням спеціальної сцени Game, яка містить гравця, камеру та всі елементи користувацького інтерфейсу (інвентар, панель здоров'я, меню паузи тощо). Дані об'єкти позначені як DontDestroyOnLoad, що унеможливлює їх знищення під час завантаження нових сцен.

Після завантаження нової локації гравець переміщується до попередньо визначеної точки, яка відповідає логіці ігрового сюжету. Такий підхід забезпечує безперервний ігровий процес без втрати прогресу, перезавантаження інтерфейсу або дублювання об'єктів. Реалізована система переходів була ретельно протестована, зокрема перевірялася коректність роботи камери, позиціювання гравця та стабільність відображення UI після кожної зміни сцени.

Таким чином, створення ігрових рівнів у прототипі гри «Adventure Guild»

поєднує продумане просторове проєктування, атмосферне візуальне оформлення та технічну реалізацію ігрових механік. Рівні «Forest» і «Factory» демонструють послідовне ускладнення ігрового процесу, логічне введення нових викликів і поступове занурення гравця у світ гри. Реалізований підхід до побудови локацій та переходів між ними забезпечує цілісність ігрового досвіду, стабільність роботи системи та створює надійну основу для подальшого розвитку контенту і розширення ігрового світу.

Таким чином, у межах підрозділу 3.3 було реалізовано повноцінну графічну складову та ігровий контент проєкту, що охоплює налаштування системи анімацій, створення ігрових рівнів і організацію візуального середовища. Використання можливостей Unity Editor дозволило інтегрувати тривимірні моделі, анімації, освітлення та елементи інтерфейсу в єдину узгоджену систему, яка підтримує цілісність ігрового стилю та атмосферу фентезійного світу. Реалізовані рішення забезпечують наочність, динамічність і зрозумілість ігрового процесу, а також створюють міцну основу для подальшого розширення контенту та вдосконалення візуальної складової гри.

У межах розділу «Розробка програмного забезпечення» було послідовно розглянуто процес створення прототипу гри «Adventure Guild», починаючи з реалізації основного функціоналу та завершуючи формуванням графічної складової й ігрового контенту. Розробка здійснювалася на основі компонентно-орієнтованої архітектури ігрового рушія Unity з використанням мови програмування C#, що дозволило забезпечити чітке розмежування відповідальності між програмними компонентами, узгоджену взаємодію підсистем та відповідність реалізації проєктним рішенням, описаним у попередніх розділах роботи.

Результати даного етапу розробки підтверджують ефективність обраних технологічних і архітектурних підходів, оскільки створений програмний продукт демонструє стабільну роботу основних ігрових механік, цілісність візуального представлення та можливість подальшого розвитку.

# 4 ТЕСТУВАННЯ, ВПРОВАДЖЕННЯ ТА ПІДРИМКА ПРОЄКТУ

На даному етапі життєвого циклу програмного продукту основна увага зосереджена на тестуванні реалізованого функціоналу прототипу гри «Adventure Guild», що дозволяє оцінити відповідність програмної реалізації проєктним рішенням і вимогам, сформульованим у попередніх розділах роботи.

Тестування розглядається як невід'ємна складова процесу розробки, спрямована на виявлення помилок, перевірку коректності взаємодії між компонентами системи та оцінку стабільності роботи гри в різних ігрових сценаріях. Окрім цього, у розділі розглядаються питання впровадження програмного забезпечення та можливості його подальшої підтримки, що дозволяє визначити готовність створеного прототипу до розширення функціональності та використання на наступних етапах розвитку проєкту.

## 4.1 Тестування розробленого функціоналу

Підрозділ «присвячений перевірці коректності роботи основних ігрових механік та програмних компонентів прототипу гри «Adventure Guild». Метою тестування є виявлення можливих помилок у реалізації, перевірка стабільності взаємодії між підсистемами та оцінка відповідності реалізованого функціоналу вимогам, сформульованим на етапах аналізу та проєктування програмного забезпечення.

Тестування виконувалося переважно вручну шляхом багаторазового проходження ігрових сценаріїв у середовищі Unity та у зібраній версії проєкту. Особлива увага приділялася перевірці бойової системи, взаємодії з об'єктами ігрового світу, роботи інвентаря, поведінки ворогів, коректності переходів між рівнями та стабільності користувацького інтерфейсу. Отримані результати дозволили оцінити надійність створеного прототипу та визначити напрями для подальшого вдосконалення програмного продукту [16,17].

Окрему увагу під час проходження тестових сценаріїв було приділено

оцінці навантаження на систему та загальної продуктивності гри. У процесі тестування перевірялася стабільність частоти кадрів, коректність роботи гри за одночасної присутності кількох активних об’єктів, зокрема ворогів, анімаційних ефектів та елементів інтерфейсу користувача. Також аналізувалася поведінка системи під час переходів між рівнями, відкриття інвентаря та виконання бойових дій, що дозволило виявити потенційні вузькі місця та оцінити рівень оптимізації реалізованого функціоналу.

Далі наведено декілька прикладів сценаріїв тестування гри (див. додаток Б):

1. Перевірка функцій руху та бойових станів персонажа.

   Мета сценарію. Перевірити коректність реалізації системи керування персонажем, переходів між станами руху та бойовими станами, а також їх синхронізацію з анімаційною системою.

   Опис сценарію. Під час тестування гравець послідовно виконує всі доступні дії керування персонажем у межах ігрової сцени. Перевіряється реакція персонажа на введення з клавіатури, коректність переходів між анімаційними станами та відсутність конфліктів між руховими й бойовими механіками. У межах сценарію перевіряються такі стани та дії:

   – стан спокою (відсутність введення);
   – ходьба персонажа;
   – біг із підвищеною швидкістю;
   – виконання перекатів як захисної та маневрової дії;
   – перехід у бойову стійку;
   – пересування персонажа у режимі бойової готовності;
   – виконання удару справа;
   – виконання удару зліва;
   – виконання комбінації з двох послідовних ударів;
   – використання блоку для захисту від атак.

   Очікуваний результат:

   – кожна дія викликає відповідний анімаційний стан;

– переходи між станами відбуваються плавно, без різких ривків або затримок;
– бойова стійка коректно обмежує або змінює рух персонажа;
– удари виконуються лише в бойовому режимі;
– комбінація атак відпрацьовується в правильній послідовності;
– блок активується та деактивується відповідно до введення користувача;
– відсутні конфлікти між анімаціями руху та бою.

Фактичний результат. Усі перевірені функції руху та бойових дій персонажа працюють коректно. Анімаційні стани змінюються відповідно до дій користувача, логіка переходів між ними не порушується. Система керування демонструє стабільну роботу як у звичайному режимі пересування, так і в бойовій стійці.

Висновок. Реалізована система руху та бойових станів персонажа відповідає проєктним вимогам і забезпечує передбачувану та зручну взаємодію гравця з ігровим персонажем. Сценарій підтверджує коректну інтеграцію керування, анімаційної системи та бойової логіки.

Результати вимірювання навантаження на систему. Завдяки використанню низькополігональних тривимірних моделей, оптимізованих анімацій і відсутності надмірної кількості активних об’єктів на сцені, навантаження на процесор і графічний адаптер залишалося незначним. Частота кадрів зберігалася стабільною, без помітних просідань, що свідчить про достатній рівень оптимізації системи керування та анімацій персонажа.

2. Перевірка бойової взаємодії з ворогом та системи здоров’я.

Мета сценарію перевірити коректність роботи бойової системи під час взаємодії гравця з ворогом, а також функціонування системи здоров’я для обох сторін — гравця та неігрового персонажа.

Опис сценарію. У межах тестування гравець вступає в бій зі звичайним ворогом на ігровому рівні. Під час бою виконуються базові та

комбіновані атаки, використовується блок та маневрові дії. Ворог, у свою чергу, реагує на присутність гравця, наближається до нього та здійснює атаки відповідно до своєї поведінкової логіки. У процесі взаємодії перевіряється коректність нанесення шкоди, зменшення рівня здоров’я та обробка стану смерті.

Очікуваний результат:

– атаки гравця коректно завдають шкоди ворогу;
– значення здоров’я ворога зменшується відповідно до нанесених ударів;
– при досягненні нульового рівня здоров’я ворога активується анімація смерті та відбувається деактивація його активних компонентів;
– атаки ворога зменшують рівень здоров’я гравця;
– індикатор здоров’я гравця коректно оновлюється в інтерфейсі користувача;
– блок зменшує або повністю запобігає отриманню шкоди;
– бій завершується без логічних або анімаційних помилок.

Фактичний результат. У ході тестування бойова взаємодія між гравцем і ворогом відбувається коректно. Система здоров’я для обох сторін працює стабільно, значення HP змінюються відповідно до нанесеної шкоди. Анімації ударів, блокування та смерті відтворюються без збоїв, а керування гравцем зберігається коректним протягом усього бойового сценарію.

Результати вимірювання навантаження на систему. Одночасне виконання анімацій, обчислення логіки штучного інтелекту та оновлення інтерфейсу не призводило до помітного зростання навантаження. Проте, при великій кількості ворогів спостерігалося незначне збільшення навантаження на оперативну пам’ять. Завдяки використанню оптимізованих, низькополігональних моделей та обмеженої кількості активних об’єктів на сцені, частота кадрів залишалася стабільною, що свідчить про ефективну реалізацію бойової

підсистеми.

Висновок: Результати сценарію підтверджують коректну інтеграцію бойової логіки та системи здоров’я в ігровий процес. Реалізований механізм бою забезпечує передбачувану та збалансовану взаємодію між гравцем і ворогами та відповідає проєктним вимогам прототипу гри «Adventure Guild».

3. Перевірка бойової взаємодії з босом та адаптивної логіки атак.

   Мета сценарію: Перевірити коректність роботи бойової системи під час протистояння з босом, зокрема правильність вибору типу атак залежно від відстані до гравця, обробку ефектів відкидання та взаємодію з механікою блокування.

   Опис сценарію. У межах тестування гравець вступає в бій із босом-автоматоном на відповідному ігровому рівні. Під час бою гравець змінює дистанцію до противника, використовує атаки, перекати та блок. Бос аналізує відстань до гравця та обирає відповідний тип атаки: ближній, середній або дальній. Частина атак боса має ефект відкидання, який змінює позицію гравця на ігровій сцені. Під час виконання сценарію перевіряється реакція системи на блокування ударів гравцем, а також коректність обробки шкоди та анімацій у різних бойових ситуаціях.

   Очікуваний результат:

   – бос коректно визначає відстань до гравця;
   – тип атаки боса змінюється залежно від дистанції;
   – удари боса відкидають гравця на визначену відстань;
   – при активному блоці гравець не отримує шкоди;
   – ефект відкидання зберігається навіть у разі блокування;
   – анімації атак, блокування та реакцій відтворюються коректно;
   – значення здоров’я гравця та боса змінюються відповідно до умов бою;
   – у разі перемоги активується анімація смерті боса.

Фактичний результат. У ході тестування бос коректно адаптує свою поведінку залежно від відстані до гравця, використовуючи відповідні типи атак. Механіка блокування працює стабільно: при активному блоці шкода не наноситься, проте ефект відкидання зберігається, що підвищує тактичну складність бою. Взаємодія бойової логіки, системи здоров’я та анімацій не викликає конфліктів або збоїв.

Результати вимірювання навантаження на систему. Під час бою з босом спостерігалося підвищене навантаження, пов’язане з одночасною роботою складної логіки штучного інтелекту, декількох анімаційних станів і візуальних ефектів. Водночас завдяки оптимізованій реалізації та використанню низькополігональних моделей частота кадрів залишалася стабільною, без критичних просідань, що свідчить про достатній рівень продуктивності бойової підсистеми навіть у складних сценаріях.

Висновок. Результати тестового сценарію підтверджують коректну реалізацію адаптивної поведінки боса та її інтеграцію з бойовою системою прототипу гри «Adventure Guild». Сценарій демонструє стабільність роботи ключових механік і підтверджує готовність даного функціоналу до подальшого розвитку та ускладнення.

4. Перевірка роботи інвентаря та збору предметів зі скринь.

   Мета сценарію. Перевірити коректність функціонування системи інвентаря, механіки взаємодії зі скринями та правильність застосування предметів під час ігрового процесу.

   Опис сценарію. У межах тестування гравець взаємодіє зі скринею, відкриває її інтерфейс та вибирає предмети, доступні для підбору. Обрані предмети додаються до інвентаря гравця та відображаються у відповідному розділі користувацького інтерфейсу. Після цього гравець використовує предмети з інвентаря, зокрема зілля лікування, для перевірки їх впливу на стан персонажа.

Під час сценарію також перевіряється коректність оновлення інтерфейсу після взяття та використання предметів, а також відсутність помилок при повторному відкритті скрині.

Очікуваний результат:

– інтерфейс скрині відкривається коректно;
– список предметів у скрині відображається правильно;
– обраний предмет зникає зі скрині після підбору;
– предмет з’являється в інвентарі гравця;
– використання предмета активує відповідну логіку;
– зілля лікування збільшує рівень здоров’я гравця на 100 одиниць;
– індикатор здоров’я в інтерфейсі оновлюється коректно;
– після використання предмет зникає з інвентаря;
– повторна взаємодія зі скринею не призводить до помилок або дублювання предметів.

Фактичний результат. У ході тестування система інвентаря та механіка взаємодії зі скринями працювали стабільно. Предмети коректно додавалися до інвентаря, а їх використання викликало очікуваний ефект. Зілля лікування збільшувало рівень здоров’я гравця на задану величину, а інтерфейс користувача своєчасно відображав зміни стану персонажа.

Результати вимірювання навантаження на систему. Під час виконання даного сценарію навантаження на систему залишалося незначним. Операції відкриття інвентаря, оновлення UI та застосування ефектів предметів не викликали помітних затримок або зниження продуктивності. Це свідчить про ефективну реалізацію системи інвентаря та її коректну інтеграцію з іншими підсистемами гри.

Висновок. Результати сценарію підтверджують коректність реалізації системи інвентаря, механіки збору предметів та логіки їх використання. Функціонал відповідає проєктним вимогам і забезпечує зручну та зрозумілу взаємодію гравця з ігровими предметами у прототипі гри «Adventure Guild».

5. Перевірка коректності роботи пасток.

Мета сценарію. Перевірити коректність функціонування ігрових пасток, зокрема їх здатність виявляти об’єкти за допомогою тригерів, наносити шкоду, застосовувати ефект відкидання та взаємодіяти як з гравцем, так і з ворогами.

Опис сценарію. У межах тестування гравець навмисно активує різні пастки, розміщені на ігровому рівні, шляхом входу в зону їх спрацювання. Після активації пастки перевіряється нанесення шкоди персонажу та коректність ефекту відкидання. Окрім цього, у сценарії перевіряється взаємодія пасток з ворогами, які можуть потрапляти в зону дії тригерів під час пересування або переслідування гравця. Під час виконання сценарію особлива увага приділяється перевірці коректності роботи тригерних колайдерів, уникненню повторного некоректного спрацювання та відповідності візуальних ефектів фактичній дії пастки.

Очікуваний результат:

– пастка коректно активується при вході об’єкта в зону тригера;
– після активації пастка наносить шкоду;
– застосовується ефект відкидання у відповідному напрямку;
– гравець отримує шкоду та реагує відповідною анімацією;
– вороги також отримують шкоду та ефект відкидання;
– пастка не активується повторно без перезарядки або виходу з тригера;
– відсутні помилкові або некоректні спрацювання.

Фактичний результат. У ході тестування пастки коректно реагували на входження гравця та ворогів у зону дії тригера. Нанесення шкоди та ефект відкидання застосовувалися відповідно до заданої логіки. Взаємодія пасток з різними типами об’єктів відбувалася стабільно, без збоїв або некоректних повторних активацій.

Результати вимірювання навантаження на систему. Під час активації пасток і виконання відповідних фізичних та анімаційних обчислень

навантаження на систему залишалося незначним. Навіть за одночасного спрацювання кількох пасток не спостерігалося суттєвого зниження продуктивності, що свідчить про ефективну реалізацію тригерної логіки та фізичних взаємодій.

Висновок. Результати тестового сценарію підтверджують коректність реалізації механіки пасток у прототипі гри «Adventure Guild». Пастки функціонують стабільно, взаємодіють як з гравцем, так і з ворогами, та додають ігровому процесу додатковий рівень складності без негативного впливу на продуктивність системи.

У результаті проведеного тестування було перевірено коректність роботи основних ігрових механік прототипу гри «Adventure Guild», зокрема системи керування персонажем, бойової взаємодії, штучного інтелекту ворогів і боса, роботи інвентаря, механіки пасток та взаємодії з об’єктами ігрового світу. Виконані тестові сценарії підтвердили узгоджену роботу програмних компонентів, стабільність їх взаємодії та відповідність реалізованого функціоналу проєктним вимогам. Оцінка продуктивності засвідчила відсутність критичних навантажень на систему, що свідчить про достатній рівень оптимізації та готовність прототипу до подальшого розвитку і розширення функціональності.

## 4.2 Підтримка програмного забезпечення

Підтримка прототипу гри «Adventure Guild» забезпечується завдяки використанню сучасних технологій розробки та продуманої архітектури програмного забезпечення. Компонентно-орієнтований підхід, реалізований у середовищі Unity, дозволяє локалізувати зміни в межах окремих модулів без необхідності суттєвого втручання в загальну структуру системи. Це значно спрощує процес виправлення помилок, оновлення окремих механік та адаптацію гри до нових вимог.

Чітке розмежування відповідальності між основними підсистемами, такими як керування персонажем, бойова логіка, штучний інтелект, інвентар та інтерфейс

користувача, створює умови для ефективної технічної підтримки. У разі виявлення дефектів або необхідності оптимізації певного функціоналу, зміни можуть бути внесені точково, без порушення роботи інших компонентів. Такий підхід знижує ризик появи нових помилок під час оновлення програмного продукту.

Використання системи контролю версій дозволяє відстежувати історію змін проєкту, повертатися до стабільних станів та паралельно розробляти нові функціональні можливості. Це є важливим елементом підтримки, особливо у разі подальшого розвитку проєкту або роботи над ним кількох розробників. Крім того, регулярне тестування ключових механік після внесення змін забезпечує стабільність роботи гри та збереження якості ігрового процесу.

Завдяки використанню низькополігональних моделей, оптимізованих анімацій і обмеженої кількості активних об’єктів на сценах, підтримка продуктивності гри не потребує значних апаратних ресурсів. Це спрощує адаптацію програмного забезпечення до різних конфігурацій апаратного забезпечення та зменшує витрати часу на оптимізацію під час подальшої експлуатації.

### 4.3 Можливості подальшого розвитку

Розроблений програмний продукт «Adventure Guild» на поточному етапі є функціональним прототипом, який демонструє базові можливості ігрової системи, реалізацію ключових механік та обґрунтовані архітектурні рішення. Створений прототип не є завершеним комерційним продуктом, однак він закладає міцну технічну та концептуальну основу для подальшого розвитку гри як повноцінного багатокористувацького RPG-проєкту.

Завдяки використанню компонентно-орієнтованої архітектури, ігрового рушія Unity та мови програмування C#, програмне забезпечення має високий потенціал до масштабування та розширення функціональності. Подальший розвиток гри може здійснюватися поетапно, з поступовим впровадженням нових

механік, контенту та технологічних рішень без необхідності кардинальної перебудови існуючої системи.

Одним із ключових напрямів подальшого розвитку є впровадження багатокористувацького режиму, що дозволить кільком гравцям одночасно взаємодіяти в межах одного ігрового світу. Передбачається реалізація кооперативного режиму, у якому гравці об’єднуються в групу для спільного виконання завдань, проходження підземель та участі у бойових зіткненнях. У багатокористувацькому режимі планується використання системи ролей, відповідно до якої кожен гравець обирає окремий клас персонажа. Передбачено чотири базові класи:

1. Лицар — персонаж ближнього бою з високим рівнем захисту та можливістю блокування атак.
2. Лучник — персонаж дальнього бою, орієнтований на нанесення шкоди з безпечної дистанції.
3. Чарівник — персонаж, що використовує магічні атаки з ефектами масового ураження або контролю.
4. Клірик — персонаж підтримки, здатний лікувати союзників та посилювати їхні характеристики.

Кожен клас матиме унікальний набір умінь, бойових можливостей та особливостей розвитку, що вимагатиме від гравців тактичного розподілу ролей і командної взаємодії. Такий підхід сприятиме підвищенню складності та глибини ігрового процесу.

Подальший розвиток гри передбачає впровадження централізованої системи завдань, пов’язаної з гільдією шукачів пригод. Гільдія виконуватиме роль ключового елемента ігрового світу, через який гравці отримуватимуть доступ до основного контенту гри.

Планується реалізація різних типів квестів, зокрема:

1. Зачищення підземель від ворогів.
2. Супровід торгових караванів або важливих персонажів.
3. Захист об’єктів від хвиль противників.

4. Пошук і здобуття рідкісних артефактів.
5. Дослідження небезпечних локацій.

Завдання можуть відрізнятися рівнем складності, кількістю учасників та умовами виконання, що дозволить адаптувати ігровий процес як для одиночної гри, так і для командної взаємодії.

Важливим напрямом розвитку є впровадження процедурної (рандомної) генерації ігрових рівнів. Такий підхід дозволить автоматично створювати унікальні варіанти підземель, кімнат і маршрутів проходження, що суттєво підвищить реіграбельність гри. Використання цього механізму зменшить залежність від ручного створення контенту та дозволить підтримувати інтерес гравців протягом тривалого часу.

Подальший розвиток гри також передбачає розширення системи прогресії персонажів. Зокрема, можливе впровадження:

1. Системи рівнів і досвіду.
2. Розвитку умінь та здібностей.
3. Покращення спорядження.
4. Спеціалізацій класів.

Ці механіки дозволять створити більш глибоку та мотивуючу систему розвитку персонажа, що є характерною рисою RPG-жанру.

Окрім функціонального розвитку, важливим аспектом є подальше вдосконалення графічної складової та візуального стилю гри. Планується розширення набору анімацій, підвищення деталізації оточення та формування цілісної художньої концепції ігрового світу [22-27].

Для представлення гри як завершеного програмного продукту доцільним є використання візуальних матеріалів, зокрема обкладинки гри «Adventure Guild», яка відображає її основну ідею, атмосферу та ключових персонажів. Така обкладинка може використовуватися в супровідних матеріалах, презентаціях та як ілюстрація до даного підрозділу, підкреслюючи цілісність і завершеність концепції проекту (див. додаток В).

Таким чином, прототип гри «Adventure Guild» має значний потенціал для подальшого розвитку та масштабування. Запропоновані напрями розширення охоплюють як технічні, так і геймдизайнерські аспекти, що дозволяє розглядати проєкт як основу для створення повноцінного багатокористувацького RPG-продукту. Реалізована архітектура та обрані технології забезпечують гнучкість, підтримуваність і готовність системи до впровадження нових ігрових механік у майбутньому.

У межах розділу було проведено комплексну перевірку розробленого прототипу гри «Adventure Guild», яка охопила тестування основних ігрових механік, взаємодію програмних компонентів та оцінку стабільності роботи системи в типових ігрових сценаріях. Реалізовані тестові сценарії дозволили підтвердити коректність функціонування системи керування персонажем, бойової логіки, штучного інтелекту ворогів і боса, інвентаря, пасток, а також механізмів переходу між рівнями. Результати тестування свідчать про узгоджену роботу підсистем та достатній рівень оптимізації програмного забезпечення.

Окрім цього, у розділі розглянуто питання впровадження та підтримки програмного продукту, а також визначено основні напрями його подальшого розвитку. Використання сучасних технологій і продуманої архітектури забезпечує зручність супроводу проєкту та створює умови для розширення функціональності, зокрема впровадження багатокористувацького режиму, нових ігрових механік і контенту. Таким чином, створений прототип можна вважати стабільною та перспективною основою для подальшого розвитку гри як повноцінного програмного продукту.

# 5 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

У процесі розробки програмного забезпечення, зокрема тривимірних ігрових застосунків, важливу роль відіграє забезпечення належних умов праці та дотримання вимог охорони праці і безпеки в надзвичайних ситуаціях. Робота програміста передбачає тривале використання комп'ютерної техніки, що супроводжується підвищеним навантаженням на зоровий апарат, опорно-руховий апарат та нервову систему, а також впливом електромагнітних і психоемоційних факторів. Тому дотримання чинних нормативно-правових вимог у сфері охорони праці, організації робочого місця та заходів безпеки в умовах можливих надзвичайних ситуацій є необхідною умовою забезпечення здоров'я працівника, безпечного виконання професійних обов'язків і безперервності робочого процесу.

## 5.1 Охорона праці

Охорона праці під час розробки програмного забезпечення є важливою складовою професійної діяльності інженера-програміста та спрямована на забезпечення безпечних і нешкідливих умов праці, збереження здоров'я і працездатності працівника. У процесі виконання магістерської роботи з розробки програмного продукту «Adventure Guild» основні роботи здійснювалися з використанням персонального комп'ютера та екранних пристроїв, що зумовлює необхідність дотримання чинних вимог охорони праці, електробезпеки, пожежної безпеки та ергономіки робочого місця.

Правові та організаційні засади охорони праці в Україні визначаються Законом України «Про охорону праці» [28], який регламентує обов'язки щодо створення безпечних умов праці, запобігання професійним захворюванням і мінімізації впливу шкідливих виробничих факторів. Відповідно до цього закону, під час організації роботи з комп'ютерною технікою повинні бути враховані всі потенційні ризики, пов'язані з використанням електронного обладнання та тривалою статичною роботою.

Під час виконання робіт з розробки програмного забезпечення основними потенційно небезпечними та шкідливими факторами є:

тривале статичне навантаження на опорно-руховий апарат;

підвищене навантаження на органи зору;

психоемоційне напруження;

ризик ураження електричним струмом;

пожежна небезпека, пов’язана з використанням електрообладнання.

Вимоги щодо безпеки та захисту здоров’я працівників при роботі з комп’ютерною технікою регламентуються НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями»[29]. Зазначений нормативний документ встановлює обов’язкові вимоги до організації робочого місця, параметрів екранних пристроїв, режимів праці та відпочинку, а також заходів щодо зменшення негативного впливу на здоров’я працівників.

Відповідно до НПАОП 0.00-7.15-18, робота з екранними пристроями повинна організовуватися таким чином, щоб забезпечити оптимальні умови зорового сприйняття інформації, зменшити м’язове напруження та запобігти перевтомі.

Організація робочого місця програміста повинна відповідати будівельним, санітарно-гігієнічним та ергономічним вимогам. Згідно з ДБН В.2.2-28:2010 «Будинки і споруди. Адміністративні та побутові будівлі» [30], робоче місце з персональним комп’ютером має забезпечувати достатню площу для розміщення обладнання та вільного переміщення працівника, а також створювати комфортні умови для виконання професійних обов’язків.

Ергономічні вимоги до взаємодії людини з комп’ютерними системами визначаються стандартами ДСТУ ISO 9241 [31], які регламентують розміщення монітора, клавіатури, миші, робочого столу та крісла. Висота та положення елементів робочого місця повинні забезпечувати природне положення тіла, зменшувати навантаження на хребет, плечовий пояс і верхні кінцівки.

Монітор має бути розташований на оптимальній відстані від очей

користувача, а кут огляду — таким, щоб уникати перенапруження зору. Клавіатура і маніпулятор повинні бути розміщені на зручній висоті, що забезпечує нейтральне положення кистей рук.

Освітлення робочого місця є одним із важливих факторів, що впливає на безпеку, працездатність та зоровий комфорт користувачів персональних комп'ютерів. Вимоги до природного та штучного освітлення регламентуються ДБН В.2.5-28:2018 «Природне і штучне освітлення» [32], який встановлює нормативні показники освітленості робочих приміщень, вимоги до рівномірності освітлення, обмеження засліплюючої дії світильників та раціональне поєднання природного і штучного світла. Дотримання зазначених вимог сприяє зниженню втоми зору, підвищенню ефективності праці та забезпеченню безпечних умов роботи.

Робоче місце програміста повинно мати комбіноване освітлення — природне та штучне — з рівномірним розподілом світла. Освітлювальні прилади мають бути розташовані таким чином, щоб уникати віддзеркалення світла на екрані монітора та появи різких тіней у робочій зоні.

Для зменшення негативного впливу тривалої роботи з екранними пристроями необхідно дотримуватися раціонального режиму праці та відпочинку. Відповідно до вимог НПАОП 0.00-7.15-18, робота за комп'ютером повинна чергуватися з перервами для відпочинку, під час яких рекомендується виконувати вправи для очей і розминку для м'язів.

Під час експлуатації комп'ютерної техніки необхідно дотримуватися вимог електробезпеки, визначених Правилами улаштування електроустановок (ПУЕ) [33] та Правилами технічної експлуатації електроустановок споживачів. Усі електричні пристрої повинні бути справними, мати надійну ізоляцію та заземлення.

Забороняється використання пошкоджених кабелів, несправних розеток та саморобних подовжувачів. Дотримання вимог електробезпеки зменшує ризик ураження електричним струмом і виникнення аварійних ситуацій.

Вимоги пожежної безпеки регламентуються Правилами пожежної безпеки в

Україні (НАПБ А.01.001-2014) [35]. Основними причинами пожеж у приміщеннях із комп’ютерною технікою є коротке замикання, перевантаження електромережі та несправність електрообладнання.

Для забезпечення пожежної безпеки необхідно:

використовувати сертифіковане електрообладнання;

не перевантажувати електромережу;

утримувати робоче місце в належному стані;

мати доступ до первинних засобів пожежогасіння.

Таким чином, аналіз умов праці під час розробки програмного забезпечення «Adventure Guild» показав, що за умови дотримання чинних нормативно-правових актів України забезпечується належний рівень охорони праці. Виконання вимог НПАОП, державних будівельних норм та національних стандартів дозволяє мінімізувати вплив небезпечних і шкідливих факторів, створити безпечні умови праці та забезпечити ефективну і стабільну професійну діяльність розробника.

## 5.2 Забезпечення електробезпеки користувачів ПК

Електробезпека є одним із найважливіших аспектів безпеки життєдіяльності, особливо у сфері використання комп’ютерної техніки на робочих місцях. Електричний струм, незважаючи на свою повсюдність у сучасному інформаційному середовищі, може бути джерелом потенційної небезпеки внаслідок порушень правил експлуатації, несправності обладнання або невідповідності робочого місця нормативним вимогам. За визначенням, до надзвичайних ситуацій належать події техногенного характеру, що можуть спричинити травматизм, ураження електричним струмом або інші ушкодження здоров’я людей та майна [37].

У практичній діяльності користувачів персональних комп’ютерів найпоширенішою небезпекою є контакт із струмоведучими частинами обладнання при порушенні цілісності ізоляції, неправильному підключенні живлення або використанні дефектних кабелів та розеток. Тому одним із

ключових завдань є створення безпечних умов праці, що передбачає не лише правильне розташування обладнання, а й дотримання електротехнічних вимог. Згідно з навчальними рекомендаціями, безпечна експлуатація електроустановок включає застосування засобів захисту, правильне заземлення та ретельну перевірку стану кабельних з’єднань перед початком роботи [36].

Одним із методів забезпечення електробезпеки є організація робочого місця відповідно до чинних стандартів. Це забезпечує підтримку не лише ергономічних, а й техніко-безпекових вимог, що значно зменшує ймовірність виникнення аварійних ситуацій через контакт зі струмом. Рекомендовано розташовувати робочі пристрої так, щоб доступ до мережевих кабелів був вільним, а вони не піддавалися механічному пошкодженню; електронні пристрої мають підключатися через сертифіковані подовжувачі з пристроями захисту від перевантаження [36].

Крім того, важливим елементом безпеки є навчання користувачів діям у разі виникнення електротехнічної надзвичайної ситуації. Це включає знання порядку відключення обладнання від джерела живлення, вміння швидко та безпечно евакуюватися при виявленні запаху горіння із розеток чи перегріванні обладнання, а також застосування первинних засобів пожежогасіння під час незначного загоряння електроприладів. Індивідуальні дії користувача повинні бути скоординовані з організаційними заходами, що передбачені системою цивільного захисту та правилами безпеки на підприємстві [37].

Особлива увага приділяється перевірці технічного стану робочих місць, що включає регулярне обстеження мережевих кабелів, адаптерів та розеток для своєчасного виявлення ознак зносу або пошкодження ізоляції. Стан обладнання необхідно перевіряти до початку робочої зміни, особливо в холодний період року, коли кабелі можуть втрачати свою еластичність та пошкоджуватись значно швидше. За результатами таких перевірок формуються акти технічного обстеження, що дозволяє системно контролювати стан електрообладнання та своєчасно усувати дефекти [36].

Важливо також відзначити, що ефективне забезпечення електробезпеки

ґрунтується не лише на технічних заходах, але й на систематичному навчанні персоналу щодо ризиків та способів їх мінімізації. Це передбачає проведення інструктажів з електробезпеки, роз'яснення потенційних наслідків недотримання правил та практичних рекомендацій щодо попередження небезпеки. Такий підхід сприяє формуванню у користувачів ПК усвідомлення важливості безпечної поведінки в робочому середовищі, що безпосередньо впливає на зниження кількості нещасних випадків [37].

Отже, забезпечення електробезпеки користувачів ПК є складовою частиною впровадження безпечних умов праці та цивільного захисту на об'єкті праці. Дотримання електротехнічних вимог, організаційних правил експлуатації обладнання, регулярне технічне обслуговування та навчання персоналу дозволяють значно зменшити ризики виникнення надзвичайних ситуацій, пов'язаних із електричним струмом, та забезпечити безпечну життєдіяльність користувачів комп'ютерної техніки.

У межах даного розділу було розглянуто основні аспекти охорони праці та безпеки життєдіяльності в надзвичайних ситуаціях, що є актуальними під час роботи з персональними комп'ютерами. Проаналізовано потенційні небезпеки, пов'язані з експлуатацією електронного обладнання, та визначено ключові заходи щодо забезпечення електробезпеки користувачів ПК. Дотримання встановлених правил організації робочого місця, вимог електробезпеки, а також знання порядку дій у разі виникнення надзвичайних ситуацій дозволяють суттєво знизити ризик травматизму та негативного впливу на здоров'я працівників. Реалізація комплексу технічних, організаційних і профілактичних заходів забезпечує безпечні умови праці та сприяє стабільному й безперебійному виконанню професійних обов'язків.

# ВИСНОВКИ

У межах магістерської кваліфікаційної роботи було виконано розробку прототипу 3D-гри жанру RPG «Adventure Guild» із використанням ігрового рушія Unity та мови програмування C#. Робота охоплювала всі основні етапи життєвого циклу програмного забезпечення — від аналізу предметної області та формування вимог до проєктування, реалізації, тестування та визначення напрямів подальшого розвитку проєкту.

На етапі аналізу предметної області було досліджено особливості рольових ігор, визначено ключові ігрові механіки, типові сценарії взаємодії користувача з ігровим середовищем, а також сформовано словник предметної області. На основі проведеного аналізу були визначені функціональні та нефункціональні вимоги до програмного продукту, що дозволило чітко окреслити межі та цілі розробки прототипу гри.

У процесі проєктування програмного забезпечення було розроблено архітектуру системи з використанням компонентно-орієнтованого підходу, характерного для середовища Unity. Для наочного представлення структури та поведінки системи були побудовані діаграми варіантів використання, діаграма класів та діаграми послідовностей, які дозволили формалізувати взаємодію між основними компонентами гри та описати часові аспекти їх роботи. Обрані архітектурні рішення забезпечили модульність, масштабованість і зручність супроводу програмного продукту.

У межах реалізації основного функціоналу було створено ключові ігрові підсистеми, зокрема систему керування персонажем, бойову систему, систему здоров’я, інвентар, механізми взаємодії з об’єктами, штучний інтелект ворогів і боса, а також інтерфейс користувача. Реалізація здійснювалася з використанням об’єктно-орієнтованого програмування, подієвого підходу та тісної інтеграції з анімаційною системою Unity, що забезпечило узгоджену та стабільну роботу ігрових механік.

Окрему увагу було приділено створенню графічної складової та контенту

гри. У середовищі Unity Editor було реалізовано два повноцінні ігрові рівні з різною атмосферою та структурою, налаштовано анімаційні контролери для гравця, ворогів і боса, а також реалізовано механізми переходу між сценами зі збереженням стану основних ігрових об’єктів. Це дозволило створити цілісний ігровий простір та забезпечити безперервний ігровий досвід.

У процесі тестування розробленого функціоналу було перевірено коректність роботи основних ігрових сценаріїв, зокрема керування персонажем, бойової взаємодії, поведінки ворогів і боса, роботи інвентаря та пасток. Проведене тестування підтвердило стабільність функціонування програмного продукту та відповідність реалізованих механік поставленим вимогам. Додатково було здійснено оцінку навантаження на систему, яка показала, що завдяки використанню оптимізованих низькополігональних моделей та раціональної архітектури прототип демонструє стабільну продуктивність без критичних просідань частоти кадрів.

У роботі також було розглянуто питання підтримки та подальшого розвитку програмного забезпечення. Визначено, що створений прототип є надійною основою для розширення функціональності гри, зокрема впровадження багатокористувацького режиму, системи класів персонажів, квестів гільдії шукачів пригод, процедурної генерації рівнів та розвитку системи прогресії. Запропоновані напрями розвитку підтверджують перспективність проєкту та можливість його трансформації у повноцінний ігровий продукт.

Таким чином, у ході виконання магістерської роботи було досягнуто поставленої мети та вирішено всі визначені завдання. Розроблений прототип гри «Adventure Guild» демонструє практичне застосування сучасних технологій розробки програмного забезпечення, підтверджує ефективність використаних архітектурних рішень та може бути використаний як основа для подальших досліджень або розвитку у сфері створення інтерактивних ігрових систем.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hocking J. Unity in Action: Multiplatform game development in C#. 2nd ed. Manning Publications, 2018. 382 с. URL: https://www.manning.com/books/unity-in-action-second-edition (дата звернення: 16.12.2025).

2. Gibson Bond J. Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#. Addison-Wesley, 2014. 944 с. URL: https://www.pearson.com/us/higher-education/program/Gibson-Bond-Introduction-to-Game-Design-Prototyping-and-Development-From-Concept-to-Playable-Game-with-Unity-and-C/PGM100000.html (дата звернення: 16.12.2025).

3. Unity Manual: Introduction to Unity. Unity Technologies, 2023. URL: https://docs.unity3d.com/Manual/index.html (дата звернення: 16.12.2025).

4. C# Scripting in Unity. Unity Technologies, 2023. URL: https://docs.unity3d.com/Manual/ScriptingSection.html (дата звернення: 16.12.2025).

5. Skeet J. C# in Depth. 4th ed. Manning Publications, 2019. 528 с. URL: https://www.manning.com/books/c-sharp-in-depth-fourth-edition (дата звернення: 16.12.2025).

6. Animator Controller Documentation. Unity Technologies, 2023. URL: https://docs.unity3d.com/Manual/class-AnimatorController.html (дата звернення: 16.12.2025).

7. Navigation and Pathfinding in Unity. Unity Technologies, 2023. URL: https://docs.unity3d.com/Manual/Navigation.html (дата звернення: 16.12.2025).

8. Coroutines in Unity. Unity Technologies, 2023. URL: https://docs.unity3d.com/Manual/Coroutines.html (дата звернення: 16.12.2025).

9. Mixamo: Free 3D animations for games. Adobe, 2023. URL: https://www.mixamo.com/ (дата звернення: 16.12.2025).

10. 3D Models for Download. CGTrader, 2023. URL: https://www.cgtrader.com/ (дата звернення: 16.12.2025).

11. AccuRIG: Free Auto Rigging Tool. Reallusion, 2023. URL: https://actorcore.reallusion.com/auto-rig (дата звернення: 16.12.2025).

12. Unity Asset Store. Unity Technologies, 2023. URL: https://assetstore.unity.com/ (дата звернення: 16.12.2025).

13. The Scrum Guide. Schwaber K., Sutherland J. Scrum.org, 2020. URL: https://scrumguides.org/scrum-guide.html (дата звернення: 16.12.2025).

14. Keith C. Agile Game Development with Scrum. Addison-Wesley, 2010. 355 с. URL: https://www.pearson.com/us/higher-education/program/Keith-Agile-Game-Development-with-Scrum/PGM100000.html (дата звернення: 16.12.2025).

15. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Addison-Wesley, 2003. 208 с. URL: https://www.pearson.com/us/higher-education/program/Fowler-UML-Distilled-A-Brief-Guide-to-the-Standard-Object-Modeling-Language-3rd-Edition/PGM100000.html (дата звернення: 16.12.2025).

16. Sommerville I. Software Engineering. 10th ed. Pearson, 2015. 816 с. URL: https://www.pearson.com/us/higher-education/program/Sommerville-Software-Engineering-10th-Edition/PGM100000.html (дата звернення: 16.12.2025).

17. Pressman R. S. Software Engineering: A Practitioner's Approach. 8th ed. McGraw-Hill Education, 2014. 976 с. URL: https://www.mheducation.com/highered/product/software-engineering-practitioner-s-approach-pressman-maxim/M9780078022128.html (дата звернення: 16.12.2025).

18. Buckland M. Programming Game AI by Example. Wordware Publishing, 2005. 395 с. URL: https://www.jblearning.com/catalog/productdetails/9781556220784 (дата звернення: 16.12.2025).

19. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 395 с. URL: https://www.pearson.com/us/higher-education/program/Gamma-Design-Patterns-Elements-of-Reusable-Object-Oriented-Software/PGM100000.html (дата звернення: 16.12.2025).

20. Nystrom R. Game Programming Patterns. Genever Benning, 2014. 354 с. URL: https://gameprogrammingpatterns.com/ (дата звернення: 16.12.2025).

21. Rollings A., Morris D. Game Architecture and Design: A New Edition.

New Riders, 2003. 936 с. URL: https://www.peachpit.com/store/game-architecture-and-design-a-new-edition-9780735713635 (дата звернення: 16.12.2025).

22. Adams E. Fundamentals of Game Design. 3rd ed. New Riders, 2014. 576 с. URL: https://www.peachpit.com/store/fundamentals-of-game-design-9780321929679 (дата звернення: 16.12.2025).

23. Rogers S. Level Up! The Guide to Great Video Game Design. 2nd ed. Wiley, 2014. 387 с. URL: https://www.wiley.com/en-us/Level+Up%21+The+Guide+to+Great+Video+Game+Design%2C+2nd+Edition-p-9781118877197 (дата звернення: 16.12.2025).

24. Schell J. The Art of Game Design: A Book of Lenses. 3rd ed. CRC Press, 2019. 652 с. URL: https://www.routledge.com/The-Art-of-Game-Design-A-Book-of-Lenses-Third-Edition/Schell/p/book/9781138632059 (дата звернення: 16.12.2025).

25. Salen K., Zimmerman E. Rules of Play: Game Design Fundamentals. MIT Press, 2003. 688 с. URL: https://mitpress.mit.edu/9780262240451/rules-of-play/ (дата звернення: 16.12.2025).

26. Unity Learn: Junior Programmer Pathway. Unity Technologies, 2023. URL: https://learn.unity.com/pathway/junior-programmer (дата звернення: 16.12.2025).

27. Blow J. Game Development: Harder Than You Think // ACM Queue. 2004. Vol. 1, № 10. P. 28–37. URL: https://queue.acm.org/detail.cfm?id=971593 (дата звернення: 16.12.2025).

28. Закон України «Про охорону праці» від 14.10.1992 № 2694-XII (зі змінами). URL: https://zakon.rada.gov.ua/laws/show/2694-12 (дата звернення: 16.12.2025).

29. НПАОП 0.00-7.15-18. Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями : затв. наказом Мінсоцполітики України від 14.02.2018 № 207. URL: https://zakon.rada.gov.ua/laws/show/z0508-18 (дата звернення: 16.12.2025).

30. ДБН В.2.2-28:2010. Будинки і споруди. Будинки адміністративного та побутового призначення. URL: https://online.budstandart.com/ua/catalog/doc-

page.html?id_doc=27263 (дата звернення: 16.12.2025).

31. ДСТУ ISO 9241 (серія). Ергономіка взаємодії людина-система (різні частини). URL: https://online.budstandart.com/ua (пошук за серією стандартів) (дата звернення: 16.12.2025).

32. ДБН В.2.5-28:2018. Природне і штучне освітлення : затв. наказом Мінрегіону України від 03.10.2018 № 264. Київ : Укрархбудінформ, 2018. URL: https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=79885 (дата звернення: 16.12.2025)

33. Правила улаштування електроустановок (ПУЕ) : затв. наказом Міненерговугілля України від 21.07.2017 № 476 (нова редакція). URL: https://zakon.rada.gov.ua/laws/show/v0476732-17 (дата звернення: 16.12.2025).

34. Правила технічної експлуатації електроустановок споживачів (ПТЕЕС) : затв. наказом Мінпаливенерго України від 25.07.2006 № 258. URL: https://zakon.rada.gov.ua/laws/show/z1143-06 (дата звернення: 16.12.2025).

35. НАПБ А.01.001-2014. Правила пожежної безпеки в Україні : затв. наказом МВС України від 30.12.2014 № 1417. URL: https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=60541 (дата звернення: 16.12.2025).

36. Стручок В. С. Безпека в надзвичайних ситуаціях : методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання. Тернопіль : ФОП Паляниця В. А., 2022. 156 с. URL: https://elartu.tntu.edu.ua/handle/lib/39196 (дата звернення: 16.12.2025).

37. Стручок В. С. Техноекологія та цивільна безпека. Частина «Цивільна безпека» : навчальний посібник. Тернопіль : ФОП Паляниця В. А., 2022. 156 с. URL: http://elartu.tntu.edu.ua/handle/lib/39424 (дата звернення: 16.12.2025).

38. Методичні вказівки до виконання кваліфікаційної роботи магістра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання / укладачі: Д. М. Михалик, Г. Б. Цуприк, В. М. Бревус, І. Я. Мудрик. Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя, 2024. 44 с. URL: https://elartu.tntu.edu.ua/handle/lib/5031.

# ДОДАТКИ

**Додаток А**

# Апробація результатів

**УДК 004.4**
**Бойко Ігор докт. фіз.-мат. наук, професор кафедри ІП, Ткачук Роман**
Тернопільський національний технічний університет імені Івана Пулюя

**РОЗРОБКА 3D-ГРИ ЖАНРУ RPG НА UNITY**

**Boiko Ihor Dr., Prof., Tkachuk Roman**
**DEVELOPMENT OF A 3D RPG GAME USING UNITY**

Розробка сучасних 3D-ігор жанру RPG потребує поєднання інженерних підходів, гнучких методологій, оптимальних архітектурних рішень та ефективного використання інструментів ігрових рушіїв. У представленій роботі описано процес створення 3D-гри «Lone Knight» на платформі Unity з використанням мови C#. Метою роботи є реалізація повноцінної ігрової системи з базовими механіками RPG, включно з бойовою системою, обробкою взаємодій, управлінням об'єктами сцени, побудовою рівнів, системою прогресу та обробкою подій.

На етапі аналізу проведено дослідження актуальних рішень у сфері розробки RPG-ігор, визначено вимоги до функціональних елементів гри, сформовано цілі та обмеження проєкту. Особливу увагу приділено модульності, розширюваності та підтримуваності архітектури.

Під час проєктування створено діаграми класів та сценаріїв використання, що визначають логіку взаємодії ключових компонентів: гравця, ворогів, камерної системи, менеджера станів, системи бою та інвентаря. Застосовано об'єктно-орієнтований підхід та низку патернів проєктування, що забезпечують гнучкість структури гри та можливість подальшого масштабування.

На етапі розробки реалізовано основні механіки: рух і керування персонажем, бойові взаємодії, систему здоров'я, відображення інтерфейсу, перемикання сцен, поведінку ворогів, контент гри. Використано можливості Unity для створення анімацій, налаштування камерних переходів та управління об'єктами шляхом скриптової логіки.

Тестування гри здійснювалось із використанням модульних та інтеграційних підходів, що дозволило виявити та усунути помилки на ранніх етапах. Перевірено коректність роботи бойової системи, обробки колізій, навігації персонажа, відображення UI та оптимізації продуктивності сцени.

Отримані результати демонструють можливість створення повноцінної 3D RPG-гри в умовах обмежених ресурсів, використовуючи сучасні інструменти Unity та підхід структурованої розробки. Створений прототип може бути основою для подальшого вдосконалення механік, розширення ігрового світу та впровадження нових функціональних можливостей.

Рисунок А.1 – Знімок екрану апробації результатів

**Додаток Б**

## Лістинги коду описаних класів системи

Лістинг Б.1 – Клас BossBehavior

```
using JetBrains.Annotations;
using UnityEngine;
using System.Collections;


public class BossBehavior : MonoBehaviour
{
    public float close, mid, far, rotateSpeed;
    public float shDamage, kiDamage, hDamage;
    public float shRadius, kiRadius, hRadius;
    public Transform shField, kiField, hField;
    private Animator animator;
    private Transform player;
    private AnimatorStateInfo stateInfo;
    void Start()
    {
        player = GameObject.Find("KnightPaladine").transform;
        animator = GetComponent<Animator>();
    }
    void Update()
    {
        float      dis      =      Vector3.Distance(player.position,
transform.position);
        stateInfo = animator.GetCurrentAnimatorStateInfo(0);
        if         (!stateInfo.IsName("RobobossStrongHit")         &&
!stateInfo.IsName("Robobos   360")   &&   !stateInfo.IsName("Robobos
Kick"))
        {
            transform.rotation = Quaternion.LookRotation(
            Vector3.RotateTowards(transform.forward,  player.position
- transform.position, rotateSpeed * Time.deltaTime, 0));

        }
        if (dis < far + 5)
        {
            GetComponent<HealthManager>().ShowHealthBar();
            if (dis < close)
            {
                animator.SetInteger("dis", 1);
            }
            else if (dis < mid)
            {
                animator.SetInteger("dis", 2);
            }
```

```
            else if (dis < far)
            {
                animator.SetInteger("dis", 3);
            }
            else if (dis > far)
            {
                animator.SetInteger("dis", 4);
            }
        }
        else
            GetComponent<HealthManager>().HideHealthBar();


    }
    public void StrongHit()
    {
        Collider[]                    colliders                    =
Physics.OverlapSphere(shField.position, shRadius);
        foreach (Collider num in colliders)
        {
            if (num.gameObject == player.gameObject)
            {

player.GetComponent<PlayerHealth>().GetDamage(shDamage);
                if
(!player.GetComponent<Animator>().GetBool("roll"))
                {player.GetComponent<Interaction>().enabled = false;
                StartCoroutine(RotateCharacter());
                player.GetComponent<Animator>().SetBool("fly",
true);}
                break;
            }
        }
    }
    public void Kick()
    {
        Collider[]                    colliders                    =
Physics.OverlapSphere(shField.position, kiRadius);
        foreach (Collider num in colliders)
        {
            if (num.gameObject == player.gameObject)
            {

player.GetComponent<PlayerHealth>().GetDamage(kiDamage);
                if
(!player.GetComponent<Animator>().GetBool("roll"))
                {
                    player.GetComponent<Interaction>().enabled      =
false;
                    StartCoroutine(RotateCharacter());
                    player.GetComponent<Animator>().SetBool("fly",
true);
                }
```

```
                break;
            }
        }
    }
    public void Hit()
    {
        Collider[]                    colliders                    =
Physics.OverlapSphere(hField.position, hRadius);
        foreach (Collider num in colliders)
        {
            if (num.gameObject == player.gameObject)
            {

player.GetComponent<PlayerHealth>().GetDamage(hDamage);
                if
(!player.GetComponent<Animator>().GetBool("roll"))
                {player.GetComponent<Interaction>().enabled = false;
                StartCoroutine(RotateCharacter());
                player.GetComponent<Animator>().SetBool("fly",
true);}
                break;
            }
        }
    }
    public IEnumerator RotateCharacter()
    {
        if (player.GetComponent<Interaction>().enabled == false)
        {
            yield return new WaitForFixedUpdate();
            player.LookAt(transform.position);
        }
        yield return new WaitForFixedUpdate();
    }
}
```

Лістинг Б.2 – Клас FightController

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

public class FightContoller : MonoBehaviour
{
    private Animator playerAnim;
    private Transform atackField;
    public float rmRadius, rmDamage, lmDamage;
    private bool lhit, rhit, block, isEnemy;
    private AnimatorStateInfo stateInfo;

    void Start()
    {
```

```
        playerAnim = GetComponent<Animator>();
        atackField = gameObject.transform.GetChild(0);
    }

    // Update is called once per frame
    void Update()
    {

        if                          (isEnemy                          &&
Input.GetKeyDown(KeyCode.Mouse0)&&CompareAnim())
        {
            rhit = true;
            if (!lhit)
                playerAnim.SetBool("hit", true);
        }
        if                          (isEnemy                          &&
Input.GetKeyDown(KeyCode.Mouse1)&&CompareAnim())
        {
            lhit = true;
            if (!rhit)
                playerAnim.SetBool("leftHit", true);
        }
        if (Input.GetKeyDown(KeyCode.Q))
        {
            playerAnim.SetBool("block", true);
            block = true;
        }
        else if (Input.GetKeyUp(KeyCode.Q))
        {
            playerAnim.SetBool("block", false);
            block = false;
        }
        //Search for enemies
        Collider[]                    colliders                     =
Physics.OverlapSphere(transform.position, 10);
        isEnemy = false;
        foreach (Collider num in colliders)
        {
            if (num.tag.Equals("Enemy"))
            {
                isEnemy = true;
                break;
            }
        }
        //Prepare to fight
        if (Input.GetKey(KeyCode.LeftControl))
            isEnemy = true;
        playerAnim.SetBool("fighting", isEnemy);
    }
    public void Combo1()
    {
        if (lhit)
        {
```

```
            playerAnim.SetBool("leftHit", true);
        }
    }
    public void Combo2()
    {
        if (rhit)
        {
            playerAnim.SetBool("hit", true);
        }
    }
    public bool isBlocking()
    {
        return block;
    }
    public void DisableAll()
    {
        lhit = false;
        rhit = false;
        playerAnim.SetBool("hit", false);
        playerAnim.SetBool("leftHit", false);
    }
    public bool CompareAnim()
    {
        stateInfo = playerAnim.GetCurrentAnimatorStateInfo(0);
        if         (stateInfo.IsName("Left         Melle")         ||
stateInfo.IsName("Right Melle") || stateInfo.IsName("Fight Idle") ||
stateInfo.IsName("Fight Walk"))
            return true;
        else
            return false;
    }
    public List<Collider> CheckForEnemies()
    {
        Collider[]                    colliders                    =
Physics.OverlapSphere(atackField.position, rmRadius);
        List<Collider> enemies = new List<Collider>();
        foreach (Collider num in colliders)
        {
            if (num.tag.Equals("Enemy"))
            {
                enemies.Add(num);
            }
        }
        return enemies;
    }
    public void RightMelle()
    {
        foreach (Collider num in CheckForEnemies())
        {
            if (num.GetComponent<HealthManager>())

num.GetComponent<HealthManager>().GetDamage(rmDamage);
        }
```

```
    }
    public void LeftMelle()
    {
        foreach (Collider num in CheckForEnemies())
        {
            if (num.GetComponent<HealthManager>())

num.GetComponent<HealthManager>().GetDamage(lmDamage);
        }
    }
    public void IncreasePower(float duration, float perCent)
    {
        StartCoroutine(increasePowerCoroutine(duration, perCent));
    }
    public IEnumerator increasePowerCoroutine(float duration, float
perCent)
    {
        float oldrmDamage = rmDamage, oldlmDamage = lmDamage;
        rmDamage += Mathf.RoundToInt(rmDamage*(perCent/100));
        lmDamage += Mathf.RoundToInt(lmDamage*(perCent/100));
        GameObject       effect       =       GameObject.Find("Main
Camera").transform.Find("Canvas").transform.Find("Effect").gameObjec
t;
        effect.SetActive(true);

effect.transform.Find("Description").GetComponent<TextMeshProUGUI>()
.text = "Збільшення сили";
        TextMeshProUGUI                    timer                    =
effect.transform.Find("Timer").GetComponent<TextMeshProUGUI>();
        timer.text                                                  =
$"{Mathf.FloorToInt(duration/60):00}:{Mathf.FloorToInt(duration%60):
00}";
        while(duration>0)
        {
            yield return new WaitForSeconds(1);
            duration-=1;
            timer.text                                              =
$"{Mathf.FloorToInt(duration/60):00}:{Mathf.FloorToInt(duration%60):
00}";
        }
        rmDamage = oldrmDamage;
        lmDamage = oldlmDamage;
        effect.SetActive(false);
    }
}
```

Лістинг Б.3 – Клас HealthManager

```
using System;
using UnityEngine;
using UnityEngine.UI;

public class HealthManager : MonoBehaviour
```

```
{
    public GameObject barPref;
    public float maxHealth;
    [NonSerialized] public float curHealth;
    private GameObject healthBar;
    private Canvas canvas;
    private Camera cam;
    private Transform place;
    private Text text;
    void Start()
    {
        curHealth = maxHealth;
        if (GameObject.Find("Main Camera"))
        {
            cam                 =                GameObject.Find("Main
Camera").GetComponent<Camera>();
            canvas                                                    =
cam.transform.GetChild(0).GetComponent<Canvas>();
            if (!canvas)
                Debug.Log("There is no canvas");
        }
        else
            Debug.Log("Camera hasn't been found");

        if (transform.Find("BarPos"))
            place = gameObject.transform.Find("BarPos");
        else
            Debug.Log("There is no tansform for health bar");
        if (!barPref)
            Debug.Log("There is no prefab for health bar");
    }

    public void ShowHealthBar()
    {
        if (!healthBar&&barPref)
        {
            healthBar = Instantiate(barPref, canvas.transform);
            text                                                      =
healthBar.transform.Find("Value").GetComponent<Text>();
            text.text = curHealth + "/" + maxHealth;
            Transform bar = healthBar.transform.Find("Health");
            bar.localScale  =  new  Vector3(curHealth  /  maxHealth,
bar.localScale.y, 0);
        }
    }
    public void HideHealthBar()
    {
        if(healthBar)
            Destroy(healthBar);
    }
    void Update()
    {
        if (healthBar)
```

```
            healthBar.transform.position                                    =
cam.WorldToScreenPoint(place.position);
    }
    public void GetDamage(float damage)
    {
        curHealth -= damage;
        if(healthBar)
        {
            Transform bar = healthBar.transform.Find("Health");
            bar.localScale  =  new  Vector3(curHealth  /  maxHealth,
bar.localScale.y, 0);
        }

        if (curHealth <= 0)
        {
            if (healthBar)
                healthBar.SetActive(false);
            GetComponent<Animator>().SetBool("death", true);
            Destroy(GetComponent<Collider>());
            if (GetComponent<EnemyBehavior>())
                GetComponent<EnemyBehavior>().enabled = false;
            else if (GetComponent<BossBehavior>())
                GetComponent<BossBehavior>().enabled = false;
        }
        else
        {
            text                                                            =
healthBar.transform.Find("Value").GetComponent<Text>();
            text.text = curHealth + "/" + maxHealth;
        }
    }
    public void DestroyAnimator()
    {
        Destroy(healthBar);
        foreach (Behaviour comp in GetComponents<Behaviour>())
        {
            Destroy(comp);
        }
    }
}
```

Лістинг Б.4 – Клас Inventory

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

public class Inventory : MonoBehaviour
{
    public GameObject inventory;
    public GameObject chestInventory;
```

```
    public GameObject itemPref;
    public UnityEngine.Vector3 firstSlot;
    public float interval;
    [NonSerialized] public LootItem choosedItem;
    public TextMeshProUGUI description;
     private List<LootItem> lootItems = new List<LootItem>();
    private List<GameObject> items = new List<GameObject>();
    public int row, col;

    void Update()
    {
        if (Input.GetKeyUp(KeyCode.I))
        {
            if (inventory.activeSelf == false)
            {
                ShowInventory();
            }
            else
            {
                inventory.SetActive(false);
                ClearAll();

if(inventory.transform.parent.Find("PasteWindow(Clone)"))

GameObject.Find(inventory.transform.parent.Find("PasteWindow(Clone)"
).Find("ObjName").GetComponent<Text>().text).GetComponent<Paster>().
CloseWindow();
            }
        }

        if  (Input.GetKeyUp(KeyCode.Escape)  &&  inventory.activeSelf
|| Input.GetKeyUp(KeyCode.Escape) && chestInventory.activeSelf)
        {
            GetComponent<Chestinventory>().ClearAll();
            GetComponent<Chestinventory>().CloseChest();
            chestInventory.SetActive(false);
            ClearAll();
            inventory.SetActive(false);

if(inventory.transform.parent.Find("PasteWindow(Clone)"))

GameObject.Find(inventory.transform.parent.Find("PasteWindow(Clone)"
).Find("ObjName").GetComponent<Text>().text).GetComponent<Paster>().
CloseWindow();
        }


    }
    public void ShowInventory()
    {
        inventory.SetActive(true);
        if (chestInventory.activeSelf)
        {
```

```
            GetComponent<Chestinventory>().ClearAll();
            GetComponent<Chestinventory>().CloseChest();
            chestInventory.SetActive(false);
        }
        GameObject temp;
        UnityEngine.Vector3 pos = new UnityEngine.Vector3();
        for (int i = 0; i < lootItems.Count; i++)
        {
            temp = Instantiate(itemPref, inventory.transform);
            temp.GetComponent<LootIButt>().TakeData(lootItems[i]);
            pos.z = 0;
            pos.x = firstSlot.x + interval * (i % col);
            pos.y = firstSlot.y - interval * Mathf.Floor(i / col);
            temp.GetComponent<RectTransform>().anchoredPosition     =
pos;
            items.Add(temp);
        }
    }
    public void ClearAll()
    {
        foreach (GameObject i in items)
            Destroy(i);
        items.Clear();
        description.text = "";
    }
    public void ChooseItem(LootItem lootItem)
    {
        choosedItem = lootItem;
        description.text = lootItem.desc;
    }
    public void UseItem()
    {
        if(choosedItem!=null)
        {
            if(choosedItem.isUsable())
            {
                choosedItem.Use();
                RemoveItem(choosedItem);
                choosedItem = null;
                description.text = "";
            }
            else
                Debug.Log("Choosed item can't be used");
        }
        else
            Debug.Log("There is no choosed item");
    }
    public void RemoveItem(LootItem lootItem)
    {
        lootItems.Remove(lootItem);
        ClearAll();
        ShowInventory();
    }
```

```
    public void AddItem(LootItem newItem)
    {
        Debug.Log("Item: " + newItem.desc);
        lootItems.Add(newItem);
    }
    public void AddItem(List<LootItem> newLootItems)
    {
        foreach(LootItem i in newLootItems)
        {
            lootItems.Add(i);
        }
    }
}
```

**Додаток В**

## Знімки екрану у процесі гри

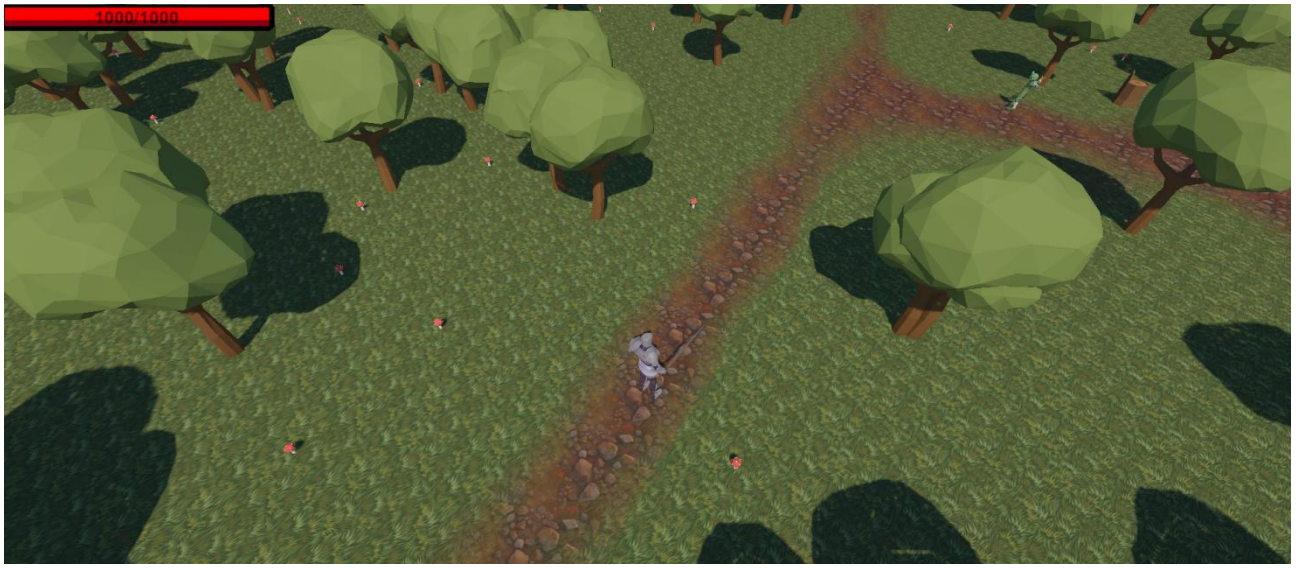


Рисунок В.1 – Пересування персонажа

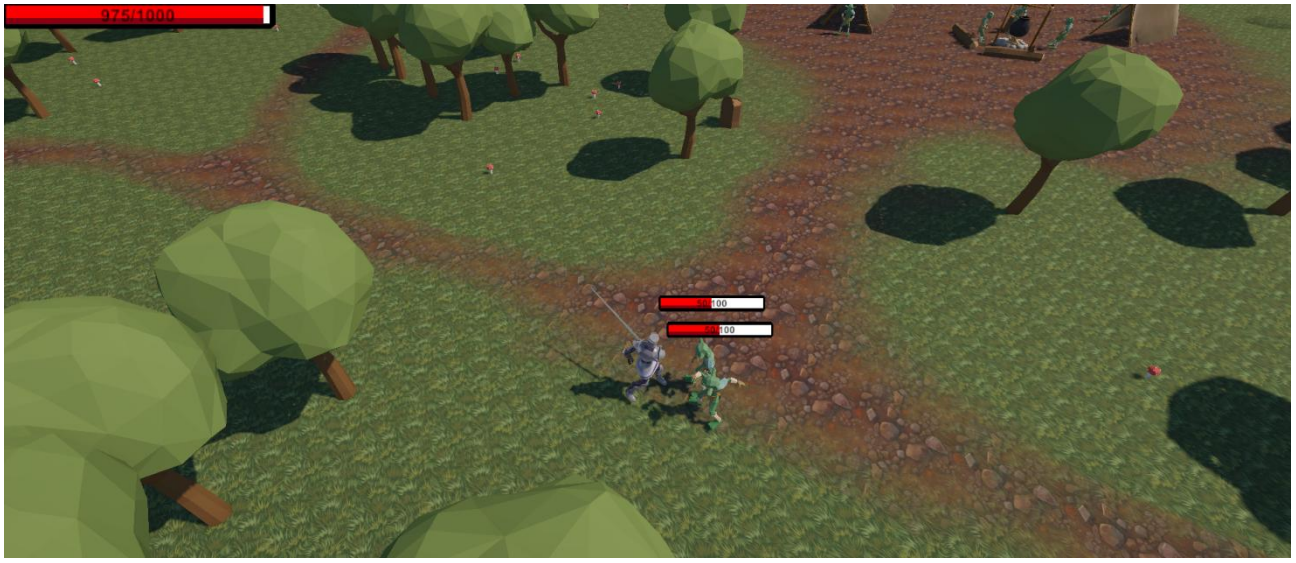


Рисунок В.2 – Ведення бою

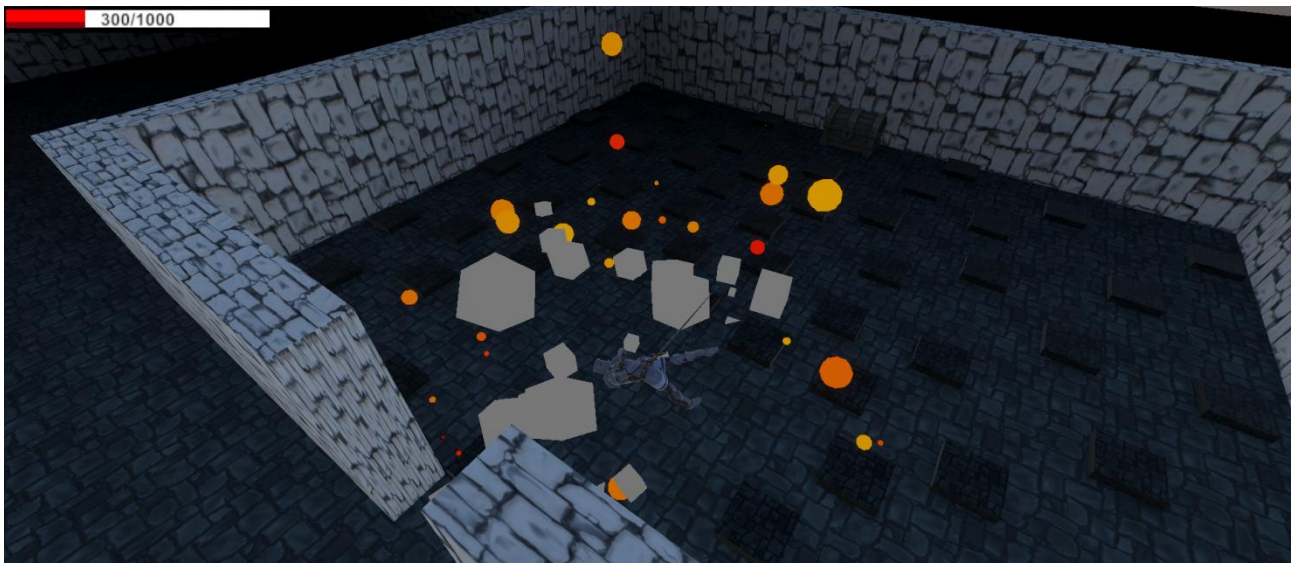


Рисунок В.3 – Реагування на пастки

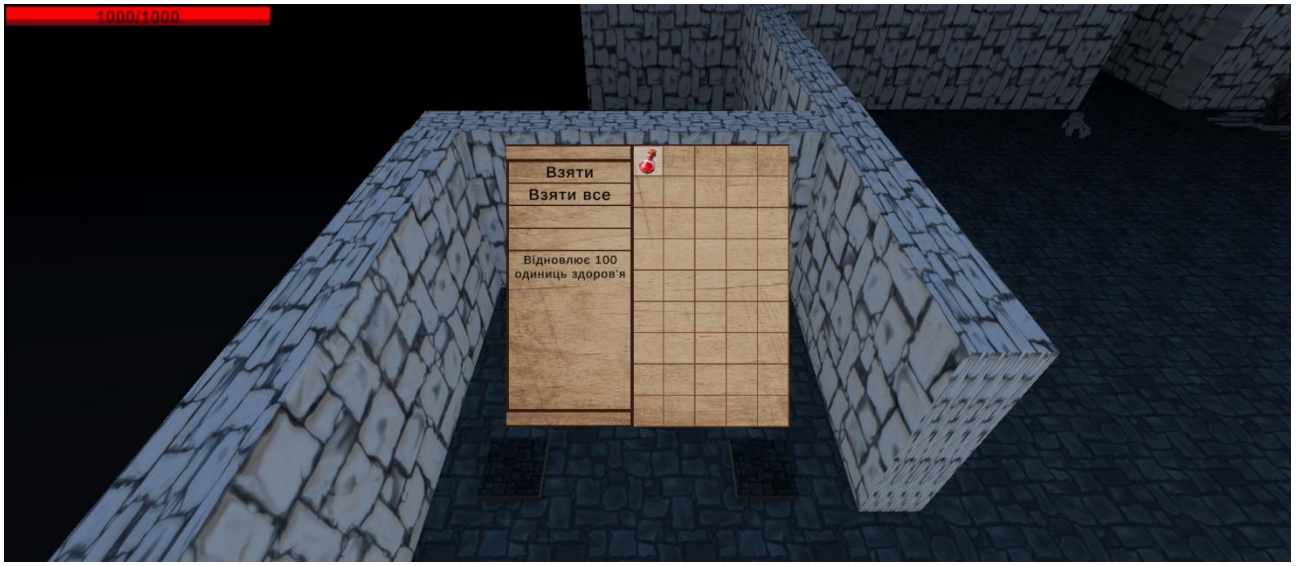


Рисунок В.4 – Взаємодія із скринею

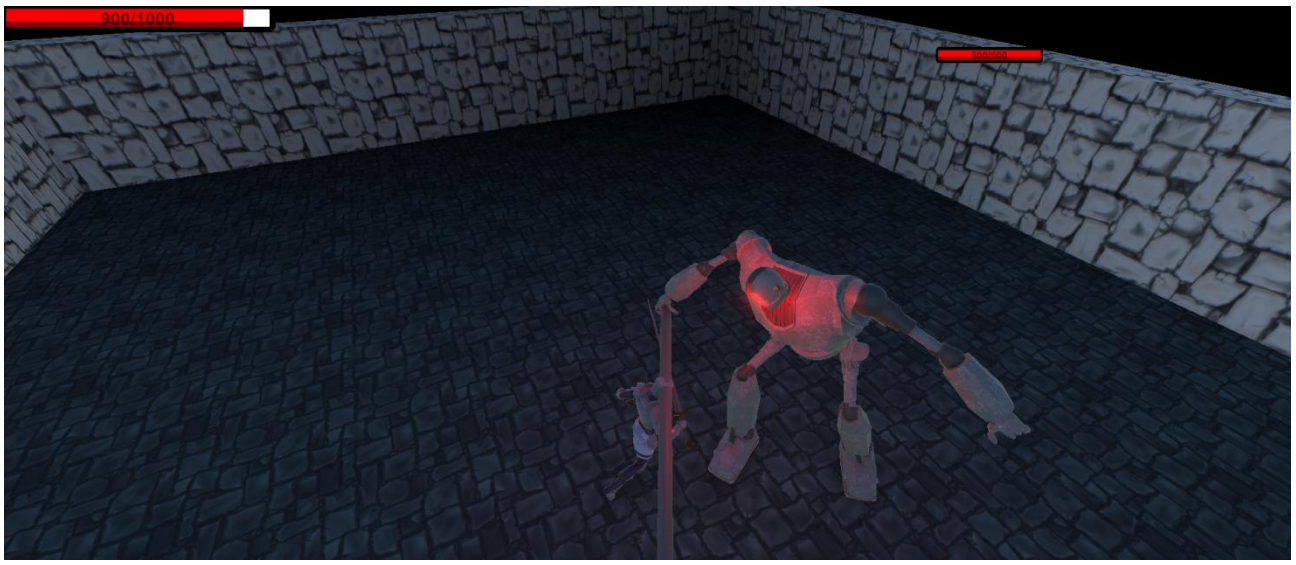


Рисунок В.5 – Бій з босом-автоматоном


Рисунок В.6 – Користування інвентарем

**Додаток Г**

**Обкладинка гри**

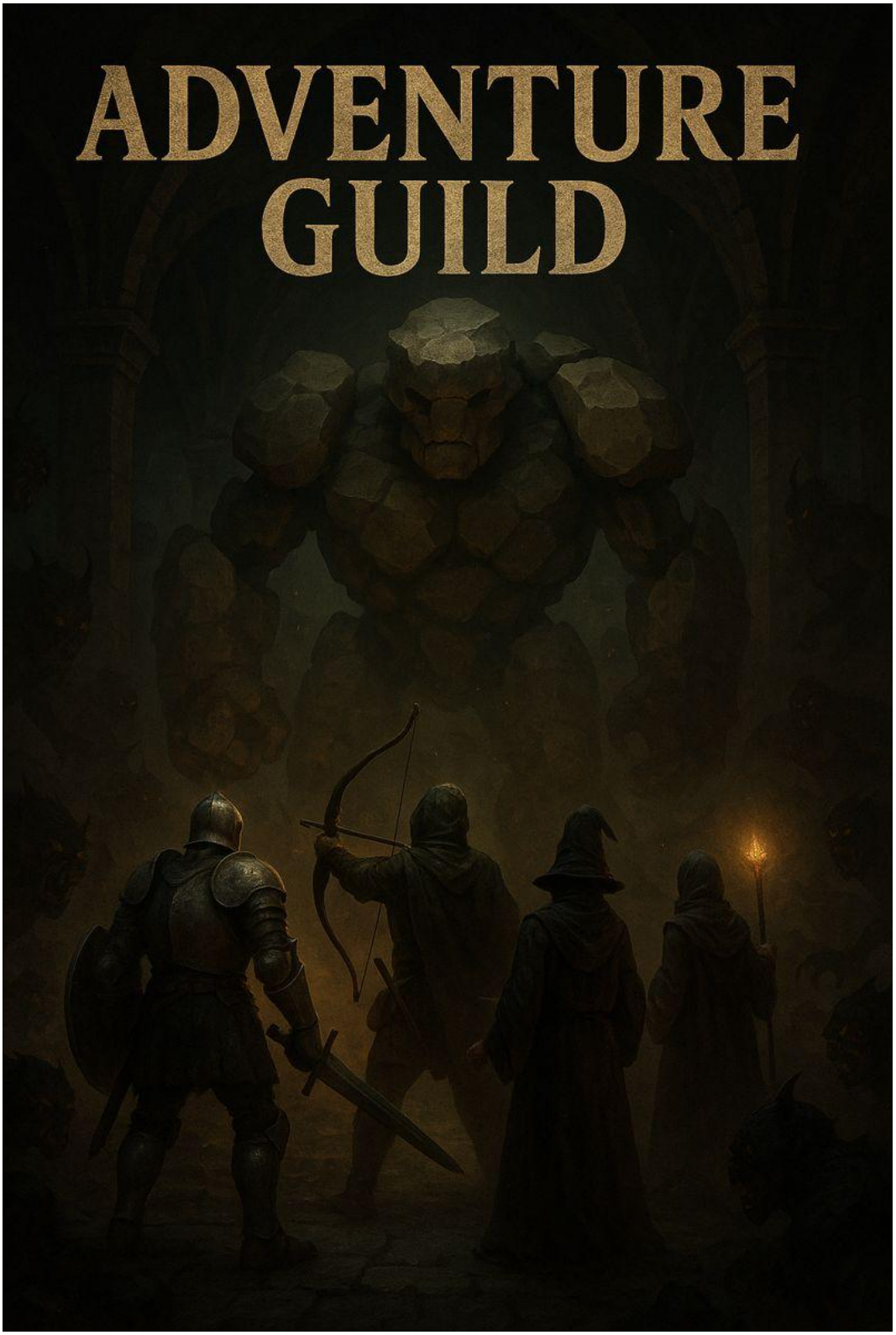


Рисунок Г.1 – Обкладинка гри «Adventure Guild»