Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ ЖАНРУ HORROR З ВИКОРИСТАННЯМ UNITY

DEVELOPMENT OF A HORROR COMPUTER GAME USING UNITY

спеціальність 122 Комп'ютерні науки

освітня програма «Комп'ютерні науки»

Виконав: здобувач вищої освіти
групи КН-41
Андрощук Богдан Олегович

_____
(підпис)

Керівник: асистент
Хиць Руслан Андрійович

_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2026 р.
Гарант освітньої програми:
д.пед.н., професор
Тулашвілі Юрій Йосипович

_____
(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерних наук*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *122 Комп'ютерні науки*
Освітня програма: *«Комп'ютерні науки»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Валерій ЛІЩИНА
«16» січня 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ПЕРШОГО (БАКАЛАВРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Андрощук Богдан Олегович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка комп'ютерної гри жанру horror з використанням Unity»
Керівник асистент Хиць Руслан Андрійович
затверджені наказом закладу вищої освіти від «*16*» *січня* *2026* р. № 32/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи: *«02» червня 2026 р.*

3. Вихідні дані до роботи: ігровий рушій Unity 2022.3 LTS, мова програмування C#, середовище розробки Visual Studio 2022, система контролю версій Git. механізм Raycast для визначення ігрових об'єктів

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):
Аналіз сучасного стану проблеми, існуючих методів і засобів її розв'язання, аналіз і вибір засобів проектування, опис функціонального наповнення об'єкта проектування, розробка й обґрунтування системного наповнення, оцінка ергономічних та надійністних параметрів проектованої системи, функціонально-структурна схема роботи об'єкта проектування

5. Перелік графічного матеріалу: 8 *зображень, презентація*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Хиць Р. А.*		
Теоретичне дослідження	*Хиць Р. А.*		
Практична реалізація	*Хиць Р. А.*		
Спеціальні розрахунки	*Хиць Р. А.*		
Показник запозичень тексту	_____ %		
Інструментальна перевірка	*Кошелюк В. А.*		
Нормоконтроль	*Хиць Р. А.*		
Гарант ОПП	*Тулашвілі Ю. Й.*		

7. Дата видачі завдання *«16» січня 2026 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	*Провести огляд літературних джерел по темі кваліфікаційної роботи*	*до 20.02.2026 р*	
2	*Провести аналіз загальної проблеми і вибір напрямків дослідження*	*до 27.02.2026 р.*	
3	*Розробити функціональну схему роботи програмного продукту*	*до 21.03.2026 р*	
4	*Описати засоби розробки об'єкта проектування*	*до 17.04.2026 р.*	
5	*Практична реалізація об'єкта проектування*	*до 26.04.2026 р.*	
6	*Провести спеціальні розрахунки*	*до 04.05.2026 р.*	
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*до 02.06.2026 р.*	

Здобувач вищої освіти _____ Богдан АНДРОЩУК

Керівник роботи _____ Руслан ХИЦЬ

АНОТАЦІЯ

Андрощук Б. О. Розробка прототипу хорор-гри на базі ігрового рушія Unity. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерні науки». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

У першому розділі виконано аналіз предметної області, розглянуто особливості ігор у жанрі хорор та подібних інтерактивних ігрових систем, визначено основні підходи до реалізації керування ігровими локаціями та взаємодії користувача з ігровим середовищем.

Другий розділ містить специфікацію функціональних і нефункціональних вимог до програмного продукту, опис структури майбутньої системи та визначення основних компонентів ігрової логіки.

У третьому розділі здійснено розробку прототипу хорор-гри на базі рушія Unity, включаючи реалізацію системи керування персонажем, системи керування камерою, механізму взаємодії з локаціями, а також системи визначення та використання точок генерації ігрових об'єктів. Також реалізовано базові візуальні ефекти, що забезпечують виділення активної локації та покращують сприйняття ігрового процесу.

Четвертий розділ містить перспективи розвитку програмного продукту, зокрема розширення ігрових механік шляхом додавання різних типів ворогів, системи ігрових предметів та покращення візуально-аудіальної складової гри.

У висновках узагальнено результати виконаної роботи та підтверджено можливість використання розробленого прототипу як бази для подальшого розвитку повноцінної гри.

Ключові слова: Unity, хорор-гра, C#, ігровий рушій, прототип, взаємодія з локаціями, камера, spawn points, game development

ANNOTATION

Androshchuk Bohdan. Development of a Horror Game Prototype Based on the Unity Game Engine. Manuscript.

Bachelor's qualification paper in the study program "Computer Science". Lutsk National Technical University. Lutsk, 2026.

The bachelor's qualification paper consists of an introduction, four chapters, conclusions, a list of references, and appendices.

The first chapter provides an analysis of the subject area, examines the features of horror games and similar interactive game systems, and defines the main approaches to implementing location-based control and user interaction with the game environment.

The second chapter contains the specification of functional and non-functional requirements for the software product, a description of the system structure, and the definition of the main components of the game logic.

The third chapter presents the development of a horror game prototype based on the Unity engine, including the implementation of a character control system, a camera control system, a location interaction mechanism, and a system for defining and using game object spawn points. Basic visual effects were also implemented to highlight the active location and improve the user experience.

The fourth chapter describes the prospects for further development of the software product, including the expansion of gameplay mechanics by adding different types of enemies, an item system, and improvements to visual and audio components.

The conclusions summarize the results of the work and confirm that the developed prototype can be used as a foundation for further development of a full-scale game.

Keywords: Unity, horror game, C#, game engine, prototype, location interaction, camera, spawn points, game development.

ЗМІСТ

ВСТУП

У сучасному світі цифрових технологій інтерактивні розважальні системи займають важливе місце в індустрії програмного забезпечення. Особливо швидкими темпами розвивається сфера комп'ютерних ігор, яка поєднує досягнення програмної інженерії, комп'ютерної графіки та інтерактивного дизайну. Сучасні ігрові продукти вже давно виходять за межі простих розваг і перетворюються на складні програмні системи з розвиненою логікою взаємодії та високим рівнем занурення користувача.

Одним із популярних жанрів є хорор-ігри, які спрямовані на створення атмосфери напруження, невизначеності та емоційного занурення гравця. Такі ігри часто використовують обмежену інформацію про ігровий світ, динамічне освітлення та аудіовізуальні ефекти для формування відповідного емоційного досвіду. У цьому контексті важливим є не лише художнє оформлення, але й технічна реалізація механік взаємодії користувача з ігровим середовищем.

З огляду на це виникає потреба у розробці прототипів ігрових систем, які дозволяють досліджувати підходи до реалізації керування персонажем, камерою та ігровими об'єктами в умовах обмеженого простору та структурованих локацій. Такі прототипи є важливим етапом у процесі створення повноцінних ігрових продуктів, оскільки дозволяють перевірити архітектурні рішення та основні механіки взаємодії.

Метою даної кваліфікаційної роботи бакалавра є розробка прототипу хорор-гри на базі ігрового рушія Unity з використанням мови програмування C#, що забезпечує реалізацію базових механік взаємодії користувача з ігровим середовищем.

Об'єктом дослідження є процес розробки інтерактивних ігрових застосунків у середовищі ігрового рушія Unity.

Предметом дослідження є методи та програмні засоби реалізації ігрових механік керування персонажем, камерою та взаємодії з локаціями у тривимірному ігровому середовищі.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

– проаналізувати особливості сучасних хорор-ігор та підходи до реалізації їхніх базових механік;

– сформулювати вимоги до програмного продукту;

– розробити архітектуру прототипу гри на основі рушія Unity;

– реалізувати основні ігрові механіки, зокрема керування персонажем, камерою та взаємодією з локаціями;

– забезпечити можливість подальшого розширення функціональності розробленого прототипу.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз сучасного стану проблеми

Сучасна індустрія комп'ютерних ігор є однією з найбільш динамічно розвинених галузей програмної інженерії. Щорічне зростання обсягу ринку, поява нових ігрових платформ та вдосконалення інструментів розробки суттєво знижують поріг входу для незалежних розробників і стимулюють появу нових ігрових концепцій. У цьому контексті особливої актуальності набуває розробка ігрових прототипів як методу дослідження та перевірки архітектурних рішень до початку повноцінного виробництва [4; 8].

Одним із жанрів, що стабільно зберігає популярність серед гравців, є horror. Ігри цього жанру орієнтовані на створення емоційного впливу через атмосферу напруження, невизначеності та страху. На відміну від більшості інших жанрів, де успіх визначається переважно технічною складністю або глибиною ігрових механік, у хорор-іграх ключову роль відіграє сукупність взаємодіючих елементів: системи керування персонажем, динамічного освітлення, аудіовізуальних ефектів та структури ігрового простору [5; 7]. Типовими представниками жанру є Outlast та Amnesia: The Dark Descent [11; 12], у яких основний акцент зроблено на дослідженні локацій, обмеженій інформації про ігровий світ та безпосередньому контролі персонажа гравцем.

З технічної точки зору реалізація хорор-гри є нетривіальним завданням, оскільки потребує інтеграції декількох складних підсистем: керування персонажем і камерою, взаємодії з ігровим середовищем, системи анімації та генерації ігрових об'єктів. Складність полягає не лише в реалізації кожної підсистеми окремо, а й у забезпеченні їхньої коректної взаємодії в межах єдиної архітектури. Помилки на етапі проєктування архітектури можуть призвести до значних витрат на рефакторинг на пізніших етапах розробки [14; 16].

Для вирішення цих завдань широко застосовується підхід прототипування – розробки спрощеної функціональної версії майбутнього

продукту, що дозволяє перевірити ключові архітектурні рішення та ігрові механіки на ранньому етапі. Ігровий рушій Unity є одним із найпоширеніших інструментів для такого роду розробки завдяки компонентно-орієнтованій архітектурі, підтримці мови C# та великій кількості вбудованих інструментів для роботи з фізикою, анімацією та інтерфейсом [1; 3]. Водночас питання оптимальної організації взаємодії між ігровими підсистемами в Unity залишається актуальним, оскільки від правильності прийнятих архітектурних рішень залежить масштабованість і підтримуваність проєкту в майбутньому [13].

1.2 Аналіз аналогічних програм, систем та обґрунтування вибраного напряму

Для визначення оптимального напряму розробки було проведено аналіз існуючих ігрових рішень, які частково реалізують необхідний функціонал.

Серед ігор із механіками управління ресурсами варто виділити Fallout Shelter, яка демонструє ефективну реалізацію системи розвитку бази, взаємодії з персонажами та управління внутрішніми процесами [9]. Гравець здійснює контроль над різними аспектами функціонування укриття, що забезпечує глибину ігрового процесу.

Іншим прикладом є This War of Mine, де реалізовано більш серйозний та емоційно насичений підхід до виживання. Гра поєднує елементи управління ресурсами з моральними виборами, що створює додатковий рівень занурення [10].

У свою чергу, класичні horror-ігри, такі як Outlast та Amnesia: The Dark Descent, забезпечують високий рівень напруги та страху, але не передбачають складних систем управління базою або ресурсами [11; 12].

Проведений аналіз дозволяє зробити висновок, що кожен із розглянутих підходів має свої переваги, але жоден із них не забезпечує повноцінного поєднання:

– систем управління ресурсами,

– розвитку інфраструктури,

– атмосфери horror.

У зв’язку з цим обрано напрям розробки гри, яка поєднує:

– механіки менеджменту та симуляції,

– систему випадкових подій,

– елементи психологічного напруження.

Як основні методи реалізації обрано подієво-орієнтований підхід, що дозволяє створювати динамічний ігровий процес, а також використання алгоритмів випадкової генерації подій для підвищення реіграбельності [16].

1.3 Аналіз і вибір засобів проєктування комп’ютерної гри

Для визначення оптимального напряму розробки та вибору архітектурних підходів було проведено аналіз існуючих ігрових рішень, які реалізують механіки, схожі до тих, що заплановані у розроблюваному прототипі. Аналіз охоплював як класичні хорор-ігри з прямим керуванням персонажем, так і ігри з непрямим керуванням через вибір локацій.

Серед представників жанру horror з прямим керуванням персонажем варто виділити Outlast [11] та Amnesia: The Dark Descent [12]. Обидві гри демонструють високий рівень реалізації атмосфери напруження та страху завдяки поєднанню динамічного освітлення, обмеженої видимості та реактивної поведінки ворогів. З технічної точки зору ці ігри реалізують складні системи керування персонажем від першої особи, фізику взаємодії з об’єктами та просунутий штучний інтелект ворогів. Однак їхня архітектура розрахована на повноцінний комерційний продукт і не є придатною для прототипування – складність підсистем унеможливлює їх відтворення в рамках навчального проєкту без суттєвих спрощень [5; 7].

Іншим релевантним прикладом є Five Nights at Freddy’s – хорор-гра, в якій гравець керує не персонажем безпосередньо, а здійснює спостереження за

локаціями через камери та взаємодіє з об’єктами інтерфейсу. Такий підхід є концептуально близьким до розроблюваного прототипу: гравець обирає локацію для фокусу уваги, а не безпосередньо переміщує персонажа в просторі. Це підтверджує перспективність обраного напряму та доводить, що механіка керування через вибір локацій може ефективно поєднуватись з атмосферою хорор [4; 5].

This War of Mine [10] є прикладом гри, де реалізовано механіку непрямого керування персонажами через вибір локацій і завдань. Гра демонструє, що переміщення персонажів між точками на карті з автоматичним виконанням дій є зрозумілою та прийнятною для гравця моделлю взаємодії. Разом із тим ця гра орієнтована на механіки виживання та морального вибору, а не на створення атмосфери страху, що обмежує можливість прямого перенесення її підходів у хорор-контекст.

Проведений аналіз дозволяє систематизувати переваги та недоліки розглянутих рішень. Порівняльну характеристику аналогів наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз характеристик аналогів

Гра	Механіка керування	Атмосфера horror	Прийнятна складність реалізації
Outlast	Пряме керування (1-ша особа)	Висока	Ні
Amnesia: TDD	Пряме керування (1-ша особа)	Висока	Ні
FNAF	Непряме (вибір камер)	Висока	Частково
This War of Mine	Непряме (вибір локацій)	Низька	Так
Розроблюваний прототип	Непряме (вибір локацій)	Середня	Так

На основі проведеного аналізу визначено напрям розробки: прототип хорор-гри з непрямим керуванням персонажем через вибір локацій. Такий підхід поєднує:

– механіку вибору локацій із автоматичним переміщенням персонажа, що знижує технічну складність реалізації порівняно з прямим керуванням від першої особи;

– візуальне виділення активної локації та систему spawn-точок як основу

для подальшого розширення ігрового контенту;

– атмосферні елементи хорор-жанру, що реалізуються через систему візуальних ефектів та заплановане аудіовізуальне оформлення.

Як основний інструмент реалізації обрано ігровий рушій Unity з мовою програмування C#, що забезпечує необхідний функціонал при відносно невисокому порозі входу та широкій документованості [1; 3]. Компонентно-орієнтована архітектура Unity добре від{повідає обраному підходу: кожна підсистема реалізується як окремий компонент MonoBehaviour, що забезпечує незалежність модулів та зручність їхнього тестування і розширення [15]

1.4 Постановка завдання на кваліфікаційну роботу бакалавра.

Метою кваліфікаційної роботи є розробка прототипу комп'ютерної гри жанру horror на базі ігрового рушія Unity з використанням мови програмування C#, що забезпечує реалізацію базових механік взаємодії користувача з ігровим середовищем через систему керування локаціями.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати сучасний стан розробки ігор жанру horror та виявити основні підходи до реалізації їхніх базових механік;

– провести огляд існуючих аналогів і обґрунтувати вибір напряму розробки та інструментальних засобів;

– розробити архітектуру прототипу гри на основі компонентно-орієнтованого підходу Unity з визначенням основних програмних модулів та зв'язків між ними;

– реалізувати систему керування персонажем із автоматичним переміщенням між локаціями та синхронізацією анімаційних станів;

– реалізувати систему керування камерою з підтримкою вільного переміщення, масштабування та автоматичного центрування;

– реалізувати систему взаємодії з локаціями на основі механізму Raycast із візуальним виділенням активної кімнати та оновленням spawn-точок;

– провести функціональне тестування розробленого прототипу та підтвердити коректність роботи всіх підсистем;

– оцінити масштабованість реалізованої архітектури та визначити напрями подальшого розвитку продукту.

Об’єктом дослідження є процес розробки інтерактивних ігрових застосунків у середовищі ігрового рушія Unity.

Предметом дослідження є методи та програмні засоби реалізації ігрових механік керування персонажем, камерою та взаємодії з локаціями у тривимірному ігровому середовищі.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання

Розробка комп'ютерної гри жанру horror з елементами симуляції потребує вибору оптимальних технологій, які забезпечать баланс між функціональністю, продуктивністю та складністю реалізації. Враховуючи особливості поставленого завдання, ключовим етапом є визначення програмного середовища, архітектурного підходу та алгоритмів реалізації ігрової логіки [1; 14].

Як основний інструмент розробки було обрано Unity, який є універсальним ігровим рушієм для створення як 2D, так і 3D проєктів. Даний вибір обумовлений рядом переваг, серед яких: підтримка мови програмування C#, наявність вбудованої фізики, системи анімацій, а також широкі можливості для створення користувацького інтерфейсу.

Використання мови програмування C# дозволяє реалізувати об'єктно-орієнтований підхід до розробки, що є особливо важливим при створенні складних систем із великою кількістю взаємодіючих компонентів, таких як персонажі, ресурси та події.

Для реалізації ігрового процесу застосовується подієво-орієнтований підхід, який передбачає обробку ігрових подій (наприклад, атака ворогів, аварії, зміни стану ресурсів) через систему тригерів і обробників. Це дозволяє створити динамічний і непередбачуваний ігровий процес.

Також використовується модульний підхід до побудови системи, при якому окремі компоненти (система ресурсів, управління персонажами, генерація подій) реалізуються як незалежні модулі, що взаємодіють між собою через чітко визначені інтерфейси [16].

Для збереження стану гри застосовуються механізми серіалізації даних, що дозволяють записувати поточний стан ігрового середовища у файл та

відновлювати його при повторному запуску.

Отже, обрані технології та підходи забезпечують ефективну реалізацію поставленого завдання та дозволяють створити масштабовану і гнучку систему.

2.2 Функціонально-структурна схема роботи об’єкта проектування

Функціонально-структурна схема розроблюваної системи відображає основні програмні компоненти гри та логіку їхньої взаємодії між собою. Систему побудовано за принципом розподілу на незалежні функціональні підсистеми, кожна з яких виконує чітко визначений набір задач і взаємодіє з іншими через спільні інтерфейси та центральне сховище стану.

Верхній рівень схеми представляє зовнішню сутність – користувача, який взаємодіє з грою виключно через модуль користувацького інтерфейсу (UI Manager). UI Manager приймає події введення (кліки миші, прокручування) і транслює їх у виклики до центрального координатора системи – Game Manager.

Game Manager є керуючим модулем системи, що відповідає за загальний стан гри та координацію роботи всіх підсистем. Він ініціалізує компоненти при запуску, управляє ігровим циклом і забезпечує передачу команд до відповідних підсистем. Від Game Manager відгалужуються чотири основні функціональні підсистеми: SelectObject, PlayerMovements, CameraManager та AnimationManager.

До основних компонентів системи належать:

– Game Manager – координує загальний стан гри та ігровий цикл;

– UI Manager – обробляє дії користувача та відображає стан інтерфейсу;

– SelectObject – реалізує механізм Raycast для визначення вибраної локації та передачі сигналу про переміщення персонажа;

– PlayerMovements – забезпечує автоматичне переміщення персонажа між waypoint-ами, зміну орієнтації та стану анімації;

– CameraManager – керує переміщенням камери по сцені, зміною масштабу та автоматичним центруванням на персонажі;

– AnimationManager – управляє станами Unity Animator, забезпечує перемикання анімаційних кліпів для персонажа і локацій;

– RoomStats – централізоване сховище поточного стану рівня: зберігає посилання на активну локацію, масиви waypoint-ів та spawn-точок;

– SaveManager – виконує серіалізацію стану гри у формат JSON та завантаження збереженого стану.

Функціонально-структурну схему наведено на рисунку 2.1.

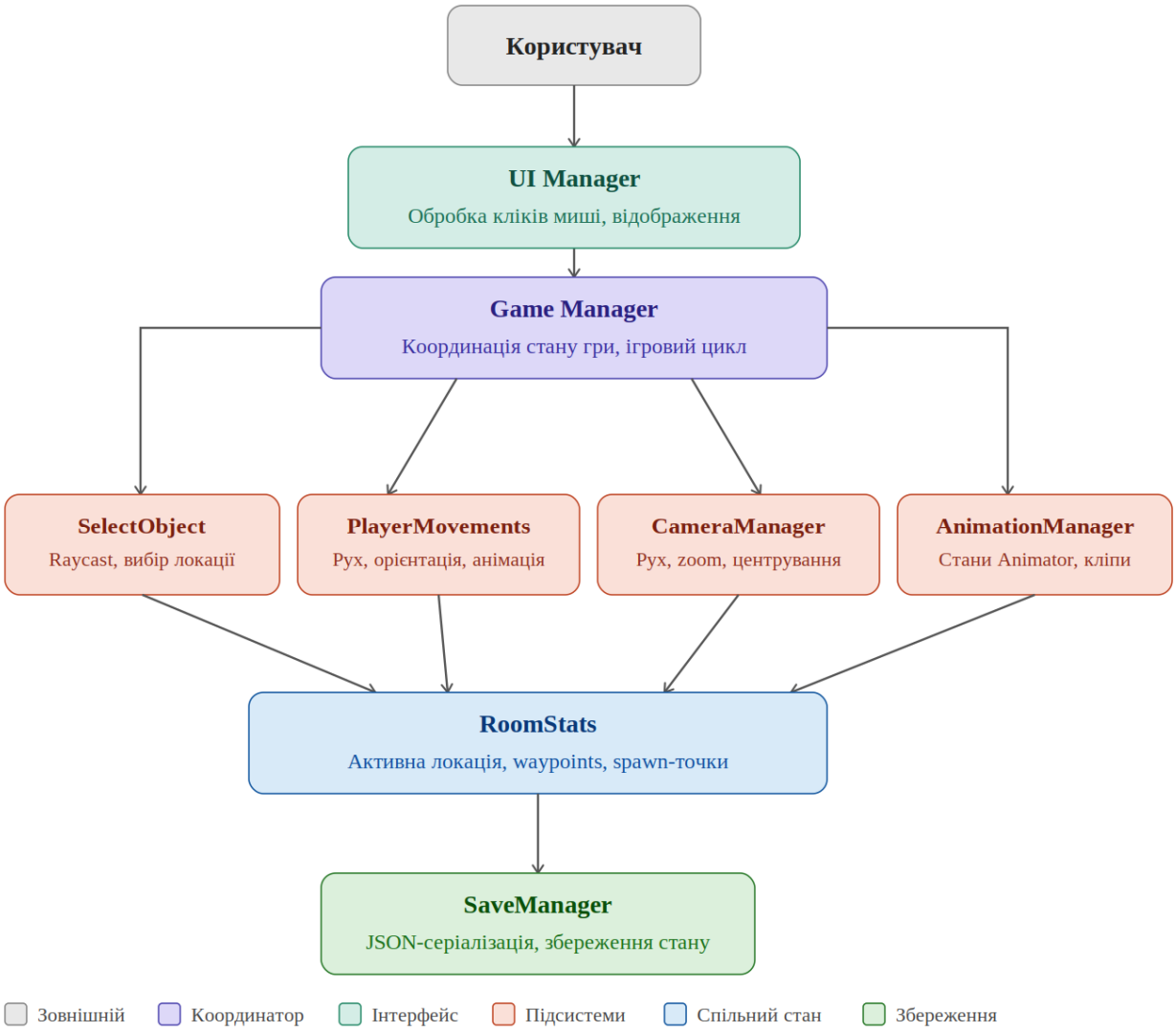

Рисунок 2.1 – Функціонально-структурна схема програмного забезпечення гри

Центральним елементом взаємодії між підсистемами є компонент RoomStats, який виступає спільним сховищем стану рівня. Після вибору

користувачем нової локації модуль SelectObject оновлює дані в RoomStats: записує посилання на активний ігровий об’єкт та формує масиви waypoint-ів, точок генерації предметів і ворогів. Компонент PlayerMovements зчитує ці дані для визначення цільової точки переміщення, а AnimationManager – для зміни анімаційного стану відповідних об’єктів. Така архітектура забезпечує слабку зв’язаність між підсистемами та єдину точку синхронізації стану.

Взаємодія між компонентами здійснюється через прямі посилання на компоненти Unity (SerializeField) та виклик відповідних публічних методів. Game Manager ініціює дії у підсистемах, підсистеми оновлюють спільний стан через RoomStats, а SaveManager зберігає знімок стану після кожної значущої події – зміни локації або завершення ігрової сесії. Така структура забезпечує гнучкість і розширюваність системи: додавання нових підсистем (наприклад, модуля ворогів або інвентаря) не потребує змін у вже реалізованих компонентах.

2.3 Створення моделі бази даних

Для забезпечення збереження та обробки даних у розроблюваній системі необхідно створити відповідну модель даних. У контексті комп’ютерної гри модель даних охоплює інформацію про поточний стан гри, характеристики персонажа, ігрові локації, генеровані об’єкти та прогрес гравця. Коректно спроєктована модель даних є фундаментом для розширення функціоналу гри у подальших ітераціях розробки.

Оскільки гра розробляється з використанням рушія Unity, доцільно застосувати вбудовані засоби збереження та організації даних, а саме: ScriptableObject для статичних конфігураційних даних (характеристики ворогів, параметри предметів) та серіалізацію у формат JSON для збереження динамічного стану гри між сесіями. Такий підхід дозволяє уникнути залежності від зовнішніх систем керування базами даних, спрощує розгортання програмного продукту та забезпечує достатню продуктивність для ігор

подібного масштабу.

На основі аналізу функціональних вимог до прототипу та запланованих напрямків його розвитку визначено п’ять основних сутностей моделі даних. Кожна сутність відповідає окремому ігровому елементу та містить набір атрибутів, що описують його стан і поведінку. Перелік сутностей із їхніми атрибутами наведено в таблиці 2.1.

Таблиця 2.1 – Основні сутності моделі даних

Сутність	Основні атрибути	Тип зберігання
Гравець (Player)	playerId, currentLocationId, healthPoints, inventoryItems[], progressFlags[]	JSON (SaveData)
Локація (Location)	locationId, locationName, isActive, animationState, connectedLocationIds[]	ScriptableObject
Персонаж (Character)	characterId, currentWaypointIndex, isMoving, moveSpeed, animationState	MonoBehaviour (Runtime)
Ігровий предмет (Item)	itemId, itemType, itemName, description, spawnWeight, effectValue	ScriptableObject
Ворог (Enemy)	enemyId, enemyType, healthPoints, moveSpeed, detectionRadius, patrolPoints[]	ScriptableObject
Подія (GameEvent)	eventId, eventType, triggerCondition, affectedEntityId, consequence	JSON (EventLog)

Сутність «Гравець» є центральною в моделі даних: вона зберігає поточний прогрес гравця, включаючи ідентифікатор активної локації, поточний стан здоров’я та список підібраних предметів. Саме ця сутність серіалізується у JSON-файл при збереженні гри та десеріалізується при її завантаженні.

Сутності «Ігровий предмет» та «Ворог» є конфігураційними: їхні дані не змінюються під час гри і тому зберігаються як ScriptableObject-активи у проєкті Unity. Це дозволяє зручно редагувати параметри балансу безпосередньо в інспекторі редактора без змін у програмному коді. Сутність «Подія» фіксує ігрові події в журналі для подальшого аналізу або реалізації системи досягнень.

Зв’язки між сутностями відображають логіку взаємодії ігрових елементів між собою. Правильно визначені зв’язки забезпечують цілісність даних та дозволяють уникнути дублювання інформації при розширенні функціоналу системи. Основні зв’язки між сутностями наведено в таблиці 2.2.

Таблиця 2.2 – Зв’язки між сутностями

Сутність 1	Тип зв’язку	Сутність 2	Опис зв’язку
Гравець	1 : 1	Локація	Гравець перебуває в одній активній локації
Гравець	1 : N	Ігровий предмет	Гравець може мати кілька предметів в інвентарі
Локація	1 : N	Ворог	У локації може знаходитись кілька ворогів
Локація	1 : N	Ігровий предмет	У локації може бути кілька предметів
Локація	M : N	Локація	Локації пов’язані між собою переходами
Подія	N : 1	Локація	Подія прив’язана до конкретної локації
Персонаж	N : 1	Локація	Персонаж переміщується в межах активної локації

Ключовим зв’язком для поточної реалізації є відношення між сутностями «Персонаж» і «Локація»: при зміні активної локації система автоматично оновлює масиви waypoint-ів персонажа через компонент RoomStats, що є програмним відображенням цього зв’язку в архітектурі гри. Зв’язок «Локація-Ворог» та «Локація-Ігровий предмет» реалізовано через масиви enemySpawnpoints та itemSpawnpoints відповідно.

Для збереження динамічного стану гри між сесіями використовується серіалізація даних у формат JSON засобами Unity. Вбудований клас JsonUtility забезпечує перетворення C#-об’єктів у JSON-рядок та у зворотному напрямку без залежностей від зовнішніх бібліотек. Файл збереження розміщується у директорії Application.persistentDataPath, що гарантує доступність шляху запису на всіх підтримуваних платформах.

Структура JSON-файлу збереження відображає стан сутності «Гравець» та пов’язаних із нею динамічних даних. Основні поля файлу збереження наведено в таблиці 2.3.

Таблиця 2.3 – Структура JSON-файлу збереження

Поле JSON	Тип даних	Опис
playerId	string	Унікальний ідентифікатор збереження
currentLocationId	string	Ідентифікатор поточної активної локації
healthPoints	int	Поточний рівень здоров’я гравця
inventoryItems	string[]	Масив ідентифікаторів предметів в інвентарі
progressFlags	bool[]	Масив прапорців виконаних завдань/подій
sessionTime	float	Загальний час поточної ігрової сесії у секундах
lastSaveTimestamp	string	Дата та час останнього збереження

Для роботи із JSON-збереженням реалізовуватиметься окремий компонент SaveManager, що інкапсулює логіку читання та запису файлу збереження. Серіалізація виконуватиметься через JsonUtility.ToJson(), десеріалізація – через JsonUtility.FromJson<SaveData>(). Виклик збереження планується здійснювати автоматично при переході між локаціями та при закритті гри, що убезпечить від втрати прогресу.

Статичні конфігураційні дані гри – параметри ворогів, предметів та локацій – зберігаються у вигляді ScriptableObject-активів Unity. ScriptableObject є контейнером даних, що існує як окремий актив у проєкті та може бути призначений будь-якому ігровому об’єкту через інспектор без дублювання даних. Це відповідає принципу єдиної відповідальності та забезпечує зручне редагування балансу гри без зміни програмного коду.

Для кожного типу ігрового об’єкта створюється окремий клас ScriptableObject. Наприклад, клас EnemyData містить поля healthPoints, moveSpeed та detectionRadius, а клас ItemData – поля itemType, description та effectValue. Конкретні активи створюються у редакторі Unity як екземпляри відповідних класів із індивідуально налаштованими значеннями параметрів.

Перевагою такого підходу є повна незалежність конфігураційних даних від логіки гри: дизайнер може змінювати параметри балансу безпосередньо в інспекторі Unity без участі програміста. Крім того, один ScriptableObject-актив може бути одночасно призначений кільком ігровим об’єктам, що виключає дублювання даних у пам’яті.

Таким чином, модель даних розробленого прототипу поєднує два підходи до організації даних: ScriptableObject для статичних конфігураційних параметрів та JSON-серіалізацію для динамічного стану гри. Визначені сутності та зв’язки між ними відображають ключові ігрові елементи та забезпечують достатню гнучкість для реалізації запланованих розширень функціоналу – системи противників, предметів та збереження прогресу – без необхідності перегляду загальної структури даних.

РОЗДІЛ 3
РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ ЖАНРУ HORROR

3.1 Практична реалізація комп'ютерної гри

Розробка прототипу хорор-гри здійснювалася із використанням ігрового рушія Unity та мови програмування C#. В основу проєктного рішення покладено компонентно-орієнтований підхід, відповідно до якого кожен ігровий об'єкт формується із набору незалежних компонентів MonoBehaviour, що відповідають за окремі аспекти його поведінки. Такий підхід є стандартним для Unity і забезпечує повторне використання коду, зручність тестування та незалежне розширення окремих підсистем.

Програмне забезпечення прототипу складається з п'яти основних функціональних модулів: системи керування персонажем (PlayerMovements), системи керування камерою (CameraManager), системи взаємодії з локаціями (SelectObject), системи анімації (AnimationManager) та центрального сховища стану рівня (RoomStats). Взаємодія між модулями організована через об'єкт RoomStats, який виконує роль спільного контексту: після вибору нової локації до нього записуються актуальні масиви waypoint-ів, spawn-точок предметів та spawn-точок ворогів, а всі інші компоненти зчитують ці дані за потребою.

Центральним елементом архітектури є об'єкт керування рівнем (RoomStats), який зберігає посилання на поточну активну локацію, масив точок переміщення персонажа (playerWaypoints), масив точок генерації предметів (itemSpawnpoints) та масив точок появи ворогів (enemySpawnpoints). Інші програмні компоненти отримують необхідні дані через звернення до цього об'єкта, що забезпечує централізоване керування станом ігрового середовища та уникнення дублювання інформації між модулями.

Система анімації Unity Animator застосовується для керування візуальним відображенням стану персонажа та стану локацій. Перемикання між анімаційними станами здійснюється через цілочисельний параметр animState, значення якого встановлюється програмно відповідно до поточних дій

користувача або стану ігрового об’єкта. Система взаємодії користувача з ігровими локаціями реалізована за допомогою механізму Raycast, який дозволяє визначати ігрові об’єкти за позицією курсора миші на екрані.

Структуру програмного забезпечення спроєктовано з урахуванням подальшого розширення функціоналу. Система spawn-точок підготовлена для генерації ворогів і предметів у наступних ітераціях розробки. Додавання нових типів ігрових об’єктів не потребуватиме суттєвих змін у вже реалізованій архітектурі.

Система керування персонажем забезпечує автоматичне переміщення ігрового персонажа між локаціями після вибору кімнати користувачем, зміну орієнтації моделі залежно від напрямку руху та синхронізоване відтворення анімаційних станів.

Після вибору користувачем нової локації модуль SelectObject оновлює масив playerWaypoints у компоненті RoomStats та активує прапорець isMoving у компоненті PlayerMovements. Отримавши цей сигнал, система керування персонажем визначає цільову точку переміщення як перший елемент масиву waypoint-ів і виконує плавний рух до неї. Подальший рух виконується автоматично, без додаткового втручання користувача (рис. 3.1).

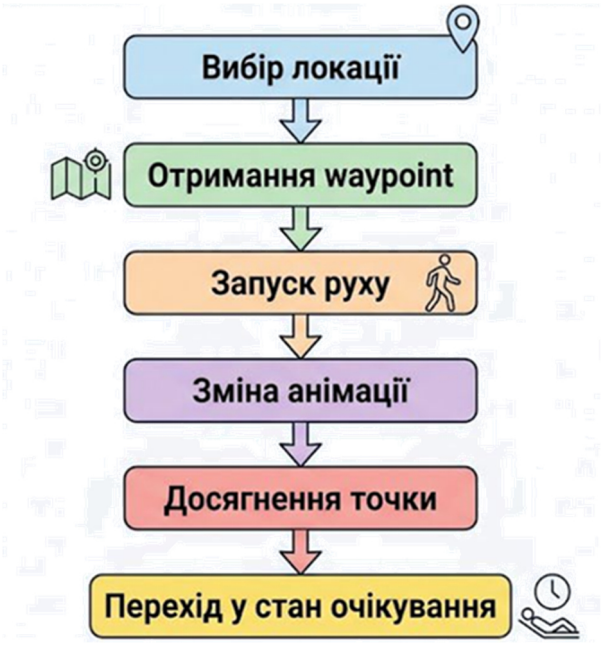

Рисунок 3.1 – Схема роботи руху персонажа

Для реалізації переміщення використовується метод Vector3.MoveTowards(), який дозволяє плавно переміщувати ігровий об’єкт до заданої точки з постійною швидкістю, незалежно від поточної частоти кадрів завдяки множнику Time.deltaTime. У методі MovePlayer() щокадрово обчислюється поточна відстань між персонажем та цільовим waypoint-ом; коли ця відстань досягає нуля, прапорець isMoving скидається та рух завершується (ліст. 3.1).

Лістинг 3.1 – Реалізація переміщення персонажа

```
private void MovePlayer()
{float distance = Vector3.Distance(
        transform.position,roomStats.playerWaypoints[0].position);
    float step = moveSpeed * Time.deltaTime;
    transform.position = Vector3.MoveTowards(
        transform.position, roomStats.playerWaypoints[0].position,step);
    if (distance == 0) {  isMoving = false;  }}
```

кінець лістингу 3.1

У результаті виконання коду (ліст. 3.1) персонаж щокадрово зміщується на крок, що визначається добутком швидкості та часу кадру. Після досягнення цільової точки рух автоматично зупиняється, що унеможливлює некоректне накопичення зміщень (рис. 3.2).

Рисунок 3.2 – Рух персонажа до waypoint

Для покращення візуальної достовірності руху реалізовано автоматичну зміну орієнтації 3D-моделі персонажа залежно від горизонтального напрямку переміщення. Метод FlipCharacter() порівнює координату персонажа з координатою цільового waypoint-а (ліст. 3.2).

Лістинг 3.2 – Реалізація зміни напрямку персонажа

```
private void FlipCharacter()
{
    if (transform.position.x <
        roomStats.playerWaypoints[0].position.x)
    {transform.rotation =
            Quaternion.Euler(0, 0, 0);}
    Else {transform.rotation = Quaternion.Euler(0, 180, 0); }}
```

кінець лістингу 3.2

Якщо ціль розташована правіше за поточну позицію персонажа, застосовується стандартна орієнтація моделі (кут обертання по осі Y дорівнює 0°). У протилежному разі модель повертається на 180° навколо вертикальної осі, що забезпечує природний вигляд руху у зворотному напрямку без заміни спрайту або додаткових ресурсів (рис. 3.3).

Рисунок 3.3 – Рух персонажа в зворотньому напрямку

Синхронізацію анімацій персонажа з його рухом забезпечує компонент AnimationManager, який отримує ціле число-ідентифікатор стану та передає його параметру animState контролера Animator (ліст. 3.3).

Лістинг 3.3 – Зміна активного стану анімації

```
public void ChangeAnimationClip(int i)
{
    _anim.SetInteger("animState", i);
}
```

кінець лістингу 3.3

Залежно від значення параметра animState контролер Animator переходить до відповідного стану: 0 – очікування, 1 – виділення активного об’єкта, 2 – повернення до нейтрального стану. Такий підхід дозволяє централізовано керувати всіма анімаційними переходами з будь-якого компонента гри через єдиний метод.

У результаті реалізації системи керування персонажем забезпечено: автоматизоване переміщення між локаціями з коректним зупиненням у цільовій точці, динамічне відзеркалення моделі відповідно до напрямку руху та синхронізацію переміщення з анімаційними станами через єдиний механізм параметра animState.

Система керування камерою забезпечує комфортну взаємодію користувача з ігровим середовищем через три основні функції: вільне ручне переміщення, автоматичне центрування на персонажі та масштабування поля зору.

Переміщення камери виконується при утриманні правої кнопки миші. Щокадрово зчитуються значення переміщення курсора по осях та Y через Input.GetAxis(), після чого ці значення помножуються на коефіцієнт швидкості та Time.deltaTime і застосовуються до позиції камери через Transform.Translate(). Для запобігання виходу камери за межі ігрового рівня після кожного переміщення виконується перевірка координат відносно заздалегідь визначених граничних значень (ліст. 3.4).

Лістинг 3.4 – Реалізація перевірки меж переміщення камери

```
private void StayInBoundaries()
{
    if (transform.position.x < boundaryXmin)
    {
        transform.position = new Vector3(
            boundaryXmin,
            transform.position.y,
            transform.position.z);
    }
    if (transform.position.x > boundaryXmax)    {
        transform.position = new Vector3(
            boundaryXmax,
            transform.position.y,
            transform.position.z);
    }
   }
```

кінець лістингу 3.4

Якщо поточна координата камери виходить за встановлені межі, вона примусово фіксується на граничному значенні. Межі задаються як серіалізовані поля компонента і налаштовуються у редакторі Unity без зміни коду (рис. 3.4). Аналогічна перевірка виконується і для координати Y, що обмежує область перегляду лише межами ігрового рівня.

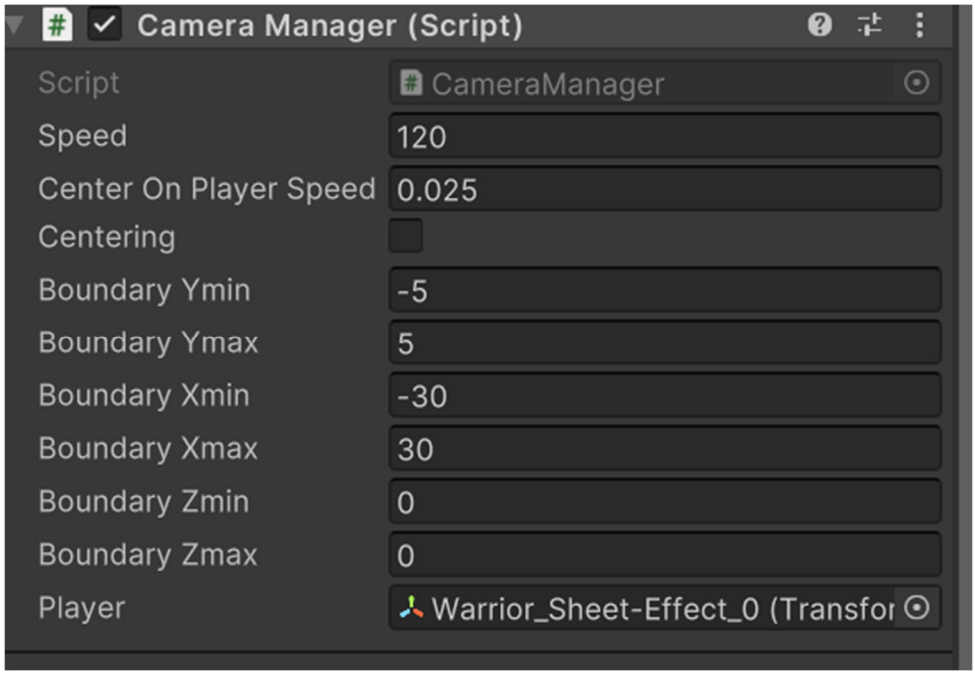

Рисунок 3.4 – Налаштування параметрів обмеження переміщення камери

Для швидкого повернення до поточного положення персонажа реалізовано механізм автоматичного центрування. Воно активується натисканням середньої кнопки миші та виконується протягом 0,5 секунди через корутину, яка встановлює прапорець centering на час виконання. Сам рух камери до персонажа здійснюється за допомогою функції Mathf.Lerp(), що забезпечує плавне наближення без різких стрибків (ліст. 3.5).

Лістинг 3.5 – Реалізація центрування камери на персонажі

```
private void CenterOnPlayer()
{
    Vector3 currentPosition = transform.position;
    float targetX = Mathf.Lerp(
        currentPosition.x,
        player.position.x,
        centerOnPlayerSpeed);
    float targetY = Mathf.Lerp(
        currentPosition.y,
        player.position.y,
        centerOnPlayerSpeed);
    transform.position =
        new Vector3(
            targetX,
            targetY,
            currentPosition.z);
}
```

кінець лістингу 3.5

Функція Lerp() щокадрово зменшує відстань між поточною позицією камери та позицією персонажа на задану частку (centerOnPlayerSpeed), що створює ефект плавного наближення. Координата Z при цьому залишається незмінною, оскільки керування глибиною огляду здійснюється окремим механізмом масштабування.

Масштабування поля зору реалізоване через зміну координати Z камери відповідно до значення прокрутки колеса миші, отриманого через Input.GetAxis("Mouse ScrollWheel") (ліст. 3.6). Оскільки рушій Unity використовує ортографічну проєкцію, зміщення камери по осі Z еквівалентне наближенню або віддаленню від ігрової сцени.

Лістинг 3.6 – Реалізація зміни масштабу камери

```
private void Zoom()
{ float scroll =
        Input.GetAxis("Mouse ScrollWheel");
    transform.position =
        new Vector3(
            transform.position.x,
            transform.position.y,
            transform.position.z + scroll);
}
```

кінець лістингу 3.6

Таким чином, система керування камерою надає користувачу повний контроль над областю перегляду: вільне переміщення з обмеженнями, зручне повернення до персонажа одним натисканням та масштабування для детального або загального огляду ігрового середовища.

Система взаємодії з локаціями є ключовим елементом ігрового процесу: вона обробляє кліки миші користувача, визначає вибрану кімнату, оновлює її візуальний стан та передає актуальні дані про активну локацію іншим підсистемам гри.

Для визначення вибраного об’єкта використовується механізм Raycast. Після натискання лівої кнопки миші формується промінь із позиції камери через координати курсора на екрані. Якщо промінь перетинає колайдер ігрового об’єкта, посилання на цей об’єкт зберігається для подальшої обробки (ліст. 3.7).

Лістинг 3.7 – Визначення вибраної локації за допомогою Raycast

```
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
RaycastHit hit;
if (Physics.Raycast(ray, out hit))
{raycastObjectHit = hit.transform.gameObject;}
```

кінець лістингу 3.7

Наведений фрагмент коду (ліст. 3.7) реалізує повний цикл обробки кліку: перш ніж застосувати виділення до нового об’єкта, система повертає попередню активну локацію до нейтрального анімаційного стану (параметр 2).

Після цього визначається новий hit-об’єкт, оновлюється інформація про кімнату та встановлюється прапорець isMoving, що запускає переміщення персонажа. Нарешті, на новому об’єкті активується анімація виділення (параметр 1).

Зміна анімаційного стану активної локації здійснюється через метод ChangeAnimation(), який делегує виклик до компонента AnimationManager відповідного ігрового об’єкта. Це дозволяє централізовано керувати зовнішнім виглядом локацій через єдиний інтерфейс (ліст. 3.8).

Лістинг 3.8 – Зміна стану вибраної локації

```
private void ChangeAnimation(int i)
{
    selectedObject_AnimationMgr
        .ChangeAnimationClip(i);
}
```

кінець лістингу 3.8

Після визначення активної кімнати виконується оновлення центрального сховища стану RoomStats. Метод UpdateRoomStatistics() записує посилання на батьківський ігровий об’єкт вибраної локації, після чого запускаються три окремі методи оновлення масивів точок (ліст. 3.9).

Лістинг 3.9 – Оновлення інформації про активну локацію

```
private void UpdateRoomStatistics()
{
    _roomStats._selectedObject =
        raycastObjectHit
        .transform.parent.gameObject;
}
```

кінець лістингу 3.9

Завдяки такому підходу всі необхідні дані про поточну кімнату автоматично синхронізуються після кожної взаємодії користувача з локаціями (рис. 3.5). Це уникає дублювання інформації між модулями та забезпечує єдину точку істини для стану ігрового рівня.

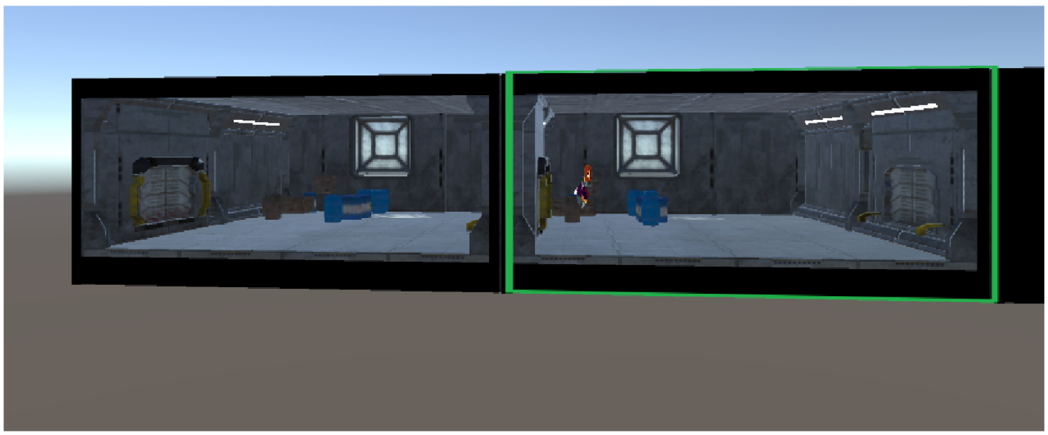

Рисунок 3.5 – Приклад вибору активної локації користувачем

Схема роботи виділення кімнат в вікні Unity Animator зобрпажена на рисунку 3.6.

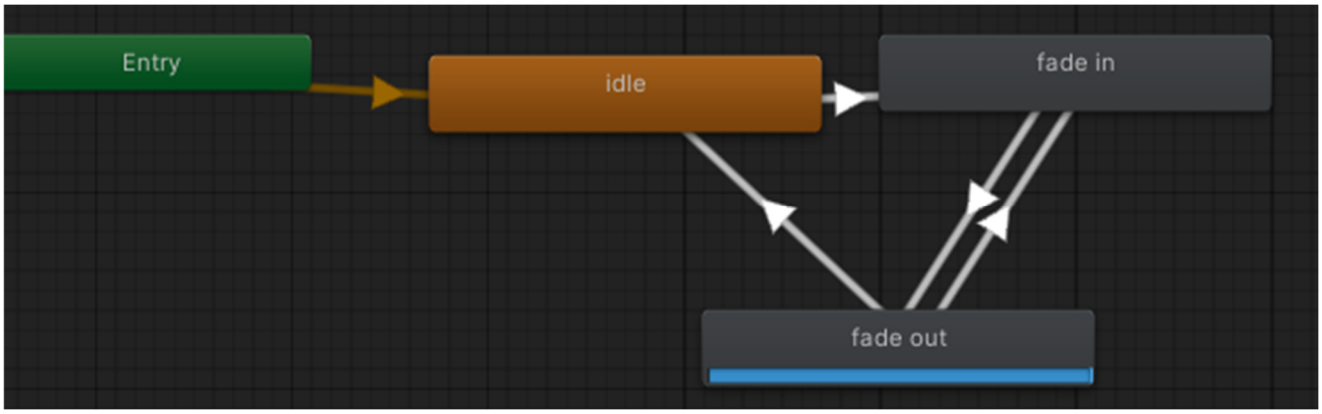

Рисунок 3.6 – Схема роботи виділення кімнат в вікні Unity Animator

Система генерації ігрових об’єктів відповідає за формування масивів spawn-точок у межах активної локації після кожної взаємодії користувача з кімнатою. Ці точки є спеціальними службовими об’єктами Unity із відповідними тегами, розміщеними як дочірні елементи ігрових локацій у ієрархії сцени.

Для визначення точок переміщення персонажа виконується пошук усіх об’єктів із тегом "PlayerWaypoints", що є дочірніми стосовно поточного активного об’єкта. LINQ-запит фільтрує результати за умовою IsChildOf() та перетворює їх у масив трансформів (ліст. 3.10).

Лістинг 3.10 – Формування масиву точок переміщення персонажа

```
_roomStats.playerWaypoints = GameObject
    .FindGameObjectsWithTag("PlayerWaypoints")
    .Where(wp => wp.transform.IsChildOf(
        _roomStats._selectedObject.transform))
    .Select(wp => wp.transform)
    .ToArray();
```

кінець лістингу 3.10

Перед формуванням нового масиву попередній масив явно очищується методом Array.Clear(), щоб уникнути використання застарілих даних від попередньо вибраної локації. Після виконання запиту отриманий масив використовується системою PlayerMovements для визначення цільової точки наступного переміщення персонажа.

Аналогічний підхід застосовується для формування масиву точок генерації предметів (ліст. 3.11) та масиву точок появи ворогів (ліст. 3.12).

Лістинг 3.11 – Формування масиву точок генерації предметів

```
_roomStats.itemSpawnpoints = GameObject
.FindGameObjectsWithTag("ItemSpawnPoints")
.Where(wp => wp.transform.IsChildOf(
_roomStats._selectedObject.transform))
.Select(wp => wp.transform) .ToArray();
```

кінець лістингу 3.11

Обидва масиви також очищуються перед наповненням новими даними відповідно до тегів "ItemSpawnPoints" та "EnemySpawnPoints".

Лістинг 3.12 – Формування масиву точок генерації ворогів

```
_roomStats.enemySpawnpoints = GameObject
.FindGameObjectsWithTag("EnemySpawnPoints")
.Where(wp => wp.transform.IsChildOf(
_roomStats._selectedObject.transform))
.Select(wp => wp.transform)
.ToArray();
```

кінець лістингу 3.12

На поточному етапі розробки сформовані масиви використовуються лише для збереження позицій потенційного розташування об’єктів. У наступних версіях проєкту ці дані будуть застосовані для автоматичної генерації предметів, статичних ворогів та мобільних противників безпосередньо в межах обраної локації (рис. 3.7).

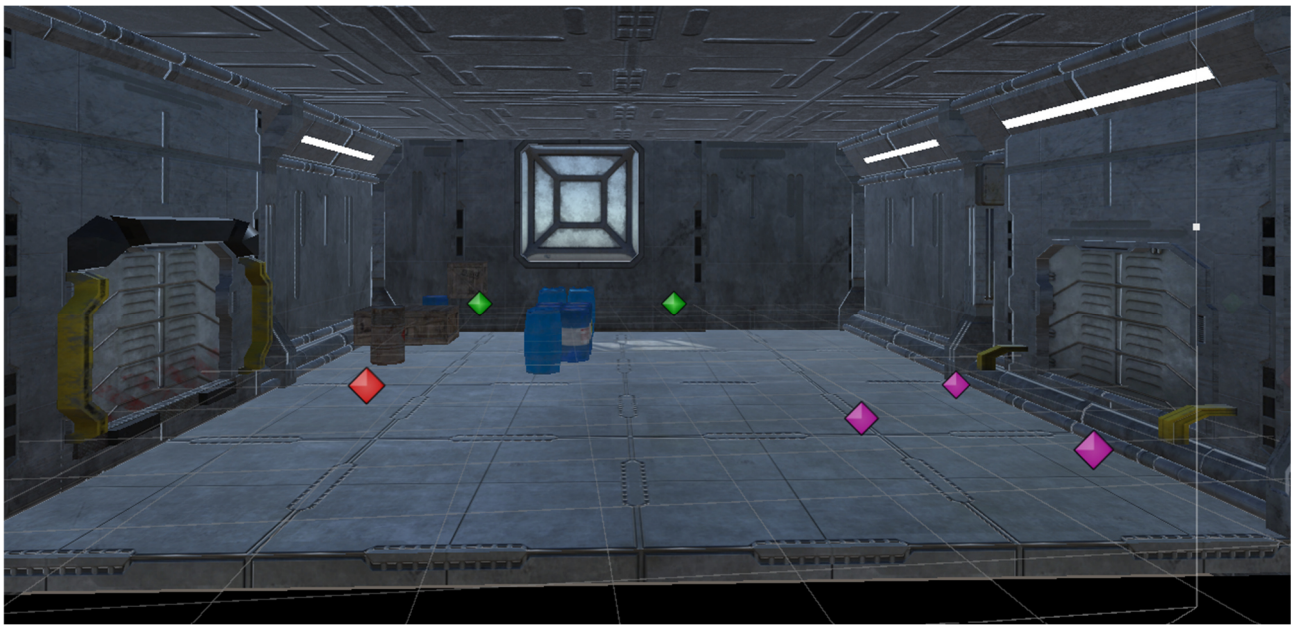

Рисунок 3.7 – Приклад розташування точок генерації об’єктів у середовищі Unity (червона – гравець, фіолетова – ворог, зелена – предмети)

Система візуальних ефектів інтерфейсу відповідає за дві функції: плавне затемнення ігрової сцени під час переходів між локаціями та виділення активної кімнати серед усіх елементів рівня. Обидві функції підвищують читабельність ігрового середовища та допомагають користувачу орієнтуватися у взаємодії з об’єктами.

Ефект затемнення реалізований через компонент CanvasGroup, що накладається поверх ігрової сцени у вигляді напівпрозорого шару UI. Метод FadeScreen() реалізований як корутина та плавно змінює значення параметра alpha від поточного до цільового протягом заданого часу (ліст. 3.13). Використання Mathf.Lerp() гарантує рівномірність переходу незалежно від частоти кадрів.

Лістинг 3.13 – Реалізація ефекту затемнення екрану

```
private IEnumerator FadeScreen(float duration, float targetAlpha)
{   float startAlpha = canvasGroup.alpha;
    float time = 0f;
    while (time < duration)
    { time += Time.deltaTime;
        canvasGroup.alpha = Mathf.Lerp(
            startAlpha,
            targetAlpha,
            time / duration);
        yield return null; }
    canvasGroup.alpha = targetAlpha; }
```

кінець лістингу 3.13

У наведеному фрагменті коду (ліст. 3.13) час виконання корутини та цільове значення прозорості передаються як аргументи, що дозволяє використовувати один і той самий метод як для затемнення (targetAlpha = 1), так і для появи сцени (targetAlpha = 0). Явне присвоєння canvasGroup.alpha = targetAlpha після виходу з циклу гарантує досягнення точного цільового значення.

Виділення активної локації виконується через зміну анімаційного стану об’єкта кімнати. Після вибору нової кімнати метод ChangeAnimation() передає ідентифікатор стану 1 до компонента AnimationManager вибраного об’єкта, а при знятті виділення – ідентифікатор 2 (ліст. 3.14). Це активує відповідний перехід в Unity Animator, що змінює зовнішній вигляд кімнати (підсвічування, зміна кольору або іншого візуального параметра) без зміни ігрової логіки.

Лістинг 3.14 – Активація візуального виділення локації

```
private void ChangeAnimation(int i)
{ selectedObject_AnimationMgr.ChangeAnimationClip(i); }
```

кінець лістингу 3.14

Система візуальних ефектів інтегрована з механізмом взаємодії з локаціями: затемнення та виділення застосовуються автоматично після кожного вибору нової кімнати. У результаті реалізації цього модуля забезпечено покращення U□-складової гри, підвищення інтуїтивності взаємодії та формування чіткого візуального фокусу на активних елементах сцени.

3.2 Тестування та налагодження системи

На завершальному етапі розробки прототипу проведено функціональне тестування у середовищі Unity шляхом запуску ігрової сцени та перевірки ключових сценаріїв взаємодії. Метою тестування є підтвердження коректної роботи кожного програмного модуля та правильності взаємодії між підсистемами гри в цілому.

Тестування здійснювалось безпосередньо у режимі Play редактора Unity Editor, що дозволяє перевіряти поведінку системи в режимі реального часу без необхідності збірки виконуваного файлу. Кожна підсистема перевірялась окремо, а потім у складі загального ігрового циклу. Для виявлення помилок використовувалась вбудована консоль Unity (Console window), яка відображає виключення, попередження та відлагоджувальні повідомлення у реальному часі.

Застосовувався метод чорного ящика: перевірялась відповідність вхідних дій користувача (кліки миші, прокручування) очікуваному результату (переміщення персонажа, зміна вигляду локацій, реакція камери). Для кожного модуля сформовано набір тест-кейсів, що охоплюють як стандартні сценарії використання, так і граничні випадки.

Окрім перевірки коректності роботи окремих модулів, тестування охоплювало інтеграційні сценарії, у яких декілька підсистем взаємодіяли одночасно. Зокрема, перевірялась послідовність «вибір локації → оновлення spawn-точок → запуск переміщення персонажа → зміна візуального стану кімнати», що є основним ігровим циклом прототипу. Коректне виконання цього ланцюжка підтверджує не лише правильність реалізації кожного компонента окремо, а й узгодженість їхньої взаємодії через спільне сховище стану RoomStats.Тест-кейси наведено в таблиці 3.1.

Таблиця 3.1 – Тест-кейси функціонального тестування прототипу

№	Тест-кейс	Очікуваний результат	Фактичний результат
1	Клік лівою кнопкою миші на кімнаті	Кімната виділяється; персонаж переміщується до waypoint; попередня кімната знімає виділення	Відповідає очікуванню
2	Клік на вже активній кімнаті	Стан не змінюється; повторне переміщення не запускається	Відповідає очікуванню
3	Переміщення камери (ПКМ + рух миші)	Камера плавно переміщується; не виходить за межі рівня	Відповідає очікуванню
4	Натискання середньої кнопки миші	Камера плавно центрується на персонажі протягом 0,5 с	Відповідає очікуванню
5	Прокручування колеса миші вперед/назад	Масштаб сцени збільшується/зменшується	Відповідає очікуванню
6	Вибір кімнати → перевірка масивів spawn-точок	Масиви playerWaypoints, itemSpawnpoints, enemySpawnpoints оновлюються лише для активної кімнати	Відповідає очікуванню
7	Перехід між кімнатами → ефект затемнення	Сцена плавно затемнюється та повертається до нормальної яскравості	Відповідає очікуванню

У процесі тестування виявлено та усунено одну проблему: при відсутності компонента AnimationManager на об’єкті, який потрапив під Raycast, виникав NullReferenceException. Помилка проявлялась при кліку на ігрових об’єктах, що не є локаціями (наприклад, на елементах фону сцени). Проблему усунено додаванням перевірки selectedObject_AnimationMgr != null перед зміною анімаційного стану (ліст. 3.7, рядок 4), що забезпечило стабільну роботу системи за будь-яких умов взаємодії користувача з об’єктами сцени.

Додатково виявлено, що масиви spawn-точок не очищувались коректно при повторному виборі тієї ж локації, якщо розмір масиву дорівнював нулю. Виправлення полягало у перевірці довжини масиву перед викликом Array.Clear(), що запобігло виникненню IndexOutOfRangeException у відповідних методах оновлення точок.

За результатами тестування підтверджено працездатність та стабільність усіх основних підсистем прототипу: переміщення персонажа, керування камерою, взаємодії з локаціями, оновлення spawn-точок та візуальних ефектів. Всі виявлені дефекти усунено до фінальної версії прототипу. Мінімальні та рекомендовані системні вимоги для запуску програмного продукту наведено в таблиці 3.2.

Таблиця 3.2 – Мінімальні та рекомендовані системні вимоги

Компонент	Мінімальні вимоги	Рекомендовані вимоги
Операційна система	Windows 10 64-bit / macOS 10.14	Windows 11 64-bit / macOS 13
Процесор	Intel Core i3 2,0 ГГц або аналог	Intel Core i5 3,0 ГГц або аналог
Оперативна пам’ять	4 ГБ RAM	8 ГБ RAM
Відеокарта	Intel HD Graphics 4000 або Direct□ 11	NVIDIA GT□ 1050 / AMD R□ 570
Дискова пам’ять	2 ГБ вільного місця	5 ГБ вільного місця
Unity	Unity 2022.3 LTS	Unity 2022.3 LTS або новіша

Наведені системні вимоги відповідають можливостям середнього сучасного комп’ютера, що підтверджує доступність розробленого прототипу для широкого кола користувачів. Використання Unity 2022.3 LTS як обов’язкової умови забезпечує стабільність середовища виконання та сумісність із усіма реалізованими програмними компонентами

3.3 Інструкція користувача з інсталяції та використання програмного продукту

Перед початком роботи необхідно переконатися, що комп’ютер відповідає мінімальним системним вимогам, наведеним у таблиці 3.2. Для запуску прототипу хорор-гри необхідно встановити ігровий рушій Unity версії 2022.3 LTS або новішої. Рекомендується використовувати саме LTS-версію, оскільки вона є найбільш стабільною та підтримується протягом тривалого часу. Unity Hub – менеджер версій рушія – доступний для завантаження за адресою https://unity.com/download.

Порядок встановлення та відкриття проєкту:

– Завантажити та встановити Unity Hub з офіційного сайту. Після встановлення запустити Unity Hub та увійти в обліковий запис Unity ID (або зареєструватися безкоштовно).

– У Unity Hub обрати вкладку «Installs» та натиснути «Install Editor».

Обрати версію Unity 2022.3 LTS. У переліку модулів позначити «Windows Build Support (IL2CPP)» для Windows або «Mac Build Support (IL2CPP)» для macOS. Натиснути «Install» та дочекатися завершення.

– Перейти до вкладки «Projects» та натиснути «Open» → «Add project from disk». У діалоговому вікні вказати шлях до папки з проєктом (папка, що містить файл Assets, ProjectSettings тощо).

– Unity Hub відкриє проєкт у відповідній версії рушія. При першому відкритті відбудеться компіляція скриптів та імпорт асетів, що може тривати 2–5 хвилин залежно від продуктивності комп'ютера.

– Після завершення завантаження у вікні Unity Editor перейти до вкладки «Project» у нижній панелі та відкрити Assets → Scenes → MainScene подвійним кліком.

– Натиснути кнопку «Play» у верхній панелі редактора. Гра запуститься у режимі тестування безпосередньо в Unity Editor.

Після запуску гри у режимі Play користувач бачить тривимірну ігрову сцену, що складається з набору кімнат-локацій та ігрового персонажа. Кімнати розміщені на сцені та мають видимі межі. Персонаж за замовчуванням знаходиться у початковій позиції та не переміщується до отримання першої команди. У верхній частині екрана відображається поле зору камери, яким можна керувати незалежно від положення персонажа.

Активна (вибрана) кімната візуально відрізняється від інших: вона підсвічується або змінює свій вигляд відповідно до налаштувань анімаційного контролера. Це дозволяє користувачу завжди чітко бачити, яка локація є поточною метою переміщення персонажа. При виборі нової кімнати попередня автоматично повертається до нейтрального стану.

Для взаємодії з грою використовується мишка. Усі дії виконуються за допомогою трьох кнопок та колеса прокручування. Клавіатура на поточному етапі розробки не задіяна. Основні елементи керування наведено в таблиці 3.3.

Таблиця 3.3 – Елементи керування

Дія	Елемент керування
Вибір кімнати для переходу персонажа	Ліва кнопка миші (клік на кімнаті)
Переміщення камери по сцені	Права кнопка миші (утримання + рух)
Центрування камери на персонажі	Середня кнопка миші (натискання)
Наближення / віддалення (zoom)	Колесо прокручування миші

Щоб відправити персонажа до кімнати, необхідно навести курсор безпосередньо на об’єкт локації та клікнути лівою кнопкою миші. Якщо клік потрапив на правильний об’єкт, кімната підсвітиться, а персонаж автоматично почне рух до відповідного waypoint-а. Якщо ж клік потрапив у порожню частину сцени або на об’єкт без компонента AnimationManager, жодних змін не відбудеться – система обробляє такі випадки коректно завдяки реалізованим перевіркам.

Для зручного огляду ігрового рівня слід утримувати праву кнопку миші та переміщувати курсор у потрібному напрямку – камера рухатиметься відповідно. При наближенні до меж рівня камера автоматично зупиниться, не виходячи за встановлені обмеження. Якщо персонаж вийшов із поля зору, достатньо натиснути середню кнопку миші – камера плавно повернеться до поточного положення персонажа протягом приблизно 0,5 секунди. Масштаб огляду регулюється колесом прокручування: прокрутка вперед наближує сцену, назад – віддаляє.

У режимі редактора Unity для зупинки гри достатньо повторно натиснути кнопку «Play» у верхній панелі, що повертає рушій до стану редагування. Важливо пам’ятати, що будь-які зміни, внесені під час режиму Play (зміна значень компонентів у інспекторі тощо), не зберігаються після зупинки – це стандартна поведінка Unity Editor.

Для створення самостійного виконуваного файлу гри без необхідності Unity Editor потрібно виконати збірку (Build). Для цього слід відкрити меню File → Build Settings, обрати цільову платформу «PC, Mac & Linux Standalone», у полі «Target Platform» вказати операційну систему (Windows або macOS),

натиснути «Switch Platform» для зміни платформи, а потім «Build» або «Build And Run» для негайного запуску після збірки. У діалоговому вікні вказати папку для збереження файлів збірки та дочекатися завершення процесу. Готовий виконуваний файл можна запускати на будь-якому комп’ютері, що відповідає мінімальним системним вимогам, без встановлення Unity.

У разі збірки окремого виконуваного файлу гру можна закрити стандартним способом – комбінацією клавіш Alt+F4 або Command+Q, або через меню закриття вікна операційної системи.

РОЗДІЛ 4
СПЕЦІАЛЬНІ РОЗРАХУНКИ

Розроблений прототип хорор-гри реалізує базову архітектуру та ключові механіки взаємодії користувача з ігровим середовищем. Модульна структура програмного забезпечення, побудована на компонентно-орієнтованому підході Unity, створює технічну основу для поетапного розширення функціоналу без необхідності кардинальної перебудови вже реалізованих підсистем. У цьому розділі розглядаються основні напрямки подальшого розвитку проєкту та оцінюється їхня технічна реалізовуваність у межах існуючої архітектури.

Поточний стан прототипу характеризується повністю функціональними механіками вибору локацій, переміщення персонажа та керування камерою. Водночас ряд ігрових елементів реалізовано лише на рівні інфраструктури: масиви spawn-точок для ворогів і предметів формуються, але відповідні об’єкти ще не генеруються. Це свідоме архітектурне рішення, що дозволяє поступово нарощувати функціонал без переписування існуючого коду. Основні напрямки розвитку наведено в таблиці 4.1.

Таблиця 4.1 – Основні напрямки розвитку прототипу

Напрямок	Поточний стан	Планований результат
Система противників	Spawn-точки сформовано; логіка ворогів відсутня	Два типи ворогів із різними алгоритмами поведінки
Система предметів	Spawn-точки сформовано; генерація відсутня	Підбирання предметів, інвентар, вплив на ігровий процес
Візуальне оформлення	Базові текстури, без постобробки	URP-освітлення, Post Processing, єдиний хорор-стиль
Звуковий супровід	Відсутній	Атмосферні звуки, ефекти взаємодії, аудіотригери
Збірка (Build)	Лише режим Play у редакторі	Виконуваний файл для Windows / macOS

Одним із пріоритетних напрямків розвитку є впровадження повноцінної системи ворогів. На поточному етапі система spawn-точок вже підготовлена до генерації ворожих об’єктів – масив enemySpawnpoints формується автоматично після вибору кожної локації. Це дозволяє реалізувати появу противників без змін у вже існуючому коді взаємодії з локаціями.

Планується реалізація двох типів ворогів, що відрізняються поведінковими алгоритмами та рівнем загрози для гравця. Порівняльну характеристику типів ворогів наведено в таблиці 4.2.

Таблиця 4.2 – Характеристика типів противників

Параметр	Тип 1: Статичний	Тип 2: Мобільний
Переміщення	Відсутнє	Патрулювання між waypoint-ами
Реакція на гравця	Відсутня	Зміна стану на переслідування
Алгоритм поведінки	Розміщення у spawn-точці	FSM (патруль / переслідування / атака)
Компонент Unity	Instantiate() у spawn-точці	NavMeshAgent + FSM-контролер
Складність реалізації	Низька	Середня

Перший тип – статичні противники – розміщуються у визначених точках локації та виконують роль базових перешкод. Їх поява реалізується через виклик методу Instantiate() Unity з передачею координат з масиву enemySpawnpoints. Другий тип – мобільні противники з базовою системою патрулювання. Для їх переміщення планується використання компонента NavMeshAgent із вбудованої системи навігації Unity NavMesh, що дозволяє автоматично прокладати маршрут між точками з урахуванням геометрії рівня.

Для керування поведінкою мобільних противників доцільно застосувати патерн «Скінченний автомат станів» (Finite State Machine, FSM). Основні стани: патрулювання, переслідування та атака. Переходи між станами визначатимуться відстанню до гравця та іншими ігровими умовами. Такий підхід є стандартним для реалізації штучного інтелекту в іграх і добре інтегрується з компонентною архітектурою Unity [16].

Другим напрямком масштабування є впровадження системи ігрових предметів. Аналогічно до системи ворогів, масив itemSpawnpoints вже формується після вибору кожної локації, що забезпечує готову інфраструктуру для розміщення предметів у сцені.

Предмети можуть виконувати різні функціональні ролі залежно від потреб ігрового дизайну. Основні категорії ігрових предметів наведено в таблиці 4.3.

Таблиця 4.3 – Категорії ігрових предметів

Категорія	Приклади	Вплив на ігровий процес
Ресурси виживання	Аптечки, батарейки для ліхтаря	Відновлення здоров'я, продовження часу дії ліхтаря
Ключові об'єкти	Ключі, картки доступу, коди	Розблокування нових локацій або механізмів
Сюжетні елементи	Нотатки, щоденники, фото	Розкриття лору та сюжету гри
Інструменти	Ліхтар, каміння, пастки	Тактичні можливості взаємодії з ворогами

Взаємодія гравця з предметом реалізовуватиметься через вже існуючий механізм Raycast – клік на об'єкті предмета запускатиме відповідну логіку підбирання або використання. Для управління інвентарем планується реалізація окремого компонента InventoryManager, що зберігатиме список підібраних предметів та забезпечуватиме доступ до них з інших підсистем гри. Відображення інвентаря користувачу здійснюватиметься через UI-панель на основі компонентів Canvas та ScrollView Unity.

Важливим напрямком розвитку є підвищення якості візуального та звукового оформлення гри для формування повноцінної атмосфери хорор-жанру. На поточному етапі прототип використовує базові текстури без постобробки та не має звукового супроводу. Заплановані покращення охоплюють дві взаємопов'язані складові.

У частині візуального оформлення планується перехід на Unity Universal Render Pipeline (URP) із впровадженням системи динамічного освітлення. Точкові та спрямовані джерела світла з параметрами кольору та інтенсивності дозволять формувати характерну атмосферу з чергуванням освітлених і затемнених зон. Додатково передбачається застосування стеку постобробки зображення (Post Processing Stack) з ефектами Bloom, Vignette та Color Grading, що підсилять відчуття напруження й невизначеності. Заміна тимчасових текстур на фінальні графічні ресурси, розроблені відповідно до єдиного хорор-стилю, завершить формування цілісного візуального образу гри.

У частині звукового оформлення планується інтеграція компонента AudioSource Unity для відтворення фонових атмосферних звуків та локальних

звукових ефектів взаємодії з об’єктами. Для централізованого керування аудіосистемою доцільно реалізувати окремий компонент AudioManager за патерном Singleton, що забезпечить уникнення конфліктів між одночасно відтворюваними звуками. Спеціальні аудіальні тригери, пов’язані з появою або наближенням противників, значно підвищать рівень емоційного занурення гравця та підсилять психологічний вплив гри.

Реалізована архітектура прототипу демонструє достатній рівень масштабованості для підтримки всіх запланованих розширень. Централізоване сховище стану RoomStats дозволяє легко додавати нові типи даних про локацію без модифікації інших компонентів: достатньо оголосити новий масив у класі та відповідний метод оновлення в SelectObject. Система spawn-точок на основі тегів Unity є незалежною від конкретних типів об’єктів, що генеруються, тому додавання нових категорій ігрових об’єктів зводиться до введення нового тегу та масиву у RoomStats без зміни логіки взаємодії з локаціями.

Для об’єктивної оцінки стану розробки виконано аналіз реалізованого функціоналу відносно запланованого повного обсягу гри. Результати наведено в таблиці 4.4.

Таблиця 4.4 – Ступінь готовності підсистем

Підсистема	**Статус**	**Готовність, %**
Керування персонажем (переміщення, анімація)	Реалізовано повністю	100
Керування камерою (рух, zoom, центрування)	Реалізовано повністю	100
Взаємодія з локаціями (Raycast, виділення)	Реалізовано повністю	100
Система spawn-точок (інфраструктура)	Реалізовано повністю	100
Генерація ворогів та їхня поведінка	Заплановано	0
Система предметів та інвентар	Заплановано	0
Візуальне оформлення (URP, постобробка)	Частково (базові текстури)	20
Звуковий супровід	Заплановано	0

Єдиним потенційним обмеженням продуктивності є використання методу FindGameObjectsWithTag() при кожній зміні локації, що може призводити до незначного зниження продуктивності при великій кількості об’єктів на сцені. Для усунення цього обмеження у подальших версіях доцільно кешувати

посилання на об’єкти під час ініціалізації сцени замість їхнього пошуку в режимі реального часу. Такий підхід є стандартною оптимізацією для ігор на рушії Unity і не потребує змін у загальній архітектурі проєкту.

Таким чином, розроблений прототип може розглядатися як стабільна технологічна основа для створення повноцінного ігрового продукту в жанрі хорор. Його архітектура забезпечує можливість поетапного нарощування функціоналу – від реалізації системи противників і предметів до повноцінного аудіовізуального оформлення – зі збереженням стабільності вже реалізованих підсистем та без необхідності суттєвого рефакторингу існуючого коду.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розроблено прототип комп’ютерної гри жанру horror на базі ігрового рушія Unity з використанням мови програмування C#. Поставлену мету досягнуто в повному обсязі: створено функціональний програмний продукт, що реалізує базові механіки взаємодії користувача з ігровим середовищем через систему керування локаціями.

У першому розділі проведено аналіз предметної області та сучасного стану розробки ігор жанру horror. Розглянуто особливості жанру, визначено ключові підходи до реалізації атмосфери напруження та взаємодії гравця з ігровим середовищем. На основі порівняльного аналізу аналогів – Outlast, Amnesia: The Dark Descent, Five Nights at Freddy’s та This War of Mine – обґрунтовано вибір механіки непрямого керування персонажем через вибір локацій як оптимального підходу з точки зору балансу між складністю реалізації та ігровою цінністю. Як основний інструмент розробки обрано Unity 2022.3 LTS у поєднанні з мовою C#.

У другому розділі сформовано специфікацію вимог до програмного продукту, розроблено функціонально-структурну схему системи та визначено модель даних. Архітектуру побудовано на компонентно-орієнтованому підході Unity, що забезпечує незалежність модулів та зручність їхнього подальшого розширення. Центральним елементом архітектури визначено компонент RoomStats як спільне сховище стану рівня.

У третьому розділі здійснено практичну реалізацію прототипу. Розроблено та реалізовано п’ять основних програмних підсистем: систему керування персонажем із автоматичним переміщенням між waypoint-ами та синхронізацією анімаційних станів, систему керування камерою з підтримкою вільного переміщення, масштабування та автоматичного центрування, систему взаємодії з локаціями на основі механізму Raycast, систему визначення та формування масивів spawn-точок для персонажа, предметів і ворогів, а також

систему візуальних ефектів інтерфейсу. Проведено функціональне тестування методом чорного ящика, підтверджено коректність роботи всіх підсистем і усунено виявлені дефекти.

У четвертому розділі виконано спеціальні розрахунки, що охоплюють аналіз перспектив розвитку та масштабування програмного продукту. Визначено основні напрями подальшого розширення: реалізацію системи противників двох типів на основі патерну FSM та компонента NavMeshAgent, впровадження системи ігрових предметів та інвентаря, вдосконалення візуального оформлення засобами Unity URP та Post Processing Stack, а також розробку повноцінного звукового супроводу. Проведено оцінку масштабованості реалізованої архітектури та підтверджено її придатність для підтримки запланованих розширень без суттєвого рефакторингу існуючого коду.

Практичне значення роботи полягає в тому, що розроблений прототип може бути використаний як технологічна основа для створення повноцінного ігрового продукту в жанрі horror. Реалізована модульна архітектура на базі Unity дозволяє поступово нарощувати функціонал шляхом додавання нових компонентів без зміни вже реалізованих підсистем, що підтверджує практичну цінність обраного архітектурного підходу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Scripting Reference. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference (дата звернення: 11.04.2026).

2. Unreal Engine 5.7 Documentation. Epic Games. URL: https://docs.unrealengine.com (дата звернення: 11.04.2026).

3. User Manual. Unity Documentation. URL: https://docs.unity3d.com (дата звернення: 11.04.2026).

4. Schell J. The art of game design. A book of lenses. ArtHuss. (дата звернення: 11.04.2026).

5. Adams E. Fundamentals of game design. 2nd ed. New Riders, 2010. 696 p. (дата звернення: 11.04.2026).

6. Tekinbas K. S. Rules of play: game design fundamentals. MIT Press, 2003. 688 p. (дата звернення: 11.04.2026).

7. Rogers S. Level up! The guide to great video game design. John Wiley and Sons Ltd, 2014. 560 c. (дата звернення: 11.04.2026).

8. Gregory J. Game engine architecture. 2nd ed. Jason Gregory : CRC Press, 2017. (дата звернення: 11.04.2026).

9. Fallout Shelter. Bethesda Game Studios. URL: https://fallout.bethesda.net/en/games/fallout-shelter (дата звернення: 11.04.2026)

10. This War of Mine. Wikipedia. URL: https://uk.wikipedia.org/wiki/This_War_of_Mine (дата звернення: 11.04.2026).

11.Outlast. Red Barrels. URL: https://redbarrelsgames.com/games/outlast (дата звернення: 11.04.2026)

12. Amnesia: The Dark Descent. Frictional Games. URL: https://www.frictionalgames.com/games/amnesia (дата звернення: 11.04.2026)

13. GitSCM. git. URL: https://git-scm.com/ (дата звернення: 01.05.2026).

14. Sommerville I. Software engineering. 10th ed. Pearson, 2016. 796 p. (дата звернення: 11.04.2026).

15. Systems and software engineering. iso. URL: https://www.iso.org/standard/78177.html (дата звернення: 11.04.2026).

16. Nystrom R. Game Programming Patterns. Game Programming Patterns. URL: https://gameprogrammingpatterns.com (дата звернення: 11.04.2026).

ДОДАТКИ

Додаток А

```
public class CameraManager : MonoBehaviour
{
    [SerializeField] private float speed = 120f;
    [SerializeField] private float centerOnPlayerSpeed = 0.025f;
    [SerializeField] private bool centering = false;
    [SerializeField] private float boundaryYmin;
    [SerializeField] private float boundaryYmax;
    [SerializeField] private float boundaryXmin;
    [SerializeField] private float boundaryXmax;
    [SerializeField] private float boundaryZmin;
    [SerializeField] private float boundaryZmax;
    [SerializeField] private Transform player;
    private void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player").transform;
    }
    private void Update()
    {
        if (Input.GetMouseButton(1))
        {
            float moveX = Input.GetAxis("Mouse X") * speed *
Time.deltaTime;
            float moveY = Input.GetAxis("Mouse Y") * speed *
Time.deltaTime;
            transform.Translate(moveX, moveY, 0);
            StayInBoundaries();
        }
        if (Input.GetMouseButtonDown(2))
        {
            StartCoroutine(CenterOnPlayerTransition(0.5f));
        }
        if (centering)
        {
            CenterOnPlayer();
        }
        Zoom();

    }
    private void StayInBoundaries()
    {
        if (transform.position.x < boundaryXmin)
        {
            transform.position = new Vector3(boundaryXmin,
this.transform.position.y, this.transform.position.z);
        }
        if (transform.position.x > boundaryXmax)
        {
```

```
            transform.position = new Vector3(boundaryXmax,
this.transform.position.y, this.transform.position.z);
        }
        if (transform.position.y < boundaryYmin)
        {
            transform.position = new Vector3(this.transform.position.x,
boundaryYmin, this.transform.position.z);
        }
        if (transform.position.y > boundaryYmax)
        {
            transform.position = new Vector3(this.transform.position.x,
boundaryYmax, this.transform.position.z);
        }
    }
    private void CenterOnPlayer()
    {
        if (player != null)
        {
            Vector3 currentPosition = transform.position;
            float targetX = Mathf.Lerp(currentPosition.x,
player.position.x, centerOnPlayerSpeed);
            float targetY = Mathf.Lerp(currentPosition.y,
player.position.y, centerOnPlayerSpeed);
            transform.position = new Vector3(targetX, targetY,
currentPosition.z);
        }

    }
    private IEnumerator CenterOnPlayerTransition(float _seconds)
    {
        centering = true;
        yield return new WaitForSeconds(_seconds);
        centering = false;
    }

    private void Zoom()
    {
        float scroll = Input.GetAxis("Mouse ScrollWheel");

        if(Input.GetAxis("Mouse ScrollWheel") > 0)
        {
            transform.position = new Vector3(transform.position.x,
transform.position.y, transform.position.z + scroll);
        }
        if (Input.GetAxis("Mouse ScrollWheel") < 0)
        {
            transform.position = new Vector3(transform.position.x,
transform.position.y, transform.position.z + scroll);
        }
    }
}
```

Додаток Б

```
public class SelectObject : MonoBehaviour
{
    /*
     * вибір обєкта
     * зміна анімації вибраного обєкту
     */
    [SerializeField]
    private GameObject raycastObjectHit;


    [SerializeField]
    private AnimationManager selectedObject_AnimationMgr;

    [Header("Script References")]
    [SerializeField] private RoomStats _roomStats;
    [SerializeField] private PlayerMovements playerMovements;

    private void Start()
    {
       if(_roomStats == null)
        {
            _roomStats = GetComponent<RoomStats>();
        }
        playerMovements =
GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerMovements>(
);
    }
    private void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit))
            {
                if (selectedObject_AnimationMgr != null)
                {
                    ChangeAnimation(2);
                }
                raycastObjectHit = hit.transform.gameObject;
                UpdateRoomStatistics();
                playerMovements.isMoving = true;
            }
            selectedObject_AnimationMgr =
raycastObjectHit.GetComponent<AnimationManager>();
            ChangeAnimation(1);
        }
    }
    //змінюємо анімацію вибраного обєкта
```

```
    private void ChangeAnimation(int i)
    {
        selectedObject_AnimationMgr.ChangeAnimationClip(i);
    }

    private void UpdateRoomStatistics()
    {
        //
        _roomStats._selectedObject =
raycastObjectHit.transform.parent.gameObject;
        AjustPlayerWaypointArray();
        AjustItemSpawnpointArray();
        AjustEnemySpawnpointArray();
    }
    private void AjustPlayerWaypointArray()
    {
        //очищуємо масив точок спавну
        System.Array.Clear(_roomStats.playerWaypoints, 0,
_roomStats.playerWaypoints.Length);
        //заповнюємо масив точок спавну
        _roomStats.playerWaypoints = GameObject
            .FindGameObjectsWithTag("PlayerWaypoints")
            .Where(wp =>
wp.transform.IsChildOf(_roomStats._selectedObject.transform))
            .Select(wp => wp.transform)
            .ToArray();
    }
    private void AjustItemSpawnpointArray()
    {
        //очищуємо масив точок спавну
        System.Array.Clear(_roomStats.itemSpawnpoints, 0,
_roomStats.itemSpawnpoints.Length);
        //заповнюємо масив точок спавну
        _roomStats.itemSpawnpoints = GameObject
            .FindGameObjectsWithTag("ItemSpawnPoints")
            .Where(wp =>
wp.transform.IsChildOf(_roomStats._selectedObject.transform))
            .Select(wp => wp.transform)
            .ToArray();
    }
    private void AjustEnemySpawnpointArray()
    {
        //очищуємо масив точок спавну ворогів, щоб уникнути залишкових
даних від попередньо вибраного обєкта
        System.Array.Clear(_roomStats.enemySpawnpoints, 0,
_roomStats.enemySpawnpoints.Length);
        //заповнюємо масив точок спавну ворогів, які є дочірніми обєктами
вибраного обєкта
        _roomStats.enemySpawnpoints = GameObject
            .FindGameObjectsWithTag("EnemySpawnPoints")
```

```
            .Where(wp =>
wp.transform.IsChildOf(_roomStats._selectedObject.transform))
            .Select(wp => wp.transform)
            .ToArray();
  }
}
```

Додаток В

```
public class RoomStats : MonoBehaviour
{
    public GameObject _selectedObject;
    public Transform[] playerWaypoints;
    public Transform[] itemSpawnpoints;
    public Transform[] enemySpawnpoints;
}
```

Додаток Г

```
public class AnimationManager : MonoBehaviour
{
    /* *reference animator
     * *change animation clip
     * *triger the animation clip */
    Animator _anim;

    [SerializeField]
    private int startingClip;

    private void Start()
    {
        _anim = GetComponent<Animator>();
        _anim.SetInteger("animState", startingClip);
    }

    public void ChangeAnimationClip(int i)
    {
        _anim.SetInteger("animState", i);
    }
```