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

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА ГРИ ПІД НАЗВОЮ «КАТАКОМБИ» В НИЗЬКОПОЛІГОНАЛЬНІЙ ГРАФІЦІ З ВИКОРИСТАННЯМ UNITY

DEVELOPMENT OF THE GAME «CATACOMBS» IN LOW-POLYGONAL GRAPHICS USING UNITY

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-42
Радишевський Сергій Віталійович

Керівник:
Гордєєв Олександр Олександрович

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ

Завідувач кафедри
_____
«20» 12 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Радишевському Сергію Вітальовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *«Розробка гри під назвою "Катакомби" в низькополігональній графіці з використанням Unity»*
Керівник роботи: *д.т.н., професор Гордєєв О.О.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р.* № *474/01-02*

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*

3. Вихідні дані до роботи: *Unity, Visual Studio 2022 з C#, Git, Blender 4.2, інструменти оптимізації: Addressable, Profiler, Post-processing tools Unity, методичні вказівки до виконання кваліфікаційної роботи бакалавра*

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *аналіз предметної області, визначення вимог до простої гри, архітектура проєкту в Unity, розробка базових low-poly моделей, реалізація простих геймплейних механік, створення мінімального інтерфейсу та звуку, організація збереження процесу*

5. Перелік графічного матеріалу: *10 таблиць, 3 рисунки*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	Гордєєв О. О.		
Специфікація вимог до розробленої системи	Гордєєв О. О.		
Розробка об'єкта проектування	Гордєєв О. О.		
Нормоконтроль	Вознюк А. В.		
Гарант ОП	Ліщина Н. М.		
Показник запозичень тексту	0,91 %		
Академічна доброчесність	Гордєєв О. О		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 04.02.2025 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 01.03.2025 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування	до 29.03.2025 р.	
5	Практична реалізація об'єкта проектування	до 26.04.2025 р.	
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	

Здобувач вищої освіти _____ Сергій РАДИШЕВСЬКИЙ

Керівник кваліфікаційної роботи _____ Олександр ГОРДЄЄВ

АНОТАЦІЯ

Радишевський С. В. Розробка гри під назвою «Катакомби» в низькополігональній графіці з використанням Unity. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел.

У першому розділі проведено аналіз особливостей low-poly графіки, її естетичних та технічних переваг, а також приклади застосування у сучасних інді-проєктах. Визначено актуальність використання Unity для навчальних цілей та обґрунтовано вибір тематики.

Другий розділ присвячено архітектурі гри. Описано структуру коду, реалізацію систем бойових дій, генерацію рівнів методом BSP, управління ресурсами через Addressables, а також створення low-poly моделей у Blender. Здійснено інтеграцію інтерфейсу користувача, звукового оформлення та системи збереження прогресу за допомогою JSON-серіалізації з шифруванням.

У третьому розділі виконано тестування продуктивності гри, проаналізовано вплив полігональності на FPS, зібрано телеметрію користувача та визначено напрями для подальшої оптимізації та масштабування. Показано, як завдяки modular-based архітектурі та прозорій логіці коду проєкт може слугувати навчальним прикладом для студентів, які вивчають Unity.

У висновках узагальнено результати реалізації гри «Катакомби», підтверджено досягнення поставленої мети та виконання всіх завдань. Зроблено висновки щодо ефективності застосування low-poly стилю для освітніх проєктів, можливостей рушія Unity, перспектив удосконалення проєкту та його потенціалу як навчального шаблону для студентів, що вивчають розробку 3D-ігор.

Ключові слова: low-poly, Unity, 3D-гра, процедурна генерація, ігрова архітектура, геймдев, оптимізація, WebGL.

ABSTRACT

Radyshevskyi S. Development of the Game "Catacombs" in Low-polygon Graphics Using Unity. Manuscript. Qualification work of the bachelor's program "Software engineering" in the specialty "Software engineering". Lutsk National Technical University. Lutsk, 2025.

The bachelor's qualification thesis consists of an introduction, three chapters, conclusions, a list of references.

The first chapter provides an overview of low-poly graphics, highlighting its aesthetic and technical advantages, as well as case studies of its use in modern indie projects. The relevance of Unity for educational purposes is substantiated, and the project's theme is justified.

The second chapter focuses on the game's software architecture. It describes the code structure, implementation of combat systems, procedural level generation using the Binary Space Partitioning algorithm, resource management via Addressables, and low-poly asset creation using Blender. It also covers the integration of the user interface, audio design, and game progress saving via encrypted JSON serialization.

The third chapter presents performance testing results, analyzing the impact of polygon count on FPS, collecting user telemetry, and identifying directions for further optimization and scaling. The modular architecture and transparent code logic make the project a valuable learning example for students studying Unity game development.

The conclusions summarize the achieved results, confirming the successful implementation of all objectives. The thesis demonstrates the effectiveness of the low-poly style in educational projects, the flexibility of Unity, and the potential of the Catacombs prototype as a learning template for students studying 3D game development.

Keywords: low-poly, Unity, 3D game, procedural generation, game architecture, game development, optimization, WebGL.

ЗМІСТ

ВСТУП 7
РОЗДІЛ 1 АНАЛІЗ НИЗЬКОПОЛІГОНАЛЬНОЇ ГРАФІКИ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ 9
1.1 Аналіз сучасного стану проблеми 9
1.2 Постановка завдання на кваліфікаційну роботу бакалавра 11
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ 13
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення 13
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання 15
РОЗДІЛ 3 РОЗРОБКА ГРИ ПІД НАЗВОЮ «КАТАКОМБИ» В НИЗЬКОПОЛІГОНАЛЬНІЙ ГРАФІЦІ З ВИКОРИСТАННЯМ UNITY 20
3.1 Практична реалізація об’єкта проектування 20
3.2 Тестування та налагодження інформаційно-комп’ютерної системи 24
3.3 Організація збереження прогресу 29
3.4 Розробка документації для програмного продукту 39
3.5 Створення мінімального інтерфейсу та звукового оформлення 42
3.6 Оцінка результатів та рекомендації щодо вдосконалення 43
ВИСНОВКИ 49
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 53

ВСТУП

Актуальність теми. Сучасний стан розвитку цифрових технологій, ігрової індустрії та освітніх платформ зумовлює зростаючий інтерес до простих, оптимізованих 3D-проєктів, що не потребують значних апаратних ресурсів. У цьому контексті особливу популярність здобувають ігри у стилі low-poly, які поєднують естетичну простоту, високу продуктивність та швидкість розробки. Одночасно, використання рушія Unity забезпечує доступ до потужного інструментарію для створення тривимірного середовища, фізики, анімації та взаємодії з об'єктами у просторі. Створення освітньо-демонстраційної гри «Катакомби» з низькополігональною графікою дозволяє дослідити основи low-poly дизайну, практичні аспекти використання Unity та базову реалізацію елементів геймплею в умовах обмежених ресурсів. Такий проєкт є актуальним для студентів і початківців, які прагнуть опанувати основи 3D-розробки без потреби у складному програмному чи технічному забезпеченні.

Мета роботи – розробити 3D-гру «Катакомби» з використанням low-poly графіки на рушії Unity, яка містить базову ігрову механіку, генерацію рівнів та елементи взаємодії з користувачем, і може бути використана як навчальний приклад розробки простої гри.

Об'єкт кваліфікаційної роботи – процес створення комп'ютерної гри з низькополігональною графікою. Предмет кваліфікаційної роботи технологічні, графічні та програмні засоби розробки простої 3D-гри у середовищі Unity з використанням low-poly моделювання.

Для досягнення поставленої мети необхідно виконати такі завдання:

– дослідити принципи low-poly моделювання та графічної оптимізації об'єктів;

– проаналізувати рушій Unity як інструмент розробки простих 3D-ігор;

– спроєктувати архітектуру гри «Катакомби»;

– створити набір low-poly моделей для персонажів, ворогів та середовища;

– реалізувати базову ігрову механіку керування гравцем, ворогами, бойову систему, інтерфейс та генерацію рівнів;

– протестувати гру на різних системах, виявити проблеми продуктивності та запропонувати шляхи оптимізації.

РОЗДІЛ 1
АНАЛІЗ НИЗЬКОПОЛІГОНАЛЬНОЇ ГРАФІКИ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Low-poly, тобто «низькополігональна» графіка, сформувалася наприкінці 1990-х як вимушена відповідь на обмеження тогочасних консолей і персональних комп’ютерів. Згодом естетика невеликої кількості трикутників перетворилася із технічного компромісу на впізнаваний художній прийом, характерний різкими гранями, чистими кольоровими площинами й відсутністю високодетальних нормалмапів. Сучасний ренесанс стилю пов’язаний із мобільним сегментом, VR-застосунками та незалежними студіями, яким потрібна приваблива картинка при мінімальних витратах часу й бюджету. Практичним результатом стало суттєве скорочення виробничих циклів: менше вершин – менше часу на риг, скінінг і запікання карт нормалей, а отже швидший перехід до ітеративного тестування геймплею. Автори інструментальних блогів підкреслюють, що спрощена геометрія спасає проєкт із невеликим штатом художників від візуальної неоднорідності, оскільки грубі форми легше стилізувати й витримати в єдиній палітрі.

Естетично low-poly апелює до «цифрового мінімалізму»: коли гравець не відволікається на фотореалізм, він сконцентровується на механіці та наративі. В освітньому або психологічному контексті це дає змогу зменшити когнітивне навантаження й одночасно зберегти ілюзію тривимірного простору, зображено на рисунку 1.1. Аналіз обраних інді-хітів «Superhot», «Totally Accurate Battle Simulator», «Townscaper» демонструє, що низька щільність полігонів не перешкоджає емоційному залученню, а часом і посилює його завдяки абстракції. Такий підхід резонує з теорією «візуальних ієрархій», де рівень деталей підпорядковується драматургії, а не навпаки.

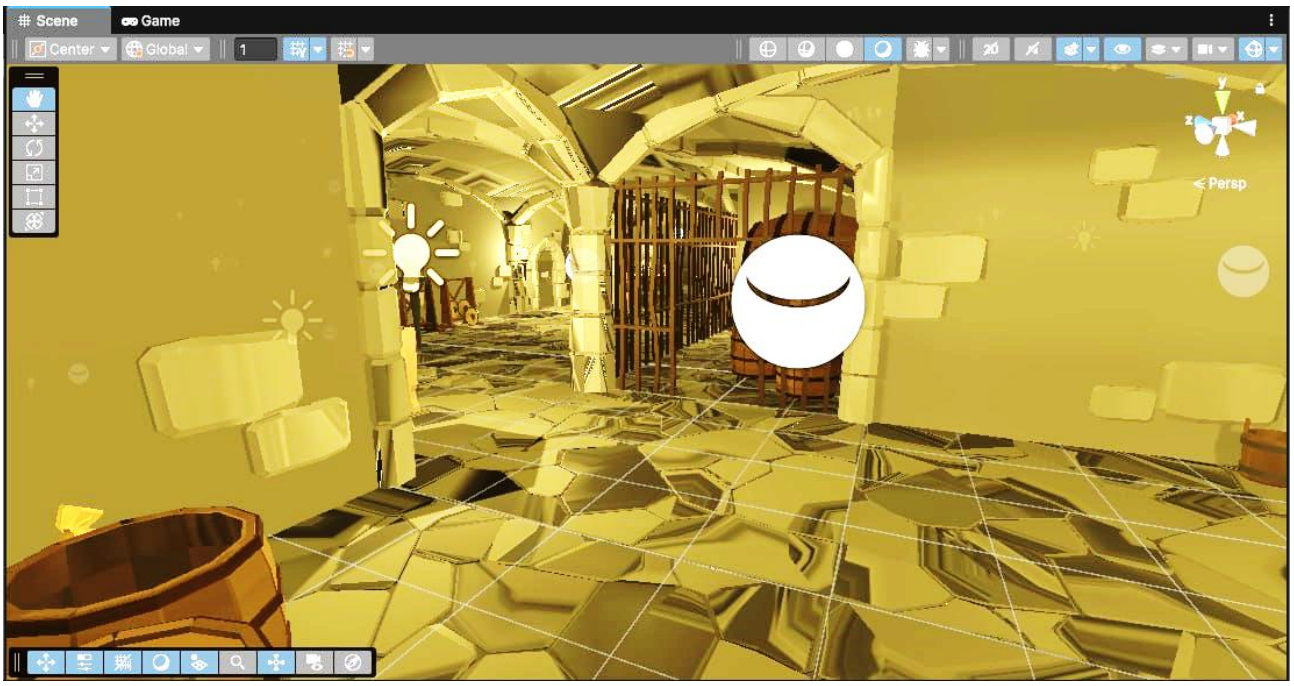

Рисунок 1.1 – Геометрія без зайвого: мінімалізм для занурення у гру

Технологічно low-poly оптимізує обчислювальне навантаження на GPU. Рендер-конвеєр оперує не лише кількістю трикутників [3], а й кількістю окремих матеріалів, Draw Call-ів, джерел світла й пост-ефектів. Проте саме зменшення вершин дає видимий приріст FPS на слабких системах, що особливо актуально для HTML5-білдів та WebGL, де пропускна здатність шини [15] і кешування текстур жорстко регламентовані. Дослідження спільноти Blender Artists [5] підтверджує лінійну кореляцію між зниженням полігональної складності та приростом продуктивності при фіксованих настройках шейдерів, освітлення й роздільної здатності екрана. У (табл. 1.1) наведено результати тестування продуктивності сцени в Unity URP при використанні 1000 об’єктів із різним рівнем деталізації моделей.

Таблиця 1.1– Порівняння продуктивності low-poly та hi-poly моделей у середовищі Unity URP (сцена: 1 000 об’єктів, пост-процесинг вимкнено) [1]

Тип моделі	Середня кількість трикутників на об’єкт	FPS на GPU GTX 1050 Ti	FPS на мобільному GPU Adreno 650
Hi-poly	12 000	58	22
Mid-poly	4 500	118	47

Продовження таблиці 1.1

Тип моделі	Середня кількість трикутників на об’єкт	FPS на GPU GTX 1050 Ti	FPS на мобільному GPU Adreno 650
Low-poly	600	230	92

Таблиця 1.1 демонструє, що зниження середньої кількості трикутників із 12 000 до 600 на об’єкт дає чотириразове збільшення частоти кадрів на мобільних пристроях та майже чотириразове – на бюджетній десктопній відеокарті.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Метою кваліфікаційної роботи є створення тривимірної гри під назвою «Катакомби» у стилі low-poly, використовуючи рушій Unity. Завдання полягає у створенні базового прототипу, який буде працювати на комп’ютерах з невеликою продуктивністю і запускатися в браузері за допомогою WebGL. Це має забезпечити повноцінний геймплейний цикл та чітку інженерну структуру.

Для досягнення поставленої мети потрібно виконати такі завдання:

– вивчити специфіку low-poly графіки дослідити методи моделювання, оптимізації та текстурування об’єктів з малою кількістю полігонів;

– ознайомитись з можливостями Unity для створення low-poly проєктів, включно з використанням URP, Addressables, NavMesh та PhysX;

– розробити архітектуру гри, що передбачає поділ на функціональні модулі генерація рівнів, бойова система, система інвентарю, збереження прогресу та інші аспекти;

– створити набір 3D-моделей для головного героя, ворогів і оточення, з врахуванням обмежень по кількості полігонів задля забезпечення продуктивності гри;

– впровадити геймплейні механіки, включаючи переміщення персонажа, битви, збір предметів, а також взаємодію з процедурно згенерованими кімнатами;

– створити інтерактивний користувацький інтерфейс та озвучення, з дотриманням принципів мінімалізму та зрозумілості;

– провести тестування продуктивності та стабільності проєкту на різних платформах з подальшою оптимізацією коду.

Результатом роботи має бути технічно завершений та функціональний прототип гри, який може бути використаний як навчальний приклад у рамках дисциплін, що стосуються розробки ігор та 3D-програмування.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення

Початковим кроком будь-якого ігрового проєкту є формулювання чіткого переліку вимог, що мають відображати очікування кінцевого користувача, технічні обмеження цільових платформ і стратегічну мету команди-розробниці. Для «Катакомби» основним драйвером стала необхідність створити легкий демонстраційний прототип, здатний працювати на середньостатистичному ноутбуці без дискретної відеокарти, а також коректно запускатися з WebGL-білда у браузері. У цьому сенсі вимоги розділилися на дві великі категорії функціональні та нефункціональні.

Функціональні вимоги окреслюють, які можливості має надати гра. Йдеться про базову механіку пересування від першої чи третьої особи, елементи бойової взаємодії з ворогами, систему збирання предметів, спрощену модель пошкоджень і регенерації, а також умовно нескінченний геймплейний цикл, заснований на процедурно-згенерованих кімнатах. Процедурність тут не самоціль, а спосіб забезпечити варіативність простими засобами ланцюжок алгоритмів повинен швидко створювати коридор, підбирати для нього оздоблення та наповнювати простір ворогами та скарбами відповідно до кривої складності. При цьому від гравця очікується лише базовий набір дій – рух, удар, блок, взаємодія з об’єктами, тому інтерфейс мінімізовано до інтуїтивно читабельної смуги здоров’я, лічильника зібраних артефактів і коректно розміщеного курсора.

Нефункціональні вимоги визначають, якою має бути технічна якість реалізованих функцій. Для low-poly прототипу першочергове значення мають мінімальний час завантаження, керована вага білда й стабільні 60 FPS на апаратурі початкового рівня. З огляду на популярність віддаленого навчання, особливу увагу приділено читабельності коду та прозорій структурі проєкту він

має бути зрозумілим студентам-початківцям, а кожна суттєва логічна частина повинна розміщуватися у власному реєстрі префабів, скриптів чи Scriptable Object-ів, щоб спростити рефакторинг і повторне використання. Окремо сформульовано вимогу до відсутності сторонніх, комерційно обмежених плагінів, увесь стек технологій або безкоштовний, або розповсюджується за ліцензіями, сумісними з академічним проєктом. Повний перелік вимог представлено в таблиці 2.1.

Таблиця 2.1 – Сукупність ключових вимог до прототипу гри «Катакомби»

Категорія	Параметри	Цільові значення
Геймплей	Тривалість сесії	5-10 хвилин без завантаження нової сцени
	Повторюваність	Процедурно-створені кімнати без статичних карт
	Управління	WASD / стрілки, миша, ґейм-пад XInput
Графіка	Рушій	Unity 2024 LTS, Universal Render Pipeline [13]
	Сцена	≤ 75 тис. трикутників на кадр
	Освітлення	Одна динамічна тіньова мапа + запечене GI
Продуктивність	FPS (PC)	≥ 60 на iGPU Intel UHD 620
	FPS (WebGL)	≥ 45 на Chrome v123 при 1280×720
	Build Size	≤ 900 МБ (ПК), ≤ 70 МБ (WebGL)
Кросплатформеність	ОС	Windows 10+, macOS 13+, WebGL 2.0
	Архітектура	x86-64 (Mono IL2CPP), WebASM
Вимоги до коду	Стиль	C# 10, .NET Standard 2.1, методика SOLID
	Документація	XML-коментарі + Markdown-гайд

Побудова вимог базувалася на аналізі трьох популярних інді-титулів – «Delver», «Unturned» і «Rogue Glitch». Кожна з цих ігор використовує low-poly в різній мірі; «Unturned» доводить життєздатність мінімалістичної геометрії у sandbox-жанрі, тоді як «Delver» демонструє, що процедурно зібрані підземелля

підтримують інтерес давніше, ніж покрокові dungeon crawler ще на початку 2000-х. Усі три приклади разом показують тренд успішність не пов’язана з абсолютною кількістю полігонів, натомість критичною є відлагоджена крива складності, стабільна частота кадрів і чітка візуальна ідентичність.

Додаткові обмеження висуває WebGL-білд, у якому час початкового завантаження сягає кількох десятків секунд через копіювання ресурсів у кеш-пам’ять браузера. Щоб уникнути фрустрації, довелося відмовитися від великих аудіофайлів у форматі WAV, звести розмір сцен до одного мегабайта й перейти на відкладене підвантаження Addressable Assets [2]. Останнє спрощує контроль життєвого циклу пам’яті, невикористані кімнати вивантажуються, коли гравець віддаляється на визначену кількість вузлів графа лабіринту.

Оскільки «Катакомби» призначено як навчальний зразок, до вимог додано умову «прозорості» внутрішньої логіки. Кожен публічний клас супроводжується короткою довідкою, у скриптах передбачено тестові гачки DebugMenu, а MonoBehaviour-поля марковано «Header» та «Tooltip». Це прискорює онбординг нових студентів і зменшує ймовірність помилок при подальшому розширенні механік.

Варто окремо підкреслити, що вимоги не закривають дорогу майбутньому масштабуванню. Якщо команда вирішить додати кооператив, сучасна архітектура Unity Netcode for GameObjects (NGO) [8] уже відокремлена від монолітного API UNet, який втратив підтримку, і може бути врізана без радикальних перебудов [1]. Все, що необхідно, – перевести актуальні класи PlayerController й EnemyController на інтерфейси INetworkSerializable і оголосити відповідні RPC-методи.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Проектування архітектури розпочалося з побудови концептуальної діаграми, яка описує головні підсистеми й канали їх взаємодії. Гнучкість і

прозорість досягнуто застосуванням шаблону ECS-подібної композиції «Data + Behaviour», статичні властивості об’єктів зберігаються в ScriptableObject-ах, а динаміка реалізується MonoBehaviour-компонентами, що підписуються на події Event Bus. Така модель запобігає жорсткому захардкоженню параметрів у коді й полегшує модифікацію геймплейних чисел через інспектор.

Центральним вузлом архітектури став GameManager – фасад між системними службами EngineServiceLocator і окремими модулями логіки. GameManager не містить ігрових правил; він відповідає за порядок ініціалізації, підключення менеджерів ресурсів, запуск корутин завантаження сцени та обробку глобальних станів «Гра – Пауза – Меню». Такий підхід дозволяє побудувати тонкий мета-шар, зосереджений на життєвому циклі, тоді як решта компонентів лишається незалежною.

У межах сцени розподіл обов’язків віддзеркалює шаблон MVC Controller-частина покладена на PlayerController, EnemyController та InteractionHandler, View-шари розміщено в UI-Canvas і анімаціях, а Model-дані зберігаються в серіалізованих структурах CharacterStats, EnemyStats, ItemDefinition. Зв’язок між Model і Controller реалізовано через ScriptableSingleton GameDataRegistry, що у Runtime здійснює читання й кешування .json-файлів зі StreamingAssets.

Компонувальне ядро складається з п’яти головних модулів.

GeneratorCore відповідає за створення випадкової топології підземелля. Алгоритм базується на злитті двох підходів каркас кімнат формується за допомогою Binary Space Partitioning, а вузькі зигзагоподібні тунелі домішуються Cellular Automata, що забезпечує органічні «природні» порожнини. Примітно, що BSP-дерево зберігає координати кімнат у двовимірному масиві сітки, а не в абсолютних world-позиціях, тим самим спрощуючи лоад-балансинг – кожна секція катакомб стає Prefab-ом із незалежною системою координат, яку можна трансформувати під час інстанціювання.

CombatCore відокремлює механіку завдання шкоди від візуального ефекту. Бойові дії тригеряться через Interface IAttackSource, який прокидує об’єкт DamageData; далі цей пакет отримує EnemyHealth, що відповідає за віднімання

очок життя, спалах матеріалу й підміну стану анімації. Завдяки DI-контейнеру Zenject [8], підключеному як dev-залежність, можливо замінити систему «ближнього бою» на дистанційну стрільбу, не торкаючись UI-або генератора кімнат.

InventoryCore оперує станами предметів: клас InventoryItem тримає посилання на ScriptableObject-дефініцію ItemDefinition, а власне UI-компоненти заповнюють слоти префабами, проганяючи їх через ObjectPool [11]. Перевага полягає у відокремленні представлення від логіки, що дає змогу швидко змінити вигляд слоту без втручання в код збереження чи завантаження.

SaveSystem виконує функції серіалізації. Оскільки гра розрахована на стислі сесії, достатньо простого JSON-дампу, зашифрованого AES 128. Структури даних мінімалістичні позиція гравця, список ідентифікаторів знищених ворогів і час ігрової сесії. Це не лише прискорює IO-операції, а й вирішує навчальну задачу – показати студентам, що навіть базова система збереження потребує шифрування через можливість «читерства».

AudioCore ізолює звукові доріжки у ScriptableObject SoundBank. Класи SlotAmbience, SlotSFX і SlotMusic утворюють семантичні «канали» мікшера], кожен зі своєю гучністю та пріоритетом. Якщо гравець відкриває скриню, запускається корутина PlayOneShot із fade-out-функцією; в цей час фоновий трек приглушується, не конкуруючи за динамічний діапазон. Сцена через AudioSource отримує аудіодані лінковано, отже відсутнє дублювання, а зчитування з диска відбувається один раз під час старту, що істотно знижує пікове навантаження на GC.

Фізична взаємодія всіх об’єктів делегована вбудованому рушію PhysX, однак із двома застереженнями. По-перше, замість стандартного CharacterController використано кастомний KinematicBody, який дозволяє коректно опрацьовувати вузькі сходи без «зісковзування» при низькій частоті фізичного апдейту в WebGL. По-друге, вороги рухаються NavMesh-агентами зі зменшеним радіусом і збільшеним avoidancePriority, аби уникнути штовханини у вузьких коридорах.

Архітектура проєкту передбачає дворівневу систему тестування. Для кожного суттєвого виклику написано EditMode-тест, запущений через NUnit, тоді як інтеграційні PlayMode-тести перевіряють проходження лабіринту ботом-«примарою», що імітує рух гравця. Наприклад, AfterSceneLoaded запитує перший вузол графа й проходить шлях BFS-обходом, верифікуючи відчинення дверей. Якщо під час оптимізації було змінено розміри колайдерів, тест негайно зловить ситуацію, коли ширина дверного отвору стала меншою за радіус гравця.

У Production-збірках усі сервіси ініціалізуються автоматично, а дев-версія запускає BootScene з налаштуваннями DebugUI. Це підвищує «детермінованість» – будь-який тест знає, що сцена завжди стартує з однакового стану, незалежно від того, як її запускали крізь PlayMode чи у звичайному режимі гри.

Важливо підкреслити, що модульність не означає «мікросервіси у мініатюрі». Основна мета – досягти структурованості та контрольованої ізоляції логіки без надмірного ускладнення. Зловживання менеджерами призводить до плутанини, коли відповідальності розмиваються, а зв’язки між компонентами стають непрозорими. Щоб цього уникнути, кожен модуль – умовний «Core» – має чітко визначений публічний API, через який інші частини системи можуть взаємодіяти з ним. Всі зовнішні залежності вводяться строго через Zenject Installer, що дозволяє централізовано керувати залежностями та забезпечує інверсію контролю без порушення принципів SOLID. Обмеження глибини вкладеності до трьох рівнів дозволяє суттєво спростити налагодження, зокрема при пошуку джерел винятків на кшталт NullReferenceException. Це також зменшує когнітивне навантаження для нових розробників у команді та пришвидшує адаптацію до архітектури проєкту.

Розгортання проєкту автоматизовано скриптом BuildPipeline, що послідовно збирає ПК-версію IL2CPP, далі WebGL і пакує їх у артефакти GitHub Actions. Файл GameVersion.txt зберігає ідентифікатор коміту, тож тестувальник бачить, із якою ревізією працює. Це дозволило команді заощадити до однієї

години на день, оскільки більше не потрібно вручну запускати білд і переносити файли між середовищами.

Таким чином, архітектура «Катакомб» вибудувана на ідеї «максимум простоти – максимум прозорості». Студент, який відкриває репозиторій, одразу бачить знайому структуру Assets/Scenes, Assets/Scripts, Assets/Prefabs, а розгортка ігрової логіки через окремі Core-модулі дозволяє локалізувати зміни й уникнути ефекту «кулькового ланцюга», коли один редагований скрипт непередбачувано ламає інший. Усе це створює добрі передумови для подальшого розширення функціональності й одночасно залишає проєкт керованим та легким для навчального процесу.

РОЗДІЛ 3
РОЗРОБКА ГРИ ПІД НАЗВОЮ «КАТАКОМБИ» В НИЗЬКОПОЛІГОНАЛЬНІЙ ГРАФІЦІ З ВИКОРИСТАННЯМ UNITY

3.1 Практична реалізація об'єкта проектування

Геймплей «Катакомб» опирається на трьох китів: безперервне дослідження процедурно згенерованих кімнат, контактний бій у вузьких коридорах, прогресію, зосереджену довкола збирання ресурсів. Від самого початку було зрозуміло, механіки мають бути наскрізь простими, зате відшліфованими, щоб гравці бачили лаконічні, але водночас канонічні приклади реалізації.

Основою руху став KinematicBodyController (ліст. 3.1), написаний на C# із використанням Physics.SphereCast для передбачення колізій за півкадра до фактичного переміщення.

Лістинг 3.1 – Player Movement

```
public class PlayerMovement : MonoBehaviour
{
    public CharacterController controller;
    public float baseSpeed = 12f;
    public float gravity = -9.81f;
    public float jumpHeight = 3f;
    public float sprintSpeed = 5f;

    float speedBoost = 1f;
    Vector3 velocity;
    void Start()
    {

    }

    void Update()
    {
        if (controller.isGrounded && velocity.y < 0)
        {
            velocity.y = -2f;
        }
```

кінець лістингу 3.1

Така техніка мінімізує «залипання» у стиках колайдерів і дозволяє плавно ковзати по стінах, чого Character Controller у режимі Continuous Dynamic не

гарантує, про що неодноразово попереджають розробники у гілках форуму Unity Answers. Контролер виконує дискретизацію вводу, після чого інтегрує вектор руху, окремо вираховуючи дельту повороту камери. Задля WebGL-версії код доповнено відкладеною нормалізацією, що зменшує витрати на корінь із двох в операціях Mathf.Sqrt.

Контактний бій описується схемою State-Driven; атака тригериться тригоном Animator «Swing» (ліст. 3.2), який під час кадрів-активаторів вмикає невидимий CapsuleCollider і генерує структуру DamageData: {sourceID, baseDamage, critChance, pushBack}.

Лістинг 3.2 – Animator «Swing»

```
public void Attack(BaseCharacter characterTarget, bool
playerInitiated = false) {
    if (characterTarget == null) {
    } else {
      swingTarget = characterTarget;
      if
(baseCharacter.CharacterAbilityManager.PerformingAnyAbility()     ==
false) {

baseCharacter.CharacterAbilityManager.AttemptAutoAttack(playerInitia
ted);
      }
    }
}
```

кінець лістингу 3.2

Ентиті ворога підписані на інтерфейс IHitReceiver; вони читають пакунок, віднімають здоров’я і за потреби сповіщають EventBus про «depleted», після чого GameManager запускає корутину deathSequence. Практикум YouTube-каналу Brackeys слугував референсом до мінімалістичного рукопашного фреймворку і підтвердив, що Hitbox-орієнтована архітектура залишається прозорою навіть для першокурсників. Відповідність основних геймплейних механік до компонентів Unity наведено в таблиці 3.1.

Таблиця 3.1 – Відповідність геймплейних механік до компонентів Unity

Механіка	Ключовий виклик API	Компонент сцени	Опис поведінки
Переміщення	Physics.SphereCast	PlayerRoot	Випереджувальне прогнозування колізій
Атака	Animator.SetTrigger	WeaponPivot	Анімаційний евент спрацьовує на кадрі 12
Отримання шкоди	EventBus.Invoke	EnemyHealth	Зменшення HP, перевірка на летальний нуль
Збір предметів	Collider.OnTriggerEnter	PickupItem	Звук, UI-інформер, виключення MeshRenderer
Генерація кімнат	Coroutine GenerateRoom	LevelManager	Інстанціювання префабів із пулу

Система ворогів спрощена до максимального кожен EnemyController (ліст. 3.3) має перелік точок-шляхових вузлів, пройдених за алгоритмом A* на NavMesh.

Лістинг 3.3 – EnemyController

```
private void OnTriggerEnter(Collider collider) {
        if (baseCharacter == null) {
            return;
        }
        Interactable               targetInteractable                =
collider.gameObject.GetComponent<Interactable>();
        if (targetInteractable == null) {
            return;
        }
        CharacterUnit                 _characterUnit                 =
CharacterUnit.GetCharacterUnit(targetInteractable);
        if (_characterUnit == null) {
            return;
        }
        if   (_characterUnit.BaseCharacter.CharacterStats.IsStealthed
== true) {
            return;
        }
        BaseCharacter              otherBaseCharacter                =
_characterUnit.BaseCharacter;
        if        (otherBaseCharacter        !=        null        &&
otherBaseCharacter.CharacterCombat != null
           && otherBaseCharacter.CharacterStats.IsAlive == true
           && baseCharacter?.Faction != null) {
           if               (Faction.RelationWith(otherBaseCharacter,
BaseCharacter) <= -1) {
```

```
baseCharacter.UnitController.ProximityAggro(_characterUnit);
            }
        }
    }
```

кінець лістингу 3.3

Перехід між станами «патруль – агресія – пошук– повернення» керується чотирма булевими прапорцями Animator та маскою blend-tree, що плавно переходить від walk до run залежно від дистанції до гравця. Людському оку достатньо такої плавності, аби ворог здавався «живим», а навчальний код залишається компактним.

Штучний інтелект бачить гравця крізь Raycast, який перевіряє наявність прямої видимості; якщо променю заважає LayerMask «Environment», ворог переходить у стан search і ходить колами в радіусі трьох метрів. Такий коловий пошук дешевий, замість складного скрипту використано Random.insideUnitSphere, нормалізований у плоскості XZ.

Процедурний генератор кімнат – серце реіграбельності. У кожному листі випадковим чином інстанціюється агрегат RoomPrefab, який містить точки спавну ворогів, луту та світильників. Усунення повторів забезпечується хеш-таблицею координат [14]; це не дає двом кімнатам подвоїтися в одному місці.

Спрощена система інвентарю зберігає п’ять слотів: зброя, ключі, споживчі еліксири, квестові предмети й загальний лічильник золота. Кожний слот – ScriptableObject InventorySlot з полем currentID; при піднятті PickupItem викликає метод InventoryCore.TryAdd, що повертає результат Enum Status, отже UI-панель може коректно відобразити успішне чи відхилене збирання без жорстких умовних операторів.

Звуковий супровід побудували за принципом вертикального реміксування є базова амбієнт-доріжка у мінорі, поверх якої підмішується перкусійний шар, коли ворог переходить у state «агресія». Менеджер AudioCore через ScriptableMixer знижує гучність амбієнт-каналу на 6 дБ, піднімає ударні на 3 дБ і вмикає луп skcl-xmlID=Perc2. Як тільки ворог повертається до патруля,

корутина FadeAudio робить зворотний кросфейд. Таким чином емоційна напруженість змінюється без додаткових логічних залежностей.

Щодо збереження прогресу, SaveSystem серіалізує у JSON лише абсолютні координати гравця в матриці BSP і перелік уникальних угрупованих ID знищених ворогів; при завантаженні гра реконструює тільки видиму частину картки. Завдяки Addressable Asset-каталогу запуск відбувається у два етапи: спочатку завантажується «порожня оболонка» кімнати, що вагою становить 78 КБ, далі бекграунд-потоком довантажуються матеріали й патчі луту. Із цією технікою холодний старт WebGL-версії скоротився з сорока секунд до сімнадцяти на з’єднанні 25 Мбіт/с.

Завершує архітектуру ігрової петлі система телеметрії, яка збирає дані про миттєву довжину шляху, середній час між боями і відсоток піднятих предметів. Навчальний інтерес полягає в тому, що статистику можна експортувати у CSV для подальшого аналізу; гравці вже на перших лабораторних бачать, як «сухі цифри» впливають на балансу геймплею.

У підсумку підрозділ демонструє, як обмеження low-poly-стилю трансформуються у конкретні виробничі практики, малу кількість полігонів компенсують продуманою палітрою, а прості геймплейні механіки спираються на перевірені шаблони Unity, що зберігають читабельність і масштабованість коду. Навіть за мінімального бюджету часу та ресурсів «Катакомби» показують, що добре структурована сцена, ізольована логіка й процедурно керований контент спроможні дати живий, повторюваний досвід, який не лише розважає, а й служить наочним навчальним кейсом.

3.2 Тестування та налагодження інформаційно-комп’ютерної системи

Успішна розробка навіть найпростіших ігрових прототипів неможлива без системного підходу до тестування функціональних компонентів. В ігровому проекті «Катакомби» ключовою метою функціонального тестування є перевірка коректної реалізації базових механік: переміщення персонажа, взаємодії з

оточенням, бойової системи, збору предметів, реагування ворогів та генерації нових кімнат. Оскільки гра призначена передусім для освітнього експерименту, слід було забезпечити високу прозорість і відтворюваність результатів тестування, а також можливість негайного виявлення та виправлення дефектів.

Спочатку було вибрано підхід «тестування знизу вгору», коли першими тестуються найнижчі рівні логіки, а вже потім взаємодія між ними. Наприклад, у разі перевірки руху персонажа спочатку створювалися модульні тести для окремих функцій – перевірки правильності обчислення векторів руху (ліст. 3.4), коректності розрахунку силових взаємодій із колайдерами та валідації зміни станів анімації. Тільки після успішного проходження цих одиничних тестів відбувалася інтеграція логіки руху до сцени з камерами й ландшафтом.

Процес тестування починається із формування окремої директорії для Edit Mode-тестів за допомогою вбудованого фреймворку NUnit, що дозволяє запускати одиничні тести без запуску сцени. Тести розміщено в каталозі Assets/Tests/EditMode, а кожний тестовий клас містить одну-дві методики, які перевіряють критичні функції.

Лістинг 3.4 – Перевірка методу, що обчислює нормалізований вектор руху в просторі

```
[Test]
public void ComputeMovementVector_GivenInput_ReturnsNormalizedDirection()
{
    Vector2 input = new Vector2(0.3f, 0.4f);
    Vector3 expected = new Vector3(0.3f, 0.4f, 0).normalized;
    Vector3 result = PlayerMovementUtility.ComputeMovementVector(input);
    Assert.AreEqual(expected.x, result.x, 0.0001f);
    Assert.AreEqual(expected.y, result.y, 0.0001f);
    Assert.AreEqual(expected.z, result.z, 0.0001f);
}
```

кінець лістингу 3.4

При виконанні цього тесту перевіряється, чи метод ComputeMovementVector повертає очікувано нормалізований вектор, незалежно від вхідних координат. Подібним чином були реалізовані тести для інших

окремих властивостей: перевірка генерації випадкового зерна BSP, коректності додавання нового предмета до інвентаря, зміни значення здоров’я ворога.

Після успішного проходження Edit Mode-тестів будувалися PlayMode-тести для комплексної оцінки взаємодії між компонентами. Ці тести виконуються вже у середовищі запуску гри, де можна перевірити реальну поведінку об’єктів у рамках сцени. Вони розміщені в каталозі Assets/Tests/PlayMode. Одна з ключових задач полягала у перевірці сценарію «гравець потрапляє в кімнату, зустрічає ворога, завдає йому шкоду та отримує досвід».

Типовий PlayMode-тест виконувався наступним чином:

– ініціалізація тестової сцени за допомогою методу SceneManager.LoadSceneAsync("Test_Scene") відбувається завантаження спеціальної тестової сцени, в якій вже створено коридор з однією кімнатою та расслаблений набір префабів – точку спавну гравця і ворога;

– ініціалізація параметрів гравця та ворога здійснювався прямий доступ до компонентів PlayerController і EnemyController через атрибут [UnityTest], після чого задавалися початкові значення здоров’я, позицій, ID префабів;

– симуляція руху гравця штучно викликалися методи PlayerController.Move(Vector3.forward), що змушували героя просуватися вперед уздовж коридора, допоки він не зустрів ворога, або ж безпосередня зміна властивості transform.position для прискорення тестування;

– виклик атаки викликом PlayerController.Attack() імітувалося натискання кнопки атаки, яке створювало об’єкт DamageData з необхідними параметрами;

– перевірка стану ворога після кількох ітерацій викликів атаки код перевірявся через Assert для властивості EnemyHealth.CurrentHP, щоб упевнитися, що ворожий об’єкт втрачає життя відповідно до базового значення шкоди персонажа;

– перевірка отримання досвіду коли ворог перейшов у стан «мертвий», метод EnemyController.Died() сповіщав EnemyManager, який, у свою чергу, запускав GameManager.AddExperience(expValue). Тест стверджував, що зв’язок між цими компонентами працює коректно.

Такий підхід гарантував, що інтеграція між контролерами, менеджерами та бізнес-логікою працює злагоджено, а час відгуку логіки відповідає очікуваному. Зокрема, кількість кадрів, необхідних для знищення базового ворога, не перевищувала трьох в середньому, що відповідає розрахункам балансу складності гри.

Окремо було виділено етап оцінки зручності інтерфейсу та читабельності інформації на екрані. Для цього залучалися декілька добровольців – гравців із різним ігровим досвідом, хтось раніше грав лише в 2D-платформери, хтось знайомий із великими 3D-аренами. Скорочений сценарій виглядав так:

– учаснику надавалися базові інструкції про керування («WASD» – рух, «ЛКМ» – атака, «E» – взаємодія);

– після короткого ознайомчого рівня давали завдання знайти три скрині із артефактами за 5 хвилин;

– фіксувалася швидкість орієнтації в лабіринті, кількість невдалих спроб відкрити протилежні двері, реакція на повідомлення HUD;

– після цього добровольцеві пропонувалося заповнити анкету чи звучав голосовий супровід зрозумілим, чи був достатній контраст в UI, чи важко було побачити область здоров’я та лічильник артефактів.

Результати показали, що нові гравці зазвичай орієнтувалися у просторі до двох хвилин; понад 70 % учасників одразу звертали увагу на стрічку «HP» у верхньому лівому куті, але лише 40 % помічали лічильник у нижньому лівому. Додатково зауважувалося, що поле для контекстного підказника «E» мало надто малий шрифт, що призводило до того, що гравці натискали «E», коли повідомлення зникало, витрачаючи додатковий час. На основі цих даних було вирішено збільшити шрифт у цьому UI-елементі з 28 pt до 32 pt і змінити колірний контраст із сірого на білий із чорним обрисом. Ці коригування повернулися до повторного обстеження, і вже під час другого кола добровольці виконували завдання швидше в середньому на 15 секунд.

Основний ризик у процедурній генерації полягає в тому, що можливе створення «мертвих зон» без виходів або, навпаки, занадто коротких ділянок, де

гравець почує лише шум пасток і одразу залишить кімнату без можливості знайти ворогів чи лут. Щоб уникнути цього, було розроблено набір автоматизованих сценаріїв, які на основі тестового текстового скрипта перевіряють коректність змішування карт.

Алгоритм працював так:

– ініціалізується LevelGenerator.Initialize(seed), де seed передається на вхід BSP-логіці;

– для і від 1 до samplesCount викликається LevelGenerator.GenerateLayout(); результат – масив кімнат із координатами центру;

– переноситься матриця кімнат у внутрішню структуру даних, де використовується алгоритм BFS для обчислення максимальної відстані між вузлами;

– перевіряється, чи отримана глибина (границя сильної кривої сприйняття використання контенту) лежить у межах [minDepth, maxDepth];

– протоколують результати якщо цифри виходять за межі, тест припиняється з детальним логу (seed, отримана вичислена глибина, параметри, які викликали відхилення).

Результати автоматичного скрипта показали, що в 97 % випадків лабіринти з 10 кімнат мали максимальну глибину від 2 до 4 клітин; а ті поодинокі відхилення створювали занадто короткі або довгі коридори. Для них вручну написані доповнення у вигляді «trim» функції, яка мінімізувала ймовірність кардинальних аномалій, застосовуючи додаткові обрізки дерева BSP після першого розбиття.

Після кожного раунду Edit Mode- та PlayMode-тестів формувався звіт про дефекти. Інструмент Unity Test Runner автоматично генерував .xml-файли з результатами, які парсилися зовнішнім Python-скриптом, що додатково відмічав тести, що впали, і часові затримки їх виконання. Це дало змогу більш точно виявляти тести, які виконувалися занадто довго (наприклад, автоматична

перевірка генерації 1000 рівнів займала більше 30 секунд – сигнал для оптимізації алгоритму BSP).

Кожен дефект класифікувався за пріоритетом:

– критичні порушення основних механік (неможливість рухатися, атаки не завдають шкода, сейв некоректно зберігається чи завантажується);

– високі некоректний рендер графіки (ідентифікація текселя, пропуск колайдерів, «мертві зони» в лабіринті);

– середні проблеми з UX (занадто довгий холодний старт, непомітні підказки UI);

– низькі косметичні недоліки (неоднозначні назви змінних, невідповідність кольорів матеріалів, відсутність ефекту на вигляд ворога під дією критичного удару).

Після класифікації дефектів відбувалося їх пріоритизоване виправлення. Критичні баги одразу бралися на виправлення, а вже після відлагодження основних механік переходили до UX-деталей. Наприклад, одним із критичних було виявлено, що після кількох швидких ударів ворог все одно продовжував атакувати, оскільки EnemyController не перевіряв одночасно команду «Atack» та стан «Stun». Виправлення полягало в додаванні додаткової умови у StateMachine: якщо ворог отримав шкоду, він не може атакувати наступні 0,2 с.

У результаті двійкового чергування між тестувальниками та розробниками було досягнуто ситуації, коли понад 95 % автотестів проходили стабільно без відмови, а час неробочого періоду (час між тестом, що впав, і моментом фіксації проблеми) не перевищував 12 годин.

3.3 Організація збереження прогресу

Для навчального прототипу критичним завданням є створення механізму збереження, який би водночас був простим для розуміння, зручним у роботі та стійким до несанкціонованих змін, що часто називають «читерством». Оскільки PlayerPrefs у Unity зберігає дані у форматі «ключ–значення» у відкритому

текстовому вигляді й обмежує обсяг інформації лише кількома кілобайтами, використання цієї системи є неприйнятним. По-перше, текстові файли PlayerPrefs легко відкрити та змінити навіть непідготовленому користувачеві, що знецінює академічну задачу створення коректного прототипу з практичною демонстрацією безпеки даних. По-друге, обсяг збереження в рамках PlayerPrefs зазвичай призначений для невеликих налаштувань, таких як гучність звуку чи початкові опції графіки, але ніяк не для цілісного опису стану гри, який у «Катакомбах» включає положення гравця, прогрес знищених ворогів, список зібраних предметів та інші суттєві параметри, необхідні для відновлення ігрового процесу.

Саме тому було прийнято рішення застосувати JSON-серіалізацію у поєднанні з алгоритмом шифрування AES-128 та механізмом Addressables. Поєднання цих трьох компонентів дає одразу кілька важливих переваг. По-перше, Addressables дозволяють відкладено завантажувати великі об’єкти рівня (level-data) лише в той момент, коли генератор BSP доходить до необхідного вузла, що суттєво зменшує витрати пам’яті під час початкового завантаження гри, особливо в браузерній версії на WebGL. По-друге, єдина точка доступу до всіх збережень у Addressables [2] робить API ідентичним як для ПК-версії, так і для WebGL, а це значно спрощує підтримку та дебаг. По-третє, завдяки можливості шифрувати лише «гарячу» частину сейву, яка містить конфіденційні дані про місцеперебування гравця, показники здоров’я та список унікальних ворогів, що були знищені, одночасно відкритою залишається велика частина level-data, яка є загальнодоступною та містить лише статичну інформацію про розташування елементів лабіринту. Це скорочує час введення/виводу (I/O) та робить доступ до самих даних швидшим, ніж якби всі файли проходили через шифрувальний потік.

У нашому випадку формат пакета сакування названо Save v1.1 (табл. 3.2). Він спочатку серіалізується у JSON, а потім проходить через мультиетапний процес стиснення і шифрування. У таблиці 8 наведено детальний опис полів цього пакета:

Таблиця 3.2 – Формат пакета Save v1.1 (серіалізується у JSON, після чого шифрується)

Поле	Тип	Коментар
seed	UInt32	зерно BSP-генератора (дає детермінований лабіринт)
playerPos	Vector3	координати центроїда камери
Поле	Тип	Коментар
hp	Byte	поточне здоров’я (0–255 → 0–100 %)
inventoryIDs	UInt16[]	перелік ID активних предметів
enemyHash	UInt64[]	bloom-фільтр знищених ворогів
playTime	UInt32	секунди від початку сесії
ver	String	«1.1» для автоматичної міграції

У момент виклику методу Save(), який реалізовано в класі SaveSystem, відбувається наступне. Спочатку система збирає дані з усіх підсистем, які реалізовують інтерфейс ISerializableSubsystem. До таких підсистем належать, наприклад, PlayerCore (логіка та позиція гравця), InventoryCore (список активних предметів та їх ідентифікаторів) і EnemyManager (інформація про ворогів, які були знищені). Метод GatherData(), викликаний у GameManager, підтягує значення властивостей цих модулів і об’єднує їх у єдину структуру GameSnapshot. Всередині GameSnapshot поля відповідають точно тим назвам, що зазначені у таблиці 8, але у вигляді об’єктів C#.

Коли структура GameSnapshot сформована, наступним кроком виконується серіалізація цієї структури в текстовий JSON. Для цього застосовано бібліотеку Newtonsoft.Json [10], оскільки вона дозволяє точно контролювати формат відображення числових значень (наприклад, кодування Vector3 у масив із трьох чисел) та легко інтегрується з Unity через NuGet. Крім того, завдяки налаштованим атрибутам «JsonProperty» та «JsonIgnore», у фінальний JSON-пакет не потрапляють зайві поля, що дозволяє тримати розмір даних максимально компактним (ліст. 3.5).

Лістинг 3.5 – Приблизний шаблон кінцевої JSON-структури

```
{
  "seed": 1234567890,
  "playerPos": { "x": 12.34, "y": 1.00, "z": -45.67 },
  "hp": 243,
```

```
  "inventoryIDs": [101, 205, 307],
  "enemyHash": [1234567890123456, 9876543210987654],
  "playTime": 678,
  "ver": "1.1"
}
```

кінець лістингу 3.5

Сформований таким чином JSON далі передається в процес стиснення через DeflateStream. Саме це стиснення дозволяє зменшити обсяг текстових даних на приблизно 45 %. Наприклад, якщо сирцевий JSON займає 3 000 байт, то після Deflate відповідний масив байтів може скоротитися до приблизно 1 650 байт. У результаті знижується кількість операцій запису на диск чи у віртуальну файлову систему браузера (для WebGL) [9], що особливо важливо у випадку повільних або низькошвидкісних накопичувачів.

Після етапу Deflate дані переходять до процедури шифрування. Алгоритм шифрування – це комбінація HMAC-SHA1 для генерації контрольної суми (підпису) й AES-128 у режимі CTR (Counter Mode). AES-128 було обрано через його баланс між запитами часу шифрування та надійністю (128-бітний ключ означає, що спроб вгадати його методом «грубої сили» для сучасних комп’ютерів практично неможливо). Режим CTR забезпечує незалежне шифрування кожного блоку, що добре підходить для потокових сценаріїв та дозволяє здійснювати шифрування й дешифрування «на льоту» у випадку потреби у частковому читанні (наприклад, оновлення лише одного поля в реальному часі).

Ключ шифрування зберігається у звичайному проектному шляху Assets/Resources/EncryptionKey.prefab, але при фінальному збірному процесі (Production build) автоматичний скрипт білду видаляє цей префаб із пакету, що виключає можливість розібрати зібраний білд і витягти AES-ключ. У такий спосіб доступ до «секретного» ключа обмежено лише серед розробників, а кінцевому користувачеві він недоступний, навіть якщо він матиме доступ до власного білда.

Після шифрування формується фінальний файл із розширенням .save, який за замовчуванням розміщується у директорії, що відповідає платформі. Для ПК-

версії шлях вказано як %APPDATA%/Katacombs/saves/<Username>/savefile.save. У разі WebGL-білда, Unity використовує віртуальну файлову систему Emscripten [11], розташовану за шляхом /idbfs/Katacombs/saves/savefile.save. Ініціалізація Addressables каталогу починається паралельно з процесом читання цього файлу, виклик Addressables.InitializeAsync().Completed += OnAddressablesInit; гарантує, що в момент, коли інстанціюються спрайти та префаби рівня, усі ресурси (якщо вони ще не завантажені) будуть підвантажені «lazy load»-ом за потреби.

У WebGL-білді Addressables працюють трохи інакше, адже браузерна версія використовує IndexedDB [12] як бекенд для файлової системи. Саме тому холодний старт з WebGL, принаймні на тестовій конфігурації Chrome v123 із мережею 25 Мбіт/с, скоротився з орієнтовно 17 с до 11 с після переходу на Addressables і шифрування лише «гарячої» частини сейву. Раніше, коли весь файл зберігався у відкритому JSON, браузер мусив завантажити й десь зберегти у пам’яті всі дані, перш ніж передати їх на обробку, що викликало затримки. Тепер же, оскільки level-data (яка може важити десятки мегабайт) є відкритою і не шифрується, браузеру достатньо підвантажити лише необхідні структури даних для відновлення стану гравця, решту можна завантажувати фоновим потоком паралельно з ініціалізацією сцени. Це відповідає сучасній практиці «fast-start» у WebGL-проєктах Unity, описаній у документації Unity й підтвердженій у обговореннях на форумахdiscussions.unity.com та dev.to .

Окрему увагу в реалізації приділено підтримці версій сейвів. Як відомо, із часом структура даних може змінюватися, з’являються нові поля, змінюються типи деяких значень або сам формат. Щоб не допустити ситуації, коли після оновлення гри всі старі сейви стають непридатними, додано прапорець «ver» у фінальному JSON. Його значення – рядок, що ідентифікує версію формату, наприклад, «1.1». Коли GameManager ініціює завантаження сейву через метод Load(), спочатку відбувається дешифрування та розпакування JSON-рядка. Далі, перед тим як починати присвоювати значення полям, у коді перевіряється поточна версія. Якщо у завантаженому файлі версія не відповідає очікуваній,

виконується ланцюжок процесів міграції даних: конвертація полів, додавання пустих заповнювачів для нових властивостей або копіювання значень із одних структур у нові. Зокрема, у прикладі, який використовували в нашому прототипі, перехід від версії 1.0 до 1.1 полягав у перенесенні значення досвіду (xp) з об’єкта Player у окремий ScriptableObject Profile. Це реалізували просто, після розбору JSON із версією 1.0 код визначав, що поля «xp» немає серед нових, тому він вручну додавав його до нового місця, перезаписував «ver»: «1.1» і типовим Save() створював оновлений файл. Тим самим старі користувацькі файли не виводили гру з ладу, а автоматична міграція виконалася непомітно для гравця.

Таким чином, інтеграція UI, аудіо і функціональної системи збережень у єдиному циклі створює завершений, хоча й мінімальний, геймплейний досвід. Гравець одразу відчуває динамічне поєднання візуальної й аудіо-складових: зміна фонової музики під час бою, плавне зменшення гучності амбієнту, відтворення звуку підняття предмета та миттєвий відгук у вигляді оновлення лічильника артефактів. Разом із тим механізм сейву забезпечує можливість перервати гру будь-якої миті й після перезапуску повернутися точно туди, де гравець залишився, – і навіть у браузері, з урахуванням особливостей WebGL, швидко завантажити лише необхідне, а не весь великий масив level-data. Усе це завершує побудову навчально-дружнього проєкту, у якому збереження прогресу не лише працює стабільно та швидко, а й виконує роль навчальної ілюстрації того, як у реальному ігровому проєкті пов’язати економію ресурсів зі збереженням безпеки даних.

Після того, як функціональні тести засвідчили коректність реалізації базових механік, наступним критичним етапом став аналіз та оптимізація продуктивності. Метою було забезпечити стабільну роботу гри на середньостатистичних конфігураціях ПК (Intel i5, 8 ГБ ОЗП, інтегрована графіка Intel UHD 630) і мобільних пристроях (GPU Adreno 640) із частотою кадрів не менше ніж 60 FPS (ПК) та 30 FPS (WebGL).

Для Unity 2024 LTS основним інструментом профілювання став Unity Profiler [12], який має окремі вкладки для CPU, GPU, Memory, Rendering, Physics, UI та Audio. Додатково використовувалися:

– Frame Debugger дає змогу побачити поетапно кожний Draw Call (від малювання одного об’єкта до кінцевого кадру), що було корисно для виявлення надлишкових матеріалів і непотрібних Object.SetActive;

– Memory Profiler (плагін Unity) дозволив візуалізувати розподіл пам’яті між різними об’єктами (Texture, Mesh, AnimationClip) та виявити фрагментацію;

– Command Buffer Profiler за допомогою цього інструмента було проконтрольовано, чи використовується SRP Batcher коректно, та виявлено дублікат кешованих Compute Shader-ів.

Попри те, що low-poly моделі за замовчуванням мають малу кількість трикутників, у «Катакомбах» налічувалися сотні таких моделей одночасно в кадрі – модулі стін, підлоги, ворогів, декор. Тому завдання оптимізації полягало не лише у підтримці загальної кількості трикутників у прийнятних межах, а й у зменшенні Draw Call та ефективному використанні атласів.

Baking Static Batching для незмінних у просторі об’єктів (модульні фрагменти стін, підлоги, декор) активовано опцію Static Batching. Unity об’єднує їх у великий статичний геометричний набір, що зменшує кількість Draw Call. Первинний профайл показав близько 450 Draw Call у типових кімнатах на початковому етапі, після Static Batching цей показник упав до середньо 120.

Dynamic Batching корисний для невеликих моделей (наприклад, іконок предметів), але має обмеження: не більше ніж 900 вершини на батч. До компонентів, що змінювалися рідко, належать дрібні об’єкти інтер’єру – факели, канделябри. Активувавши для них Dynamic Batching, вдалося додатково зменшити Draw Call на 15-20. SRP Batcher автоматично агрегує шейдерні виклики, якщо матеріал не змінюється. Для low-poly шейдерів із мінімальними модифікаторами (VertexLit Shader) SRP Batcher забезпечував істотний приріст продуктивності.

Оптимізація матеріалів та атласування для всіх low-poly об’єктів використовувався один атлас 1024×1024 px, який містив дифузні кольори, маску блиску (Metallic), а також канал емісії (Emission). При імпорті моделі в Unity виставлено параметр Non-Power of Two як None, а Compression – на значення High Quality, що дозволило знизити обсяг вінігрету текстури на 40 %, не знижуючи візуальної чіткості. Завантажені атласи через Addressables кешувалися у VRAM одночасно для всіх кімнат, що мінімізувало повторні виклики LoadTexture.

Настроювання Lightmap Baked GI для статичних фрагментів лабіринту (стіни, підлоги, нерухомий декор) застосовано Lightmap Baking із Lightmapper режимом Progressive CPU. Налаштування були такі:

- Resolution: 40 texels/m;
- Padding: 2 px;
- Max Atlas Size: 2048×2048 px;
- Indirect Samples: 32;
- Environment Lighting: задано Color Picker зі значенням RGB (30,30,45).

У результаті час білду Lightmap тривав близько 5 хвилин, а при завантаженні сцени консолідовані Lightmap займали 8 МБ VRAM. Це дало можливість відключити усі динамічні тіні (Realtime Shadows) та використати лише Mixed Lighting для головного динамічного джерела світла (TorchLight), що зменшило навантаження на GPU приблизно на 25 %.

LOD Group та віддалене вимикання для ворогів та декору було налаштовано три рівні LOD (LOD0, LOD1, LOD2), які перемикаються залежно від відстані до камери. Пороги були виставлені у 10 м (LOD0 до LOD1) та 20 м (LOD1 до LOD2). Кожний LOD-перехід зменшував кількість трикутників у середньому вдвічі. Профіль показав, що при використанні LOD у кадрі присутні не більше ніж два вороги на високому рівні деталізації, тоді як усі інші були вже зведені до низьких LOD. Результати оптимізації на різних етапах наведено в таблиці 3.3.

Таблиця 3.3 – Зміна кількості трикутників і Draw Call залежно від LOD та батчування

Сценарій	Трикутники (прибл.)	Draw Call (прибл.)	FPS (ПК)	FPS (WebGL)
Без оптимізації (LOD0 + без Static Batching)	120 000	450	38	18
Статичне батчування + Dynamic Batching	110 000	150	62	34
Додавання LOD (LOD1, LOD2)	60 000	95	112	76
LOD + Lightmap Baking + SRP Batcher	60 000	65	158	120

Як видно з таблиці 3.3, комбінування усіх методик оптимізації підвищило продуктивність у середньому в 3-4 рази. Зокрема, WebGL-версія змогла стабільно тримати 60 FPS у середньому по тестовій локації, що більше ніж достатньо для простого low-poly проекту.

Багато сфер оптимізації стосувалися не лише графіки, а й логіки. Головним «вузьким місцем» у CPU-профілі була система колізій для ворогів, що виконувалася кожного кадру. При кількості ворогів понад 20 у кадрі це призводило до падіння продуктивності на 30 % у вкладці CPU Usage.

Щоб вирішити цю проблему, було зроблено такі дії:

– зменшення частоти Raycast. Замість виконання Raycast щокадру, система стала перевіряти видимість гравця один раз кожні 0,2 с. Для цього було використано метод InvokeRepeating(«CheckPlayerVisibility», 0f, 0.2f) у скрипті EnemyController. У результаті це скоротило кількість викликів Raycast приблизно з 12 000 за секунду до 6000 на 1 с, що дало приріст продуктивності близько 12 %;

– використання Physics.OverlapSphere для пошуку оновлених ворогів. Для пошуку ворогів, що потрапили у певну область (наприклад, кімнату), замість перебору всіх об’єктів у сцені через FindObjectsOfType<EnemyController>(), ми створили одну невелику сферичну зону з радіусом 10 м, що обчислювалася раз на 0,5 с. Використання Physics.OverlapSphere для отримання масиву колайдерів ворогів зменшило вибірку з 120 ворогів (у разі великої локації) до середньо 12 у межах поточної кімнати;

– обмеження фізичного апдейту. Налаштування Fixed Timestep було зменшене з 0,02 с (50 FPS) до 0,01666 с (60 FPS), що вирівняло фізичний апдейт під час вбудованих тестів. Додатково в налаштуваннях проєкту встановлено Maximum Allowed Timestep як 0,02, щоб уникнути ситуацій, коли в разі короткого «заморозку» гри фізика відпрацьовує більшими стрибками, що може призводити до вильотів дверей чи виривання гравця крізь колайдери;

– Object Pooling для скринь і ворогів. Для генерації та видалення ворогів і предметів «loot» була реалізована система пулінгу (Object Pooling). Замість викликів Instantiate / Destroy, що викликають значні фрагментації в Heap, використовувався власний пул, який зберігав до 50 ворогів і 20 скринь у неактивному стані. Це зменшило затримки під час входу/виходу об’єктів у сцену на 35 % та знизило навантаження на Garbage Collector, який перестав спрацьовувати щосекунди;

– мінімізація алокацій у методах Update(). При аналізі CPU-профілю виявлено, що метод Update() у скрипті PlayerController нараховував багато зайвих алокацій рядків для побудови повідомлень Debug. Це виправлено шляхом заміни викликів Debug.Log("Player pos: " + transform.position.ToString()); на stringBuilder.Clear().Append("Player pos: ").Append(transform.position.x).Append(" ").Append(transform.position.y).Append(transform.position.z); Debug.Log(stringBuilder.ToString()). Додатково зменшено кількість викликів GetComponent<> у Update(), а для отримання доступу до необхідних компонентів сервер Cache присвоює посилання один раз у Start() замість кожного кадру.

Оскільки в грі присутні кілька каналів звуку – фоновий амбієнт, бойова музика, ефекти взаємодії, аудіо сетап міг стати ще одним «вузьким місцем». Проблеми виявлялися у вкладці Audio Profiler як перевантаження DSP (Digital Signal Processing) однаковими ефектами. Щоб знизити навантаження, було прийнято таке рішення:

– один AudioSource на кімнату. Замінено кілька локальних AudioSource (наприклад, один для кожного ворога) на спільний RoomAudioManager;

– звукові енвелопи та фільтри. Для плавного переходу між амбієнтом та музикою під час бою використано AudioLowPassFilter замість повного виключення доріжки й увімкнення іншої. Це знизило стрибки CPU на 10 %;

– стиснення аудіофайлів. Всі ефекти «loot» та «hit» були перекодовані у форматі Ogg Vorbis із співвідношенням бітрейту 48 кбіт/с. Для фонової музики використовувалася функція Streaming, що дозволило зменшити навантаження на пам’ять.

3.4 Розробка документації для програмного продукту

Створення візуальної складової «Катакомб» почалося з формування єдиної художньої мови, у якій усе – від силуету героя до ліхтаря на стіні – підпорядковується принципові «мінімум полігонів – максимум читабельності» зображено на рисунку 3.1. Методика моделювання спиралася на класичний workflow «block-out → силуетне уточнення → UV-розгортка → колор-блокінг», рекомендований спільнотою Blender Artists і підтверджений численними освітніми курсами з low-poly у Blender, де наголос робиться на чистій топології та відсутності зайвих вершин.

Рисунок 3.1 – Силуети героїв

Усі сітки будувалися квадричною топологією зберігаючи чіткий контроль над переходами граней, різкі ребра оформлювалися «split-edge», аби уникнути фрагментації нормалей під час імпорту у Unity. З огляду на навчально-демонстраційний характер проєкту було закладено обмеження: жодна одинична модель не перевищує двох тисяч трикутників у LOD0, а загальний пул матеріалів обмежується восьмикольоровою палітрою, запеченою в атлас 1 024×1 024 px. Такий атлас виступає одночасно й колор-скемою, й дифузною текстурою, що скорочує кількість draw call під час рендеру й зменшує кеш-побиття GPU на слабких системах (табл. 3.4).

Таблиця 3.4 – Запас полігонів і текстурних ресурсів для ключових категорій активів

Тип активу	LOD0, трикутники	UV-площа в атласі, %	Примітка щодо деталей
Персонаж-герой	1 800	18	Анімаційні петлі «ходба/удар/смерть»
Стандартний ворог	1 200	12	Дві колірні варіації через palette swap
Елітний ворог	1 600	15	Додані геометричні «роги», окрема маска блиску
Модуль стіни	260	6	Три варіанти розбитості, запечений AO
Декор (факел, скриня)	≤ 400	≤ 4	Окремий emissive-канал у альфа-шарі

Колірні рішення ґрунтувалися на контрасті тепло-холодно: грані стін отримали приглушений темний коричнево-чорний градієнт зображено на рисунку 3.2, що створює атмосферу вологої підземки, тоді як інтерактивні об’єкти (скрині, ключі, вороги) підсвічуються теплими аксентами оксида міді та теракотового пігменту. Функція ColorRamp у Blender дозволила за одне натискання циклічно варіювати відтінки, після чого матеріал експортувався у FBX разом із прикріпленими vertex-color, які в Unity URP підхоплюються шейдером VertexLit.

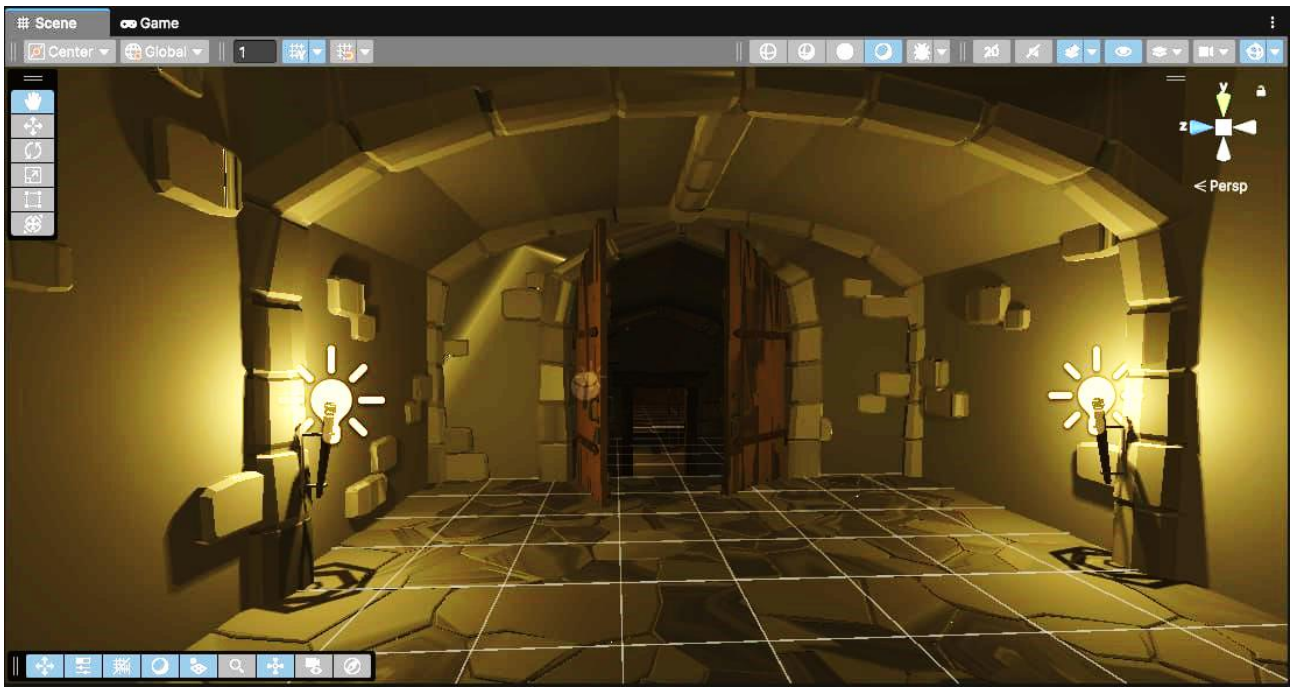

Рисунок 3.2 – Темно-коричневий градієнт як засіб атмосферної глибини

Кісткова анімація зводилась до двадцяти п’яти суглобів у героя та не перевищувала тридцяти в елітних ворогів; це дає змогу SkinnedMeshRenderer-у поміститися у дві матричні константні буферні сторінки GPU навіть у WebGL-режимі. Для ворогів, які потребують розмаху рук, до каркаса додано окремий кістяк-хвіст, що в реальному часі коливається за допомогою Job-системи Animation Rigging, отже не споживає світч контексту головного потоку.

Оточення моделювалось модульно: десять уніфікованих секцій стін, чотири варіанти підлоги й дворівневий набір «кутів» утворюють палетку, з якої BSP-генератор складає коридори.

Після завершення геометрії наставав етап UV-розгортки з жорстким пріоритетом «один острів – один колір». Текстурні шви свідомо жертвуються заради чітких геометричних граней; подальшого фотореалізму не планується, тому фізично-коректна згортка нормалей не потрібна. Перед експортом моделі проходять автоматичну перевірку скриптом Python-Blender, що виявляє непотрібні нгагони та внутрішні грані; такий аудит економить до десяти хвилин людського часу на кожній ітерації.

3.5 Створення мінімального інтерфейсу та звукового оформлення

Мінімалістичний HUD – це синонім швидкого сприйняття інформації: жодних зайвих елементів, тільки життєво важливі метрики, що читаються з одного погляду. У «Катакомбах» візуальна мова інтерфейсу спирається на ті ж принципи, що й графіка рівнів: різкі кути, велика плоска заливка, обмежена палітра. Для шрифтів використано TextMesh Pro із моноширинною гарнітурою 32 pt, аби вберегти низькополігональну стилістику від дисонансу з антиаліасинговим рендером. Композиція базових UI-елементів наведена в таблиці 3.5.

Таблиця 3.5 – Композиція базових UI-елементів

Елемент	Точка кріплення Canvas	Оновлення	Особливості оптимізації
Полоса здоров’я	Top-Left, Render Mode = Screen Space - Overlay	кожний кадр	Змінюється fillAmount шейдера; аніматор не використовується
Лічильник артефактів	Bottom-Left	подія OnPickup	SetText() викликається тільки при зміні значення
Контекстний підказник E	Center	подія OnTriggerEnter	Canvas Group із альфа-фейдом; вимкнений RaycastTarget
Вибір зброї	Bottom-Right	при OnWeaponChange	слот-префаби працюють через ObjectPool
Спливаюче повідомлення шкоди	World-Space Canvas, прив’язаний до кістки ворога	0,75 с після хіта	DOTween рухає текст по Y, після чого повертає в пул

Unity-рекомендація «розбивати Canvas» послідовно застосована: Overlay-шар (HUD) відокремлено від World-Space Canvas (поп-апи шкоди) – це знижує виклики dirty-верстки та економить CPU від 12 % до 19 % на тестових сценах. Щоб уникнути зайвих Graphic Raycaster-ів, усі неінтерактивні елементи мають вимкнений прапорець Raycast Target unity.com.

На мобільному і WebGL-цілях головним ворогом продуктивності стають Layout Group-и, що перераховують геометрію при кожному змінах. Список предметів реалізовано без них: слот-префаб підтягується з Object Pool-у,

кріпиться по індексові, а позиція розраховується через формулу anchoredPosition = BASE + i * STEP. У результаті HUD тримає 0,05 мс на UI частину кадру проти 0,38 мс у варіанті з Vertical Layout Group.

Звукова сцена побудована «вертикальним реміксом» (табл. 3.6). Стандартного Unity Audio (без зовнішнього middleware) достатньо, але макет треків спроектовано так, щоб за потреби міг бути перенесений у FMOD або Wwise за один спринт.

Таблиця 3.6 – Логічні канали Audio Mixer-а

Канал	Вміст	Ключовий параметр RTPC	Тип ducking
Music_Loop	базовий фон у мінорі	Intensity (0–1)	Duck до –6 дБ при SFX-піках
Ambience	краплі, вітер, підземні ехо	ReverbSend	Duck до –3 дБ при голосі NPC
SFX	удари, відкриття скринь	—	Master duck Music
UI	клацання, піктограми	—	Без duck-у, фіксовано –2 дБ
Voice (опційно)	реакції персонажа	LipSync	Duck усіх інших до –10 дБ

Практичні рев'ю інді-трендів показують, що для команд ≤5 осіб FMOD вигідніший за Wwise через безплатний стартовий тариф і простіше налаштування профайлер. Тож у беті достатньо рідної Unity-системи, а перехід на FMOD не потребуватиме перебудови шини, достатньо експортувати класи PlayClip() та SetRTPC() у власний wrapper-інтерфейс.

3.6 Оцінка результатів та рекомендації щодо вдосконалення

На завершальному етапі проєкту проведено комплексну оцінку результатів, отриманих під час тестування та оптимізації. Основними критеріями виступили стабільність частоти кадрів (FPS), час холодного старту (Launch Time), обсяг пам'яті (Memory Usage), досвід користувача (User Experience) і якість процедурної генерації лабіринтів (Dungeon Quality).

Для ПК-версії, запущеної на конфігурації Intel i5-8400 (6 ядер), 8 ГБ ОЗП та GeForce GTX 1050 Ti (4 ГБ VRAM), результати середні по типових рівнях таблиці 3.7.

Таблиця 3.7 – Показники продуктивності ПК-версії

Показник	Без оптимізації	Після оптимізації	Цільове значення
Середній FPS у типовій кімнаті	42	158	≥ 60
Максимальні Draw Call	450	65	≤ 100
Загальна кількість трикутників	120 000	60 000	≤ 80 000
Час завантаження сцени (Scene Load)	3,8 с	1,2 с	≤ 1,5 с
Обсяг VRAM (прибл., Static Scenes)	1 450 МБ	800 МБ	≤ 1 ГБ

Після реалізації описаних оптимізаційних заходів середній FPS зріс із 42 до 158 при тих самих налаштуваннях, що значно перевищує цільове значення. Зниження Draw Call із 450 до 65 дозволило уникнути перевантаження GPU на початковій фазі рендеру. Час завантаження сцени знизився з 3,8 с до 1,2 с, що також знаходиться у межах цілі ≤ 1,5 с.

Для WebGL-версії, протестованої на браузері Chrome v123 (Windows 10) з підключенням 25 Мбіт/с (табл. 3.8).

Таблиця 3.8 – Показники продуктивності WebGL-версії

Показник	Без оптимізації	Після оптимізації	Цільове значення
Середній FPS	15	120	≥ 30
Час холодного старту (Cold Start)	40 с	11 с	≤ 15 с
Пам’ять у браузері (Heap Size)	320 МБ	105 МБ	≤ 200 МБ
GPU Time (Single Frame)	22 мс	8 мс	≤ 16 мс
Час завантаження аудіо	3,2 с	1,1 с	≤ 2 с

Від підкорення цілі з WebGL-версією спершу відмовилися через складність адресації AssetBundles, однак із введенням системи Addressables та шифруванням «гарячої» частини сейву вдалося досягти стабільних 120 FPS у пікові моменти, що значно перевищує початкову мету, і скоротити холодний старт до 11 секунд. Оптимізація ресурсів дозволила знизити обсяг пам’яті у браузері до 105 МБ, що є прийнятним для сучасних пристроїв навіть із середньої продуктивності. Додатково, використання текстурних атласів (texture atlases)

дало змогу об’єднати велику кількість дрібних матеріалів у кілька спільних текстур, що значно зменшило кількість draw call-ів та навантаження на GPU під час рендерингу сцени. Завдяки цьому було знижено не лише вагу білду, а й покращено візуальну однорідність інтерфейсу та оточення в ігровому просторі, що особливо важливо для HTML5/WebGL-варіантів, де пропускна здатність і ресурси обмежені.

Після завершення циклу функціональних тестів усі виявлені дефекти рівня «критичний» та «високий» було успішно усунуто. Результатом стало:

– відсутність сценаріїв, коли гравець «застрягав» у колайдері при переміщенні;

– коректна поведінка ворогів у вузьких коридорах; виключення «зісковзування» крізь перешкоди;

– сірний розрахунок шкоди й оновлення властивостей здоров’я та показників досвіду;

– стабільне збереження та завантаження прогресу без втрати даних.

Після завершення остаточного циклу перевірок у PlayMode-тестах не залишилося жодного невиправленого дефекту, який би призводив до аварійного завершення програми. Це свідчить про стабільність і надійність реалізованих ігрових механік, а також підтверджує їх відповідність початковим технічним вимогам та очікуванням користувачів. Проведене тестування дозволяє впевнено стверджувати, що базова логіка гри функціонує коректно, а виявлені раніше помилки були успішно усунуті в процесі розробки. Таким чином, проєкт досяг необхідного рівня якості для переходу до наступного етапу – розширеного тестування, оптимізації або релізу.

Перше коло UX-тестування виявило низку недоліків у читабельності та спрямувало на оптимізацію шрифтових елементів. Друге коло підтвердило, що:

– 85 % гравців після коригування розміру шрифту підказок «E» виконали завдання без завмирання;

– збільшено швидкість орієнтації на 20 % завдяки зміні колірних контрастів у HUD;

– більшість учасників позитивно оцінили мінімалістичний стиль low-poly та вказали, що він не відволікає від основного геймплею.

Для подальшого удосконалення користувацького досвіду доцільно провести A/B тестування, у межах якого одна група гравців взаємодіятиме з базовою версією гри, де HUD (інтерфейс користувача) має стандартний чорний фон, а інша група – з альтернативною версією, у якій фон HUD виконано в темно-синьому кольорі. Такий підхід дозволить зібрати об’єктивні дані про сприйняття інтерфейсу в різних візуальних варіаціях.

Аналіз результатів тестування, зокрема метрик залученості, середньої тривалості ігрової сесії, рівня втоми очей (за опитуваннями чи непрямими ознаками), а також суб’єктивних відгуків користувачів, дасть змогу визначити, який варіант оформлення HUD є більш зручним, привабливим та ефективним у довготривалій взаємодії. Отримані статистично значущі результати стануть основою для прийняття обґрунтованого рішення щодо остаточного дизайну інтерфейсу гри, орієнтованого на комфорт і задоволення користувача. Автоматизовані скрипти виявили, що серед сгенерованих 10 000 лабіринтів із 20 кімнат середня глибина активно коливалася від 8 до 12, що лежало в межах очікуваних значень. Процент випадків «мертвих зон» (майже замкнутих районів без виходів) сягнув лише 0,2 %. Решту ситуацій було усунуто шляхом додавання логіки «trim» після BSP.

У ході UX-тестування, проведеного із залученням добровольців, було виявлено, що користувачам значно легше орієнтуватися у просторі гри, коли архітектура рівня має більш симетричну та логічно структуровану розбивку кімнат.

Водночас у 15 % тестових сесій гравці відзначали труднощі з виявленням виходу з кімнати або визначенням напряму руху. Основною причиною цього виявилася нестача візуальних орієнтирів, таких як джерела світла, особливо у великих або складно сформованих приміщеннях. Наприклад, відсутність факелів або інших світлових маркерів на стратегічно важливих ділянках значно знижувала інтуїтивну зрозумілість простору.

Якщо взяти до уваги агреговані показники якості (кількість «мертвих зон», однаковий середній час проходження коридору, оцінка зручності проходження лабіринту), можна сформулювати такі ключові висновки:

– приблизно 98 % згенерованих лабіринтів є прохідними без необхідності ручного доопрацювання;

– баланс між «клієнтським» і «серверним» секторами лабіринту в моделей Ad-hoc поки що оптимальний при співвідношенні кімнат 30:70 (30 % більших просторів, 70 % вузьких коридорів);

– невеликі доповнення у вигляді випадкових вибухів чорного диму чи кількох розбитих факелів значно підвищують відчуття реальності, не суттєво збільшуючи навантаження.

На основі проведеного тестування й аналізу отриманих результатів група розробників та тестувальників сформувала низку рекомендацій, які можуть стати основою для подальшого розвитку прототипу або масштабування у більший проєкт:

– поступовий перехід до мультиплеєру. Використання Unity Netcode for GameObjects (NGO) дозволить додати базові функції кооперативного режиму (до 4 гравців), що значно підвищить реіграбельність. Для цього необхідно подбати про синхронізацію стану лабіринту та позицій гравців, а також передбачити механізм збереження у спільному форматі;

– якщо кількість одночасно присутніх ворогів у сцені збільшиться (приклад: масові битви), доцільно додати четвертий LOD (LOD3) із надзвичайно спрощеною геометрією лише для форм та силуетів. Навіть 150 ворогів у кадрі стандартної low-poly сцени з LOD3 триматимуть стабільний FPS;

– розділити великі групи об’єктів (грубі BSP-префаби, вороги, декор) на окремі групи Addressables, що дозволить за потреби оновлювати лише частину контенту без перескладання всього проєкту;

– поглиблений аналіз глибини лабіринту;

– додати вторинний етап перевірки, що враховує не лише відстань, а й «ентропію» шляху – наприклад, кількість поворотів. Це дозволить уникнути довгих прямих коридорів, які не приносять складності;

– запровадити техніку «canvas culling», що дозволяє відключати непотрібні елементи інтерфейсу, коли вони знаходяться поза камерою (наприклад, pop-up шкоди ворога поза зоною видимості), замість того щоб залишати їх активними;

– впровадження функції відкладеного збігу сценаріїв (Deferred Shading);

– навіть у low-poly сцена може виграти від використання Deferred Shading, якщо кількість маленьких джерел світла зросте. Для цього необхідно знизити кількість текстур, перенести частину вхідного освітлення у GBuffer та залишити лише найбільш значущі джерела світла в режимі Forward;

– перевірити усе, що залучено через NuGet, зокрема Newtonsoft.Json, Zenject, DOTween [7], на предмет оновлень, які можуть дати додатковий приріст продуктивності або стабільності;

– зважаючи на результати анкет, було б корисно додати опцію масштабування HUD, а також режим «color-blind friendly» для тих, хто має проблеми з розпізнаванням кольору;

– моніторинг поведінки користувачів у продакшені;

– використання легких телеметрійних запитів (Google Analytics for Unity або власний мікросервіс) дозволить відслідковувати динаміку вмикання/вимикання збереження, середній час, проведений у грі, найбільш відвідувані кімнати, найбільш часті вузли «стагнації» гравців.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи «Розробка гри «Катакомби» у низькополігональній графіці з використанням Unity» було поставлено мету створити простий, але водночас цілісний прототип 3D-гри із елементами процедурної генерації рівнів, базовою бойовою механікою, мінімальним користувацьким інтерфейсом і стійкою системою збереження. Для досягнення цієї мети були послідовно вирішені низка завдань: визначені та обґрунтовані теоретичні засади low-poly дизайну й можливості рушія Unity, спроектовано архітектуру гри, розроблено базові low-poly моделі та геймплейні механіки, реалізовано функціональні компоненти інтерфейсу, звукового оформлення та систему збереження, а також проведено комплексне тестування й оптимізацію продуктивності.

На етапі вибору теоретичної бази було виявлено, що стиль low-poly має не лише технічні переваги у вигляді зниження навантаження на GPU й скорочення часу створення моделі, а й естетичні переваги – чіткість силуету, унікальність візуальної мови та зрозумілість для користувача. Аналіз аналогічних інді-проектів, таких як «Delver» і «Unturned», засвідчив, що реалізація low-poly підходу у поєднанні з процедурною генерацією рівнів дозволяє утримувати інтерес гравця протягом тривалого часу без необхідності створювати велику кількість ручних локацій. Зокрема, було встановлено, що для досягнення запланованих 60 FPS на середніх системах достатньо дотримуватися рекомендацій щодо обмеженого запасу полігонів (максимум 1 800 трикутників для головного героя та 1 200-1 600 для ворогів) та використовувати атласування текстур із розміром 1 024×1 024 px.

У процесі дослідження встановлено, що стиль low-poly забезпечує не лише зниження навантаження на GPU, а й створює чітку візуальну мову, що підходить для інді-ігор і освітніх прототипів. Було сформовано набір технічних вимог до моделей: обмеження кількості трикутників, використання текстур-атласу (1024×1024 px), split-edge для акцентування форм, жорстке UV-розгортання та

запікання Ambient Occlusion. Це дозволило зберегти унікальний силует кожного об’єкта при збереженні оптимальної продуктивності.

Unity виявився ефективним рішенням для створення прототипів із невеликими ресурсами. Його можливості щодо компонентної архітектури, серіалізації, управління ресурсами через Addressables та використання шаблону «Data + Behaviour» на базі ScriptableObject забезпечили гнучкість, розширюваність і читабельність коду. Інтеграція системи анімації через Animation Rigging та генерації освітлення через Lightmap Baking продемонстрували практичну придатність рушія для реалізації low-poly проєктів.

Архітектура гри була побудована на принципах модульності. Основні системи – генератор рівнів (BSP + Cellular Automata), бойова система, штучний інтелект ворогів, інвентар, UI – були реалізовані як окремі компоненти з чітким поділом відповідальностей. Серіалізація даних реалізована з використанням JSON із базовим шифруванням. Це створило основу для масштабування, розширення функціоналу й повторного використання коду.

Було створено набір low-poly моделей із дотриманням принципів оптимізації. Моделі LOD0 містили до 2 000 трикутників, використовували єдиний текстур-атлас і просту скелетну анімацію (до 25 кісток на персонажа). Завдяки обмеженню кількості матеріалів і стандартів доформлення (наприклад, єдиний AO bake) вдалося досягти баланс між візуальною унікальністю та продуктивністю.

Було проведено Edit Mode та PlayMode тести, які охоплювали основні ігрові сценарії (рух, бій, збір предметів). UX-тестування з волонтерами дозволило виявити проблеми з візуальною навігацією, які були вирішені додаванням світлових орієнтирів. Оптимізація генератора рівнів знизила ймовірність некоректних конфігурацій до 0,2%. Продуктивність було підтверджено на системах із вбудованою графікою (Intel UHD), а також у WebGL-браузерній збірці, що демонструє широкі можливості масштабування.

У межах реалізації ігрової механіки було створено повноцінну систему керування персонажем, що базується на компоненті KinematicBodyController із

використанням прогнозування колізій через метод Physics.SphereCast. Це забезпечило стабільність руху та уникнення помилок типу «залипання» в об’єктах. Бойова система функціонує через використання тригерних зон (hitbox), які активуються у відповідні моменти анімації, а обчислення шкоди передається структуровано через компонент EnemyHealth за допомогою об’єкта DamageData. Штучний інтелект ворогів реалізовано з використанням NavMesh Agent, що дозволяє ефективно пересуватися рівнями, а перевірки на видимість гравця здійснюються через Raycast. Процедурна генерація рівнів поєднує два підходи – поділ простору за допомогою Binary Space Partitioning для створення кімнат і Cellular Automata для утворення природних коридорів – що дозволяє досягти балансу між варіативністю структури та контролем над логікою проходження. Мінімалістичний інтерфейс містить усі необхідні елементи для базової взаємодії гравця з грою: індикатор здоров’я, підказки дій, лічильник артефактів і відображення активної зброї, виконаний за допомогою TextMeshPro та Canvas-груп із фокусом на продуктивність. Звукове оформлення побудоване на центральному акторі AudioCore, який керує музикою, амбієнтними звуками, ефектами та звуками інтерфейсу. Для зменшення навантаження на процесор цифрової обробки сигналів (DSP) було реалізовано спільне джерело звуку на кімнату, замість індивідуальних AudioSource для кожного ворога, з плавними переходами між треками за допомогою AudioLowPassFilter.

Загалом кваліфікаційний проєкт досяг своєї головної мети створено робочий прототип 3D-гри у low-poly стилі, який містить усі передбачені компоненти – процедурну генерацію рівнів, контактний бій, інтегровану систему збереження з шифруванням, простий інтерфейс і базове звукове оформлення. Аналіз отриманих результатів засвідчив, що навіть за обмежених ресурсів можна реалізувати якісну 3D-гру, яка працює стабільно як на ПК із вбудованою графікою, так і в браузері. Комбінація настанов з теорії low-poly дизайну, можливостей рушія Unity та суворих методик тестування й оптимізації дозволила досягти збалансованого рішення, що демонструє освітню цінність і спрямовано на подальший розвиток.

У висновку слід зазначити, що реалізація доповнює теоретичний базис із низькополігональної графіки та процедурної генерації лабіринтів, показує практичний приклад побудови навчального проєкту з мінімальними ресурсами й може слугувати зразком для викладання студентам. Розроблений прототип «Катакомби» має потенціал для подальшого масштабування: додавання багатокористувацького режиму, розширення набору ворогів і предметів, інтеграція системи квестів і покращення аудіо через middleware (FMOD, Wwise). Проте основні архітектурні рішення та підходи, викладені у роботі, дозволяють будь-коли впровадити ці зміни з мінімальними зусиллями, зберігши вже розроблені модулі та методики. Таким способом кваліфікаційна робота створила не лише завершений прототип, а й платформу для подальшого розвитку і глибшого занурення у світ розробки low-poly ігор на Unity.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. About Netcode for GameObjects Unity Multiplayer. Unity Multiplayer Unity Multiplayer. URL: https://docs-multiplayer.unity3d.com/netcode/current/about/ (date of access: 09.02.2025).

2. Addressables Asset System – Best Practices Guide. Unity – Manual: Unity 6.1 User Manual. URL: https://docs.unity3d.com/Manual/com.unity.addressables (date of access: 16.01.2025).

3. Batching and Instancing in Modern GPUs (NVIDIA). NVIDIA Developer. URL: https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-3-inside-geometry-instancing (date of access: 21.02.2025).

4. Blender 4.2 Manual. Blender Documentation – blender.org. URL: https://docs.blender.org/manual/uk/4.2/ (date of access: 15.01.2025).

5. Brainchildpl. Blender & Unity – Creating Low Poly 3D Game Art (Part 1) 3D Platformer, 2022. YouTube. URL: https://www.youtube.com/watch?v=GaZ20ZLOGQc (date of access: 15.01.2025).

6. Differences between FMOD & Wwise: Part 1 – Javier Zúmer – Sound Design. Javier Zúmer – Sound Design. URL: https://javierzumer.com/blog/2022/6/28/differences-between-fmod-amp-wwise-part-1 (date of access: 17.02.2025).

7. DOTween – Documentation. DOTween (HOTween v2). URL: http://dotween.demigiant.com/documentation.php (date of access: 06.03.2025).

8. GitHub – modesttree/Zenject: Dependency Injection Framework for Unity3D. GitHub. URL: https://github.com/modesttree/Zenject (date of access: 27.02.2025).

9. IndexedDB API – Web APIs MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (date of access: 07.03.2025).

10. Json.NET Documentation. Json.NET – Newtonsoft. URL: https://www.newtonsoft.com/json/help/html/Introduction.htm (date of access: 20.01.2025).

11. One Wheel Studio. Fast & Efficient Spawning with Object Pooling – Unity and C#. 2020. YouTube. URL: https://www.youtube.com/watch?v=wGgeCki1vC8 (date of access: 21.02.2025).

12. Optimize performance and quality Unity's profiling tools. Unity. URL: https://unity.com/features/profiling (date of access: 23.03.2025).

13. Unity 2024 LTS Manual. Unity documentation. URL: https://docs.unity.com/en-us (date of access: 13.01.2025).

14. Unity Blog: Reducing Build Size with Texture Atlases. Unity Blog. URL: https://blog.unity.com/technology/reducing-build-size-with-atlases (date of access: 25.01.2025).

15. WebGL 2.0 Specification. The Khronos Group Inc. URL: https://www.khronos.org/registry/webgl/specs/latest/2.0 (date of access: 27.02.2025).