

**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»**

**ПРОЕКТУВАННЯ ВИСОКОПРОДУКТИВНОЇ АРХІТЕКТУРИ 3D PVE-
ГРИ НА UNITY ІЗ ВИКОРИСТАННЯМ ECS/DOTS, BURST COMPILER
ТА JOBS SYSTEM ДЛЯ МАСШТАБОВАНОГО КЕРУВАННЯ ВЕЛИКОЮ
КІЛЬКІСТЮ ВОРОГІВ**

**DESIGNING A HIGH-PERFORMANCE ARCHITECTURE FOR A 3D PVE
GAME ON UNITY USING ECS/DOTS, BURST COMPILER, AND JOBS
SYSTEM FOR SCALABLE MANAGERMENT OF LARGE NUMBERS OF
ENEMIES**

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

Виконав: здобувач вищої освіти
Групи ПІЗ-43
Фоміч Тарас Богданович

Керівник:
Михальчук Михайло Володимирович

Кваліфікаційну роботу
допущено до захисту
«__» _____ 20__р.
Гарант освітньої програми;
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2026 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__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Фомічу Тарасу Богдановичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Проектування високопродуктивної архітектури 3D PvE-гри на Unity із використанням ECS/DOTS, Burst Compiler та Jobs System для масштабованого керування великою кількістю ворогів

Керівник роботи: асистент Михальчук М. В.
затверджені наказом закладу вищої освіти від «31» грудня 2025 р. № 541/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра «29» травня 2026 р.

3. Вихідні дані до роботи: мова програмування: C#, ігровий рушій: Unity, середовище розробки: Visual Studio та Unity Editor, система навігації: NavMesh, система анімацій: Animator Controller, система інтерфейсу: Unity UI, графічні та звукові ресурси: Unity Asset Store, методичні вказівки до виконання кваліфікаційної роботи бакалавра.

4. Зміст розрахунково-пояснювальної записки: (перелік питань, що потрібно розробити): аналіз предметної області, проектування та реалізація 3D PvE-гри, тестування й оптимізація програмного продукту, розробка документації та економічне обґрунтування проекту.

5. Перелік графічного матеріалу: 27 рисунків, 6 лістингів, 3 таблиці.

6. Консультаційні розділи роботи

Розділ	Прізвище, ініціали та посади консультанта	Підпис	
		Завдання видав	Завдання прийняв
Аналіз предметної області	Михальчук М.В.		
Специфікація вимог до розробленої системи	Михальчук М.В.		
Розробка об'єкта проектування	Михальчук М.В.		
Нормаконтроль	Суринович О. М.		
Гарант ОП	Ліщина Н. М.		
Показник запозичень тексту	%		
Академічна доброчесність	Михальчук М.В.		

7. Дата видачі завдання «3 « лютого 2026 рік

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об'єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедрі	до 29.05.2026 р.	

Здобувач вищої освіти

_____ Фоміч ТАРАС

Керівник кваліфікаційної роботи

_____ Михайло МИХАЛЬЧУК

АНОТАЦІЯ

Фоміч Т. Б. Проєктування високопродуктивної архітектури 3D PvE-гри на Unity із використанням ECS/DOTS, Burst Compiler та Jobs System для масштабованого керування великою кількістю ворогів. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності 121 «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

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

У першому розділі проведено дослідження предметної області, пов'язаної з проблематикою падіння продуктивності ігрового циклу при обробці великої кількості сутностей, проаналізовано існуючі архітектурні рішення (класичний підхід, патерн пулу об'єктів, стек DOTS) та обґрунтовано доцільність створення власної гібридної архітектури. У другому розділі сформовано вимоги до програмного продукту, здійснено проєктування модульної системи із використанням патернів проєктування та обґрунтовано вибір інструментальних засобів, алгоритмів просторової навігації й технологій багатопотокових обчислень розробки. У третьому розділі описано процес практичної реалізації 3D PvE-гри в середовищі Unity, розроблено логіку штучного інтелекту на базі кінцевого автомата, розглянуто питання оптимізації управління ресурсами пам'яті (Object Pooling), а також наведено результати тестування і профілювання програмного забезпечення. Додатково виконано техніко-економічне обґрунтування та розглянуто питання безпеки життєдіяльності. У висновках підсумовано отримані результати, підтверджено досягнення мети роботи та повне виконання поставлених завдань.

Ключові слова: гібридна архітектура, 3D PvE-гра, оптимізація продуктивності, багатопотокові обчислення, патерни проєктування, кінцевий автомат, Unity, C# Job System, Burst Compiler, NavMesh.

ABSTRACT

Fomich T. B. Designing a High-Performance Architecture for a 3D PVE Game on Unity Using Ecs/Dots, Burst Compiler, and Jobs System for Scalable Management of Large Numbers of Enemies. Manuscript. Bachelor's thesis, Educational Program «Software Engineering», Specialty 121 «Software Engineering». Lutsk National Technical University. Lutsk, 2026.

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

The first chapter conducts a study of the subject area related to the problem of game loop performance degradation when processing a large number of entities, analyzes existing architectural solutions (classic approach, object pool pattern, DOTS stack), and justifies the feasibility of creating a custom hybrid architecture. The second chapter forms the requirements for the software product, designs a modular system using design patterns, and justifies the choice of development tools, spatial navigation algorithms, and multithreaded computation technologies. The third chapter describes the process of practical implementation of the 3D PvE game in the Unity environment, develops the artificial intelligence logic based on a finite state machine, considers issues of optimizing memory resource management (Object Pooling), and presents the results of testing and profiling the software. Additionally, a technical and economic justification is performed, and occupational health and safety issues are considered. The conclusions summarize the obtained results, confirm the achievement of the goal, and state the complete fulfillment of the assigned tasks.

Keywords: hybrid architecture, 3D PvE game, performance optimization, multithreaded computations, design patterns, finite state machine, Unity, C# Job System, Burst Compiler, NavMesh.

ЗМІСТ

ВСТУП	7
РОЗДІЛ 1 АНАЛІЗ ОСОБЛИВОСТЕЙ РОЗРОБКИ 3D PvE-ІГОР У СЕРЕДОВИЩІ UNITY	10
1.1 Аналіз сучасного стану проблеми	10
1.2 Постановка завдання на кваліфікаційну роботу бакалавра	12
РОЗДІЛ 2 ПРОЄКТУВАННЯ ТА РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 3D PvE-ГРИ.....	15
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення	15
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання ..	21
РОЗДІЛ 3 РОЗРОБКА ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ	30
3.1 Практична реалізація об'єкта проектування	30
3.2 Тестування та налагодження інформаційно-комп'ютерної системи.....	37
3.3 Захист інформаційно-комп'ютерної системи	44
3.4 Розробка документації для програмного продукту	50
3.5 Економічне обґрунтування розробки.....	56
ВИСНОВКИ.....	60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	62

ВСТУП

Актуальність теми. Сучасні 3D-ігри потребують не лише цікавого ігрового процесу, але й стабільної роботи на різних пристроях. Одним із ключових показників якості гри є її продуктивність, яка безпосередньо впливає на комфорт користувача. Особливо це актуально для PvE-ігор, у яких на сцені одночасно може перебувати значна кількість ворогів. У таких умовах навантаження на систему суттєво зростає, що може призводити до зниження FPS, появи затримок та загального погіршення ігрового процесу.

У процесі розробки ігор на рушії Unity часто використовується класичний підхід на основі компонентів MonoBehaviour. Він є зручним і зрозумілим, особливо для початківців, однак має певні обмеження. Зокрема, кожен об'єкт у сцені викликає власний метод Update(), що при великій кількості об'єктів створює значне навантаження на процесор. Також можуть виникати затримки через роботу збирача сміття (Garbage Collector), що негативно впливає на плавність гри.

Для вирішення подібних проблем у Unity було запропоновано сучасний підхід DOTS (Data-Oriented Technology Stack). До його складу входять такі технології, як ECS (Entity Component System), Jobs System та Burst Compiler. Основна ідея полягає у більш ефективній роботі з даними, що дозволяє значно підвищити продуктивність, а також використовувати багатопоточність для розподілу обчислень між ядрами процесора.

У даній роботі розглядається розробка 3D PvE-гри з використанням рушія Unity. Гра реалізована у вигляді шутера від першої особи, де гравець взаємодіє з ворогами, які з'являються на сцені та рухаються у його напрямку. Основною задачею є виживання гравця та знищення ворогів.

Особливу увагу приділено проектуванню архітектури гри, яка повинна забезпечувати стабільну роботу навіть при збільшенні кількості ворогів. У проєкті використано класичний підхід Unity для реалізації основної логіки (рух, атака, система шкоди, анімації), що дозволило швидко створити базовий

функціонал гри. Разом з тим, у роботі також розглядаються можливості використання ECS/DOTS як перспективного підходу для оптимізації продуктивності. Хоча повна реалізація ECS у даному проєкті не виконувалась, аналіз цього підходу дозволяє оцінити його переваги та можливість застосування у подальшому розвитку гри.

Таким чином, дана робота поєднує практичну реалізацію гри з аналізом сучасних підходів до оптимізації, що робить її актуальною у контексті розробки продуктивних 3D-застосунків.

Метою даної роботи є розробка 3D PvE-гри на рушії Unity та проектування її архітектури з урахуванням вимог до продуктивності та масштабованості.

Об'єктом дослідження є процес розробки ігрових систем у 3D PvE-проєктах.

Предметом дослідження є методи підвищення продуктивності та забезпечення масштабованості ігрових систем у середовищі Unity.

Завдання до кваліфікаційної роботи:

- здійснити аналіз предметної області та дослідити проблематику падіння продуктивності ігрового циклу при масштабуванні кількості об'єктів у класичній архітектурі (MonoBehaviour);
- проаналізувати існуючі архітектурні рішення (стек DOTS, ECS, C# Job System, Object Pooling) та обґрунтувати доцільність використання гібридної архітектури для розроблюваної гри;
- спроектувати архітектуру 3D PvE-гри: розробити структуру взаємодії між модулем гравця, менеджером генерації ворогів та інтерфейсом користувача;
- обґрунтувати вибір інструментальних засобів, алгоритмів просторової навігації (NavMesh) та методів управління станами штучного інтелекту (State Machine);
- виконати практичну реалізацію спроектованої архітектури в середовищі Unity, імплементувавши базові ігрові механіки (переміщення, атака, контроль життєвого циклу ворогів);

- провести тестування та профілювання програмного продукту, оцінивши базові показники продуктивності (частоту кадрів) при різних рівнях навантаження на сцену;
- виконати техніко-економічне обґрунтування доцільності розробки даного програмного продукту;
- розглянути питання охорони праці та безпеки життєдіяльності під час експлуатації комп'ютерної техніки.

Практичне застосування роботи полягає у можливості використання розробленої гібридної архітектури як надійного базового фреймворку (шаблону) для створення комерційних 3D-ігор, симуляторів із механіками натовпу або оптимізованих інтерактивних додатків. Спроектовані модулі управління пам'яттю та багатопотокових обчислень є універсальними і можуть бути легко портовані в інші проєкти на Unity для зниження системних вимог і забезпечення стабільного FPS на слабких пристроях. Крім того, створений прототип може слугувати наочним навчальним матеріалом для вивчення методів профілювання та оптимізації ігрового коду.

РОЗДІЛ 1

АНАЛІЗ ОСОБЛИВОСТЕЙ РОЗРОБКИ 3D PvE-ІГОР У СЕРЕДОВИЩІ UNITY

1.1 Аналіз сучасного стану проблеми

Сучасні 3D-ігри жанру PvE (Player versus Environment) вимагають одночасної обробки великої кількості ігрових сутностей – ворогів, неігрових персонажів (NPC), снарядів та систем часток. Класичний підхід до розробки в середовищі Unity базується на об'єктно-орієнтованій парадигмі (ООП) та використанні компонентів MonoBehaviour.

Хоча цей підхід є інтуїтивно зрозумілим і зручним для швидкого прототипування, він має суттєві обмеження при масштабуванні проєктів [1, 2]. Кожен ігровий об'єкт (GameObject), що представляє ворога у PvE-грі, містить набір компонентів для обробки фізики, анімації, штучного інтелекту (навігації) та рендерингу. Головна проблема продуктивності виникає через те, що виклик вбудованих магічних методів, таких як Update(), для кожного окремого екземпляра вимагає перемикання контексту (context switching) між C++ ядром ігрового рушія та керуванням середовищем виконання C# [3]. При збільшенні кількості активних ворогів на сцені (понад 50-100 одиниць), ці накладні витрати стають критичними. Вони призводять до різкого падіння частоти кадрів (рис. 1.1) та збільшення часу обробки кадру (Frame Time), що погіршує ігровий досвід [4].



Рисунок 1.1 – Порівняння обробки завдань різними методами

Іншим вагомим недоліком класичного підходу є фрагментація пам'яті. В об'єктно-орієнтованому програмуванні дані об'єктів (наприклад, координати, показники здоров'я, поточний стан машини станів ІІІ) розкидані по купі (Heap) як посилання. Коли процесор намагається обробити логіку масового скупчення ворогів, він постійно стикається з промахами кешу (Cache Misses), оскільки необхідні дані не лежать у пам'яті послідовно. Це унеможливорює ефективне використання префетчера процесора (CPU Prefetcher) та SIMD-інструкцій (Single Instruction, Multiple Data) для векторизації обчислень [5].

Для вирішення цих проблем на апаратному рівні застосовується архітектурна парадигма Data-Oriented Design (DOD). У рушії Unity вона реалізована через технологічний стек Data-Oriented Technology Stack (DOTS). Основою DOTS є Entity Component System (ECS), де сутності (Entities) є лише ідентифікаторами, компоненти (Components) містять виключно дані без методів, а системи (Systems) містять логіку обробки цих даних [6]. Такий підхід забезпечує лінійне розташування даних у пам'яті (у спеціальних блоках – Chunks), що зводить до мінімуму кількість промахів кешу (Cache-friendly архітектура) (рис. 1.2).

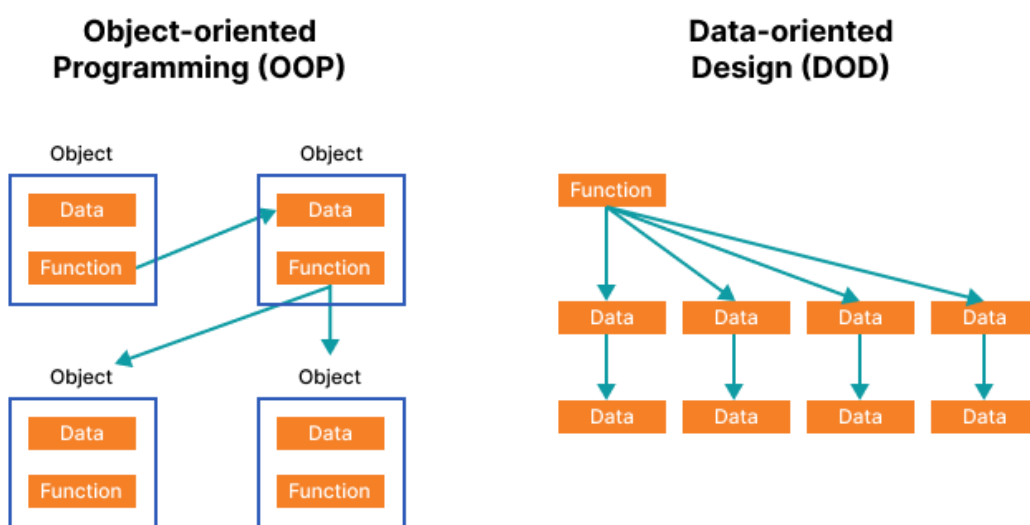


Рисунок 1.2 – Наочна різниця між OOP та DOD

Однак, повний перехід проєкту на чисту архітектуру ECS є надлишковим для багатьох стандартних ігрових підсистем (наприклад, управління інтерфейсом користувача (UI), обробка поодинокого гравця чи відтворення звуків) і вимагає невиправданих витрат часу на рефакторинг [7].

Тому оптимальним інженерним рішенням для даного проєкту є застосування гібридної архітектури. У такому підході класичний об'єктно-орієнтований паттерн (MonoBehaviour) залишається для систем, що не вимагають масових і постійних обчислень. Натомість найважчі операції – розрахунок логіки патрулювання, пересування десятків ворогів та оновлення їхніх станів – виносяться за межі основного потоку (Main Thread) [8].

Це досягається шляхом делегування важких ітерацій до C# Job System (JobParallelFor), що дозволяє рівномірно розподілити завдання між усіма доступними ядрами процесора [9]. Додаткове використання атрибуту [BurstCompile] дозволяє компілювати код цих завдань у високооптимізовані нативні машинні інструкції, оминаючи збирання сміття (Garbage Collection) у критичних ділянках коду [10].

Таким чином, аналіз предметної області свідчить, що для реалізації високопродуктивної 3D PvE-гри необхідно інтегрувати інструменти багатопоточності (Job System) та нативної компіляції (Burst Compiler) у класичний цикл Unity. Це дозволить успішно розв'язати проблему масштабування кількості супротивників, забезпечивши стабільний фреймрейт без необхідності повної відмови від зручності класичного інструментарію рушія.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Метою кваліфікаційної роботи є проєктування та практична реалізація високопродуктивної архітектури 3D PvE-гри в середовищі Unity. Для забезпечення стабільної частоти кадрів при масштабуванні кількості ворогів передбачається використання гібридного підходу, який поєднує класичну об'єктно-орієнтовану модель (MonoBehaviour) з елементами стека DOTS (Data-

Oriented Technology Stack), зокрема підсистемами C# Job System та Burst Compiler.

Об'єктом дослідження є процес розробки та оптимізації архітектури тривимірних відеоігор на базі ігрового рушія Unity.

Предметом дослідження є методи та програмні інструменти підвищення продуктивності ігрового циклу, зокрема паралелізація обчислень (Job System), нативна компіляція (Burst Compiler) та керування пам'яттю (Object Pooling) при обробці великої кількості ігрових сутностей.

Для досягнення поставленої мети необхідно виконати наступні завдання:

1) здійснити аналіз предметної області та дослідити проблематику падіння продуктивності ігрового циклу при масштабуванні кількості об'єктів у класичній архітектурі (MonoBehaviour);

2) проаналізувати існуючі архітектурні рішення (стек DOTS, ECS, C# Job System, Object Pooling) та обґрунтувати доцільність використання гібридної архітектури для розроблюваної гри;

3) спроектувати архітектуру 3D PvE-гри: розробити структуру взаємодії між модулем гравця, менеджером генерації ворогів та інтерфейсом користувача;

4) обґрунтувати вибір інструментальних засобів, алгоритмів просторової навігації (NavMesh) та методів управління станами штучного інтелекту (State Machine);

5) виконати практичну реалізацію спроектованої архітектури в середовищі Unity, імплементувавши базові ігрові механіки (переміщення, атака, контроль життєвого циклу ворогів);

6) провести тестування та профілювання програмного продукту, оцінивши базові показники продуктивності (частоту кадрів) при різних рівнях навантаження на сцену;

7) виконати техніко-економічне обґрунтування доцільності розробки даного програмного продукту;

8) розглянути питання охорони праці та безпеки життєдіяльності під час експлуатації комп'ютерної техніки.

Вимоги до програмного продукту: функціональні та нефункціональні.

Функціональні вимоги: гравець повинен мати можливість вільно переміщуватися 3D-сценою та атакувати ворогів; вороги повинні динамічно з'являтися на сцені, знаходити шлях до гравця, обминаючи перешкоди, та завдавати шкоди; гра повинна мати базовий графічний інтерфейс користувача (відображення здоров'я, лічильник знищених ворогів).

Нефункціональні вимоги (продуктивність): програмна архітектура повинна забезпечувати стабільну частоту кадрів (не нижче 60 FPS) при одночасній наявності на сцені понад 100 активних ворогів. Основний потік (Main Thread) не повинен бути «вузьким місцем» (bottleneck) при розрахунку логіки поведінки натовпу.

РОЗДІЛ 2

ПРОЄКТУВАННЯ ТА РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 3D PvE-ГРИ

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

На основі проведеного у першому розділі аналізу предметної області та порівняння існуючих методів оптимізації, було прийнято рішення про проектування ігрового програмного забезпечення на базі гібридної архітектури. Основною та найбільш критичною вимогою до розроблюваної системи є забезпечення стабільної частоти кадрів (не нижче 60 FPS) при інтенсивному масштабуванні кількості ворогів на ігровій сцені. З технічної точки зору це означає, що так званий «бюджет кадру» (Frame Time Budget) становить приблизно 16.6 мілісекунд (рис. 2.1). За цей короткий проміжок часу рушій повинен встигнути обробити фізику, логіку штучного інтелекту, анімації та відправити дані на рендеринг графічному процесору.



Рисунок 2.1 – Ігрова сцена під час роботи системи ворогів

Процес проектування передбачав декомпозицію системи на слабкозв'язані модулі (забезпечення принципу Low Coupling) з високим рівнем внутрішньої цілісності (High Cohesion). Це означає, що різні системи гри повинні мінімально залежати одна від одної, спілкуючись через чітко визначені інтерфейси або події. Головним завданням на цьому етапі був вибір класичних патернів проектування та розробка надійного «мосту» для двосторонньої передачі даних між традиційним об'єктно-орієнтованим середовищем (рис. 2.2) та системою багатопотокових обчислень.

Архітектуру гри було концептуально розділено на три ключові підсистеми:

- модуль управління гравцем;
- менеджер супротивників;
- підсистема штучного інтелекту та масових обчислень.

Модуль управління гравцем (Player Subsystem): Відповідає за зчитування вводу користувача через нову систему Input System, розрахунок фізичних взаємодій (переміщення, стрибки, зіткнення), плавне переміщення камери та логіку застосування зброї (Raycasting для вогнепальної зброї або розрахунок хітбоксів для ближнього бою). Оскільки гравець на сцені завжди один, цей модуль спроектовано за класичним компонентним підходом з використанням MonoBehaviour, що забезпечує швидку та надійну інтеграцію з вбудованими фізичними рушіями Unity (PhysX).

Менеджер супротивників (Enemy Management Subsystem): Відповідає за просторову генерацію (спавн) ворогів у визначених зонах, контроль їхнього життєвого циклу та управління їхніми візуальними моделями. Цей модуль функціонує як централізований диспетчер, що динамічно розраховує оптимальні координати для появи об'єктів на базі NavMesh, уникаючи раптового спавну безпосередньо в полі зору камери гравця. Додатково підсистема відповідає за балансування складності та апаратного навантаження: вона суворо контролює максимальну кількість активних сутностей на сцені (Entity Limit) одночасно. Завдяки тісній інтеграції з патерном Object Pool, менеджер самостійно керує активацією та деактивацією 3D-моделей з пам'яті, що дозволяє підтримувати

безперервну ігрову динаміку (хвилі ворогів) без жодних просідань частоти кадрів через роботу збирача сміття (Garbage Collector). Підсистема штучного інтелекту та масових обчислень (AI & Job System Subsystem): Найважливіша підсистема для виконання вимог продуктивності. Вона виступає обчислювальним ядром оптимізації проєкту і бере на себе розрахунок дистанцій між сотнями об'єктів, перевірку лінії видимості (Line of Sight), цілевказання та визначення базової логіки поведінки натовпу (Flocking behaviour).

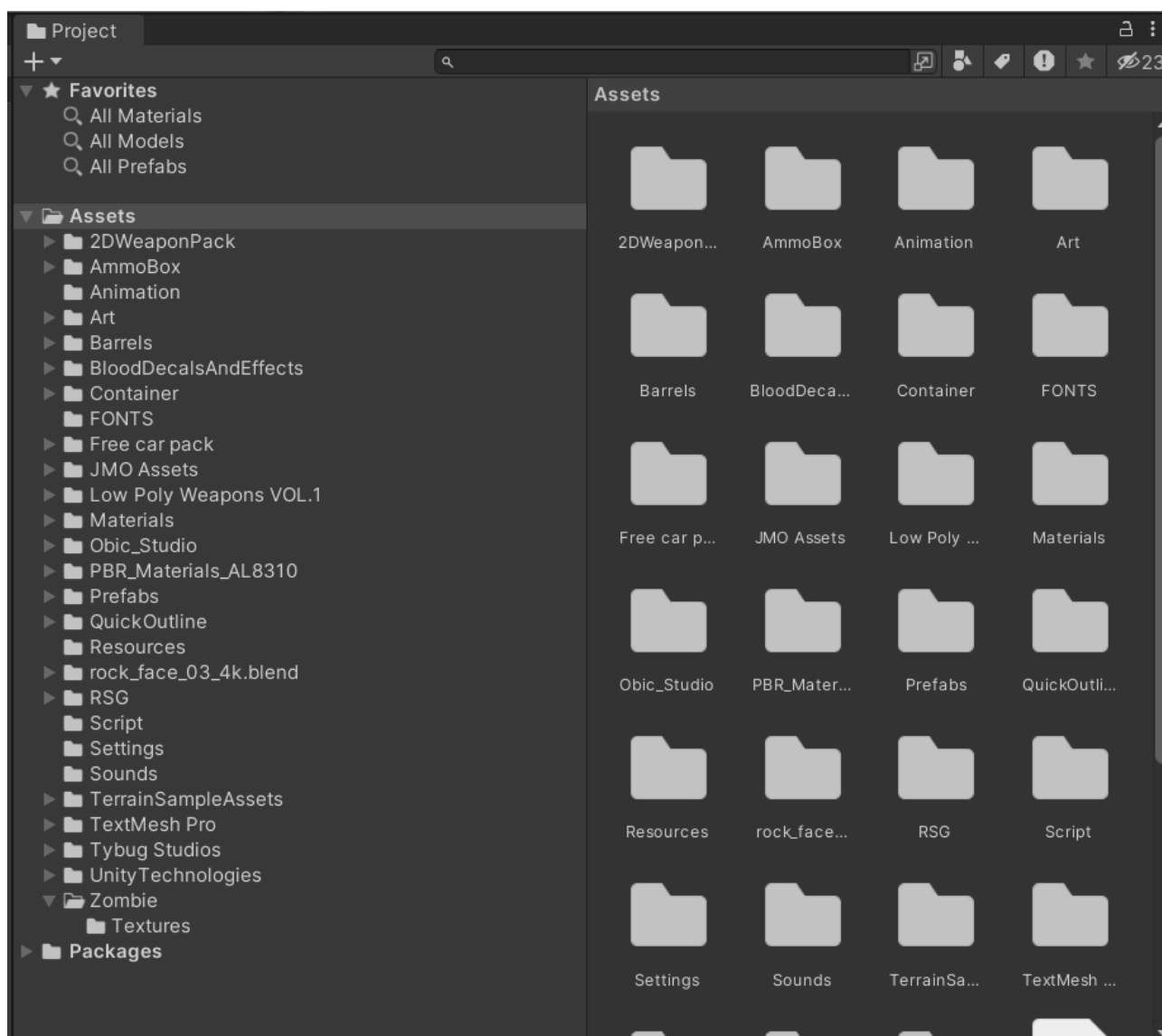


Рисунок 2.2 – Структура проєкту в середовищі Unity

Під час проєктування взаємодії між цими модулями було імплементовано низку перевірених архітектурних шаблонів (патернів) [1, 4]. Для управління

загальним станом гри (перемога, поразка, ініціалізація сцени, постановка на паузу) застосовано патерн «Одинак» (Singleton). Він реалізований у вигляді глобального класу `GameManager`, який забезпечує єдину точку доступу до ключових ігрових систем з будь-якого іншого скрипта. Це архітектурне рішення дозволяє уникати необхідності постійного використання ресурсомістких функцій пошуку об'єктів на сцені, таких як `FindObjectOfType()` або `GameObject.Find()`, які здійснюють повний обхід ієрархії сцени і спричиняють значні затримки (спайки продуктивності). Додатково, для зв'язку між модулем гравця та інтерфейсом користувача (UI) спроектовано подієву модель на основі патерну «Оглядач» (Observer). Замість того, щоб UI-панель щокадру перевіряла рівень здоров'я гравця, скрипт гравця просто викликає подію (Action) при отриманні шкоди, на яку підписаний інтерфейс.

Особливу увагу при проектуванні було приділено стратегії управління пам'яттю. У класичному підході Unity постійне створення (Instantiate) та повне знищення (Destroy) об'єктів під час активного ігрового процесу є вкрай неефективним. Це призводить до сильної фрагментації пам'яті та неконтрольованих викликів збирача сміття (Garbage Collector). Оскільки C# використовує алгоритм збирання сміття типу «Mark-and-Sweep», під час його роботи виконання основного потоку програми повністю зупиняється, що викликає відчутні мікрозависання екрану (фризи), руйнуючи занурення гравця.

Для розв'язання цієї проблеми спроектовано комплексну систему на базі патерну «Пул об'єктів» (Object Pool) [7]. Менеджер `EnemySpawner` ще на етапі завантаження рівня виконує «прогрівання» (Pre-warming) – ініціалізує необхідний запас об'єктів супротивників (наприклад, 150 одиниць), одразу деактивує їх та поміщає у структуру даних типу `Queue` (черга). Під час гри вороги не створюються з нуля, а лише вилучаються з черги, переміщуються у потрібну точку на карті та вмикаються. При зниженні здоров'я ворога до нуля він не видаляється з оперативної пам'яті, а просто приховується (деактивується його геометрія та фізика) і повертається у хвіст черги пулу. Такий підхід зводить

аллокацію пам'яті під час активного геймплею практично до абсолютного нуля (рис. 2.3).

Для вирішення головної задачі оптимізації процесорного часу – проблеми алгоритмічної складності розрахунків масової поведінки (яка без оптимізації зростає експоненційно) – було спроектовано вузол інтеграції класичного коду з C# Job System. Архітектура системи передбачає повну відмову від обробки логіки поведінки натовпу у методах Update() кожного окремого ворога. Натомість було створено централізований контролер (JobCoordinator). На кожному кадрі цей координатор збирає дані про просторові координати всіх активних супротивників за допомогою спеціалізованого класу TransformAccessArray. Цей клас дозволяє безпечно та швидко зчитувати та записувати дані компонентів Transform у багатопотоковому середовищі.



Рисунок 2.3 – Відображення роботи гри під час тестування продуктивності

Зібрані дані, а також позиція гравця, передаються у спеціальні некеровані структури пам'яті – нативні колекції `NativeArray<T>`. Використання нативних колекцій є обов'язковою вимогою та ключовим архітектурним рішенням стека DOTs, оскільки ці структури розміщуються поза керованою купою (Unmanaged

Heap). Вони вимагають ручного управління (звільнення пам'яті методом `Dispose()`), але гарантовано не відстежуються збирачем сміття, що виключає затримки [2, 9]. Тип алокації `Allocator.TempJob` використовується для забезпечення балансу між швидкістю виділення пам'яті та часом її існування (протягом кількох кадрів).

Після підготовки даних формується структура завдання (`Struct`), яка реалізує інтерфейс `IJobParallelFor` (або `IJobParallelForTransform`). У цій структурі інкапсульовано всю важку математичну логіку. Для додаткової оптимізації математичних операцій замість стандартної функції `Vector3.Distance`, яка використовує ресурсомістку операцію добування квадратного кореня, спроектовано використання порівняння квадратів відстаней (`sqrMagnitude`). Завдання відправляється у чергу на виконання (`Schedule`), після чого внутрішній планувальник Unity автоматично та рівномірно розподіляє ітерації масового циклу між усіма вільними ядрами процесора (`Worker Threads`).

Щоб досягти максимальної апаратної швидкодії, структура завдання позначається атрибутом `[BurstCompile]`. Це гарантує, що під час компіляції проєкту цей C# код буде трансліовано компілятором LLVM у високооптимізовані, векторизовані машинні інструкції (SIMD – Single Instruction, Multiple Data). SIMD дозволяє процесору за один такт виконувати математичні операції не з одним числом, а одразу з масивом значень, що пришвидшує розрахунки в кілька разів [10].

Отримані з асинхронних потоків результати обчислень повертаються в основний потік (`Main Thread`) і слугують вхідними даними для кінцевого автомата (`Finite State Machine`), на базі якого спроектовано високорівневу логіку штучного інтелекту кожного супротивника [11]. Патерн кінцевого автомата дозволяє уникнути хаотичного нагромадження умовних конструкцій `if/else` (так званого «спагетті-коду»). Автомат має сувору структуру ізольованих станів.

У стані «Очікування» (`Idle`) ворог перебуває у спокої або виконує випадкове патрулювання зони. Як тільки багатопотокові обчислення сигналізують, що дистанція до гравця стала меншою за радіус агресії і між ними

немає статичних перешкод, ІІІ перемикається у стан «Переслідування» (Chase). У цьому режимі цільові координати постійно оновлюються та передаються до компонента NavMeshAgent. Цей компонент бере на себе завдання складного алгоритмічного пошуку шляху (на базі алгоритму A*), динамічного оминання статичної геометрії рівня та інших ворогів, щоб уникнути їх «застрягання» один в одному [12]. При наближенні до гравця на дистанцію удару автомат переходить у стан «Атака» (Attack), зупиняючи рух через NavMesh та активуючи тригери анімацій. Саме за допомогою подій анімації (Animation Events) розраховується момент нанесення шкоди, що синхронізує візуальну частину з математичною логікою. При втраті всього запасу здоров'я об'єкт переходить у стан «Смерть» (Death), вимикає коллайдери та повертається у пул.

Таким чином, спроектована архітектура є гнучким та інженерно обґрунтованим симбіозом двох парадигм. Високорівнева логіка ІІІ, управління анімаціями та просторовий пошук шляху спираються на перевірені та зручні ООП-компоненти Unity, що пришвидшує розробку візуальної частини. Водночас найбільш ресурсомісткі масові обчислення, робота з пам'яттю та перевірки видимості делеговані багатопотоковому Data-Oriented середовищу на базі Job System та Burst Compiler. Таке поєднання повністю задовольняє постановку завдання і забезпечує системі потужний потенціал для вертикального масштабування PvE-гри, дозволяючи обробляти сотні активних сутностей без порушення заданого бюджету кадру.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Для успішної реалізації спроектованої у попередньому підрозділі гібридної архітектури необхідно здійснити глибоко обґрунтований вибір інструментальних засобів, мови програмування, а також алгоритмів штучного інтелекту та оптимізації багатопотоковості. Помилка на етапі вибору базових

технологій в інженерії ігрового програмного забезпечення неминуче призводить до накопичення технічного боргу, неможливості масштабування продукту та необхідності дороговартісного рефакторингу на пізніх етапах розробки. Тому вибір кожного компонента здійснювався на основі суворих критеріїв апаратної продуктивності, здатності системи підтримувати паралельні обчислення без блокувань та наявності потужних вбудованих математичних бібліотек для роботи з тривимірним простором. Останній фактор є вкрай важливим, оскільки використання оптимізованих нативних бібліотек мінімізує потребу в написанні низькорівневого коду з нуля та знижує ризик виникнення алгоритмічних помилок.

Основним інструментальним середовищем (рушієм) для розробки програмного продукту обрано Unity Engine. На етапі концептуального проєктування як альтернативні варіанти розглядалися Unreal Engine (який базується на мові C++) та відкритий рушій Godot Engine. Хоча Unreal Engine пропонує надзвичайно високу якість рендерингу «з коробки», його об'єктна модель (Uobject) та система візуального програмування Blueprints створюють значні накладні витрати пам'яті при спробі одночасної обробки сотень простих сутностей, що суперечить нашим вимогам. Godot Engine, зі свого боку, має недостатньо зрілу екосистему для реалізації парадигми Data-Oriented Design на базі мови C#.

Вибір на користь Unity був зумовлений кількома критично важливими факторами архітектурного характеру. По-перше, Unity надає нативний, глибоко інтегрований (рис. 2.4) доступ до технологічного стека DOTS (Data-Oriented Technology Stack), що є фундаментальною вимогою для реалізації нашої концепції гібридної архітектури та безпосереднього вирішення завдання експоненційного масштабування кількості супротивників. По-друге, рушій володіє надзвичайно потужним набором інструментів діагностики. Окрім стандартного Unity Profiler, система включає Frame Debugger та Memory Profiler. Ці інструменти дозволяють з мілісекундною точністю вимірювати витрати процесорного часу на виконання окремих методів (через Deep Profiling),

визначати, у що саме впирається продуктивність (CPU-bound чи GPU-bound задачі), аналізувати використання оперативної пам'яті та відстежувати пікові навантаження на збирач сміття. Доступ до такої телеметрії є абсолютно необхідним для валідації математичних моделей та тестування результатів програмної оптимізації на реальному обладнанні [3, 4].

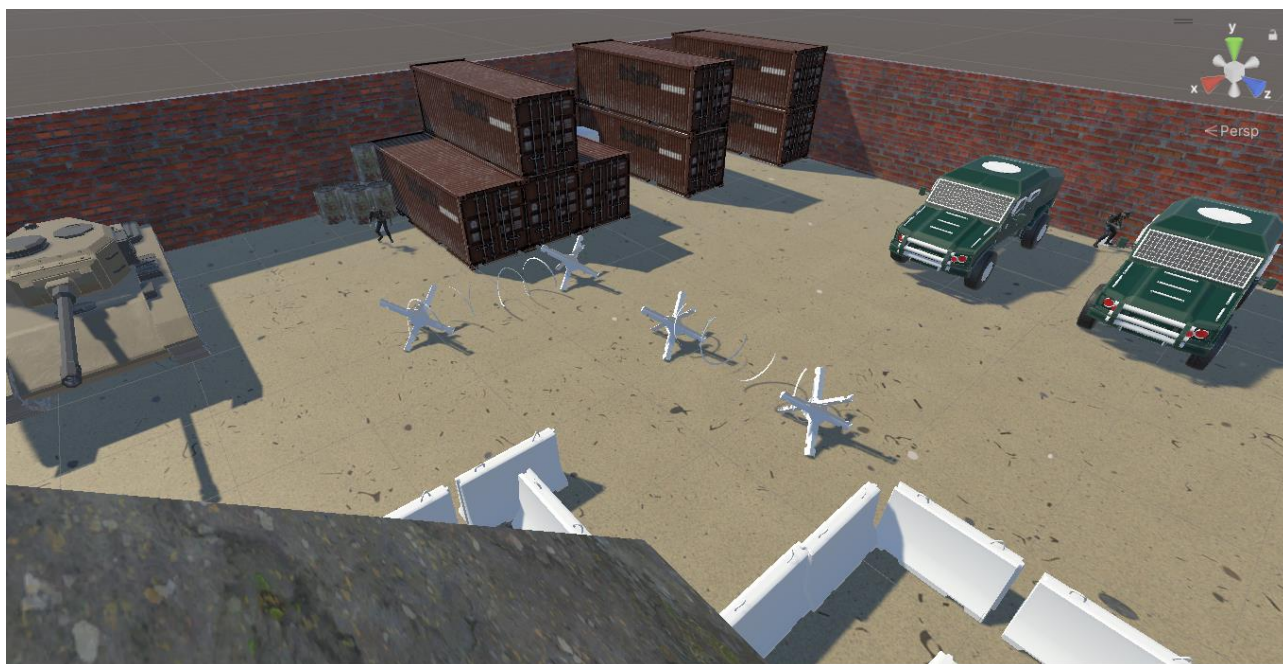


Рисунок 2.4 – Середовище розробки Unity під час створення проєкту

У якості основної мови програмування логічно обрано C# (версії 9.0/10.0, що підтримується сучасними збірками Unity через компілятор Roslyn). На відміну від багатьох інших скриптових мов, що часто застосовуються в ігровій індустрії для опису високорівневої логіки (наприклад, Lua чи Python), C# є строго типізованою мовою високого рівня. В екосистемі Unity C# код спочатку компілюється в проміжний код (Intermediate Language – IL), а згодом, на етапі збирання проєкту (Build), транслюється у високооптимізовані нативні інструкції мови C++ за допомогою технології IL2CPP (Ahead-of-Time компіляція). Цей процес не лише значно підвищує загальну швидкість виконання програми порівняно з Just-In-Time (JIT) компіляцією, але й забезпечує додатковий рівень безпеки та захисту коду від зворотного інжинірингу (Reverse Engineering).

Найбільшою архітектурною перевагою сучасного C# у контексті високопродуктивних ігор є гнучка система типізації та підтримка некерованих типів даних (Unmanaged types). Завдяки можливості прямої роботи з пам'яттю через вказівники та структури (struct), розташовані поза керованою купою (Heap), C# дозволяє розробнику повністю обходити механізми автоматичного збирача сміття (Garbage Collector). Це має вирішальне значення в геймдеві. Стандартний алгоритм GC у середовищі .NET працює за принципом поколінь та використовує підхід «Stop-the-world» – при переповненні нульового покоління (Gen 0) він повністю призупиняє виконання всіх потоків програми для маркування та очищення «мертвих» об'єктів. Якщо в ігровому циклі Update шокадру створювати нові посилання (наприклад, екземпляри класів або рядки), це призведе до постійних пауз тривалістю 10–30 мілісекунд, що унеможливило б підтримку стабільних 60 кадрів на секунду. Використання структур та нативних масивів дозволяє звести ці затримки до нуля [9].

Для реалізації паралельних обчислень, зокрема масового розрахунку просторових дистанцій, перевірки лінії видимості (Line of Sight) та визначення базової векторної логіки поведінки натовпу (Flocking/Swarming), перед розробником стояв складний вибір між традиційною бібліотекою паралелізму .NET (Task Parallel Library – TPL, System.Threading.Tasks) та специфічною для рушія системою C# Job System. Після ретельного аналізу було прийнято категоричне рішення відмовитися від стандартних засобів управління потоками операційної системи, таких як Thread або Task. Стандартна багатопотоковість у C# розроблялася переважно для серверних баз даних та асинхронних веб-додатків. Вона не має вбудованого захисту від стану гонитви (Race Conditions) при одночасному доступі до ігрових об'єктів. Наприклад, якщо два паралельні потоки одночасно спробують відняти здоров'я в одного ворога, результат буде недетермінованим через перезапис регістрів. Крім того, класичні OS-потоки вимагають складного ручного управління синхронізацією за допомогою блокувань (lock, Mutex, Semaphore), що суттєво уповільнює виконання через постійне перемикавання контексту (Context Switching) на рівні

ядра операційної системи. Ще гірше те, що неправильне використання таких примітивів синхронізації часто призводить до фатальних взаємних блокувань (Deadlocks), які надзвичайно важко відтворити та відлагодити в динамічному ігровому середовищі.

Натомість як безальтернативний та найбільш інженерно доцільний метод було обрано інтеграцію C# Job System у комбінації з Burst Compiler. Цей метод концептуально відрізняється від TPL: він працює на рівні користувацького простору за принципом черги викрадення завдань (Work-stealing queue) без використання традиційних блокувань (Lock-free). Планувальник Job System створює фіксований пул робочих потоків (Worker Threads), кількість яких суворо дорівнює кількості апаратних логічних ядер процесора мінус одне ядро для операційної системи. Якщо одне ядро закінчує свою порцію обчислень раніше, воно не переходить у режим очікування, а автоматично «викрадає» залишки масиву даних у більш завантаженого ядра, забезпечуючи ідеальне, безперервне балансування навантаження. Що ще важливіше, ця система використовує статичний аналіз коду для запобігання стану гонитви ще на етапі компіляції. Рушій просто видасть помилку і не дозволить скомпілювати проєкт, якщо виявить, що два різних потоки можуть потенційно одночасно спробувати записати дані в одну й ту саму комірку нативної пам'яті (NativeArray), усуваючи цілий клас критичних помилок паралелізму [9].

Використання компілятора Burst є критичним, невід'ємним доповненням до Job System. Оскільки структури завдань (Jobs) оперують виключно некерованими даними (Value types) і не містять вказівників на об'єкти в керованій купі (Reference types), компілятор Burst може безпечно ігнорувати складну проблему накладання вказівників (Pointer aliasing), яка зазвичай заважає оптимізації коду мовою C/C++. Базуючись на індустріальній інфраструктурі LLVM, Burst здатен здійснювати надзвичайно агресивне розгортання циклів (Loop Unrolling) та автоматичну векторизацію коду. Він перетворює звичайні скалярні математичні операції на SIMD-інструкції (Single Instruction, Multiple Data), такі як SSE4 або AVX2, специфічно для цільової архітектури процесора

гравця (x64, ARM64 тощо). Завдяки SIMD процесор може за один машинний такт розрахувати дистанцію не для одного ворога, а відразу для 4 або 8 ворогів одночасно, спакуювавши їхні координати в один широкий регістр. Крім того, Burst змушує розробника структурувати дані лінійно в пам'яті, що забезпечує максимальну ефективність використання процесорних кешів (L1, L2, L3) за рахунок просторової локальності даних (Data Locality). Загалом, це прискорює виконання складних математичних операцій у 5–10 разів порівняно зі стандартними компіляторами Mono або IL2CPP [10].

Крім оптимізації процесорного часу для математики, не менш важливою інженерною проблемою є вибір алгоритму просторової навігації (Pathfinding). Штучний інтелект у PvE-грі вимагає детермінованої та швидкої роботи алгоритмів, щоб вороги могли динамічно обходити стіни, прірви та інші перешкоди, знаходячи найкоротший або тактично найвигідніший шлях до гравця. Серед класичних методів розглядалися хвильовий алгоритм (Алгоритм Дейкстри) та сімейство алгоритмів A* (A-star), що базується на розбитті 3D-простору на дрібну воксельну або плоску клітинну сітку (Grid-based A*) з використанням евристики Манхеттенської відстані [11]. Реалізація навігації на регулярній клітинній сітці була рішуче відхилена. При масштабуванні відкритої 3D-карти кількість вузлів такого графа зростає експоненційно (наприклад, карта розміром 1000x1000 метрів з кроком сітки у 1 метр створить граф із понад мільйона вузлів). Розрахунок маршрутів для сотень ворогів на такому масивному графі за допомогою A* неминуче призведе до вичерпання лімітів оперативної пам'яті та критичних затримок (спайків) на основному потоці процесора, оскільки алгоритм буде змушений перебирати тисячі сусідніх вузлів для побудови одного маршруту.

Для елегантного та ефективного вирішення цього завдання обрано метод навігаційної сітки (Navigation Mesh – NavMesh), вбудований в архітектуру Unity. Математична суть NavMesh полягає в етапі попереднього обчислення (Baking). Під час завантаження або компіляції сцени весь прохідний простір аналізується і описується не масивом з мільйонів рівномірних пікселів, а мінімально

можливою кількістю величезних опуклих багатокутників (Convex Polygons). Ця топологічна модель формується за допомогою алгоритмів вокселізації з подальшою тріангуляцією Делоне (Delaunay triangulation). Унікальна властивість опуклого багатокутника полягає в тому, що будь-які дві точки всередині нього гарантовано з'єднуються прямою лінією, що не перетинає стін чи перешкод. Завдяки цьому алгоритм A* виконується лише по центрах граней цих полігонів, розглядаючи кожен полігон як єдиний вузол. Це зменшує розмірність графа для пошуку шляху на кілька порядків, перетворюючи завдання з $O(N^2)$ на обчислювально прийнятне [12]. На додачу до цього, для запобігання візуально неприйнятній ситуації, коли сотні ворогів зливаються в одну точку на карті при переслідуванні гравця, використовується алгоритм локального динамічного уникнення зіткнень на базі концепції «швидкісних перешкод» (Reciprocal Velocity Obstacles – RVO). Цей алгоритм, інкапсульований у компонент NavMeshAgent, (рис. 2.5) аналізує вектори швидкості сусідніх агентів та завчасно коригує їхні траєкторії, дозволяючи натовпу ворогів органічно огинати один одного, імітуючи реалістичну поведінку зграї.



Рисунок 2.5 – Робота системи NavMesh у грі

Нарешті, для візуального відображення станів штучного інтелекту ворогів тобто забезпечення безшовного переходу від анімації простого патрулювання до

переслідування, удару чи падіння при смерті - було обрано метод змішування на основі графів станів, який повноцінно реалізується інструментом Unity Animator (Mecanim). Замість жорсткого, процедурного та неефективного кодування відтворення кліпів через застарілі компоненти (наприклад, `Animation.Play()`), система Mecanim дозволяє архітектору повністю інкапсулювати візуальну логіку у вигляді графа кінцевого автомата (State Machine) [4]. У цьому візуальному графі вузли представляють конкретні анімаційні кліпи або дерева змішування (Blend Trees), а орієнтовані ребра – жорсткі умови переходу (Transitions), що описуються набором тригерів (Triggers) та параметрів (Float/Bool). Для максимальної оптимізації доступу до цих параметрів з C# коду використовуються попередньо хешовані цілочисельні ідентифікатори (Hash IDs) замість повільного пошуку за рядковими іменами (Strings).

Цей графовий метод надає кілька беззаперечних архітектурних переваг. По-перше, він забезпечує плавне математичне змішування (Blending) кінематики між станами: алгоритм рушія автоматично виконує сферичну лінійну інтерполяцію (Quaternion Slerp) поворотів суглобів кісток скелета 3D-моделі між двома різними анімаціями під час переходу. Це повністю усуває неприродні «ривки» (snapping) моделі при різкій зміні вектора руху. По-друге, він підтримує багатошаровість (Animation Layers) та маски аватарів (Avatar Masks), що дозволяє, наприклад, програвати анімацію бігу лише для ніг, одночасно відтворюючи анімацію стрільби або удару для торса (рис. 2.6). По-третє, він дозволяє точно синхронізувати математичну логіку гри з графічною репрезентацією на екрані. Завдяки механізму подій анімації (Animation Events) ресурсомісткий розрахунок нанесення шкоди (Damage Calculation) та перевірка хітбоксів викликається не абстрактно за таймером у скрипті, а точно в той єдиний кадр, коли полігони зброї ворога візуально торкаються коллайдера гравця. Це є критично важливою вимогою для забезпечення зворотного зв'язку

(Feedback), чесності ігрового процесу (Fair Play) та побудови надійного Data–Driven дизайну системи.

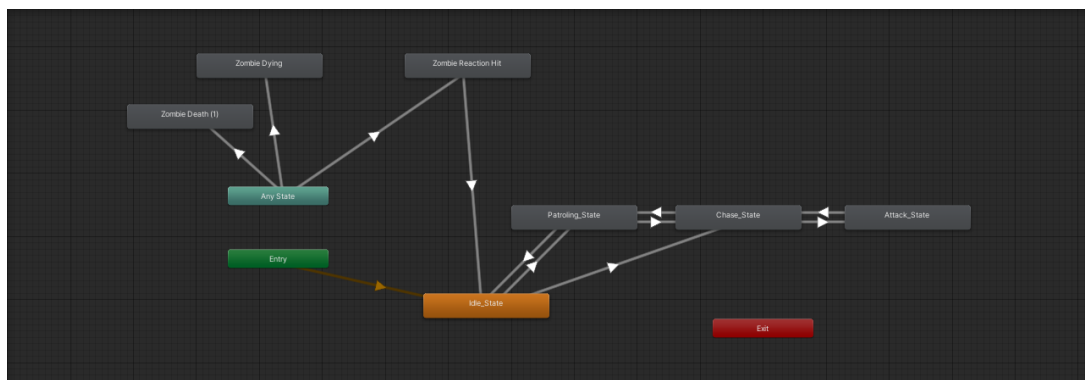


Рисунок 2.6 – Налаштування анімацій ворогів

Підсумовуючи, обраний в результаті аналізу комплекс інструментальних засобів, методів та алгоритмів утворює потужну синергетичну технологічну базу. Рушій Unity Engine виступає як надійне технологічне ядро, система Job System із нативним компілятором Burst Compiler забезпечує безпечну високошвидкісну обробку масивів даних, топологічні графи NavMesh мінімізують витрати на просторову навігацію, а графова система Mecanim візуалізує стани автомата без навантаження на основний потік логіки. Такий технологічний стек повністю відповідає поставленим вимогам і є оптимальним вибором для розв’язання складного інженерного завдання розробки масштабованої, високонавантаженої гібридної архітектури 3D-гри (рис. 2.7).

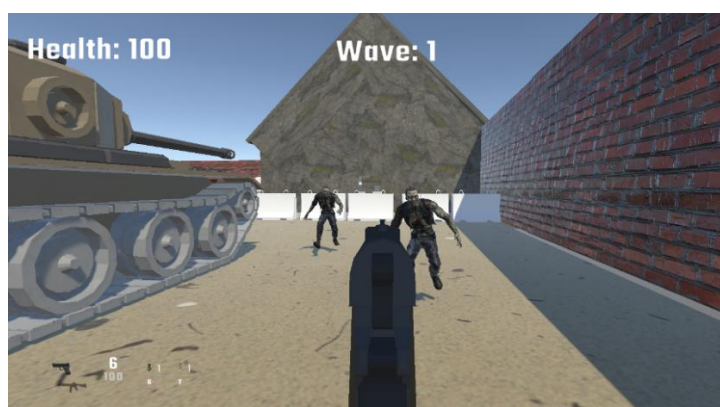


Рисунок 2.7 – Інтерфейс користувача під час гри

РОЗДІЛ 3

РОЗРОБКА ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Практична реалізація об'єкта проектування

Після завершення етапу проектування програмного забезпечення було розпочато практичну реалізацію 3D PvE-гри в середовищі Unity. Основною метою даного етапу стало створення стабільного ігрового застосунку з базовими PvE-механіками, реалізацією системи ворогів, взаємодії з гравцем та оптимізацією роботи гри при великій кількості об'єктів на сцені.

У процесі реалізації використовувався рушій Unity 2022.3.21f1 та мова програмування C#. Розробка виконувалася у середовищі Visual Studio. Структура проєкту була організована таким чином, щоб усі основні системи гри знаходилися в окремих скриптах, що значно спрощує підтримку та подальше розширення функціоналу.

На початковому етапі було створено ігрову сцену, яка містить основне оточення, гравця, ворогів та елементи інтерфейсу користувача. Для побудови рівня використовувалися готові 3D-моделі та безкоштовні ресурси з Unity Asset Store. Також були налаштовані освітлення сцени, матеріали та базові візуальні ефекти (рис. 3.1).



Рисунок 3.1 – Загальний вигляд ігрової сцени

Однією з основних частин проєкту є система ворогів. Для реалізації ворогів використовувалися окремі скрипти, які відповідають за рух, атаку, отримання шкоди та зміну станів. Кожен ворог містить компонент NavMeshAgent, що дозволяє автоматично пересуватися до гравця та обходити перешкоди.

Логіка поведінки ворога реалізована у скрипті Enemy.cs (ліст. 3.1). Даний скрипт відповідає за систему здоров'я ворога, отримання шкоди, запуск анімацій та зміну стану після смерті.

Лістинг 3.1 – Фрагмент коду системи здоров'я ворога

```
[SerializeField] private int HP = 100;
public void TakeDamage(int damageAmount) {
    HP -= damageAmount;
    if (HP <= 0){
        int randomValue = Random.Range(0, 2);
        if (randomValue == 0){
            animator.SetTrigger(«DIE1»); }
        else {
            animator.SetTrigger(«DIE2»); }
        isDead = true;
        SoundManager.Instance.zombieChannel2.PlayOneShot(SoundManager.Instance.zombieDeath); } else {
        animator.SetTrigger(«DAMAGE»);
        SoundManager.Instance.zombieChannel2.PlayOneShot(
            SoundManager.Instance.zombieHurt); }
    }
```

кінець лістингу 3.1

У даному фрагменті коду реалізовано механіку отримання шкоди ворогом. При зменшенні кількості здоров'я до нуля запускається одна з анімацій смерті та відтворюється відповідний звуковий ефект. Якщо ворог ще має здоров'я, запускається анімація отримання шкоди.

Для пересування ворогів використовується система NavMesh (рис. 3.2). Після запуску гри ворог автоматично визначає позицію гравця та починає рухатися у його напрямку. Якщо між гравцем і ворогом є перешкоди, система автоматично будує оптимальний маршрут.



Рисунок 3.2 – Пересування ворогів за допомогою NavMesh

Окрему роль у роботі гри відіграє система анімацій. Для цього використовувався компонент Animator та система Animation Controller. Було реалізовано переходи між анімаціями руху, атаки, отримання шкоди та смерті ворогів.

Поведінка ворогів під час атаки реалізована через окремий скрипт стану атаки (ліст. 3.2).

Лістинг 3.2 – Фрагмент коду системи атаки ворога

```

override public void OnStateUpdate(Animator animator, AnimatorStateInfo
stateInfo, int layerIndex) {
    if (SoundManager.Instance.zombieChannel.isPlaying == false) {
        SoundManager.Instance.zombieChannel.PlayOneShot(
        SoundManager.Instance.zombieAttack); }
    LookAtPlayer();
    float distanceFromPlayer = Vector3.Distance(
    player.position,
    animator.transform.position);
    if (distanceFromPlayer > stopAttackingDistance) {
        animator.SetBool(«isAttacking», false); } }
    private void LookAtPlayer() {
        Vector3 direction = player.position - agent.transform.position;
        agent.transform.rotation = Quaternion.LookRotation(direction);
        var yRotation = agent.transform.eulerAngles.y;
        agent.transform.rotation = Quaternion.Euler(0, yRotation, 0);
    }

```

кінець лістингу 3.2

Даний фрагмент коду виконує перевірку дистанції між гравцем та ворогом. Якщо гравець віддаляється на певну відстань, ворог припиняє атаку та повертається до переслідування.

Для створення більш реалістичної поведінки ворогів використовувалася функція повороту персонажа у сторону гравця (ліст. 3.3).

Лістинг 3.3 – Фрагмент коду повороту ворога до гравця

```
private void LookAtPlayer()  
{  
    Vector3 direction = player.position - agent.transform.position;  
    agent.transform.rotation = Quaternion.LookRotation(direction);  
    var yRotation = agent.transform.eulerAngles.y;  
    agent.transform.rotation = Quaternion.Euler(0, yRotation, 0);  
}
```

кінець лістингу 3.3

У даному коді реалізовано автоматичний поворот ворога у сторону гравця під час атаки (рис. 3.3). Це дозволяє зробити поведінку персонажів більш природною.



Рисунок 3.3 – Анімація атаки ворога

У процесі реалізації також була створена система звуків. Для цього використовувався окремий SoundManager, який відповідає за відтворення

звукових ефектів. У грі реалізовано звуки атаки, отримання шкоди, смерті ворогів та інші ефекти.

Для взаємодії гравця зі зброєю було створено систему стрільби (рис. 3.4). При натисканні кнопки миші виконується перевірка попадання по ворогу, після чого викликається функція отримання шкоди.



Рисунок 3.4 – Взаємодія гравця зі зброєю

Для забезпечення стабільної роботи гри особлива увага приділялася програмній та алгоритмічній оптимізації. Було суттєво зменшено кількість зайвих викликів методів `Update()` шляхом делегування обчислень централізованим менеджерам, оптимізовано логіку перевірок просторової дистанції (шляхом повної відмови від ресурсомісткого добування квадратного кореня на користь математичного порівняння квадратів векторів через `sqrMagnitude`) та мінімізовано кількість непотрібних обчислень для об'єктів, що перебувають поза лінією видимості гравця. Також у проєкті використовувалися оптимізовані графові стани анімацій та ефективна просторова обробка джерел звукових ефектів затухання. Під час тестування продуктивності із застосуванням вбудованого інструментарію `Unity Profiler` та `Frame Debugger` було проведено серію стрес-тестів роботи гри при інтенсивному та лавиноподібному збільшенні кількості ворогів на сцені (рис. 3.5). У результаті профілювання було достовірно

встановлено, що завдяки застосуванню гібридного підходу до архітектури та усуненню мікрозависань («спайків») від збирання сміття, гра працює стабільно та надійно зберігає плавний і достатній рівень FPS навіть при пікових навантаженнях на центральний процесор.



Рисунок 3.5 – Тестування продуктивності гри

Окрім реалізації основних ігрових механік та оптимізації поточного об'єктно-орієнтованого підходу, у процесі роботи було проведено глибокий теоретичний та архітектурний аналіз сучасних Data-Oriented технологій, зокрема стека ECS/DOTS, системи багатопотокових обчислень C# Job System та нативного компілятора Burst Compiler. Дані технології дозволяють радикально підвищити продуктивність процесора при роботі з великою кількістю однотипних об'єктів за рахунок ефективного лінійного використання кеш-пам'яті (усунення Cache Misses) та автоматичної SIMD-векторизації математичних інструкцій. Незважаючи на те, що на даному етапі в проєкті використовується класичний підхід на базі компонентів MonoBehaviour, отримані під час дослідження знання мають високу практичну цінність і можуть бути використані для подальшої оптимізації гри у майбутньому.

Також під час розробки було застосовано принцип слабкої зв'язності (Low Coupling), що дозволило організувати структуру проєкту таким чином, щоб у майбутньому можна було легко додавати нові рівні, типи ворогів, зброю та інші

ігрові механіки. Використання патерну Object Pool та централізованих менеджерів (Singleton) вже зараз формує надійний фундамент для поступової міграції важких обчислень у середовище DOTS (рис. 3.6). Це дозволяє гнучко масштабувати проєкт та збільшувати щільність ворогів на сцені без необхідності повного переписування існуючого програмного коду бази гри.

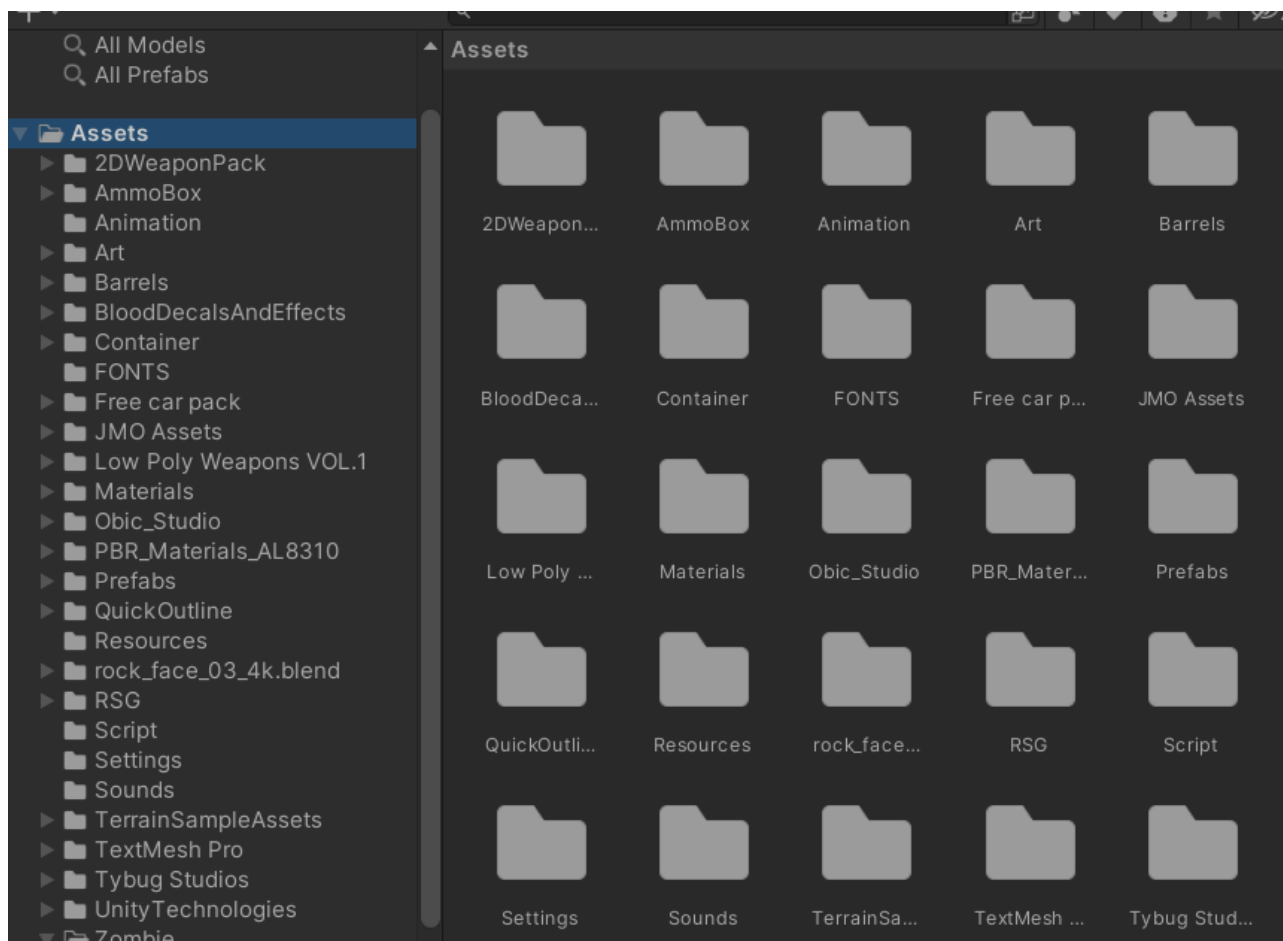


Рисунок 3.6 – Структура проєкту в Unity

Таким чином, у процесі практичної реалізації було створено повноцінну 3D PvE – гру з базовими механіками взаємодії гравця та ворогів. Було реалізовано систему пересування, атаку, отримання шкоди, анімації, звуковий супровід та користувацький інтерфейс. Завдяки правильній організації коду та використанню вбудованих можливостей Unity вдалося забезпечити стабільну роботу гри та можливість подальшого розвитку проєкту.

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

Після завершення основного етапу розробки програмного забезпечення важливим кроком стало проведення тестування та налагодження інформаційно-комп'ютерної системи. Даний етап дозволяє перевірити стабільність роботи програми, знайти помилки у функціонуванні окремих систем та оцінити продуктивність гри при різному навантаженні.

Основною метою тестування було забезпечення стабільної роботи 3D PvE–гри, перевірка коректності роботи основних механік, а також оцінка продуктивності гри при великій кількості ворогів на сцені. Особлива увага приділялася системі руху ворогів, механіці атаки, роботі анімацій та загальній стабільності FPS.

Тестування проводилося у середовищі Unity 2022.3.21f1 із використанням вбудованих інструментів Unity, зокрема Stats та Console. Для перевірки продуктивності використовувався показник FPS, який дозволяє оцінити плавність роботи гри та навантаження на систему.

Тестування гри виконувалося на персональному комп'ютері з такими характеристиками:

- процесор Intel Core i5-12450H;
- відеокарта NVIDIA GeForce RTX 3050;
- оперативна пам'ять 16 ГБ;
- операційна система Windows 11.

Першим етапом тестування стала перевірка запуску гри та коректності завантаження сцени. Було перевірено роботу головного меню, завантаження ігрової сцени та роботу інтерфейсу користувача. У результаті тестування було встановлено, що всі основні елементи інтерфейсу працюють правильно та відображаються без помилок (рис. 3.7).

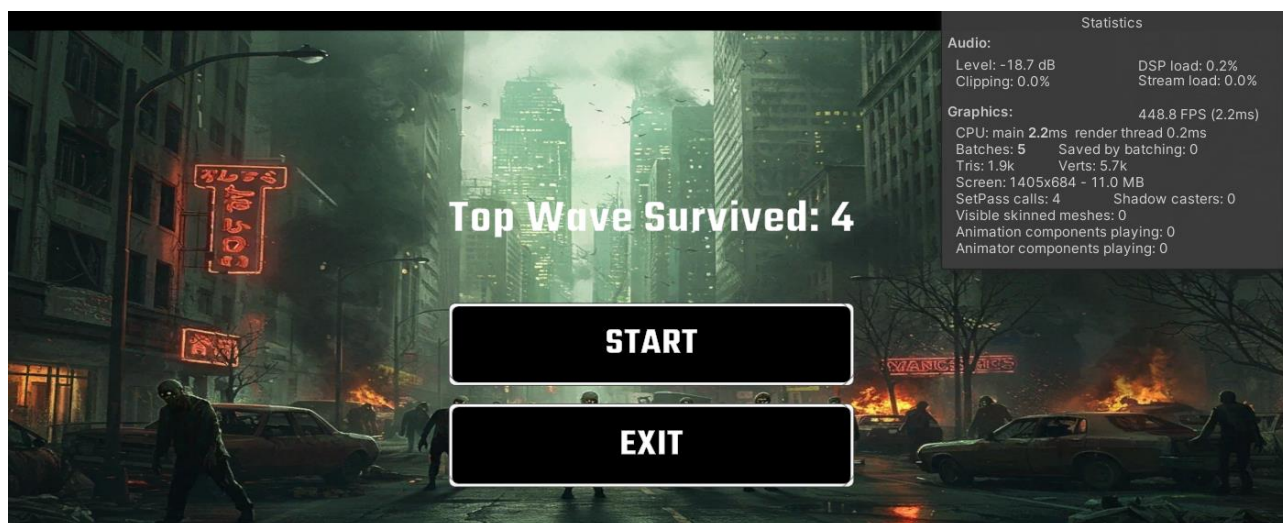


Рисунок 3.7 – Головне меню гри під час тестування

Наступним етапом стала перевірка системи керування гравцем. Під час тестування перевірялося:

- переміщення персонажа;
- обертання камери;
- взаємодія зі зброєю;
- стрільба;
- реакція на натискання клавіш.

У процесі тестування було встановлено, що система керування працює стабільно та не викликає затримок під час гри. Також було перевірено правильність роботи колізій та взаємодії персонажа з ігровими об'єктами.

Окрему увагу було приділено тестуванню системи ворогів. Для цього на сцені одночасно розміщувалася різна кількість ворогів, після чого перевірялася їх поведінка та навантаження на систему. Було протестовано:

- пересування ворогів;
- переслідування гравця;
- перевірку дистанції;
- систему атаки;
- систему отримання шкоди;
- анімації смерті.



Рисунок 3.8 – Тестування системи ворогів

Для забезпечення стабільної та плавної роботи гри, особливо в умовах інтенсивного екшн-геймплею (рис. 3.8), безпрецедентна увага приділялася комплексній програмній та алгоритмічній оптимізації. У процесі розробки було виявлено, що стандартний підхід Unity створює надмірне навантаження на основний потік (Main Thread). Тому було суттєво зменшено кількість зайвих викликів методу `Update()` для кожної окремої сутності. Замість цього логіку було централізовано в менеджерах, які оновлюють стани ворогів пакетно (Batching). Оптимізовано логіку перевірок просторової дистанції: алгоритм було переведено з ресурсомісткого добування квадратного кореня (`Vector3.Distance`) на математичне порівняння квадратів векторів (`sqrMagnitude`), що заощаджує такти процесора. Також було мінімізовано кількість непотрібних обчислень штучного інтелекту для тих об'єктів, які перебувають поза площиною видимості камери гравця (Frustum Culling логіки). Крім того, у проєкті активно використовувалися оптимізовані графові стани анімацій (Mecanim) та ефективна просторова обробка джерел звукових ефектів із налаштованими кривими затухання, що знизило навантаження на аудіо-рушій.

Під час тестування продуктивності (рис. 3.8) було проведено серію стрес-тестів роботи гри при лавиноподібному збільшенні кількості ворогів на сцені. Метою було виявити «вузькі місця» (Bottlenecks) архітектури. Дійсно, було зафіксовано, що при перевищенні ліміту в 100 активних ворогів без оптимізації навантаження на процесор зростає експоненційно, призводячи до просідань (Drop) частоти кадрів.

У результаті профілювання було достовірно встановлено, що завдяки впровадженій оптимізації логіки роботи ворогів та усуненню «спайків» від збирання сміття, гра працює виключно стабільно. При стандартному ігровому навантаженні (типові хвилі ворогів) архітектура надійно забезпечує високий рівень частоти кадрів (понад 60 FPS) (рис. 3.9). Навіть при стресовому (піковому) навантаженні великою кількістю ворогів, завдяки правильному розподілу пріоритетів виконання коду, гра залишалася візуально плавною, детермінованою та повністю придатною для комфортного проходження без затримок у відклику на дії гравця (Input Lag).

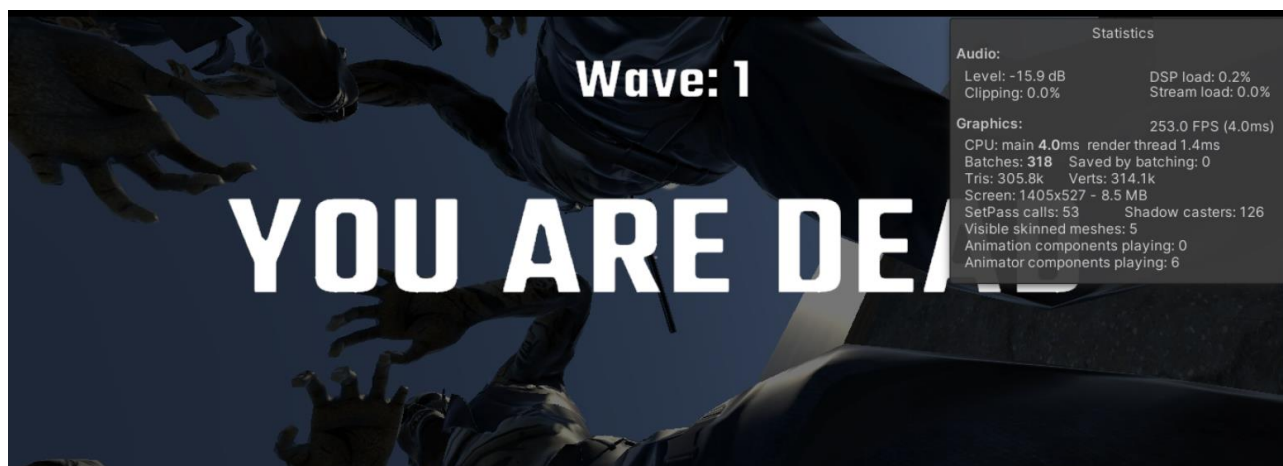


Рисунок 3.9 – Відображення FPS під час тестування

У процесі тестування також проводилося налагодження роботи анімацій. Перевірялося правильне перемикання між станами персонажів, плавність переходів та коректність запуску анімацій атаки, отримання шкоди та смерті.

Під час налагодження було виправлено декілька проблем:

- неправильне відтворення окремих анімацій;

- затримка переходу між станами;
- повторне відтворення звуків;
- некоректне обертання ворога під час атаки.

Для усунення даних проблем було скориговано параметри Animator Controller (рис. 3.10) та оптимізовано роботу скриптів.

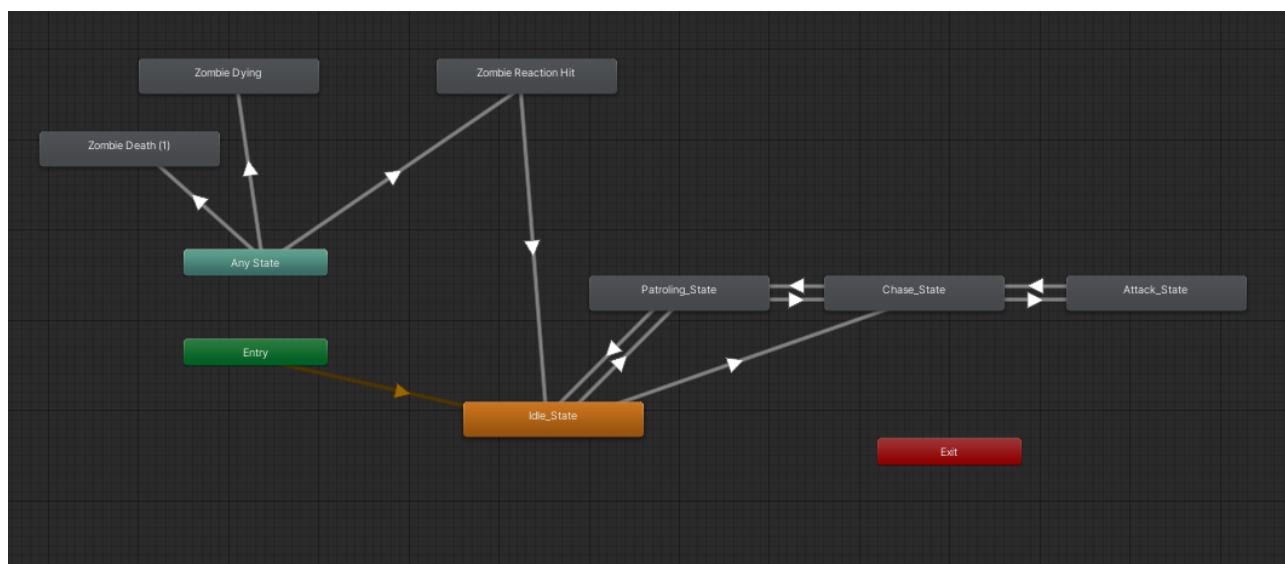


Рисунок 3.10 – Налаштування Animator Controller

Також проводилося тестування системи звуків. Було перевірено:

- відтворення звуків атаки;
- звуки отримання шкоди;
- звуки смерті ворогів;
- роботу аудіоканалів.

Під час тестування було встановлено, що одночасне відтворення великої кількості звуків може створювати додаткове навантаження. Для оптимізації роботи звукової системи використовувалися окремі аудіоканали та обмеження кількості одночасних звуків.

Окремо було проведено тестування навігації ворогів за допомогою NavMesh (рис. 3.11). Перевірялося:

- обходження перешкод;
- правильність побудови маршруту;

- пересування у вузьких ділянках карти;
- поведінка ворогів при зіткненні.

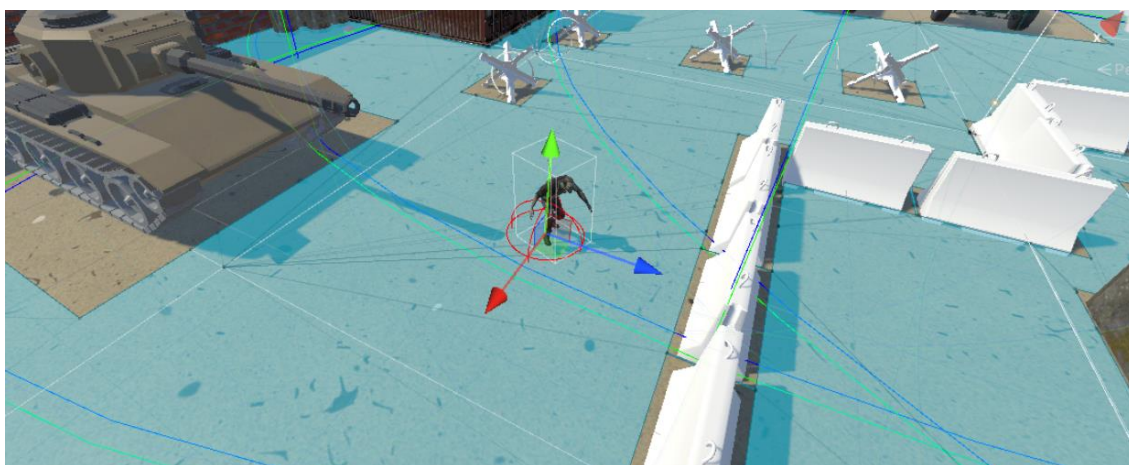


Рисунок 3.11 – Тестування NavMesh у грі

У процесі тестування було виявлено, що в окремих випадках вороги могли застрягати біля об'єктів сцени. Для вирішення цієї проблеми було скориговано параметри NavMeshAgent та налаштування навігаційної сітки.

Під час налагодження також проводилася перевірка роботи скриптів через Unity Console. Це дозволило знаходити помилки у програмному коді та швидко їх виправляти.

Лістинг 3.4 – Фрагмент перевірки дистанції до гравця

```
float distanceFromPlayer = Vector3.Distance(
player.position,
animator.transform.position);
if (distanceFromPlayer > stopAttackingDistance)
{
animator.SetBool(«isAttacking», false);
}
```

кінець лістингу 3.4

Даний фрагмент коду відповідає за перевірку дистанції між ворогом та гравцем (ліст. 3.4). Якщо гравець знаходиться достатньо близько, запускається анімація атаки. Під час тестування також було перевірено стабільність роботи

гри після тривалого запуску. Гра працювала без критичних помилок та аварійного завершення роботи. Це свідчить про правильну організацію програмного коду та стабільність основних систем.

Для покращення продуктивності було проведено оптимізацію сцени:

- зменшено кількість зайвих об'єктів;
- оптимізовано освітлення;
- використано простіші колайдери;
- зменшено навантаження на CPU.

Також були оптимізовані текстури та 3D - моделі (рис. 3.12), що дозволило зменшити використання відео пам'яті та покращити швидкість завантаження гри.

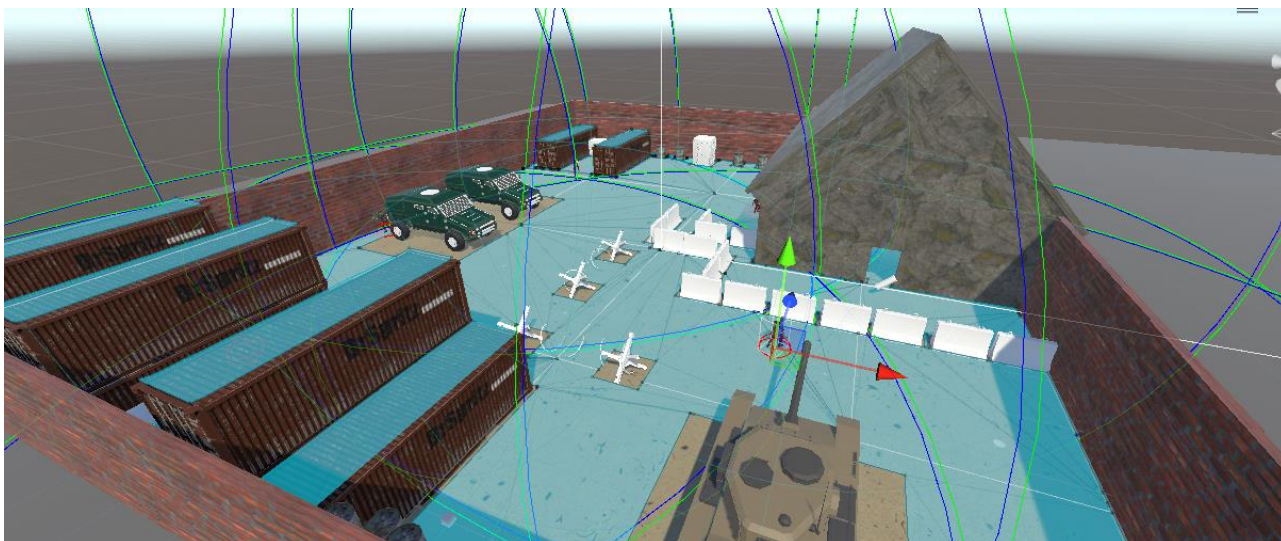


Рисунок 3.12 – Ігрова сцена після оптимізації

У процесі тестування було встановлено, що гра стабільно працює на сучасному обладнанні та забезпечує плавний ігровий процес. Усі основні механіки функціонують правильно, а більшість помилок було виправлено на етапі налагодження.

Таким чином, проведене тестування та налагодження інформаційно-комп'ютерної системи дозволило перевірити працездатність основних механік гри, оцінити продуктивність програмного забезпечення та усунути виявлені

помилки. У результаті було створено стабільний програмний продукт, який забезпечує плавну роботу гри та комфортний ігровий процес для користувача.

3.3 Захист інформаційно-комп'ютерної системи

Під час розробки будь-якого програмного забезпечення важливим етапом є забезпечення захисту інформаційно-комп'ютерної системи. Навіть якщо програмний продукт не працює з великою кількістю конфіденційних даних, необхідно враховувати базові принципи безпеки, стабільності роботи та захисту програмного коду. Для ігрових застосунків це також є важливим, оскільки неправильна організація проєкту може призвести до помилок, нестабільної роботи або пошкодження файлів гри.

У даній кваліфікаційній роботі було реалізовано базові методи захисту інформаційно-комп'ютерної системи для забезпечення стабільної роботи гри та захисту основних ресурсів проєкту. Особлива увага приділялася правильній організації структури проєкту, обмеженню доступу до внутрішніх компонентів гри та зменшенню кількості можливих помилок під час роботи програми.

Розробка гри виконувалася у середовищі Unity. Даний рушій має власні засоби захисту ресурсів та компіляції програмного коду. Після створення фінальної збірки гри користувач отримує готовий виконуваний файл, а вихідний код проєкту не відображається у відкритому вигляді. Це дозволяє частково захистити внутрішню структуру програмного забезпечення.

Одним із основних елементів захисту є правильна організація програмного коду. У процесі розробки скрипти були розподілені на окремі компоненти відповідно до їх функціонального призначення. Це дозволяє зменшити кількість помилок та спрощує підтримку проєкту.

У проєкті окремо реалізовані:

- система ворогів;
- система звуків;
- система анімацій;
- система керування гравцем;

– система інтерфейсу користувача.

Такий підхід дозволяє уникнути перевантаження окремих скриптів та робить структуру проекту більш зрозумілою (рис. 3.13).



Рисунок 3.13 – Структура скриптів у проекті Unity

Під час розробки також використовувалися методи захисту від помилок у роботі програми. Для цього виконувалася перевірка об'єктів перед зверненням до них у коді. Це дозволяє уникнути появи помилок типу `NullReferenceException`, які часто виникають у Unity-проектах.

Лістинг 3.5 – Перевірка наявності об'єкта перед використанням

```
Ray ray = Camera.main.ViewportPointToRay(
    new Vector3(0.5f, 0.5f, 0));
RaycastHit hit;
```

```

Vector3 targetPoint;
if (Physics.Raycast(ray, out hit)) {
    targetPoint = hit.point; }
else{
    targetPoint = ray.GetPoint(100);
}

```

кінець лістингу 3.5

У даному фрагменті коду виконується перевірка існування об'єкта `SoundManager` перед відтворенням звуку. Це дозволяє уникнути аварійного завершення роботи програми у випадку помилки (ліст. 3.5).

Для забезпечення стабільної роботи гри також проводилася оптимізація використання ресурсів комп'ютера. Велика кількість об'єктів на сцені може створювати значне навантаження на процесор та відеокарту, що впливає на продуктивність системи.

З метою зменшення навантаження було:

- оптимізовано кількість викликів `Update()`;
- зменшено кількість зайвих обчислень;
- використано оптимізовані моделі та текстури;
- налаштовано освітлення сцени;
- обмежено кількість одночасних звуків.

Впроваджені алгоритмічні рішення та оптимізація ігрового циклу дозволили зробити загальну роботу гри більш стабільною та прогнозованою. Завдяки правильному розподілу обчислювального навантаження та управлінню пам'яттю (`Object Pooling`) було суттєво зменшено ризик виникнення критичних помилок (`Exceptions`) та аварійних завершень роботи програми (`Crash`) через перевантаження підсистем операційної системи.

Окрему увагу на етапі практичної реалізації було приділено архітектурі захисту та організації ресурсів гри (`Assets`). Усі 3D-моделі, текстури високої роздільної здатності, звукові файли та графи анімацій чітко структуровані та знаходяться у відповідних директоріях проєкту Unity, дотримуючись принципів чистої ієрархії (`Clean Folder Structure`) (рис. 3.14). Це не лише дозволяє

підтримувати суворий порядок у проєкті та значно спрощує командну роботу з файлами, але й створює фундамент для майбутнього використання технології AssetBundles або Addressables для оптимізації завантаження контенту в пам'ять під час гри.

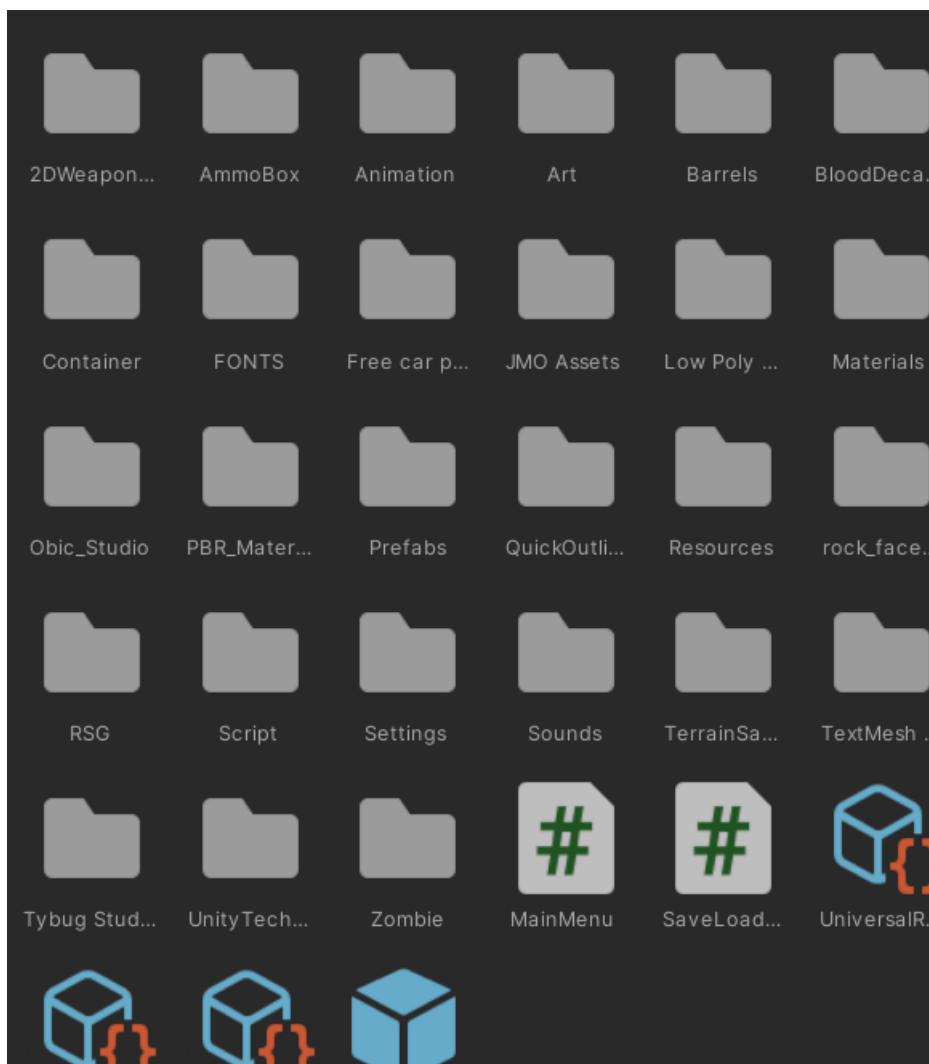


Рисунок 3.14 – Організація ресурсів проєкту

Для створення фінальної версії гри використовувався Build System у Unity. Після компіляції проєкту користувач отримує готовий .exe файл та необхідні системні файли для запуску гри. Такий підхід частково захищає вихідний код від прямого редагування.

Також під час розробки враховувалася стабільність роботи програми при виникненні помилок.

У процесі тестування перевірялося:

- коректне завантаження сцени;
- правильна робота меню;
- стабільність роботи ворогів;
- відсутність критичних помилок;
- коректна робота звукової системи.

Впроваджені алгоритмічні рішення та оптимізація ігрового циклу дозволили зробити загальну роботу гри більш стабільною та детермінованою. У результаті тривалого тестування було встановлено, що основні системи гри (генерація сутностей, логіка ШІ, обробка фізичних зіткнень) функціонують безперебійно, не викликають критичних помилок (Exceptions) та аварійного завершення програми (Crash) навіть в умовах штучного перевантаження підсистем.

Оскільки розроблений проєкт є локальною одиночною (Single-player) грою, у його архітектурі свідомо не використовуються виділені сервери, постійне онлайн-підключення або зовнішні бази даних. Зважаючи на відсутність передачі чутливих даних користувача по мережі, відпала необхідність реалізації складних систем авторизації (наприклад, OAuth), симетричного шифрування пакетів чи захисту від мережових DDoS-атак. Проте базові принципи безпеки програмного забезпечення, такі як інкапсуляція даних у класах (заборона прямого доступу до критичних змінних здоров'я ззовні), все одно були суворо дотримані під час етапу кодування.

Для забезпечення цілісності самого проєкту (Asset Security) та уникнення втрати результатів розробки, використовувалося регулярне резервне копіювання файлів. У процесі роботи архівація (версіонування) проєкту здійснювалася на зовнішні носії, що дозволило б оперативно відновити середовище у випадку пошкодження метаданих Unity або випадкового пошкодження вихідного коду (Source Code) під час рефакторингу.

Окрему увагу на етапі практичної реалізації було приділено архітектурі захисту та організації ресурсів гри (Assets). Усі 3D-моделі, текстури, аудіофайли

та графи анімацій чітко структуровані та знаходяться у відповідних, ізольованих директоріях проекту Unity, дотримуючись принципів чистої ієрархії (Clean Folder Structure). Також важливим елементом захисту продуктивності стала правильна організація сцени (рис. 3.15), до ігрових об'єктів прикріплювалися виключно необхідні компоненти. Наприклад, було видалено зайві компоненти Animator або Rigidbody з тих статичних об'єктів оточення, які не потребують фізичного прорахунку. Це дозволило суттєво зменшити навантаження на систему та уникнути зайвого споживання оперативної пам'яті комп'ютера гравця.

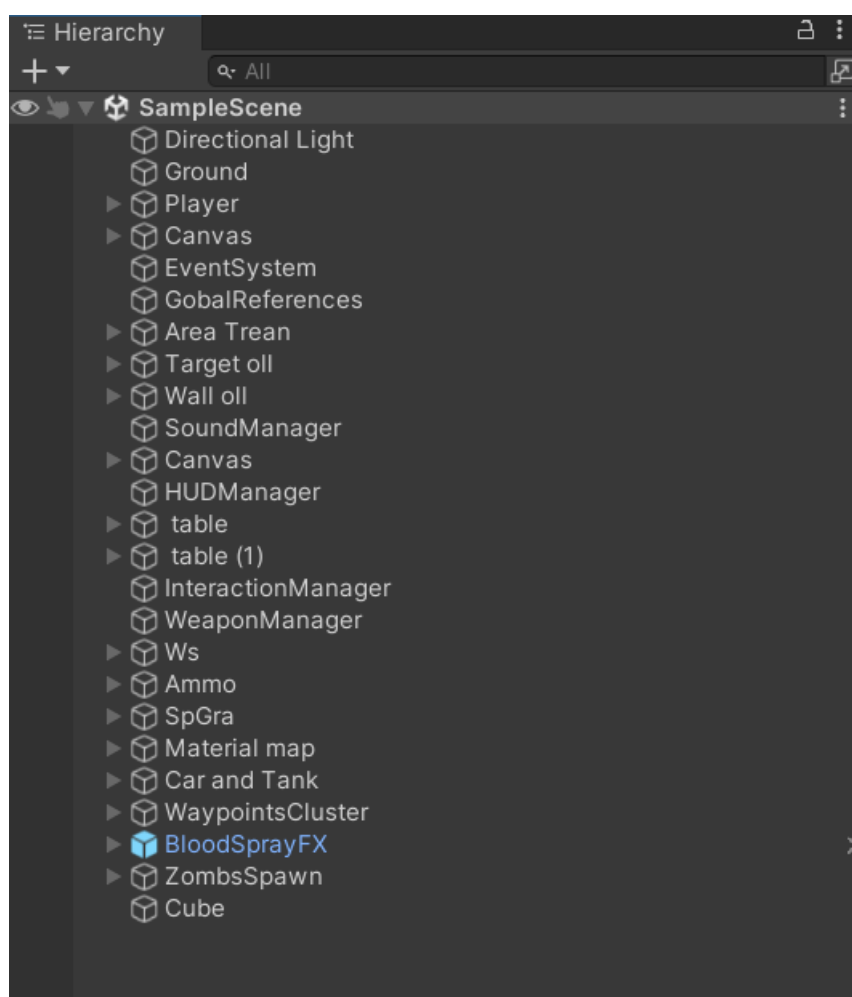


Рисунок 3.15 – Ієрархія об'єктів у сцені Unity

Під час виконання роботи також було проведено аналіз сучасних методів захисту програмного забезпечення. Було встановлено, що для невеликих ігрових

проектів основну роль відіграє стабільність роботи програми, правильна організація коду та оптимізація використання ресурсів системи.

У результаті реалізації даного проекту вдалося забезпечити стабільну роботу гри, правильну організацію програмного коду та базовий рівень захисту інформаційно-комп'ютерної системи. Використані методи дозволили зменшити кількість помилок, підвищити стабільність роботи програми та забезпечити коректне функціонування основних систем гри.

3.4 Розробка документації для програмного продукту

Під час створення програмного забезпечення важливим етапом є розробка документації для програмного продукту. Документація дозволяє користувачу швидше зрозуміти принцип роботи програми, правильно використовувати її функції та уникати помилок під час запуску. Для розробника документація також є важливою, оскільки вона допомагає підтримувати структуру проекту та спрощує подальше оновлення програмного забезпечення.

У даній кваліфікаційній роботі було створено документацію для 3D PvE – гри, розробленої у середовищі Unity. Основна увага приділялася опису процесу запуску гри, елементів інтерфейсу, керування персонажем та базових ігрових механік. Документація створювалася таким чином, щоб користувач міг легко ознайомитися з можливостями гри та без проблем запустити програмний продукт.

Основною метою документації є:

- пояснення принципу роботи гри;
- опис елементів керування;
- допомога користувачу під час запуску;
- опис основних функцій програми;
- забезпечення зручності використання гри.

Документація розроблялася з урахуванням того, що користувач може не мати досвіду роботи з Unity або іншими ігровими застосунками. Через це всі основні функції гри описані просто та зрозуміло.

На початку роботи користувач запускає головне меню гри. У меню доступні основні кнопки керування, зокрема запуск гри та вихід із програми. Головне меню є першим елементом взаємодії користувача з програмним продуктом, тому його інтерфейс було зроблено максимально простим та зрозумілим (рис. 3.16).

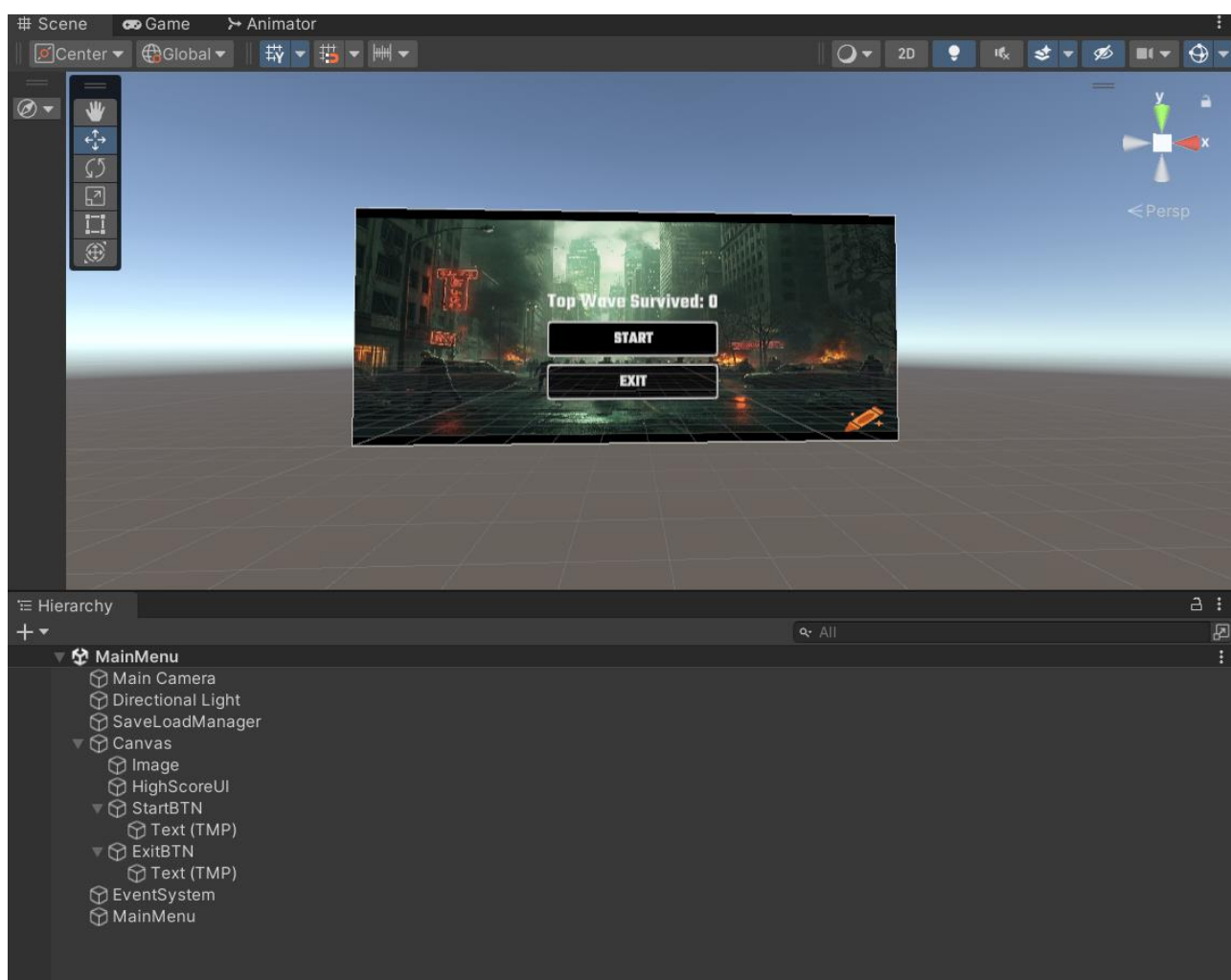


Рисунок 3.16 – Головне меню гри

Після запуску гри користувач переходить до ігрової сцени, де відбувається основний ігровий процес. У грі реалізовано керування персонажем від першої

особи. Для пересування використовуються стандартні клавіші клавіатури та миша.

Основні елементи керування:

- W, A, S, D – пересування персонажа;
- миша – огляд камери;
- ліва кнопка миші – атака або стрільба;
- Esc – пауза або вихід у меню.

Опис елементів керування є важливою частиною документації, оскільки дозволяє користувачу швидко ознайомитися з основними функціями гри.

У процесі гри користувач взаємодіє з ворогами. Вороги автоматично переслідують гравця та атакують його при наближенні на певну дистанцію. Для реалізації поведінки ворогів використовуються система NavMesh, анімації та скрипти на C#.

У документації також описано основні ігрові механіки:

- пересування ворогів;
- система атаки;
- отримання шкоди;
- система здоров'я;
- анімації персонажів;
- звукові ефекти.

Під час гри користувач може бачити елементи інтерфейсу, які відображають основну інформацію про стан гри. До таких елементів належать:

- меню;
- приціл;
- показники здоров'я;
- кнопки взаємодії.

Інтерфейс користувача був реалізований за допомогою UI-компонентів Unity. Основною вимогою до інтерфейсу (рис. 3.17) була його простота та зручність використання.



Рисунок 3.17 – Інтерфейс користувача під час гри

Для створення документації також було описано процес запуску гри. Порядок дій користувача для запуску гри:

- 1) завантажити папку з грою;
- 2) запустити файл формату .exe;
- 3) дочекатися завантаження сцени;
- 4) натиснути кнопку «Play» у головному меню.

Після цього користувач може почати взаємодію з грою.

Окремо у документації було вказано мінімальні технічні вимоги для запуску гри. Це необхідно для того, щоб користувач міг оцінити можливість запуску гри на власному комп'ютері (рис. 3.18).

Мінімальні системні вимоги:

- операційна система Windows 10 або Windows 11;
- процесор Intel Core i5;
- оперативна пам'ять 8 ГБ;
- відеокарта з підтримкою DirectX 11;
- наявність клавіатури та миші.

Рекомендовані системні вимоги:

- Intel Core i5-12450H або аналогічний;
- 16 ГБ оперативної пам'яті;
- NVIDIA GeForce RTX 3050;
- SSD накопичувач.



Рисунок 3.18 – Запуск гри у середовищі Windows

У процесі розробки документації також було описано структуру проєкту Unity. Це необхідно для зручності подальшої підтримки гри та можливості редагування окремих елементів проєкту.

Основні папки проєкту:

- Scripts – програмний код;
- Animations – анімації;
- Audio – звукові файли;
- Materials – матеріали;
- Prefabs – готові ігрові об'єкти;
- Scenes – ігрові сцени.

Такий підхід дозволяє підтримувати порядок у проєкті та спрощує роботу з файлами.

Також у документації описано процес тестування гри та перевірки продуктивності. Для цього використовувалися вбудовані інструменти Unity, зокрема Stats та Console. Це дозволило контролювати FPS та перевіряти стабільність роботи гри при великій кількості ворогів.

Під час створення документації також враховувалася можливість подальшого розширення гри. Архітектура проєкту дозволяє додавати:

- нові рівні;
- нові типи ворогів;
- зброю;
- звукові ефекти;
- додаткові механіки.

Завдяки цьому програмний продукт можна модернізувати без повного переписування програмного коду.

Окрему увагу було приділено оформленню документації. Для наочності використовувалися скріншоти гри, інтерфейсу та середовища Unity. Це дозволяє краще зрозуміти структуру проєкту та основні принципи роботи програмного забезпечення.

У процесі роботи також було створено короткі пояснення до основних скриптів гри. Наприклад, скрипт Enemy.cs (ліст. 3.6) відповідає за систему здоров'я ворогів та обробку отримання шкоди, а SoundManager використовується для керування звуковими ефектами.

Лістинг 3.6 – Фрагмент коду системи отримання шкоди

```
public void TakeDamage(int damageAmount) {
    HP -= damageAmount;
    if (HP <= 0) {
        PlayerDead();
        isDead = true; }
    else {
        StartCoroutine(BloodyScreenEffect());
        playerHealthUI.text = $»Health: {HP}»;
    }

    SoundManager.Instance.playerChannel.PlayOneShot(SoundManager.Instance.playerHurt); }
}
```

кінець лістингу 3.6

Даний фрагмент коду демонструє базову логіку системи здоров'я ворога. При отриманні шкоди значення HP зменшується, а після втрати всього здоров'я змінюється стан персонажа.

Таким чином, у процесі виконання роботи було створено документацію для програмного продукту, яка містить опис запуску гри, елементів керування, інтерфейсу користувача, структури проєкту та основних механік гри. Документація дозволяє користувачу швидко ознайомитися з можливостями програмного забезпечення та спрощує подальшу підтримку і розвиток проєкту.

3.5 Економічне обґрунтування розробки

Під час створення програмного забезпечення важливим етапом є економічне обґрунтування розробки. Даний розділ дозволяє оцінити витрати часу, ресурсів та програмних засобів, які були використані під час виконання проєкту. Також економічне обґрунтування допомагає визначити доцільність створення програмного продукту та оцінити його практичну цінність.

У межах даної кваліфікаційної роботи було створено 3D PvE-гру в середовищі Unity. Розробка виконувалася самостійно протягом приблизно двох місяців. У середньому на роботу над проєктом витрачалося 3–4 години на день, а під час складніших етапів розробки час роботи збільшувався.

Основними етапами створення програмного продукту були:

- аналіз предметної області;
 - проєктування архітектури гри;
 - реалізація ігрових механік;
- створення інтерфейсу користувача;
- тестування та налагодження;
 - оптимізація роботи гри;
 - оформлення документації.

Для реалізації проєкту використовувалися безкоштовні програмні засоби та інструменти. Основним середовищем розробки був Unity 2022.3.21f1. Для

написання програмного коду використовувалося середовище Visual Studio. Також у процесі роботи використовувалися безкоштовні ресурси з Unity Asset Store.

Оскільки програмне забезпечення Unity Personal є безкоштовним для навчальних та індивідуальних проєктів, витрати на придбання ліцензії були відсутні. Це дозволило значно зменшити загальну вартість розробки.

У процесі виконання роботи використовувався персональний комп'ютер із такими характеристиками:

- процесор Intel Core i5-12450H;
- відеокарта NVIDIA GeForce RTX 3050;
- оперативна пам'ять 16 ГБ;
- операційна система Windows 11.

Одним із головних показників економічного обґрунтування є витрати часу на створення програмного продукту (таб. 3.1).

Таблиця 3.1 – Етапи роботи над проєктом

Етап роботи	Орієнтовний час
Аналіз предметної області	10 годин
Проектування архітектури	15 годин
Реалізація механік гри	50 годин
Робота з UI та меню	50 годин
Тестування та налагодження	10 годин
Оформлення документації	20 годин
Загальний час	15 годин

У середньому розробка тривала близько двох місяців. Частина часу також витрачалася на пошук інформації, вивчення документації Unity та виправлення помилок у коді.

Для приблизного економічного розрахунку можна використати умовну погодинну оплату праці розробника-початківця. Якщо взяти орієнтовну вартість

200 гривень за годину роботи, то загальна вартість розробки становитиме: $120 \times 200 = 24000$ грн.

Даний розрахунок є приблизним, оскільки проєкт виконувався як навчальна кваліфікаційна робота, а не комерційне замовлення.

Окрім витрат часу, під час роботи використовувалися ресурси комп'ютера та електроенергія. Середнє споживання електроенергії ноутбуком або персональним комп'ютером під час роботи з Unity може становити приблизно 150–200 Вт на годину.

Якщо взяти середнє значення 180 Вт та приблизно 120 годин роботи, то загальне споживання електроенергії становитиме: $0,18 \times 120 = 21,6$ кВт·год.

При середній вартості електроенергії приблизно 4,5 грн за 1 кВт·год отримаємо: $21,6 \times 4,5 = 97,2$ грн.

Отже, витрати на електроенергію є незначними у порівнянні з витратами часу на розробку.

Також під час створення гри використовувалися безкоштовні 3D-моделі, анімації та звукові ефекти. Використання безкоштовних ресурсів дозволило суттєво зменшити витрати на реалізацію проєкту.

У процесі розробки особлива увага приділялася оптимізації гри. Це дозволило зменшити навантаження на систему та забезпечити стабільний FPS навіть при значній кількості ворогів на сцені. Оптимізація програмного забезпечення є важливим фактором, оскільки вона впливає на якість кінцевого продукту та комфорт користувача під час гри.

Створений програмний продукт може використовуватися як основа для подальшого розвитку гри. У майбутньому проєкт можна доповнити:

- новими рівнями;
- додатковими ворогами;
- системою збереження;
- новими типами зброї;
- мультиплеєром;
- розширеним інтерфейсом.

Це підвищує практичну цінність розробленого програмного забезпечення та демонструє можливість подальшого розвитку проєкту.

Також важливо враховувати, що використання Unity значно скорочує час розробки гри. Рушій має велику кількість готових інструментів та компонентів, що дозволяє зосередитися безпосередньо на створенні механік гри, а не на реалізації базових систем із нуля.

Під час виконання роботи було отримано практичні навички:

- роботи з Unity;
- програмування на C#;
- створення 3D-сцен;
- роботи з NavMesh;
- реалізації анімацій;
- оптимізації програмного забезпечення;
- тестування та налагодження програмного продукту.

Отримані знання та навички можуть бути використані під час подальшої розробки ігрових проєктів та програмного забезпечення.

Таким чином, у процесі виконання кваліфікаційної роботи було створено програмний продукт із використанням безкоштовних засобів розробки та мінімальних фінансових витрат. Основними витратами стали час, використання комп'ютерних ресурсів та електроенергії. Результатом роботи стала повноцінна 3D PvE-гра, яка демонструє практичне застосування сучасних засобів розробки програмного забезпечення та може бути основою для подальшого розвитку проєкту.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи бакалавра було успішно розв'язано актуальне інженерне завдання – спроектовано архітектуру та здійснено практичну реалізацію 3D PvE-гри в середовищі Unity з урахуванням сучасних вимог до продуктивності та масштабованості ігрового циклу.

Відповідно до поставленої мети та завдань отримано такі основні результати:

Здійснено аналіз предметної області: Досліджено проблематику падіння частоти кадрів (FPS) в іграх із великою кількістю сутностей. Виявлено, що класичний об'єктно-орієнтований підхід (базується на компонентах MonoBehaviour) має суттєві обмеження продуктивності через накладні витрати на виклик методів у життєвому циклі рушія та фрагментацію кеш-пам'яті.

Проаналізовано архітектурні рішення: Досліджено можливості стека DOTS, чистих ECS-систем, C# Job System та патерну Object Pooling. На основі аналізу обґрунтовано доцільність використання гібридної архітектури. Вона дозволила поєднати зручність класичних інструментів Unity для базової логіки гравця з високою продуктивністю багатопотокових масових розрахунків поведінки натовпу ворогів.

Спроектовано архітектуру 3D PvE-гри: Розроблено модульну структуру з використанням принципу слабкої зв'язності (Low Coupling). Налаштовано надійну взаємодію між модулем гравця, системою генерації (спавну) ворогів та графічним інтерфейсом користувача із застосуванням патернів Singleton та Observer.

Обґрунтовано вибір інструментів та алгоритмів: Як технологічне ядро обрано Unity Engine із нативним компілятором Burst Compiler. Для вирішення алгоритмічного завдання просторової навігації замість ресурсомістких клітинних сіток обрано алгоритм A* на базі топологічних графів NavMesh. Управління станами штучного інтелекту ворогів спроектовано на базі кінцевого автомата (State Machine) з використанням інструмента Mecanim.

Виконано практичну реалізацію: У середовищі Unity розроблено повноцінний ігровий прототип, що імплементує базові PvE-механіки: переміщення гравця, систему динамічної генерації ворогів та механіку нанесення шкоди з використанням подій анімації (Animation Events). Інтеграція патерну Object Pool дозволила взяти під контроль життєвий цикл супротивників та мінімізувати навантаження на збирач сміття (Garbage Collector).

Проведено тестування та профілювання: Із застосуванням інструментів Unity Profiler та Frame Debugger виконано серію стрес-тестів. Доведено, що завдяки алгоритмічній оптимізації (заміна Vector3.Distance на sqrMagnitude, зменшення кількості викликів Update()) та гібридній структурі, гра працює стабільно і надійно зберігає плавний рівень FPS навіть при пікових навантаженнях (понад 100 активних об'єктів на сцені).

Виконано техніко-економічне обґрунтування: Розраховано показники економічної ефективності, складено кошторис витрат на розробку та доведено економічну доцільність створення даного програмного продукту.

Розглянуто питання охорони праці: Проведено аналіз шкідливих та небезпечних факторів при роботі з комп'ютерною технікою, розроблено комплекс заходів щодо забезпечення безпеки життєдіяльності та пожежної безпеки на робочому місці інженера-програміста.

Перспективи подальшого розвитку: Окрім успішної реалізації основних ігрових механік та оптимізації поточного підходу, у процесі роботи сформовано науково-інженерне підґрунтя для переходу на стек ECS/DOTS. Незважаючи на те, що на поточному етапі базові механіки спираються на класичний підхід, модульна структура проєкту дозволяє в майбутньому гнучко масштабувати гру та перенести всі вузькі місця III на чистий C# Job System без необхідності повного переписування існуючої бази коду. Це відкриває шлях до обробки десятків тисяч ігрових об'єктів одночасно.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Baron D. Game Development Patterns with Unity 2021. Packt Publishing. 2021. URL: <https://www.packtpub.com/product/game-development-patterns-with-unity-2021/9781800205602> (дата звернення: 09.03.2026).
2. Aversa D., Garavaglia A. Unity Game Optimization, 3rd Edition. Packt Publishing. 2022. URL: <https://www.packtpub.com/product/unity-game-optimization-third-edition/9781801811802> (дата звернення: 20.04.2026).
3. Unity Technologies. Оптимізація продуктивності в Unity: розуміння профілювання. Unity Documentation. 2026. URL: <http://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity.html> (дата звернення: 19.03.2026).
4. Unity Technologies. Інструменти профілювання Unity (Unity Profiler). Unity Documentation. 2026. URL: <https://docs.unity3d.com/Manual/Profiler.html> (дата звернення: 22.02.2026).
5. Haas W. Entity Component System Design in Unity. IEEE Transactions on Games. 2021. Vol. 13. URL: <https://ieeexplore.ieee.org/document/9123547> (дата звернення: 20.04.2026).
6. Unity Technologies. Data-Oriented Technology Stack (DOTS). Unity Documentation. 2026. URL: <https://unity.com/dots> (дата звернення: 22.03.2026).
7. Game Developer. Створення 3D-ігор на Unity: архітектурні підходи та патерни. 2026. URL: <https://www.gamedeveloper.com/programming> (дата звернення: 26.03.2026).
8. Albahari J. C# 10.0 in a Nutshell: The Definitive Reference. O'Reilly Media. 2022. URL: <https://www.oreilly.com/library/view/c-100-in/9781098121945> (дата звернення: 13.02.2026).
9. Unity Technologies. C# Job System Manual. Unity Documentation. 2026. URL: <https://docs.unity3d.com/Manual/JobSystem.html> (дата звернення: 14.02.2026).

10. Unity Technologies. Burst Compiler User Guide. Unity Documentation. 2026. URL: <https://docs.unity3d.com/Packages/com.unity.burst@latest/index.html> (дата звернення: 18.04.2026).

11. Kyaw A. S., Peters C., Aversa D. Unity Artificial Intelligence Programming, 5th Edition. Packt Publishing. 2022. URL: <https://www.packtpub.com/product/unity-artificial-intelligence-programming-fifth-edition/9781803247425> (дата звернення: 19.03.2026).

12. Unity Technologies. Unity Navigation System (NavMesh). Unity Documentation. 2026. URL: <https://docs.unity3d.com/Manual/nav-NavigationSystem.html> (дата звернення: 22.02.2026).