Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Гайдученка Артема Сергійовича
Прізвище, ім'я, по батькові студента(ки)

на здобуття ступеня вищої освіти Бакалавра

«Програмне забезпечення гри-шутера на Unity»
Назва теми

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.2101069.01.02.ПЗ

Виконав студент IV курсу, група ІПЗ-21-1 _____ Артем ГАЙДУЧЕНКО
Підпис Ім'я, ПРІЗВИЩЕ

Керівник канд. техн. наук, доцент _____ Оксана ЯШИНА
Науковий ступінь, звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер канд. техн. наук, доцент _____ Юрій ФОРКУН
Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

5 червня 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
02. 01. 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гайдученку Артему Сергійовичу

Прізвище, ім'я, по батькові студента

1. Тема кваліфікаційної роботи Програмне забезпечення гри-шутера на Unity

Керівник кваліфікаційної роботи Яшина Оксана Миколаївна, канд. техн. наук, доцент

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 07.02.2025 р. № 23

2. Строк подання студентом роботи на кафедру 01.06.2025 р.

3. Вихідні дані до роботи Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Дослідження предметної області та постановка задачі, проєктування програмного забезпечення (гри), програмна реалізація гри, тестування

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди, 15 шт.), Контекстна діаграма А-0 «Геймплей гри», Діаграма декомпозиції А-1 «Геймплей гри», діаграма варіантів використання гри, діаграма послідовностей сірих сценаріїв, блок схема для сценарію «Переміщення в просторі»

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Форкун Ю. В., канд. техн. наук, доцент	12.05.2025	13.05.2025
Антиплагіат	Форкун Ю. В., канд. техн. наук, доцент	17.05.2025	28.05.2025

7. Дата видачі завдання «02» 01 2025р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Ознайомлення з тематикою дипломного проєктування, визначення та узгодження індивідуальних тем кваліфікаційних робіт (КвР)	01.12 – 31.12.2024	
2 Збір матеріалу за темою КвР; дослідження предметної області, в якій планується використання програмного забезпечення (ПЗ), визначення задач та вимог, розробка технічного завдання	01.01 – 20.02.2025	
3 Проєктування програмного забезпечення	21.02 – 20.03.2025	
4 Програмна реалізація з використанням відповідних засобів розробки. Тестування ПЗ	21.03 – 30.04.2025	
5 Написання вступу, загальних висновків, оформлення переліку джерел посилання та додатків. Оформлення пояснювальної записки КвР згідно вимог	01.05 – 25.05.2025	
6 Попередній захист КвР	Травень 2025	Згідно графіка
7 Перевірка КвР на плагіат, нормоконтроль, отримання відгуків, рецензій та інших супровідних документів. Брошурування (зшиття) пояснювальної записки.	26.05 – 30.05.2025	
8 Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії ХНУ; підготовка до захисту та захист КвР	з 01.06.2025	

Студент _____ Артем ГАЙДУЧЕНКО
Підпис Ім'я, прізвище

Керівник роботи _____ Оксана ЯШИНА
Підпис Ім'я, прізвище

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Програмне забезпечення гри-шутера на базі Unity».

Автор роботи: Гайдученко Артем Сергійович.

Керівник роботи: Яшина Оксана Миколаївна.

Пояснювальна записка: 72 с., 32 рис., 3 табл., 2 дод., 44 джерела.

Графічна частина: 15 слайдів.

Метою кваліфікаційної роботи є проєктування та розробка повноцінного шутера на платформі Unity з реалізацією основних ігрових механік. Також передбачено створення базового алгоритму поведінки супротивників та оптимізацією продуктивності гри для різних пристроїв.

У кваліфікаційній роботі виконана комплексна розробка гри, що включає аналіз предметної області, огляд існуючих рішень, проєктування архітектури, реалізацію ключових механік, створення ігрового інтерфейсу, базового візуального дизайну, впровадження алгоритму поведінки супротивників і загальну оптимізацію під різні платформи.

04.05.2025
Дата


Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документу	Найменування документу	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ.2101069.01.02.ПЗ	Пояснювальна записка	72		
2	А4		Завдання на кваліфікаційну роботу	1		
3	А4		Анотація	1		
			Графічні документи			
4	А4	КвРІПЗ.2101069.01.02.Е8	Контекстна діаграма	1		
			А-0 «Геймплей гри»			
5	А4	КвРІПЗ.2101069.01.02.Е8	Діаграма декомпозиції	1		
			А-1 «Геймплей гри»			
6	А4	КвРІПЗ.2101069.01.02.Е8	Діаграма варіантів	1		
			використання гри			
7	А4	КвРІПЗ.2101069.01.02.Е8	Діаграма послідовностей	1		
			сірих сценаріїв			
8	А4	КвРІПЗ.2101069.01.02.Е8	Блок схема для сценарію	1		
			«Переміщення в просторі»			
9	А4		Презентаційні матеріали	15		

Змн.	Арк	№ докум.	Підпис	Дата	КвРІПЗ.2101069.01.02.ВД		
Виконав		Гайдученко А. С.		04.06	Програмне забезпечення гри-шутера на базі Unity	Літ.	Арк. 1 / Аркушів 1
Керівник		Яшина О. М.		04.06			
Н. контр.		Форкун Ю. В.		04.06	Відомість документів	ХНУ, ІПЗ-21-1	
Зав. каф.		Бедратюк Л. П.		05.06			

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

GDD – Game Design Document, описує аспекти розробки гри

ОП – Оперативна пам’ять

DLC – Downloadable Content, додатковий вміст до гри, який можна завантажити

FPS – First-Person Shooter, ігри з видом від першої особи

TPS – Third-Person Shooter, ігри з видом від третьої особи

VHS – Video Home System, аналоговий формат відеокасет для домашніх відеомагнітофонів, стиль

ООП – об'єктно-орієнтоване програмування

КОП – компонентно-орієнтована архітектура

ECS – Entity Component System, архітектура в ігровій розробці для гнучкої побудови об'єктів

DOTS – Data-Oriented Technology Stack

DOD – Data-Oriented Design, підхід до програмування з фокусом на ефективну роботу з даними

IDEF0 – Integration DEFinition for Function Modeling, метод моделювання функцій системи

ПЗ – Програмне забезпечення

VR/AR – Virtual Reality / Augmented Reality, технології віртуальної та доповненої реальності

UVC – Unity Version Control, система керування версіями проєктів в Unity

UTF – Unity Test Framework, інструмент для автоматичного тестування коду в Unity

ВСТУП

Розвиток ігрових технологій та інтерактивного програмного забезпечення відкриває досить широкі можливості для створення ігор різних жанрів, забезпечуючи нові горизонти для гравців різних фінансових можливостей. Одним з найбільш популярних та затребуваних напрямків є розробка ігор-шутерів, які займають важливе та, напевне, найпопулярніше, місце в культурі відеоігор, маючи великий вплив на геймерське середовище та ставши невід'ємною частиною багатьох поколінь гравців.

Шутери часто поєднують захоплюючий ігровий процес, напружену атмосферу та інтерактивну взаємодію з іншими учасниками, що дозволяє створювати динамічні ігрові світи, проте як правило це саме частина ігроладу, на яку вже накладаються різні жанри.

Однак, для успішної реалізації таких проєктів розробникам необхідно враховувати не лише цікаву концепцію, а й правильне поєднання механік, що забезпечать динамічний, захоплюючий процес гри. Важливу роль у цьому відіграє створення штучного інтелекту супротивників, як алгоритму дій, який має бути здатним реагувати на дії гравців, підтримуючи баланс між складністю та доступністю гри.

Актуальність розробки шутера на базі Unity зумовлена не лише популярністю цього жанру, але й широким використанням платформи Unity, яка є однією з найбільш доступних і потужних для створення ігор завдяки багатому інструментарію та підтримці мультиплатформеності. Unity надає розробникам гнучкість та широкий набір інструментів, що дозволяють реалізувати ігрові механіки легко, швидко, та без потреби довго вивчати ігновий двигун.

Розробка гри-шутера включає в себе не лише технічні аспекти, такі як вибір архітектури гри, програмування механік та оптимізація продуктивності, але й креативні елементи: дизайн локацій, створення персонажів, розробка візуальних ефектів, атмосфери, що є важливою частиною створення занурення в гру.

Метою кваліфікаційної роботи є проєктування та розробка повноцінного шутера на платформі Unity з реалізацією основних ігрових механік. Також передбачено створення базового алгоритму поведінки супротивників та оптимізацією продуктивності гри для різних пристроїв. Крок за кроком буде розроблятися архітектура гри, визначатимуться вимоги до системи, створюватимуться ключові ігрові модулі, інтерфейс та інтеграція з іншими компонентами.

Завданнями кваліфікаційної роботи є:

– аналіз предметної області;

– аналіз сучасних тенденцій в розробці ігор-шутерів та використання Unity для створення проєктів;

– визначення функціональних та нефункціональних вимог до проєктованого програмного забезпечення та його компонентів;

– розробка та реалізація основних ігрових механік, таких як: стрільба, рух персонажів та взаємодія з навколишнім середовищем;

– реалізація базового інтелекту супротивників та оптимізація гри для різних платформ.

Загалом розробка шутера на базі Unity є складним та цікавим процесом, який вимагає не лише технічних навичок, але й креативного підходу до дизайну. Це дозволяє створювати ігри, які дарують незабутні емоції. Ураховуючи постійний розвиток ігрових технологій та популярність жанру відносно розвитку – ця тема буде завжди актуальною.

Перспективи розвитку роботи включають не тільки розширення функціональності гри, але й адаптацію до новітніх технологічних трендів, що дозволить створювати ще більш захоплюючі, інноваційні та якісні ігрові програмні продукти. У майбутньому планується вдосконалення графічної складової, оптимізація продуктивності та впровадження багатокористувацького режиму. Такий підхід забезпечить подальший розвиток проєкту та підвищить його привабливість серед цільової аудиторії.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей.

Ігрова індустрія почала розвиток в далеких 80-90 роках минулого століття, та вже до середини 21-го сильно вкорінилась як серед молодшого, так і серед старшого поколінь. Популярність відеоігор настільки стрімко росла і продовжує рости, що поки в межах пост-радянських країн їх вважали дитячими розвагами або, навпаки, небезпечними та викликаючими залежність – у 2011 році, Національним фондом мистецтв США та урядом США відеоігри були офіційно визнані видом мистецтва. А на даний момент в відеоігри так чи інакше грали навіть люди старші 40-ка, 50-ти чи навіть 60 років по всьому світу [1]. На рисунку 1.1 зображений розподіл віку гравців.

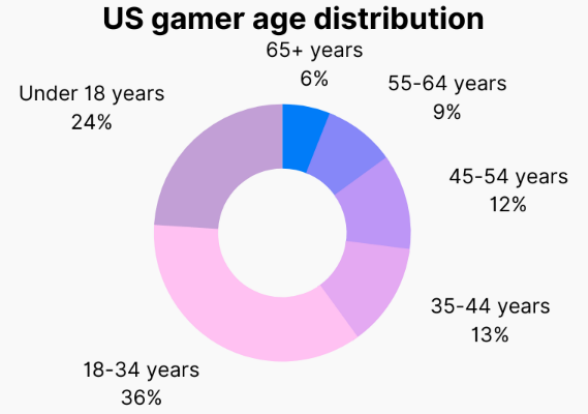


Рисунок 1.1 – Віковий розподіл гравців

Оскільки відеоігри давно вийшли за межі простих розваг – вони перетворились на інструмент спілкування та джерело розвитку творчого й аналітичного мислення. Завдяки прориву в комп'ютерній графіці, штучному інтелекті та віртуальній реальності – також стали неймовірно захопливими.

У цій сфері домінують гіганти індустрії, зокрема: FromSoftware, Ubisoft, Xbox Game Studios, EA і CD Projekt Red. Багато компаній із різних країн невпинно змагаються за увагу геймерів через постійний випуск нових ігор [2]. Загалом найбільші компанії були представлені на рисунку 1.2.

Рисунок 1.2 – Ігрові компанії-гіганти

У рамках аналізу предметної області розглянуто питання створення відносно популярних ігор, і отримано наступні висновки.

Ідея гри – це основа всього проєкту. Вона визначає жанр, цільову аудиторію, сюжет, механіки та платформу (ПК, консолі, мобільні пристрої). На етапі визначення ідеї розробники формують загальний образ гри: чи буде це динамічний шутер, стратегія чи певна інтелектуальна гра.

Документ концепції гри (з англ. Game Design Document або скорочено GDD) є головним інструментом цього етапу. Він описує основні ідеї, механіки, основні ігроладні моменти та графічний стиль. Це може бути план на кілька сторінок, але важливо, щоб він був чітким і конкретним. У цьому документі визначаються також усі цілі та вимоги до продукту. Загалом кожен проєкт повинен мати своє чітко розроблене технічне завдання.

Розподіл ролей у команді є критично важливим для досягнення успіху на цьому етапі. Розробники, дизайнери, художники, аніматори, звукорежисери – усі повинні бути чітко визначені. Залежно від масштабу проєкту, можуть бути додаткові спеціалісти: тестувальники, сценаристи, тощо.

Вибір рушія – це один із найбільш важливих кроків, оскільки він визначатиме багато аспектів подальшої розробки. Раніше частим вирішенням питання ігрового рушія (движка) було створення свого особистого, проте із збільшенням популярності UE 4 та 5 великі студії все частіше відмовляються від

такої ідеї. В свою чергу невеликі незалежні студії завжди обирали певний готовий рушій накшталт Unity або Godot. Рішення про вибір рушія загалом повинно бути засноване грі, яка розробляється, розмірах, очікуваній від неї графіці.

На етапі прототипування створюється спрощений варіант гри для тестування основних механік. Він може не містити фінальних моделей чи текстур, але має демонструвати ключовий геймплей. Це дозволяє оцінити ідею, зацікавленість користувачів і визначити оптимальний напрямок подальшої розробки, а також обсяг необхідних ресурсів.

Коли основні механіки та концепція гри протестовані, варто переходити до основної розробки. Цей етап розділяють на кілька ключових напрямків: графіка, програмування, звук.

На напрямку графіки та дизайну відбувається створення моделей персонажів, локацій, анімацій, текстур та ін. Художники працюють над створенням візуальної складової гри, що визначає її стиль та атмосферу.

Програмування – на цьому напрямку розробники працюють над впровадженням усіх механік: керування, логіка боїв, фізика руху, алгоритм поведінки для ворогів, а також створення системи збереження прогресу.

Звук та музика – напрямок, під час якого звукорежисери створюють аудіофайли для фонової музики, ефектів, а також для голосових акторів, якщо в грі є діалоги. Робота повинна бути ретельно скоординована, оскільки на цьому етапі важливо створити функціональну та приємну зовнішньо гру.

Тестування є критичним етапом на всіх етапах розробки. Ігрові механіки повинні бути протестовані на баги, щоб не виникали критичні помилки під час гри. Крім того, важливо тестувати гру на різних конфігураціях ПК, щоб забезпечити хорошу сумісність і продуктивність на різних пристроях.

Після закінчення роботи на основною частиною гри необхідно працювати над її оптимізацією. Це включає в себе покращення продуктивності, зменшення використання оперативної пам’яті (ОП), оптимізацію графіки, зменшення часу завантаження рівнів та багатьох інших аспектів, що можуть вплинути на комфорт

гри. Цей етап є дуже важливим, адже хороші ігри часто не можуть мати достатнього рівня успіху, якщо мають погану оптимізацію.

Коли всі попередні етапи завершені, наступним кроком є сам реліз гри. Гра завантажується на платформи, такі як Steam, Epic Games Store, GOG або інші, залежно від того, на яких платформах планується її випуск. Можливими варіантами можуть бути промо-матеріали, організація публікації гри в медіа та анонси на популярних ігрових платформах.

Розвиток гри не закінчується після релізу. Взаємодія з гравцями через форуми, соцмережі та надання технічної підтримки – важливий аспект тривалої життєздатності гри. Відповіді на запитання, виправлення помилок, створення нових оновлень дозволяють зберігати інтерес до гри на довгий час. Також після випуску гри важливо підтримувати її в актуальному стані, виправляючи баги та оновлюючи контент. Це включає в себе розробку патчів для виправлення помилок і поліпшення гри чи публікацію додаткового контенту, такого як: нові завдання, персонажі чи загалом, різного рівня величини, додатковий контент (DLC) [3].

У сучасних умовах ігрова індустрія стикається з низкою питань. Розробка програмного забезпечення повинна враховувати потреби кінцевих користувачів, що мають різні рівні доступу та вимоги до збереження прогресу, механік геймплею та загальної атмосфери гри.

Гра яка розроблялась в ході кваліфікаційної роботи має жанр шутеру з елементами жанрів: хорору (жахи) та виживання. Відповідно першочерговою ціллю була розробка основних елементів основного жанру.

Шутер – ігровий жанр, зосереджений на взаємодії гравця з ворогами через стрільбу. В таких іграх основний фокус – швидкість, точність, тактика та боротьба з ворогами. Шутери можуть бути «від першої особи» (FPS), де гравець бачить світ очима персонажа, або «від третьої особи» (TPS), де камера доступна гравцю зміщена від його персонажа так, що персонаж видимий зі спини. Часто завданням для головного героя є ліквідація усіх необхідних цілей на рівні, набір найбільшої кількості очок, а не проходження сюжету [4].

Хорор – це жанр, призначений для створення в гравців відчуття страху і напруги. Ігри цього жанру зазвичай містять елементи психологічного чи фізичного страху, а також частіше застосовують атмосферні ефекти, щоб налякати гравця. Сюжети можуть бути побудовані навколо міфів, монстрів, надприродних явищ або психічних проблем, що приводять до темних і страшних подій [5].

Ігри жанру виживання зосереджені на підтримці життя персонажа в умовах, де ресурси обмежені, а небезпеки – постійні. Гравець повинен взаємодіяти з навколишнім світом, збирати ресурси, будувати притулки, боротися за їжу, воду та безпеку. Виживання може бути в різних умовах: у дикій природі, після катастрофи чи апокаліпсису, або навіть в умовах космосу [6].

Коли ці жанри комбінуються, гра отримує додаткову глибину та варіативність. Наприклад, поєднання шутера і хорора може створити інтенсивну атмосферу, де гравець повинен стріляти для своєї безпеки, але одночасно переживає відчуття постійної небезпеки, що додає жаху і напруги. Додавання елементів виживання робить гру ще більш складною і вимогливою, оскільки гравець не лише бореться з ворогами, але й повинен постійно піклуватися про свої основні потреби: їжу, воду, відновлення здоров'я.

Аналізуючи предметну область варто звернути увагу на платформи обраного жанру, зображену на рисунку 1.3, вікову категорію можливої майбутньої аудиторії, яка зображена на рисунку 1.5, та стать аудиторії, на рисунку 1.4.

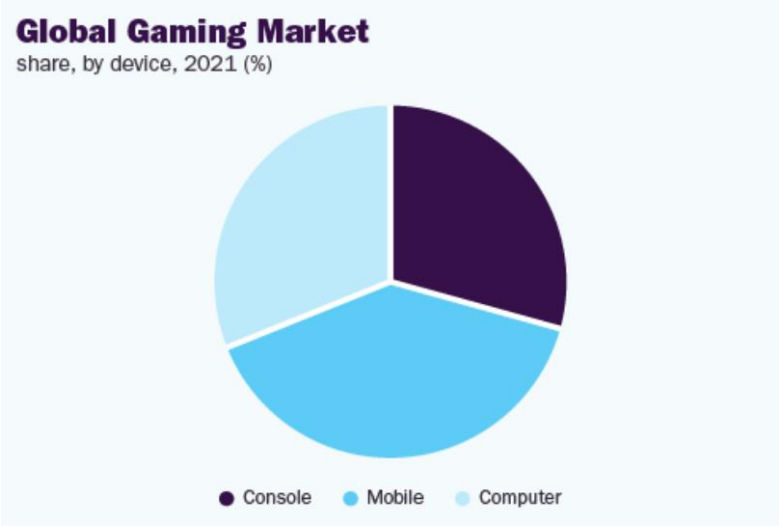


Рисунок 1.3 – Платформи обраного жанру

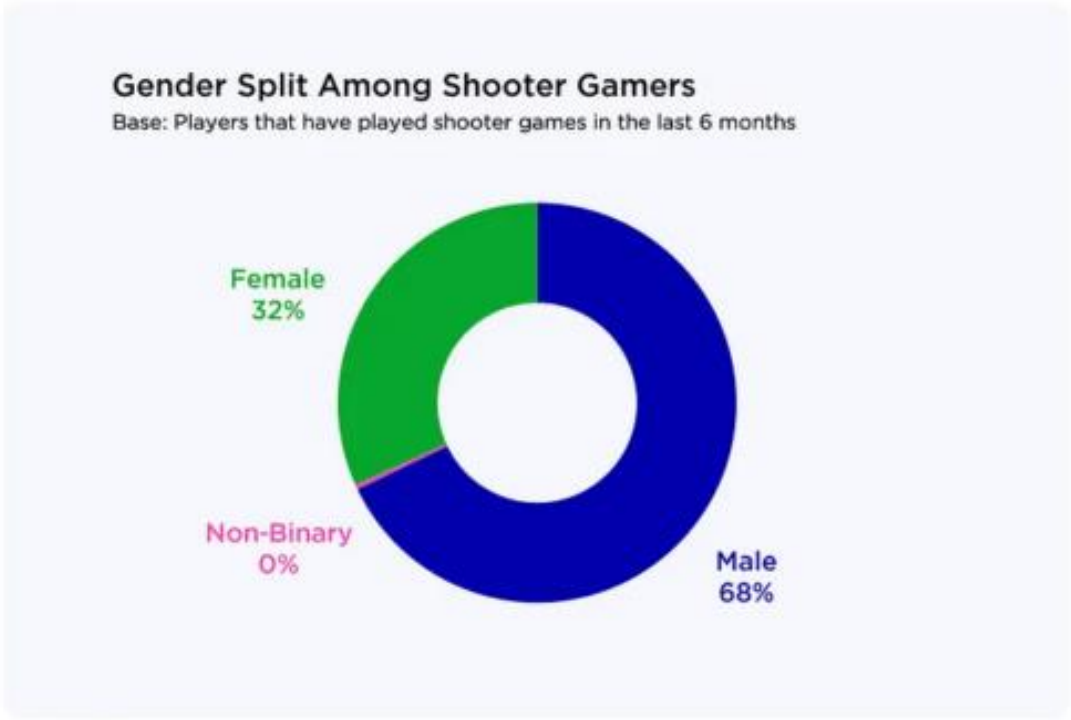


Рисунок 1.4 – Майбутні гравці: поділ за статтю

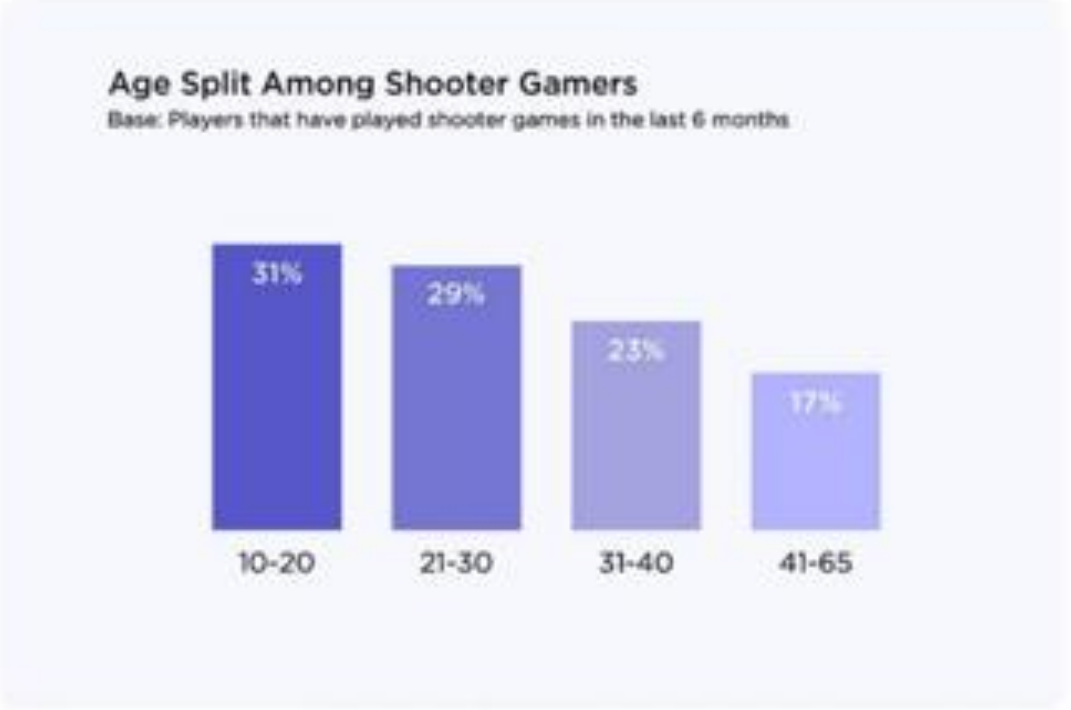


Рисунок 1.5 – Майбутні гравці: поділ за віком

З огляду на діаграми приведені вище та вказані джерела, а також вікові обмеження в багатьох країнах робимо висновки про те, що більша частина очікуваної аудиторії: чоловіки віком від 18-ти до 30-ти років. Більшу частину гравців також будуть складати ПК-користувачі [1].

Сетинг – загальне оточення, стиль, атмосфера, в яких розгортається гра. Він формує уявлення про світ, у якому перебуває гравець, і значною мірою визначає настрій ігрового процесу. До його складових належать: локації, архітектурні стилі, озброєння, типи ворогів, сюжетні події, тощо [7].

У випадку кваліфікаційної роботи сетинг шутеру проєктується так, що ігровий світ буде темним, напруженим і сповненим загроз. Гравець опиняється в атмосфері постійної небезпеки, де вороги – це моторошні істоти, а ресурси для виживання, як-от набої – обмежені.

Візуальний стиль гри також відіграє важливу роль у формуванні вражень гравця. У даному випадку він заснований на поєднанні естетики VHS-хорору та аналогового хорору.

VHS-хорор – візуальний стиль, що імітує вигляд старих касетних записів, характерний для відеокасет 1980–1990-х років. Йому притаманні візуальні шуми, статичні перешкоди, викривлення кольорів, «глюки» зображення та загальне відчуття низької якості, яке створює ефект тривожного ретро-матеріалу.

Аналоговий хорор, в свою чергу, є піджанром, який використовує стилістику застарілих медіа – відеокасет, архівних записів, тощо. Він базується на навмисному спотворенні реальності та подачі інформації фрагментами, що не піддаються логічному поясненню [8].

Комбінація цих стилів у проєктованій грі покликана підсилити ефект присутності, сформувати у гравця стійке відчуття занепокоєння та зробити візуальну подачу максимально автентичною й влучною для заданого сетингу.

Відповідно до аналізу вище далі приведено список вимог для проєкту:

- графіка та анімація: стилізовані моделі персонажів, зброї, тощо;
- звукові ефекти: атмосферні звуки, музика для створення атмосфери;
- система ресурсів: збір та управління ресурсами (їжа, ліки, боєприпаси);
- механіки виживання: контроль за здоров'ям, голодом, спрагою, сном;
- геймплей: динамічні бойові та приховані механіки;
- атмосфера: створення напруги через освітлення, ефекти, тощо.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

Для подальшого визначення функціональних та нефункціональних вимог, проєктування, розробки та тестування програмного продукту кваліфікаційної роботи було розглянуто популярні рішення в рамках предметної області.

Найпопулярнішим, та, в певному сенсі, одним із основоположників ігрової індустрії є жанр ігор-шутерів або «стрілянок». Справедливо буде визнати, що далеко не першою грою, проте тою – яка сильно популяризувала геймдев є Doom. Doom – відеогра в жанрі шутер від першої особи, випущена компанією idSoftware 10 грудня 1993 року, один з найбільш відомих та популярних шутерів в світі, що багато в чому здійснив великий ідейний вплив на розвиток свого жанру [9]. На рисунку 1.6 зображено оригінальний постер гри.

Рисунок 1.6 – Постер до гри «Doom»

Гравець керує піхотинцем, що опинився на базі Фобоса після невдалого експерименту, який відкрив портал для демонів. Основне завдання – знищення ворогів та виживання в лабіринтах, що містять двері, ліфти, секретні приміщення, небезпечні зони (лава, кислоти, отруйні відходи) та телепорти. Піхотинець використовує до 7 видів зброї, включаючи пістолет, дробовик, кулемет, ракетомет, плазмову рушницю та BFG 9000. Інтерфейс відображає стан персонажа, зокрема його реакцію на події.

Гра створила загалом наступний «класичний» тип та вигляд шутерів на багато років вперед: динамічний бойовий процес у від першої особи, де гравець веде боротьбу проти численних ворогів у закритих рівнях. Важливими елементами стали швидкість пересування, пошук ресурсів, багаторівневе оточення з секретами та інтерактивними елементами. Структура рівнів включала коридори, кімнати, двері з ключами, ліфти та пастки, що вимагали від гравця орієнтування і тактичного мислення.

Арсенал складався з різнотипної зброї, кожна з якої мала своє призначення, що заклало стандарт для балансування озброєння в майбутніх іграх жанру. Інтерфейс з індикаторами здоров'я, броні та боєприпасів став класичним, а наявність візуального відображення емоцій головного героя додала атмосферності. Doom закріпив концепцію безперервного екшену, інтенсивних перестрілок, суворої ресурсної економії та рівнів, спроєктованих для дослідження, що стало основою для розвитку FPS на десятиліття вперед.

Окрім суто геймплейних механік, Doom вплинув на розробку ігор з точки зору технічних рішень та спільноти гравців. Гра стала однією з перших, що підтримувала створення модифікацій, що стимулювало розвиток користувацького контенту і зародження фанатських спільнот. Саме завдяки відкритій структурі файлів гри з'явилися численні користувацькі рівні та альтернативні версії гри, що сприяло її довговічності. Doom також започаткував тенденцію багатокористувацьких режимів, зокрема deathmatch, що в подальшому стало стандартом для FPS-ігор.

Іншим жанром, який проте почав набирати популярність уже ближче до кінця 10-тих років 21 століття є жанр «хорор» або «жахи». Жанр більш популярний в незалежних розробників та спрямований на створення відчуття страху, напруження та тривоги у гравця. Використовує різні прийоми, такі як: моторошна атмосфера, несподівані скрімери, обмежені ресурси. Особливої популярності набули ігри з елементами виживання, де гравець змушений приймати рішення під тиском і постійною загрозою.

Сучасні хорор-ігри можуть мати різні піджанри: від класичних виживальних жахів (Resident Evil, Silent Hill) до інтерактивних трилерів (Until Dawn, The Quarry) і психологічних хоророів (Amnesia, Outlast). Крім того, зростання популярності VR-технологій дозволило значно посилити ефект занурення, що зробило ігри жанру ще більш лякаючими та реалістичними.

Хорор-шутер з елементами виживання поєднує в собі екшен, тактичні перестрілки та напружену атмосферу страху. У таких іграх головний герой часто має обмежену кількість ресурсів, що змушує гравця ретельно планувати свої дії. Основні особливості жанру:

- взаємодія з довкіллям та пошук ресурсів для виживання;
- обмежена кількість боєприпасів і необхідність стратегічного використання зброї;
- напружене оточення, що створює постійний психологічний тиск;
- підвищена складність ігрового процесу, що змушує гравця адаптуватися до ситуації;
- часто використовується незвичайна подача історії через записки, флешбеки або загадкові видіння.

Одним із найяскравіших прикладів хорор-шутера з елементами виживання є «F.E.A.R.» – гра, що поєднує напружену атмосферу жахів, динамічні перестрілки та елементи виживання. Гравець бере на себе роль спецагента елітного підрозділу F.E.A.R., який розслідує паранормальні явища, пов'язані з загадковою дівчинкою на ім'я Альма [10].

Гра пропонує унікальну суміш екшену та хорору: штучний інтелект ворогів розроблений на високому рівні, що робить бої складними та непередбачуваними. Крім того, гра має кінематографічну подачу сюжету, використовуючи флешбеки та раптові видіння, що поступово розкривають страшну правду про експерименти над Альмою. Напружена атмосфера, звуковий супровід та темний, індустріальний дизайн додають грі неповторного стилю. Усе це перетворює ігровий процес на глибоке переживання, яке надовго залишає слід у пам’яті гравця.

На рисунках 1.7 та 1.8 нижче, зображені постер гри та її ігровий скріншот із елементами інтерфейсу та неігровими персонажами відповідно.

Рисунок 1.7 – Постер гри F.E.A.R.

Рисунок 1.8 – Скріншот гри F.E.A.R.

Ще більш цікавим прикладом шутера з елементами хорору та виживання є гра Cry of Fear. Вона вирізняється серед інших тим, що була створена як модифікація для гри Half-Life, проте з часом перетворилася на повноцінний самостійний проєкт. Гравець бере на себе роль молодого чоловіка на ім'я Саймон, який опиняється в темному і зловісному місті. Його основна мета – розібратися в тому, що з ним сталося, і знайти вихід із жахіття, що розгортається навколо.

Cry of Fear має чудову атмосферу завдяки унікальному візуальному стилю, низькополігональним, але детально продуманим локаціям, що створюють відчуття пригніченості та страху. Гра поєднує елементи виживання та шутера: у гравця обмежені ресурси, а зустрічі з ворогами – це завжди серйозний виклик. Крім класичних монстрів, значну роль у грі відіграють психологічні аспекти – галюцинації головного героя, його страхи та минулі травми [11].

Сюжет Cry of Fear розкривається поступово через розмови, знайдені записки та візуальні підказки. Він зачіпає глибокі теми самотності, депресії та психічних розладів, що робить гру не просто жахастиком, а й психологічним дослідженням внутрішнього світу людини. Загалом, Cry of Fear – це не тільки гра, а й сильний емоційний досвід, який запам'ятовується надовго. На рисунках 1.9 та 1.10 нижче, зображені постер гри та її ігровий скріншот із елементами інтерфейсу та неігровими персонажами відповідно.

Рисунок 1.9 – Постер гри Cry of Fear

Рисунок 1.10 – Скріншот гри Cry of Fear

Загалом виділимо такі елементи, що будуть, так чи інакше, знаходитись в кожній грі, та які можна та навіть варто буде використати при розробці рішення:

– усі ігри містять динамічну механіку ведення бою, з використанням різноманітної зброї;

– важливим елементом геймплею є пошук і економне використання боєприпасів, аптечок тощо;

– ігри мають лабіринтоподібні рівні з секретними кімнатами, схованками та пастками а також дверми, ліфтами, телепортами чи загалмо об'єктами, що можна активувати;

– присутнє використання світла, тіней, звукового супроводу для створення відповідного настрою;

– штучний інтелект ворогів особливо виражений у F.E.A.R., що робить бої динамічнішими та цікавішими;

– використано психологічний вплив, який найкраще видно у хорор проєктах, як Cry of Fear та F.E.A.R., що використовують флешбеки, галюцинації та сюжетні елементи для посилення напруги;

– Doom та Cry of Fear підтримували створення користувацького контенту;

– є поєднання екшену (шутера) та жахів (хорору) для посилення напруги;

– присутнє поступове розкриття історії через знайдені документи, розмови або флешбеки в ігровому світі.

Аналіз показує, що успіх гри залежить від поєднання динамічних механік, глибокої атмосфери та інтелектуальних викликів. Це створює унікальний баланс між активним ігровим процесом і зануренням у сюжетну складову. Важливі не лише технічні аспекти, але й емоційний зв’язок через психологічні елементи, що поглиблюють досвід гравця, змушуючи його відчувати себе частиною подій.

Оскільки на даний момент присутнє чітке розуміння необхідних елементів для створення успішного продукту, наступним кроком буде постановка конкретних задач для їх реалізації та визначення вимог до рішення. Це дозволить ефективно спланувати подальший етап розробки та уникнути зайвих витрат.

1.3 Визначення функціональних та нефункціональних вимог

Проєкт передбачає проєктування та розробку шутера з елементами жанрів «хорор» та «виживання», стилізованого під VHS/аналоговий формат. Події розгортатимуться у світі, де катастрофа сталася нещодавно, а головний герой намагається зрозуміти ситуацію та вижити. Гра акцентує увагу на напружених боях, дослідженні оточення та глибокому психологічному впливі через звук і візуальні ефекти.

Геймплей базується на вигляді від першого лиця, обмежених ресурсах, дослідженні локацій та постійному відстрілу противників. Вороги не лише фізично загрожують, а й виклюкають відчуття тривоги та невизначеності. На рисунку 1.11 зображений стандартний вигляд від першого лиця в грі «Cry of Fear».

Рисунок 1.11 – Вигляд від першого лиця

Проєкт орієнтований на персональні комп'ютери з можливістю портування на консолі в майбутньому. Реалістичне аудіо, VHS-ефекти, деталізоване середовище забезпечать максимальне занурення. Розповідь історії здійснюватиметься через знайдені документи, записи та середовище загалом [40].

Щоб забезпечити якісний ігровий досвід, необхідно врахувати як функціональні, так і нефункціональні вимоги до проєкту. Вони визначають основні механіки геймплею, взаємодію з оточенням, поведінку ворогів, а також технічні аспекти, що впливають на продуктивність і оптимізацію. Надалі буде представлено перелік вимог [12].

Функціональні вимоги:

- вид від першої особи;
- бойова система з холодною та вогнепальною зброєю;
- обмежені ресурси (боєприпаси, здоров'я, спорядження);
- VHS-стилізація (шум, спотворення, артефакти плівки);
- динамічне освітлення та глибокі тіні;
- атмосферний саундтрек з аналоговими шумами та радіоперешкодами;
- реалістична звукова картина (кроки, дихання, перезарядка зброї);
- динамічна зміна аудіофону (звук ворогів, скрімери).

Нефункціональні вимоги:

- стабільний FPS (мінімум 60 на середніх налаштуваннях);
- підтримка Windows (Linux і Mac – за можливості);
- підтримка контролерів та клавіатури/миші;
- захист від збоїв, автозбереження прогресу;
- мінімалістичний інтерфейс для кращого занурення.

1.4 Висновки. Дослідження предметної області

Проведений аналіз предметної області засвідчив, що індустрія відеоігор є складною, багаторівневою системою з чітко структурованими етапами розробки, які охоплюють як технічні, так і творчі аспекти. Особливу увагу приділено специфіці жанру шутерів із елементами хорору та виживання, що потребує збалансованого підходу до геймплейних механік, атмосфери та візуального стилю. Жанрова структура таких ігор передбачає тісну інтеграцію емоційного впливу, напруги та динаміки подій, що ускладнює процес проєктування. Визначено ключові вимоги до майбутнього проєкту та підтверджено релевантність обраної концепції з огляду на сучасні тенденції та очікування цільової аудиторії. Отримані результати стануть надійним підґрунтям для подальшої розробки гри та формування її унікального ігрового досвіду.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Проєктування архітектури та структури системи

Розробка ігор на Unity є складним процесом, який вимагає від розробників вибору правильного підходу до побудови архітектури гри. Від цього залежить продуктивність, масштабованість та можливості гри в майбутньому, а також її розробка загалом. Серед найпопулярніших підходів виділяють об'єктно-орієнтоване програмування (ООП), компонентно-орієнтовану архітектуру (КОП), Entity Component System (ECS) і використання Scriptable Objects. Розглянемо кожен з них детальніше [13].

Entity Component System (ECS) – новий підхід, орієнтований на високу продуктивність. Архітектурний патерн для розробки ігор, що дозволяє ефективно обробляти велику кількість об'єктів за рахунок відокремлення логіки від даних. ECS підносить до абсолюту принцип Composition Over Inheritance (композиція більш важлива, ніж успадкування) і може бути окремим прикладом Data Oriented Design (орієнтованого на дані дизайну, далі DOD), однак це вже залежить від інтерпретації патерну в конкретній ситуації конкретною реалізацією.

Цей підхід використовується в таких великих ігрових проєктах, як «Cities: Skylines II», де важливо керувати тисячами об'єктів у реальному часі [14]. Завдяки цьому забезпечується висока масштабованість та ефективність обробки ігрових даних. Архітектура Entity Component System була зображена на рисунку 2.1.

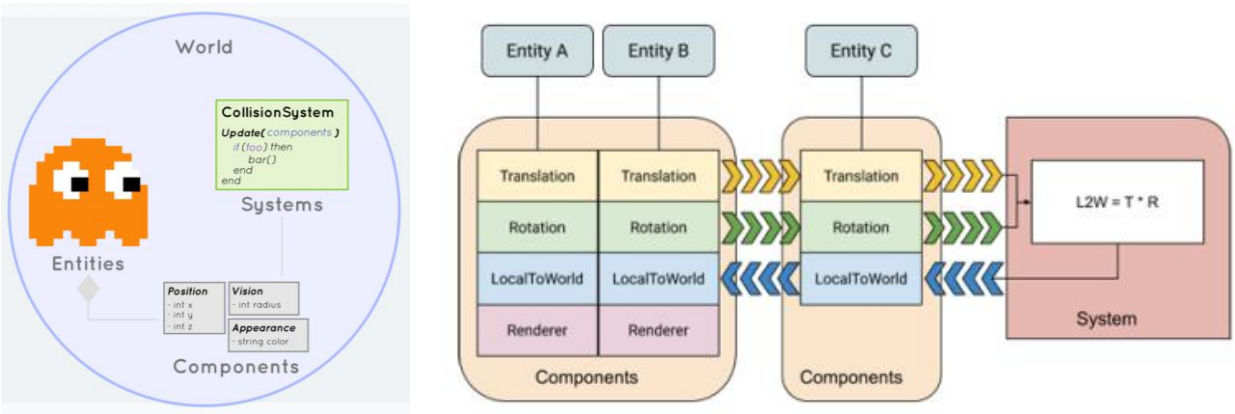


Рисунок 2.1 – Зображення ECS

Об'єктно-орієнтоване програмування (ООП) є класичним підходом, де логіка гри реалізується через класи, які взаємодіють між собою через методи та змінні. ООП використовується в багатьох інді-проєктах, де гра має невелику кількість об'єктів, що активно взаємодіють між собою. Наприклад, гра «Hollow Knight», створена на Unity, демонструє ефективність ООП у побудові глибокої системи ворогів та персонажів [15].

Data-Oriented Design (DOD) та DOTS (Data-Oriented Technology Stack) – це сучасні підходи, які використовуються для підвищення продуктивності та оптимізації роботи гри в Unity. DOD фокусується на ефективному розташуванні даних у пам'яті, що дозволяє зменшити кеш-промахи процесора та підвищити швидкість обробки. Він використовується для складних обчислень, таких як фізика, обробка великих масивів об'єктів та AI [16]. На рисунку 2.2 зображено архітектуру ООП та DOD.

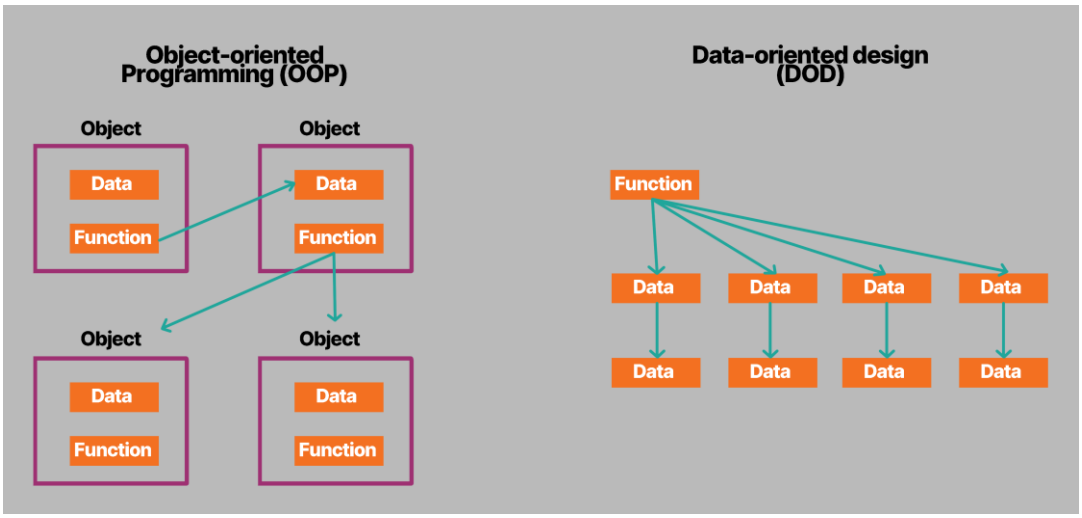


Рисунок 2.2 – ООП та DOD

DOTS є реалізацією DOD в Unity, що включає ECS, Burst Compiler і C# Job System. Ця технологія дозволяє значно підвищити продуктивність, використовуючи багатопотокову обробку та ефективне управління пам'яттю. Вона оптимізована для сучасних процесорів, де паралелізм відіграє ключову роль у досягненні високої частоти кадрів. Наприклад, Unity DOTS використовується в таких проєктах, як «Megacity», що демонструє здатність рушія обробляти тисячі об'єктів одночасно.

Об'єктно-орієнтоване програмування (ООП) та Data-Oriented Design (DOD) / Data-Oriented Technology Stack (DOTS) представляють собою два різні підходи до побудови ПЗ. ООП передбачає структурування коду навколо об'єктів, які інкапсулюють як дані, так і логіку, що забезпечує зручність розширення та підтримки програмної архітектури. Натомість DOD орієнтований на оптимізацію розташування даних у пам’яті з метою підвищення продуктивності, особливо у випадках обробки великих обсягів однотипних об'єктів. Спроектований вигляд системи Unity DOTS зображений на рисунку 2.3.

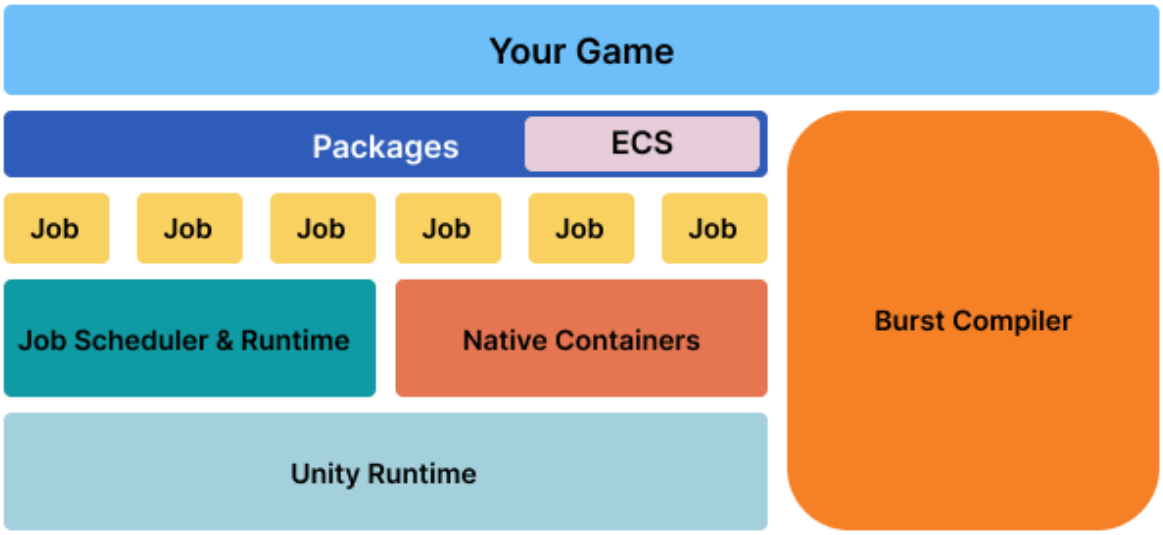


Рисунок 2.3 – Unity DOTS

Зважаючи на те, що рушій Unity використовує мову програмування C#, яка за своєю природою є об'єктно-орієнтованою, повна відмова від ООП у процесі розробки є недоцільною. У більшості випадків взаємодія з внутрішніми механізмами Unity відбувається саме через ООП так чи інкаше. Відтак, доцільно не протиставляти ці два підходи, а, навпаки, зосередитися на визначенні ефективних способів їх поєднання залежно від характеру завдань, що вирішуються в межах проєкту.

Так, структурна логіка взаємодії між об’єктами гри може реалізовуватися за допомогою ООП, тоді як компоненти, що потребують високої продуктивності (наприклад: фізика чи управління великою кількістю однотипних сутностей), доцільно реалізовувати із використанням підходів DOD або технологій DOTS.

Останній включає такі інструменти, як Entity Component System (ECS), C# Job System і Burst Compiler, які дозволяють реалізувати високопродуктивну багатопотокову обробку даних.

Прикладом поєднання обох підходів є використання Scriptable Objects – інструменту Unity, що дозволяє зберігати конфігураційні дані поза межами конкретної сцени. З одного боку, це відповідає принципам ООП, забезпечуючи структурованість і зручність у роботі з даними, а з іншого – дозволяє підвищити ефективність керування ресурсами проєкту та спростити процес балансування ігрових параметрів [17]. Взаємодія Scriptable Objects з іншими елементами проекту зображена на рисунку 2.4.

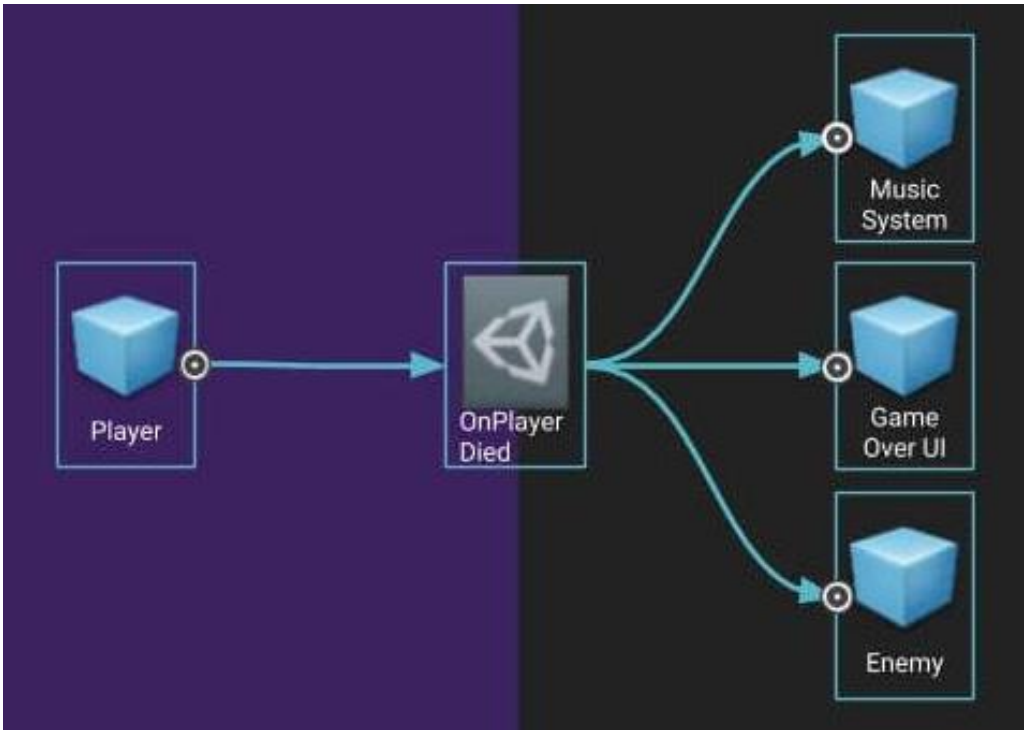


Рисунок 2.4 – Зображення Scriptable Objects

У контексті розробки проєкту, що включає бойову систему від першої особи, певну поведінку ворогів, взаємодію з навколишнім середовищем, доцільним є використання комбінованого підходу: застосування ООП для логічної структури та DOD/DOTS – для оптимізації продуктивності. Такий підхід дозволяє забезпечити як гнучкість архітектури, так і високу ефективність при обробці складних обчислювальних задач, характерних для сучасних ігор [34].

2.2 Проєктування модулів гри

Для правильного проведення проєктування гри в першу чергу варто створити її загальний план, щоб розуміти те, з чим прийдеться працювати уже на абстрактному, початковому рівні. Найкращим вибором для виділення основних осіб, механізмів, інструкцій, вхідних та вихідних даних є діаграма типу IDEF0.

IDEF0 – це графічний метод подання функцій та процесів у певних проєктованих системах, що корисний як для невеликих систем управління, так і для складних систем загалом. Такий метод подання надає можливість проєктуваня, аналізувати, так, як наслідок, створювати або покращувати бізнес-системи відмінно зображуючи бізнес-процеси та полегшуючи їх розуміння [18].

Загалом діаграми типу IDEF0 зображують у вигляді стрілок вхідні та вихідні дані, інструкції або вказівки та механізми. Окрім вказаних елементів також існують процеси, функції, що часто зображені квадратами. Стрілки показують також залежності між процесами, порядок їх виконання.

Першочерговим в створенні плану за допомогою IDEF0 є створення контекстної діаграми, в нашому випадку така матиме назву «Геймплей гри». Контекстна діаграма зображена на рисунку 2.5.

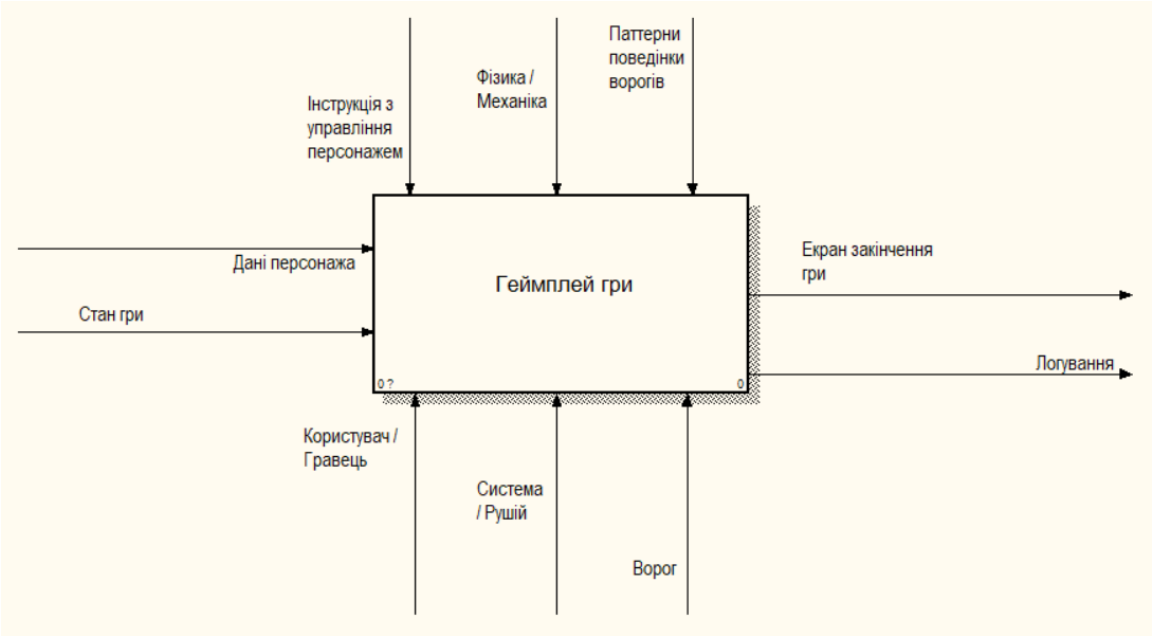


Рисунок 2.5 – Контекстна діаграма А-0 «Геймплей гри»

Геймплей – це все, що стосується того, як ми граємо в гру. Простими словами, це те, як організовано взаємодію гравця з грою: як він керує персонажами, які задачі потрібно виконати, які механіки та правила гри використовуються. Геймплей включає в себе все: від того, як персонаж рухається, до того, як вирішуються проблеми або відбуваються битви.

Наступним кроком в описі гри є декомпозиція створеної контекстної діаграми. Декомпозиція створеної контекстної діаграми – це процес розбиття загальної системи, що зображена на контекстній діаграмі, на менші, більш детальні підсистеми або етапи. Контекстна діаграма зазвичай дає загальне уявлення про систему, показуючи її основні компоненти та їхні взаємодії з зовнішнім середовищем (наприклад, користувачами або іншими системами). Діаграма декомпозиція представлена на рисунку 2.6.

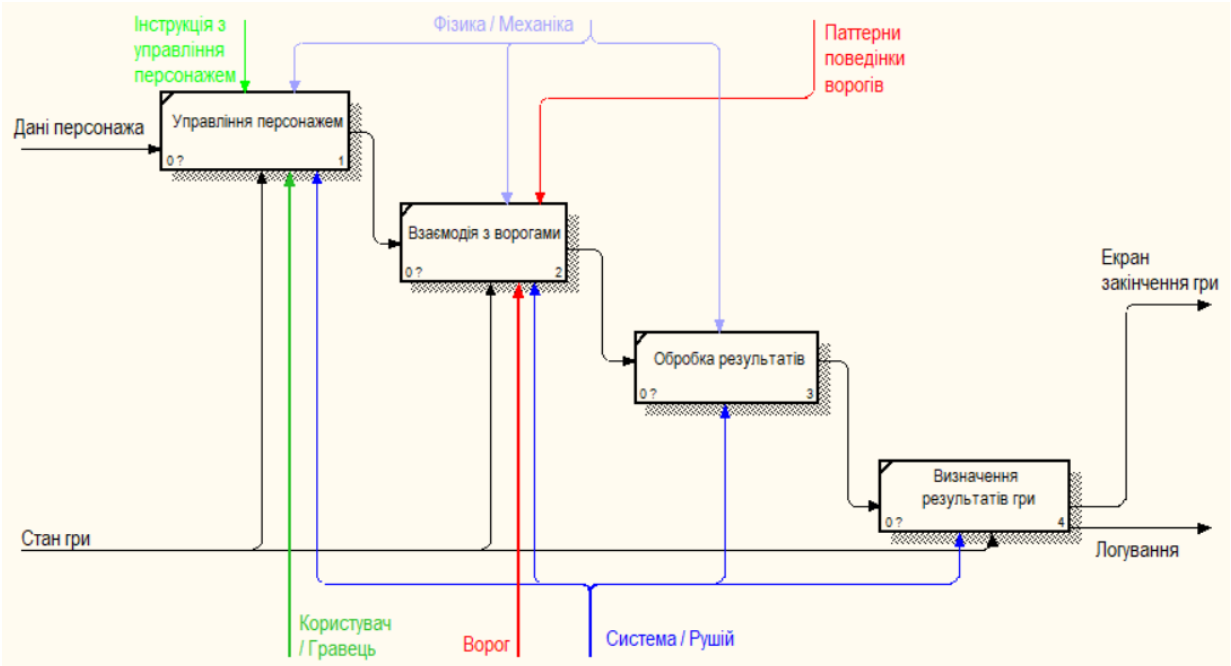


Рисунок 2.6 – Діаграма декомпозиції А-1 «Геймплей гри»

У підсумку, створення загального плану гри через діаграму IDEF0 дозволяє чітко визначити основні елементи та процеси гри, а також їх взаємодії. Це дає змогу побудувати структуру, яка відображає ключові функції, механізми та дані, що важливо для подальшої розробки та створення більш детальних діаграм, як діаграми: станів, варіантів використання, тощо.

2.3 Детальне проєктування

У пункті «Детальне проєктування» варто, опираючись на проведений аналіз в попередньому пункті: визначити основні механіки гри, спроєктувати їх у загальному вигляді та створити відповідні діаграми для кращого розуміння та візуалізації. Для цього планується використання діаграм UML.

UML (Unified Modeling Language) – це стандартна графічна мова для моделювання об'єктно-орієнтованих систем, яка дозволяє візуалізувати, різні аспекти програмного забезпечення (ПЗ). Діаграми UML допомагають краще розуміти структуру, поведінку та взаємодії компонентів системи.

Загалом в цьому пункті будуть використані нижче приведені діаграми:

– варіантів використання (use case diagram) – діаграма для ілюстрації основних сценаріїв взаємодії гравця з системою, наприклад: сутичка з ворогами, дослідження середовища, збір ресурсів;

– діаграма послідовності (sequence diagram) відповідає за деталізацію взаємодій між об'єктами в межах певних сценаріїв, наприклад, під час атаки чи взаємодії з предметами;

– блок-схема (flowchart) використовується для моделювання послідовності дій, які виконуються в межах конкретних механік гри, таких як рух персонажа чи бій, перестрілка, споживання ресурсів [19]. На рисунку 2.7 зображене розшифрування абревіатури.

Рисунок 2.7 – Unified Modeling Language

Для проєктування логіки гри: написання діаграм та завдань для програмної та графічної реалізації, варто визначити основних акторів проєкту, описати систему продукту загалом, та її механіки.

Головний герой (Гравець) – персонаж, яким керує користувач, зазвичай має різні характеристики, в нашому випадку наступні: показники здоров'я та витривалості, голоду та морального стану.

Головний герой має описаний далі функціонал:

– може переміщатися по ігрових рівнях, досліджуючи навколишнє середовище та виконуючи завдання;

– має можливість вибирати зброю для боротьби з ворогами, включаючи ближній бій та вогнепальну зброю;

– контролює обмежені ресурси, такі як боєприпаси, ліки та спорядження;

– збирає предмети, включаючи зброю, медикаменти, документи та інші ресурси, необхідні для виживання;

– може ховатися від ворогів, щоб уникнути небезпеки або спробувати підійти з іншого боку.

Початковий ворог (Ворог) – звичайний противник, що буде лякати та атакувати персонажа, створений на першому рівні для демонстрації механік користувачу та вводу його в гру.

Початковий ворог має описаний далі функціонал:

– може реагувати на шуми, залишені гравцем, і переслідувати його;

– негайно нападає на гравця, коли той знаходиться в його зоні досяжності.

Функціональність системи:

– система забезпечує зміну навколишнього середовища, зокрема освітлення, звуків та інших елементи.

– в залежності від освітлення та розташування об'єктів, система може змінювати видимість гравця та ворогів.

– система дозволяє ворогам реагувати на звуки, наприклад, постріли або падіння предметів, щоб залучити їх увагу.

Загальний функціонал:

– «Гравець» має бути здатним пересуватися по рівнях, взаємодіяти з об'єктами, збирати предмети та вести боротьбу з ворогами.

– «Ворог» повинен мати можливість реагувати на поведінку гравця, переслідувати його та атакувати.

– «Система» відповідає за адаптацію до змін, таких як освітлення, звуки, а також управління механіками взаємодії гравця та ворогів.

Поведінку ворогів, а також загалом більшості можливих неігрових персонажів, зображено на рисунку 2.8.

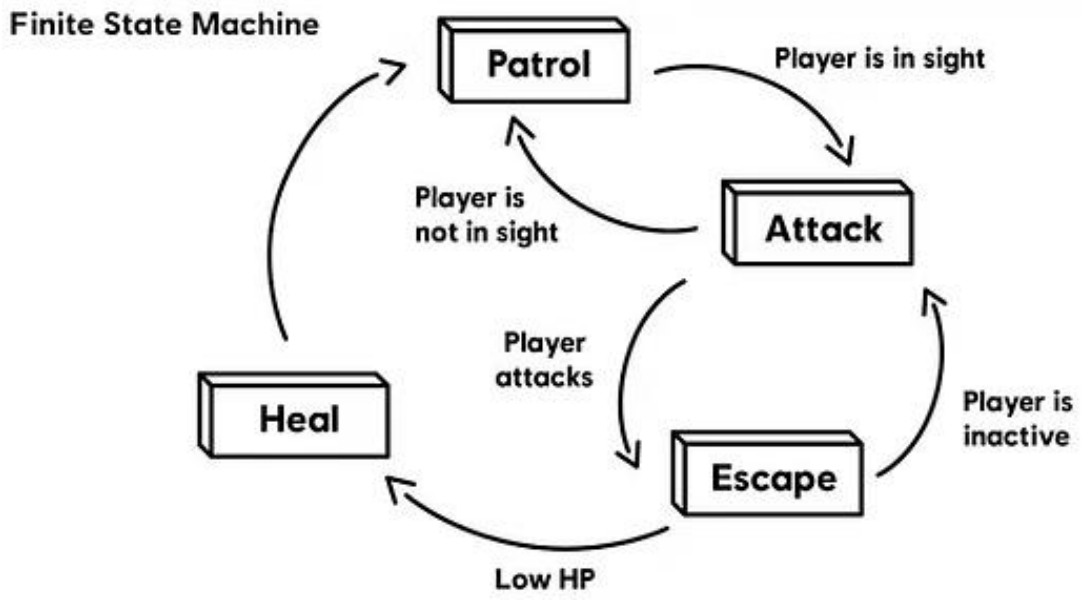


Рисунок 2.8 – Стандартна схема поведінки ворогів та НПС

Після проведення первинного аналізу та етапу проєктування було здійснене ретельне планування функціоналу основних компонентів та визначення ключових акторів проекту.

Для наочного відображення взаємодії між користувачами та системою була розроблена діаграма варіантів використання. Вона дозволяє у графічному вигляді продемонструвати основні можливості проекту, основні сценарії взаємодії гравця з системою, що сприяє ефективнішому сприйняттю його логіки та функціональності.

Зображені варіанти використання позначенні кольорами, що відповідаю більше гравцю – зелений, ворогу – червони, та спільні варіанти – сірим. Спроектована діаграма варіантів використання зображена на рисунку 2.9.

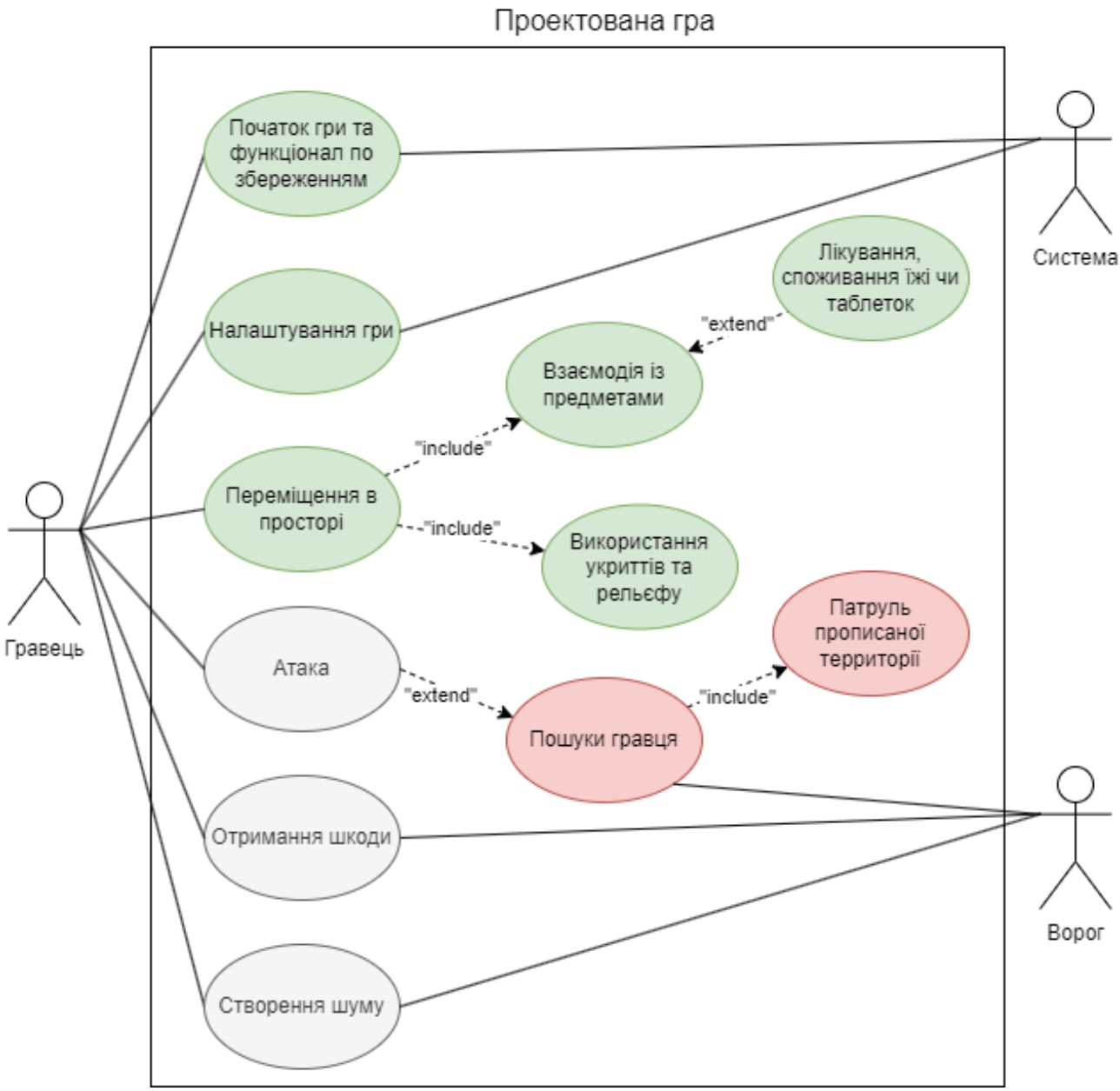


Рисунок 2.9 – Діаграма варіантів використання гри

Відповідно до проведених аналізу та проєктуванню діаграми варіантів використання будуть обрані основні загальні сценарії відповідно: «Атака», «Отримання шкоди» та «Створення шуму». Сценарії усі використовуються як гравцем, так і ворогом (на стороні ПК). Перераховані сценарії будуть розписані більш детально за допомогою діаграм послідовностей (sequence diagram).

Для моделювання послідовності дій, які виконуються в межах конкретних механік гри або загалом певних сценаріїв варто спроектувати діаграму Flowchart. Ця діаграма послідовності описує взаємодію між гравцем, грою та ворогом під час трьох основних сценаріїв: атака, отримання шкоди та створення шуму. Вона показує перевірки умов, обробку результатів (наприклад, втрата HP чи смерть), а також реакцію ворога на дії гравця. Спроектована діаграма послідовності зображена на рисунку 2.10.

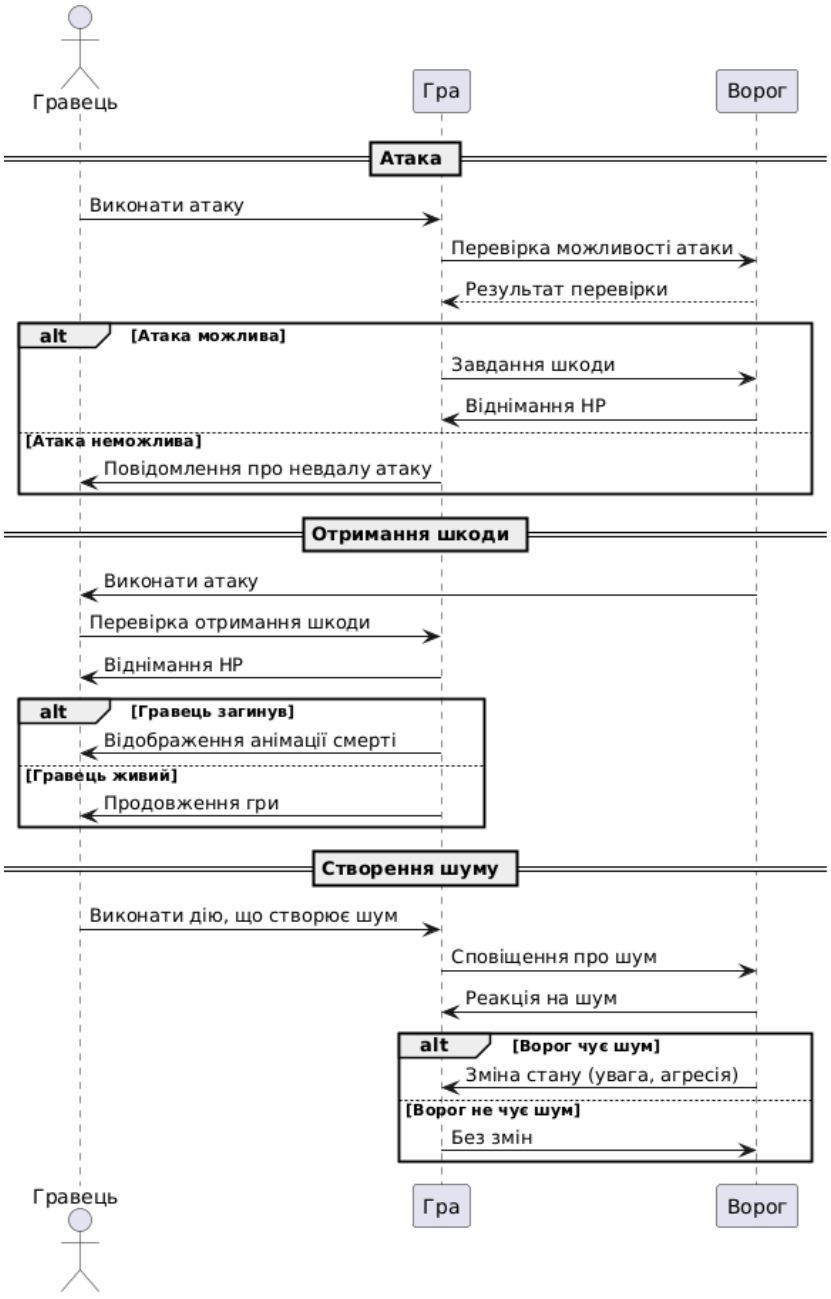


Рисунок 2.10 – Діаграма послідовностей сірих сценаріїв

Діаграма діяльності ілюструє послідовність дій гравця під час переміщення, взаємодії з об’єктами та виживання. Рішення про рух запускає вибір типу переміщення: ходьба, біг, стрибок або скрадання, яке впливає на рівень шуму та поведінку персонажа. У разі наявності об’єктів гравець може підібрати предмет або відкрити двері. При потребі він також може вилікуватися або вжити їжу, що впливає на його стан. Діаграма відображає логіку адаптивної поведінки гравця у змінному ігровому середовищі.

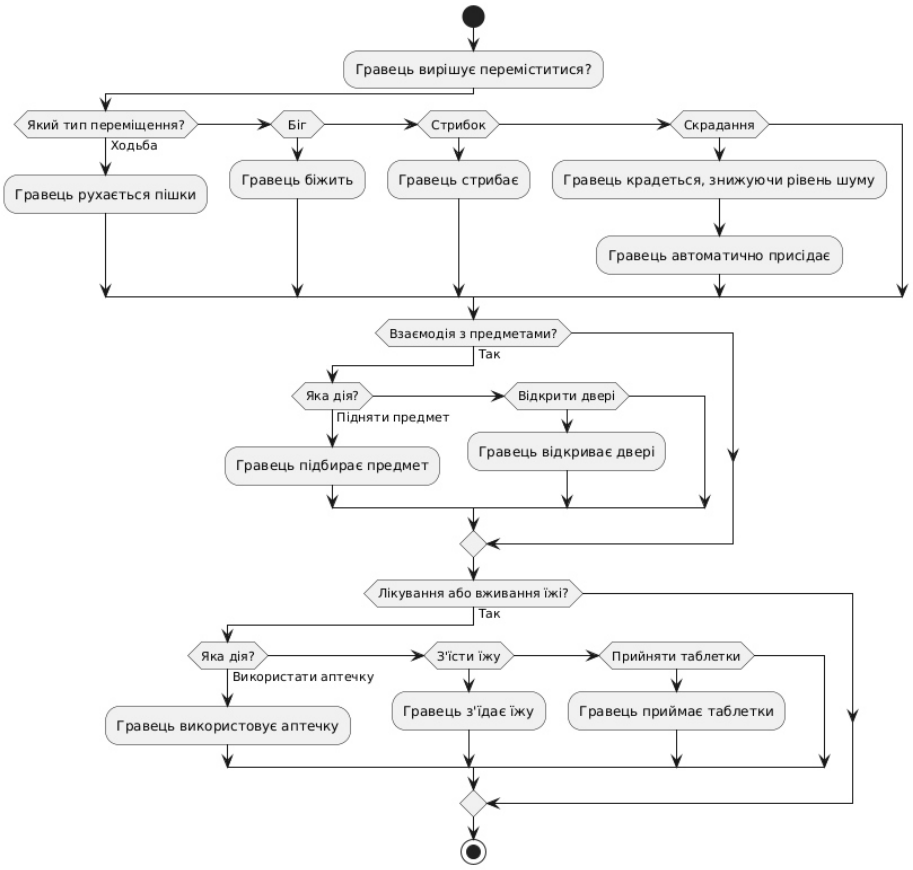


Рисунок 2.11 – Блок схема для сценарію «Переміщення в просторі»

У цьому розділі було спроєктовано основні механіки гри за допомогою UML-діаграм в вигляді сценаріїв для кращого розуміння взаємодії гравця та системи та логічному рівні. Використовувалися діаграми варіантів використання, послідовності та блок-схеми для моделювання сценаріїв, таких як переміщення, бій і взаємодія з предметами. Були визначені ключові актори: гравець, ворог та система, з їхніми функціями та ролями в грі. Проведена робота дає чітке підґрунтя для детального моделювання системи, проєктування модулів та реалізації надалі.

2.4 Аналіз та вибір технологій для реалізації гри

Для успішного створення проєкту та проведення тестування одним з перших і найважливіших етапів після етапу проєктування є підбір інструментів для реалізації гри. Це включає вибір технологій, які повинні не лише відповідати вимогам проєкту, а й задовольняти можливості та досвід розробників. Адже вибір правильних інструментів безпосередньо впливає на функціональність гри, ефективність розробки, здатність підтримувати різні платформи та зручність роботи над проєктом.

Серед усіх можливих інструментів для розробки гри дві складові є особливо важливими: мова програмування та середовище розробки проєкту. Мова програмування визначає, як буде реалізована основна логіка гри, алгоритми, а також як буде організовано взаємодію між гравцем і грою. В свою чергу, середовище розробки надає всі необхідні інструменти та ресурси для зручної, продуктивної роботи над проєктом, забезпечуючи підтримку різних платформ, оптимізацію продуктивності та полегшуючи тестування. Вибір двох складових є ключовим, оскільки від них залежить не лише успіх реалізації проєкту, а й ефективність роботи команди та здатність адаптувати гру до різних умов.

Для вибору найбільш підходящої мови програмування для розробки ігор необхідно врахувати технічні можливості мови, її сумісність з середовищами, продуктивність, зручність. Плюсом також будуть: підтримка платформ окрім Windows. У наступній таблиці порівнюються майже єдині та найбільш популярні мови програмування для створення ігор: C#, C++ [21-22].

Таблиця 1 – Порівняння мов програмування

Критерій	C#	C++
Тип мови	Високорівнева, об'єктно-орієнтована, керована	Мова низького рівня, підтримує процедурне, об'єктно-орієнтоване програмування

Продовження таблиці 1 – Порівняння мов програмування

Платформи	Переважно Windows (з підтримкою Linux через .NET Core, macOS через Xamarin)	Кросплатформена, використовується на Windows, Linux, macOS та інших платформах
Продуктивність	Трохи нижча, завдяки керуваному середовищу	Висока продуктивність, оскільки код компілюється в машинний код
Система типів	Сильна типізація, всі змінні мають визначений тип	Сильна типізація, але надає більше можливостей для низькорівневого доступу
Простота у використанні	Легкий синтаксис, вища абстракція, зручна для швидкої розробки	Складний синтаксис, потребує знань низькорівневих аспектів
Управління потоком	Підтримує багатопоточність	Підтримує багатопоточність
Інструменти та IDE	Visual Studio, JetBrains Rider, Xamarin Studio, Unity для розробки ігор	Visual Studio, CLion, Eclipse, Qt Creator, підтримка CMake
Бібліотеки	Величезна бібліотека .NET, зокрема для розробки додатків, веб-сервісів, ігор	Стандартна бібліотека C++ з великою кількістю низькорівневих інструментів
Інтероперабельність	Легко взаємодіє з іншими мовами через COM, P/Invoke	Може працювати з іншими мовами, не без додаткових зусиль

Продовження таблиці 1 – Порівняння мов програмування

Підтримка об'єктно-орієнтованого програмування	Повна підтримка	Повна підтримка
Підтримка функціонального програмування	Обмежена, але присутня через делегати та LINQ	Обмежена підтримка через шаблони та функціональні обгортки
Користувацька база	Велика, особливо серед розробників веб-додатків, мобільних додатків та ігор (Unity)	Використовується в системному програмуванні, обчисленнях, вбудованих системах
Основне застосування	Розробка бізнес-додатків, мобільних додатків, веб-додатків, ігор на Unity	Розробка високопродуктивних додатків, ігор,
Управління пам'яттю	Автоматичне збирання	Ручне управління

З огляду на проведений аналіз можна зробити наступні висновки. Мова C++ підходить для уже навчених розробників, що вміють працювати в першу чергу з нею, оскільки мова має високий поріг входження. C# в свою чергу має нижчий поріг входження, відповідно на цьому моменті менш затратна. Вона відповідає архітектурі, шаблонам. Мова програмування зображена на рисунку 2.12.

Рисунок 2.12 – Обрана мова

Для вибору оптимальних технологій для розробки ігор важливо порівняти різні доступні середовища, оцінюючи їхні можливості, зручність використання та відповідність специфікаціям проєкту [23-25].

Таблиця 2 – Порівняння ігрових движків на C#

Критерій	Unity	MonoGame	Godot (C#)
Платформи	Підтримка багатьох платформ: PC, консоль, мобільні.	Підтримка Windows, macOS, Linux, мобільні платформи	Підтримка Windows, Linux, macOS, мобільні платформи
Документація та ресурси	Величезна кількість онлайн-ресурсів, документація, спільнота	Добре розвинута документація, менш активна спільнота	Розвивається, але обмежена документація та ресурси
Легкість використання	Легкий старт, хороша підтримка C#, готові шаблони	Потрібно більше налаштувань, не такий зручний для початківців	Легкий для початківців, але сфокусованіший на своїх можливостях
Ігрові механіки та фреймворки	Багато готових інструментів, включаючи фізику, анімацію	Базовий набір інструментів, потрібно більше налаштувань	Хороші можливості для 2D ігор, обмежена підтримка 3D
Підтримка C#	Повна підтримка C# як основної мови	Підтримка C# через XNA	Повна підтримка C# як основної мови
Кросплат-форменість	Підтримка практично всіх платформ, включаючи VR/AR	Обмежена кількість платформ	Підтримка основних платформ, але не так масштабно

З огляду на проведений аналіз, створену таблицю, в якій було приведено порівняння середовищ програмування відповідно до певного функціоналу, можна зробити висновки про те, що найкращим вибором є саме двигун Unity. Навіть враховуючи його умовну платність та певні сумнівні дії зі сторони Unity Software Inc останнім часом – двигун, в порівнянні з іншими, все ще має досить сильну аудиторію, ця спільнота досвідчена та широка, що дозволить в крайньому випадку знайти допомогу, уже готові рішення або ж навіть людей в компанію для розробки або розширення проєкту. Окрім того Unity має:

– безкоштовні тренувальні курси, для ознайомлення з азами двигуна;
– підтримку різних платформ, в тому числі VR;
– повну підтримка C# як основної мови;
– кращу оптимізацію ніж у конкурентів;
– простий для розуміння редактор;
– величезну кількість онлайн-ресурсів, документацію.

Середовище розробки обраного двигуна, а точніше – робочий простір цього середовища, зображений на рисунку 2.13.

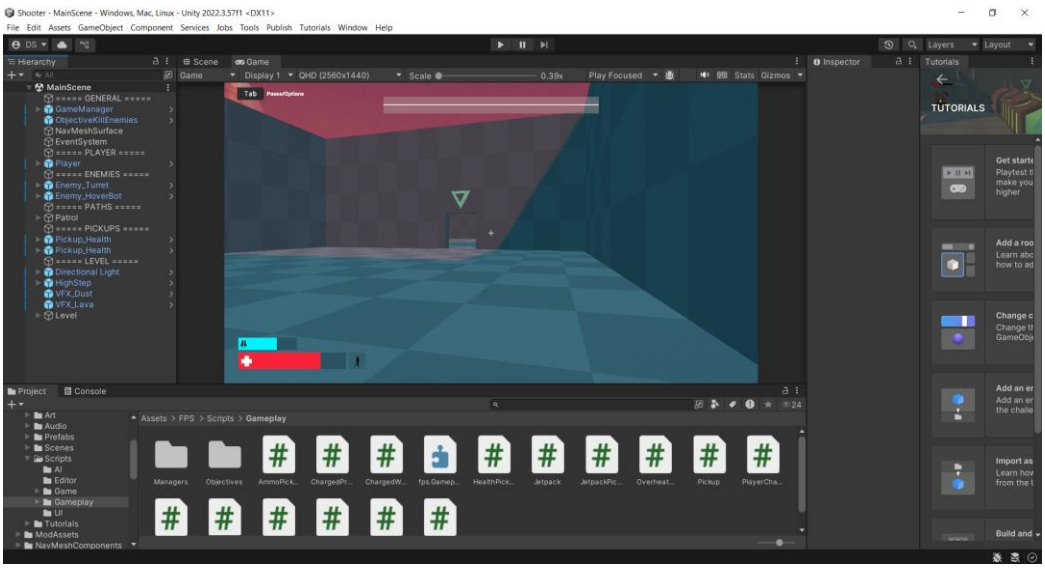


Рисунок 2.13 – Робочий простір Unity

Окрім вибору основних інструментів варто також звернути увагу на відео- та аудіо- супровід в грі, тобто анімація та звуки, музика відповідно. Для збереження ресурсів, оптимізації розробки та зосередженні в першу чергу на

проєктуванні та написанні програмного коду, варто звернутись до використання близьких по вигляду до першочергової задумки, проте безкоштовних і швидких до підключення асетів. Асет (від англ. *asset*) – це уже готовий до використання ресурс або елемент, що використовується у розробці ігор. Асетами можуть бути як графічні, так і звукові ресурси. Також сценарії, програмні елементи, шаблони інтерфейсів, рівнів та частини UI. Асети використовуються для швидшого створення гри без необхідності вручну розробляти кожен елемент. Багато ресурсів можна знайти в онлайн-магазинах для розробників, наприклад, Unity Asset Store, що дозволяє заощаджувати час, використовуючи вже готові матеріали. В свою чергу Unity Asset Store – це бібліотека, що надає доступ до великої кількості безкоштовних асетів у Unity, що створені як активними членами спільноти, так і видані від лиця самої компанії [28]. Бібліотека асетів та додаткових матеріалів загалом зображена на рисунку 2.14.

Рисунок 2.14 – Unity Asset Store

Щоб запобігти випадковому знищенню проєкту та надати можливість розробнику зберігати вчасно свій прогрес, з можливістю відкотити поточну версію до минулої варто обрати систему керування версіями як черговий інструмент. Для цього доцільно буде провести аналіз та порівняти найпопулярнішу систему «Git», вебсервіс «GitHub» та офіційну систему від обраного движка – Unity Version Control [26-27].

Критерій	GitHub (Git)	UVC
Легкість налаштування	Потрібно встановити Git, створити .gitignore для Unity	Вбудований у Unity, простіше почати
Гнучкість	Підтримує гілки, мержі, відкат змін, Git LFS	Обмежений функціонал, гілкування менш зручне
Робота з великими файлами	Git LFS підтримує великі ассети (моделі, текстури)	Може мати проблеми з великими файлами
Сховище в хмарі	Безкоштовно	Безкоштовно до 5GB
Швидкість роботи	Локальні операції швидкі, але пуш/пул вимагає інтернету	Операції відбуваються в реальному часі в Unity
Контроль змін	Точний контроль за кодом та ассетами, історія комітів	Менш детальний контроль, зручний для ассетів
Простота відкату	Можна легко повернутися до будь-якої версії	Відкат можливий, але не так гнучко як у Git
Інтеграція з іншими інструментами	Підтримує CI/CD, автотести, аналітику	Тільки інтеграція з Unity

З першого погляду Git – класичний та чудовий вибір для кожного розробника, проте в нього є критична проблема для проєктованого продукту. Цією проблемою є відсутність такої ж оптимізації для роботи з асетами та бінарними файлами, яка є в Unity Version Control (UVC). Git неефективно обробляє великі бінарні файли, оскільки зберігає кожну їхню версію в історії, що призводить до швидкого збільшення розміру репозиторію. Також, що критично, як стало відомо – Git не має вбудованих механізмів для коректного злиття змін у файлах сцен та префабів Unity, що може спричинити конфлікти під час командної роботи.

Для ефективного тестування гри, розробленої на Unity, рекомендується використовувати Unity Test Framework (UTF). Цей інструмент дозволяє розробникам створювати та виконувати автоматизовані тести як у режимі редагування (Edit Mode), так і під час гри (Play Mode). UTF інтегрується з бібліотекою NUnit, яка є популярним інструментом для модульного тестування в середовищі .NET.

Ця інтеграція забезпечує гнучкість у написанні тестів для різних платформ, таких як Windows, Android, iOS тощо. Для більш глибокого розуміння процесу тестування в Unity, включаючи практичні демонстрації, корисно переглянути спеціалізовані вебінари та навчальні матеріали, які детально пояснюють налаштування та використання UTF [29].

2.5 Висновки. Проєктування програмного продукту

У результаті опрацювання цього етапу було обґрунтовано доцільність вибору відповідних інструментів розробки з урахуванням поставленого завдання, архітектурної моделі та наявних ресурсів. Встановлено, що платформа Unity у поєднанні з мовою програмування C# є оптимальним рішенням для реалізації ігрового проєкту завдяки описаним вище позитивним сторонам.

Для ефективного контролю версій у проєктах, що містять велику кількість даних та графічних ресурсів, перевагу надано системі Unity Version Control. Застосування Unity Asset Store суттєво оптимізує процес розробки, забезпечуючи швидкий доступ до необхідних ресурсів. Крім того, передбачено використання Unity Test Framework для забезпечення якісного тестування ігрової логіки та функціоналу на різних етапах проєктування.

Такий підхід дозволяє не лише підвищити продуктивність розробки, а й забезпечує високий рівень масштабованості проєкту. Інтеграція обраних інструментів у робочий процес сприяє зменшенню витрат часу на реалізацію типових задач та підвищенню загальної якості програмного продукту.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОДУКТУ

3.1 Реалізація програмної частини продукту

Для демонстрації реалізації продукту, розробленого в ході кваліфікаційної роботи, зі сторони коду буде наведено приклад одного з важливих скриптів, що відповідає за базову механіку взаємодії зі зброєю – зокрема, стрільбу, перезарядку, прицілювання та створення куль.

Спочатку було створено скрипт у середовищі Unity з використанням мови програмування C#. Скрипт отримав назву Weapon.cs. Далі було створено змінні для збереження стану зброї, параметрів стрільби, прицілювання, візуальних ефектів, а також поточного боєзапасу. Це дозволило надалі легко керувати всіма ключовими параметрами зброї через інтерфейс Unity Editor.

Код:

```
public bool isActiveWeapon;
public bool isShooting, readyToShoot;
public float shootingDelay = 2f;
public int bulletsPerBurst = 3;
public float spreadIntensity;
public GameObject bulletPrefab;
public float bulletVelocity = 100;
public GameObject muzzleEffect;
public float reloadTime;
public int magazineSize, bulletsLeft;
public bool isReloading;
```

Потім було створено ShootingMode та WeaponModel – для структурування режимів стрільби та відображення типу поточної зброї. Такі переліки корисні для того, щоб у коді легко визначати і використовувати режими стрільби чи типи зброї без «магічних чисел» або строкових значень.

Код:

```
public enum ShootingMode { Single, Burst, Auto }
public enum WeaponModel { pistol1911, m4 }
```

Далі було необхідно реалізувати метод Awake(), у якому ініціалізувалися всі важливі значення, а також підключався компонент Animator, що відповідав за анімації зброї, розробленої в рамках двигуна. Для подальшого тестування йому було призначено модифікатор доступу internal.

Код:

```
internal void Awake()
{
readyToShoot = true;
burstBulletsLeft = bulletsPerBurst;
animator = GetComponent<Animator>();
bulletsLeft = magazineSize;
spreadIntensity = hipSpreadIntensity;
}
```

Наступним створеним методом був метод Update, в якому реалізувалася перевірка активності зброї та взаємодії користувача зі зброєю (стрільба, прицілювання, перезарядка). Цей метод є ключовим для реалізації безперервного оновлення стану об’єктів гри в режимі реального часу, що забезпечує динамічну та інтуїтивну взаємодію користувача з ігровим середовищем.

Код:

```
if (isActiveWeapon)
{
    if (Input.GetMouseButtonDown(1)) EnterADS();
    if (Input.GetMouseButtonUp(1)) ExitADS();

    if (bulletsLeft == 0 && isShooting)
        SoundManager.Instance.emptyMagazineSound1911.Play();

    if (currentShootingMode == ShootingMode.Auto)
        isShooting = Input.GetKey(KeyCode.Mouse0);
    else
        isShooting = Input.GetKeyDown(KeyCode.Mouse0);

    if (Input.GetKeyDown(KeyCode.R) && bulletsLeft < magazineSize
&& !isReloading &&
WeaponManager.Instance.CheckAmmoLeftFor(weaponModel) > 0)
        Reload();

    if (readyToShoot && isShooting && bulletsLeft > 0)
    {
        burstBulletsLeft = bulletsPerBurst;
        FireWeapon();
    }
}
```

Реалізовано два допоміжні методи – EnterADS() та ExitADS(), що відповідали за перемикання в режим прицілювання (ADS – Aim Down Sights) та вихід з нього, в тому числі і щоб уникнути збоїв із анімаціями в Unity та хрестовиною. Механіка ADS дозволила зробити стрільбу точнішою в прицільному режимі та надати візуальний ефект занурення у процес.

Код:

```
private void EnterADS()
{
    animator.SetTrigger("enterADS");
    isADS = true;
    HUDManager.Instance.middleDot.SetActive(false);
    spreadIntensity = adsSpreadIntensity;
}
private void ExitADS()
{
    animator.SetTrigger("exitADS");
    isADS = false;
    HUDManager.Instance.middleDot.SetActive(true);
    spreadIntensity = hipSpreadIntensity;
}
```

Ключова частина логіки стрільби була реалізована у методі FireWeapon(). У даній частині було реалізовано логіку стрільби, що включала зменшення кількості набоїв, запуск візуального ефекту пострілу, відтворення анімації віддачі, запуск відповідного звукового ефекту, створення кулі з фізичним імпульсом у напрямку пострілу, таймер на знищення кулі, керування затримкою між пострілами та підтримку режиму черги.

Код:

```
private void FireWeapon()
{
    bulletsLeft--;

    muzzleEffect.GetComponent<ParticleSystem>().Play();

    if (isADS)
    {
        animator.SetTrigger("RECOIL_ADS");
    }
    else
    {
        animator.SetTrigger("RECOIL");
    }

    SoundManager.Instance.PlayShootingSound(weaponModel);
```

```
        readyToShoot = false;
        Vector3 shootingDirection =
    CalculateDirectionAndSpread().normalized;
        GameObject bullet = Instantiate(bulletPrefab,
    bulletSpawn.position, Quaternion.identity);
        bullet.transform.forward = shootingDirection;
        bullet.GetComponent<Rigidbody>().AddForce(shootingDirection *
    bulletVelocity, ForceMode.Impulse);
        StartCoroutine(DestroyBulletTime(bullet,
    bulletPrefabLifeTime));

        if (allowReset)
        {
            Invoke("ResetShot", shootingDelay);
            allowReset = false;
        }
        if (currentShootingMode == ShootingMode.Burst &&
    burstBulletsLeft > 1)
        {
            burstBulletsLeft--;
            Invoke("FireWeapon", shootingDelay);
        }
    }
```

Перезарядку зброї було реалізовано наступним чином: перевірка наявності боєприпасів, запуск анімації перезарядки, відтворення відповідного звукового ефекту, а також оновлення кількості патронів після завершення перезарядки.

Код:

```
private void Reload()
{
    SoundManager.Instance.PlayReloadSound(weaponModel);
    animator.SetTrigger("RELOAD");
    isReloading = true;
    Invoke("ReloadCompleted", reloadTime);
}
private void ReloadCompleted()
{
    if (WeaponManager.Instance.CheckAmmoLeftFor(weaponModel) >
magazineSize)
    {
        bulletsLeft = magazineSize;
        WeaponManager.Instance.DecreaseTotalAmmo(bulletsLeft,
weaponModel);
    }
    else
    {
        bulletsLeft =
WeaponManager.Instance.CheckAmmoLeftFor(weaponModel);
        WeaponManager.Instance.DecreaseTotalAmmo(bulletsLeft,
weaponModel);
    }

    isReloading = false;
}
```

Напрямок та розсіювання кулі були реалізовані для роботи логіки двох режимів прицілу, щоб зробити стрільбу більш реалістичною. Метод CalculateDirectionAndSpread() відповідає за розрахунок напрямку, в якому повинна летіти куля, враховуючи розсіювання. Для цього використовувався Raycast, щоб знайти точку, на яку потрапляє постріл, і випадковим чином додавався розкид по осях.

Код:

```
public Vector3 CalculateDirectionAndSpread()
{
    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);
    }
    Vector3 direction = targetPoint - bulletSpawn.position;
    float z = UnityEngine.Random.Range(-spreadIntensity,
spreadIntensity);
    float y = UnityEngine.Random.Range(-spreadIntensity,
spreadIntensity);
    return direction + new Vector3(0, y, z);
}
```

І оскільки куля в Unity була реалізована через GameObject – важливо додати функціонал для її знищення, в іншому випадку велика кількість створених на екрані куль просто зломають гру. Метод DestroyBulletTime() забезпечує автоматичне видалення кулі з гри через певний час після її створення. Для цього використовувався корутин, який чекає задану кількість секунд, а потім знищує її.

Код:

```
private IEnumerator DestroyBulletTime(GameObject bullet, float
delay)
{
    yield return new WaitForSeconds(delay);
    Destroy(bullet);
}return direction + new Vector3(0, y, z);
}
```

У останньому блоці коду було об’єднано кілька важливих функцій. Спочатку реалізовано скидання стану стрільби, яке забезпечує готовність зброї до наступного пострілу після затримки. Далі, додано перевірку наявності патронів і можливість перезарядки при натисканні клавіші R, коли магазин не порожній та немає активного процесу перезарядки. Окрім цього, було впроваджено підтримку різних режимів стрільби (одиничний, черга та автоматичний), що змінює поведінку зброї в залежності від вибраного режиму в її описі.

Код:

```
if (Input.GetKeyDown(KeyCode.R) && bulletsLeft < magazineSize &&
!isReloading &&
WeaponManager.Instance.CheckAmmoLeftFor(weaponModel) > 0)
{
    Reload();
}

if (currentShootingMode == ShootingMode.Auto)
{
    isShooting = Input.GetKey(KeyCode.Mouse0);
}
else if (currentShootingMode == ShootingMode.Single ||
currentShootingMode == ShootingMode.Burst)
{
    isShooting = Input.GetKeyDown(KeyCode.Mouse0);
}

private void ResetShot()
{
    readyToShoot = true;
    allowReset = true;
}
```

Ці доповнення забезпечують гнучку взаємодію з користувачем через різні режими стрільби та можливість перезарядки в залежності від ситуації та по суті є останніми рядками коду в майже головному скрипті проєкту.

Також в коді був використаний патерн «Сінглтон» – структурний шаблон проєктування, який гарантує наявність лише одного екземпляра класу та забезпечує до нього глобальну точку доступу. У вигляді сінглотону було реалізовано клас WeaponManager, що відповідав за керування зброєю гравця, її слотами та кількістю патронів. Застосування цього шаблону дозволило уникнути надмірного дублювання логіки та спростило доступ до загальних ресурсів.

Оглядаючи WeaponManager можна виділити описані в ньому далі методи.

У методі Awake перевіряється, чи є вже існуючий екземпляр класу WeaponManager. Якщо він існує – новий екземпляр знищується, щоб залишився тільки один;

Всередині Update реалізовано функціональність для перемикання між слотами зброї за допомогою клавіш 1 та 2. Активна зброя показується, інші приховуються. Клавіш можна зробити більше.

Методи PickUpWeapon та PickUpAmmo відповідають за взаємодію з об'єктами зброї та патронів у світі. Зброя додається в активний слот, а патрони додаються до відповідної категорії залежно від зброї.

DecreaseTotalAmmo та CheckAmmoLeftFor зменшують кількість патронів після використання зброї та перевіряють кількість наявних патронів для конкретного типу зброї.

Код:

```
public static WeaponManager Instance { get; set; }
private void Awake()
{
    if (Instance != null && Instance != this)
    {
        Destroy(gameObject);
    }
    else
    {
        Instance = this;
    }
}
public void SwitchActiveSlot(int slotNumber)
{
    if (activeWeaponSlot.transform.childCount > 0)
    {
        Weapon currentWeapon =
activeWeaponSlot.transform.GetChild(0).GetComponent<Weapon>();
        currentWeapon.isActiveWeapon = false;
    }

    activeWeaponSlot = weaponSlots[slotNumber];

    if (activeWeaponSlot.transform.childCount > 0)
    {
        Weapon newWeapon =
activeWeaponSlot.transform.GetChild(0).GetComponent<Weapon>();
        newWeapon.isActiveWeapon = true;
    }
}
```

Код:

```
internal void PickUpAmmo(AmmoBox ammo)
{
    switch (ammo.ammoType)
    {
        case AmmoBox.AmmoType.PistolAmmo:
            totalPistolAmmo += ammo.ammoAmount;
            break;
        case AmmoBox.AmmoType.RifleAmmo:
            totalRifleAmmo += ammo.ammoAmount;
            break;
    }
}
```

Крім проведеної роботи над більш звичайними і загальними скриптами, у проєкті був використаний патерн проєктування Singleton. Загалом такий пратерн виявився доволі корисним для роботи з движком та створеними через нього елементами, тому надалі буде описно його використання та логіку роботи на прикладі глобального керівника зброї – WeaponManager. Головним завданням скрипта є забезпечення точки доступу до інформації про поточну зброю, кількість патронів, перемикання між слотами та взаємодію з об'єктами у світі. Застосування сінглтона гарантує, що в грі існує лише один екземпляр цього менеджера, що забезпечує передбачувану логіку роботи зі зброєю [30].

Спочатку у методі Awake скрипта WeaponManager перевіряється, чи вже існує інший екземпляр, і якщо так – поточний об'єкт знищується. Це дозволяє забезпечити суворе дотримання принципу єдиного екземпляра, що є критично важливим для уникнення конфліктів під час керування зброєю.

Частина коду WeaponManager.cs:

```
public static WeaponManager Instance { get; set; }

private void Awake()
{
    if (Instance != null && Instance != this)
    {Destroy(gameObject);}
    else
    {Instance = this;}
}
```

Клас WeaponManager містить список слотів зброї, а також активний слот. У методі Start за замовчуванням активується перший слот зі списку. Це дозволяє відразу працювати з конкретною зброєю на старті гри.

Частина коду WeaponManager.cs:

```
public List<GameObject> weaponSlots;
public GameObject activeWeaponSlot;

private void Start()
{
    activeWeaponSlot = weaponSlots[0];
}
```

У методі Update менеджер постійно перевіряє, який слот є активним, і відображає лише його, деактивуючи інші. Також тут реалізовано перемикання між слотами за допомогою клавіш Alpha1 і Alpha2. Цей механізм забезпечує швидку та зручну зміну зброї під час гри, що позитивно впливає на динаміку ігрового процесу та зменшує когнітивне навантаження на користувача.

Частина коду WeaponManager.cs:

```
public void Update()
{
    foreach (GameObject weaponSlot in weaponSlots)
    {
        weaponSlot.SetActive(weaponSlot == activeWeaponSlot);
    }

    if (Input.GetKeyDown(KeyCode.Alpha1))
    {
        SwitchActiveSlot(0);
    }
    if(Input.GetKeyDown(KeyCode.Alpha2))
    {
        SwitchActiveSlot(1);
    }
}
```

Метод PickUpWeapon викликається при піднятті нової зброї. Він передає отриману зброю у метод, який розміщує її в активному слоті, замінюючи попередню. Це дозволяє автоматизувати процес оновлення екіпірування без додаткових дій з боку гравця.

Частина коду WeaponManager.cs:

```
public void PickUpWeapon(GameObject pickedUpWeapon)
{
    AddWeaponIntoActiveSlot(pickedUpWeapon);
}
private void AddWeaponIntoActiveSlot(GameObject pickedUpWeapon)
{
    DropCurrentWeapon(pickedUpWeapon);


pickedUpWeapon.transform.SetParent(activeWeaponSlot.transform,
false);

    Weapon weapon = pickedUpWeapon.GetComponent<Weapon>();

    pickedUpWeapon.transform.localPosition = new
Vector3(weapon.spawnPosition.x, weapon.spawnPosition.y,
weapon.spawnPosition.z);
    pickedUpWeapon.transform.localRotation =
Quaternion.Euler(weapon.spawnRotation.x, weapon.spawnRotation.y,
weapon.spawnRotation.z);

    weapon.isActiveWeapon = true;
    weapon.animator.enabled = true;
}
```

Додавання зброї в активний слот передбачає спершу скидання поточної зброї, після чого новий об’єкт прикріплюється до слота з урахуванням позиції та обертання. Також активується відповідний аніматор, вказується активність зброї.

Скидання поточної зброї реалізовано через зміну батьківського об'єкта для старого екземпляра, а також відключення анімації та встановлення статусу неактивної зброї.

Частина коду WeaponManager.cs:

```
private void DropCurrentWeapon(GameObject pickedUpWeapon)
{
    if (activeWeaponSlot.transform.childCount > 0)
    {
        var weaponToDrop =
activeWeaponSlot.transform.GetChild(0).gameObject;
        weaponToDrop.GetComponent<Weapon>().isActiveWeapon =
false;
        weaponToDrop.GetComponent<Weapon>().animator.enabled =
false;
weaponToDrop.transform.SetParent(pickedUpWeapon.transform.parent);
        weaponToDrop.transform.localPosition =
pickedUpWeapon.transform.localPosition;
        weaponToDrop.transform.localRotation =
pickedUpWeapon.transform.localRotation;
    }
}
```

Метод `SwitchActiveSlot` дозволяє змінити активний слот. Поточна зброя деактивується, нова – активується. Це дає змогу гравцеві миттєво перемикатися між кількома зразками озброєння, що є особливо важливим у динамічних ситуаціях бою, коли швидка реакція має вирішальне значення.

Частина коду WeaponManager.cs:

```
public void SwitchActiveSlot(int slotNumber)
{
    if (activeWeaponSlot.transform.childCount > 0)
    {Weapon currentWeapon =
activeWeaponSlot.transform.GetChild(0).GetComponent<Weapon>();
        currentWeapon.isActiveWeapon = false;}
    activeWeaponSlot = weaponSlots[slotNumber];
    if (activeWeaponSlot.transform.childCount > 0)
    {Weapon newWeapon =
activeWeaponSlot.transform.GetChild(0).GetComponent<Weapon>();
        newWeapon.isActiveWeapon = true;}
}
```

Завдяки методам PickUpAmmo, DecreaseTotalAmmo та CheckAmmoLeftFor, клас керує загальним запасом патронів для різних типів зброї. Патрони додаються при піднятті коробки, зменшуються при стрільбі або перезарядці, а також можуть бути перевірені наявні залишки відповідно.

Приклад методу PickUpAmmo:

```
internal void PickUpAmmo(AmmoBox ammo)
{
    switch (ammo.ammoType)
    {
        case AmmoBox.AmmoType.PistolAmmo:
            totalPistolAmmo += ammo.ammoAmount; break;
        case AmmoBox.AmmoType.RifleAmmo:
            totalRifleAmmo += ammo.ammoAmount; break;
    }
}
```

Загалом структура реалізованої програмної частини проєкту виглядає подібним чином: скрипти певних об'єктів пов'язані між собою скриптами-Сінглтонами, що так чи інакше взаємодіють із елементами.

Оптимізація проєкту реалізована за допомогою вказаного вище підходу DOTS. Приклад – кулі та їх життєвий цикл. Кулі це річ яка може в великій кількості створюватись в дуже короткий проміжок часу, тому, якщо за три секунди випустити весь магазин автомату в повітря – гравець отримає падіння кадрів. Для оптимізації кулі було перетворено із об'єктів в сутності. Для цього були створені: struct BulletData (для зберігання даних) та клас для системи обробки логіки куль – BulletMovementSystem. Для створення куль були використані EnitityManager. Мінусом використання такого підходу є періодичні конфлікти сутностей та об'єктів, а також із створеною особисто фізикою [37].

Відповідно до пройденої роботи у цьому пункті була представлена реалізація базової системи стрільби в Unity, яка включає в себе: прицілювання, перезарядку та створення куль. Код побудовано так, щоб було зручно керувати усіма параметрами через Unity Editor. Також застосовано патерн Singleton для керування зброєю, що спростило взаємодію між скриптами. Для оптимізації продуктивності певні об'єкти були переведені в сутності за допомогою DOTS. Загалом, система працює стабільно та відповідає цілям гри, забезпечуючи надійну взаємодію між об'єктами та узгоджену логіку поведінки.

3.2 Реалізація графічної частини продукту

Для створення візуальної частини гри, анімацій та загалом об’єктів різного функціоналу першочергово варто було звернутись до Unity Assets Store для отримання необхідних додаткових матеріалів та інструментів. Це дозволило суттєво скоротити час на розробку базових компонентів і зосередитися на унікальних аспектах ігрової механіки. Нижче наведено перелік використаних ресурсів із коротким описом їхнього призначення в проєкті.

– Ammo (ID: 157327): набір 3D-моделей набоїв різних типів, що використовувався для декорування середовища, а також як інвентарні об'єкти.

– StampIt! Collection – Free Examples (ID: 218286): приклади шаблонів рельєфу, які застосовувались для створення варіативного ландшафту.

– Environmental Asset Pack (ID: 170036): набір об’єктів навколишнього середовища (рослинність, будівлі, деталі оточення), що слугував для наповнення ігрового світу.

– FPS Icons Pack (ID: 45240) – набір іконок інтерфейсу користувача (UI), використаних для відображення статусу гравця, індикаторів здоров’я, патронів.

– Free Horror Ambient Music Pack – Desperation (ID: 240918): колекція хорор-музики, яка використовувалася для створення напруженої звукової атмосфери на рівнях.

– Human Character Dummy (ID: 178395): модель персонажа, що тимчасово використовувалася для тестування анімацій в якості заповнювача (placeholder).

– Legacy Particle Pack (ID: 73777): набір ефектів частинок (дим, вогонь, вибухи), застосований для реалізації візуальних ефектів пострілів, пошкоджень.

– Low Poly Weapons Vol. 1 (ID: 151980): колекція низькополігональної зброї, яка використовувалася як моделі основної зброї в руках гравця та ворогів.

– Monster Mutant 7 (ID: 188552) – модель мутанта, що слугувала ворогом.

– Zombie Mutant Free (ID: 310842) – модель зомбі-мутанта, яка виступала в ролі супротивника у хорор-частинах гри.

Після додавання асетів першочергово було спроєктовано карту локації через об’єкт Terrain та безкоштовну колекцію StampIt! вказану вище. Terrain – це спеціальний тип об’єкта, призначений для створення великих, деталізованих ландшафтів. У межах проєкту він використовувався для формування основи ігрового середовища.

Геометрія рельєфу створювалась за допомогою кистей та шаблонів з безкоштовної колекції StampIt! Collection – Free Examples, що дозволило швидко створити горбисту місцевість, схили та інші природні форми. Створений рельєф локації представлено на рисунку 3.1.

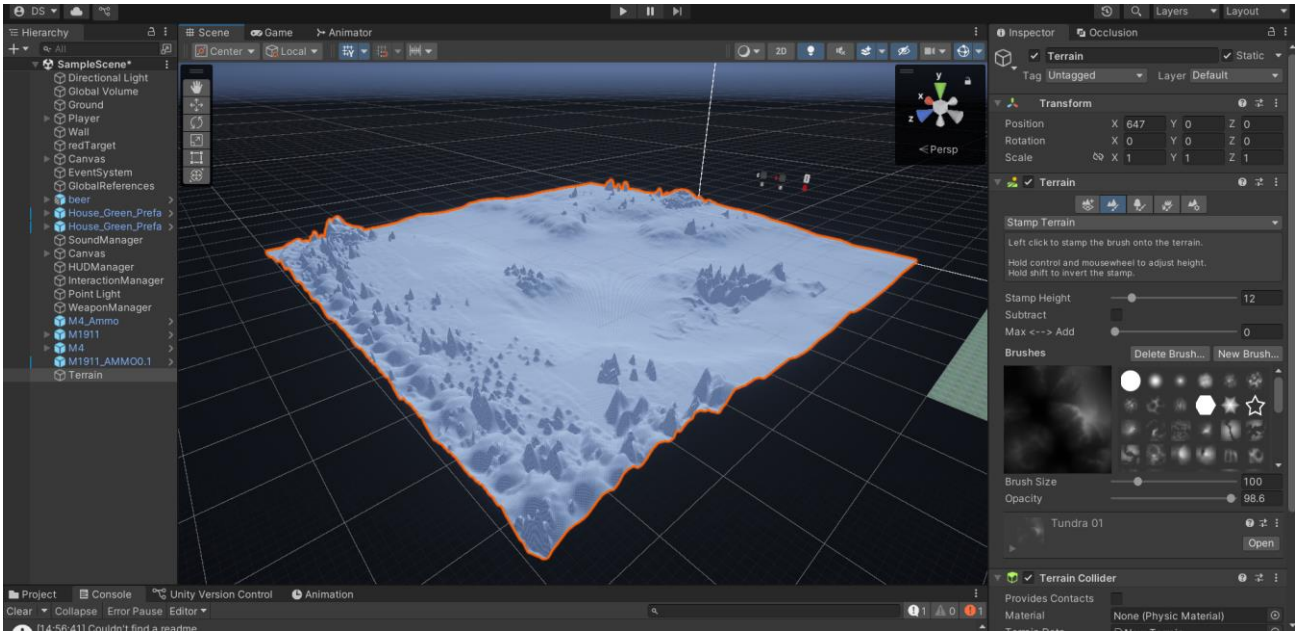


Рисунок 3.1 – Рельєф локації

Текстурування виконувалось за допомогою декількох шарів ґрунтових і кам’яних текстур, що імітують реалістичну поверхню землі.

Додаткове наповнення включало інтеграцію дерев, кущів і трави, що імпортувалися з інших ассетів, зокрема Environmental Asset Pack, для створення природного середовища.

Фарбування рельєфу здійснювалось за допомогою інструменту Procedural Terrain Painter, який дозволяє автоматично накладати текстури відповідно до параметрів поверхні, таких як висота, крутизна та інші. Це значно спростило процес створення реалістичного покриття без потреби у ручному малюванні.

Перша локація гри, що і є фінальним, деталізованим рельєфом, зображена на рисунку 3.2.

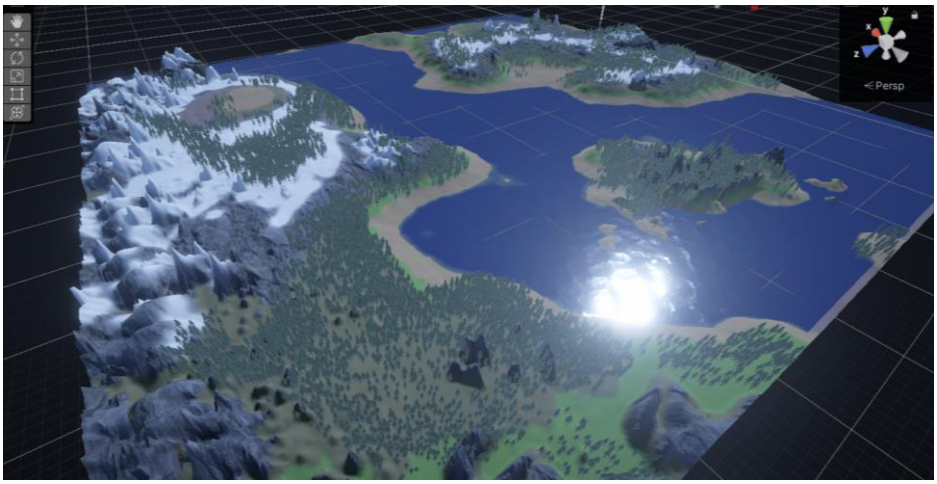


Рисунок 3.2 – Пророблена локація

Зі сторони графіки та анімацій для механік гри, наприклад – механіки стрільби, описаної в пункті 3.1 Реалізація програмної частини продукту, також було проведено певний обсяг робіт по роботі з моделями, їх положенню в просторі та анімаціями, як: віддача від стрільби в звичайному положенні та при прицілюванні, саме прицілювання, перезарядка.

Для створення анімацій був використаний функціонал движка, а саме Animator. Animator – це компонент двигуна, що керує анімаціями певного обраного об’єкта за допомогою Animator Controller. Цей компонент дозволяє визначати стани (вони ж анімації), параметри (певні змінні) та переходи між ними на основі логіки гри або взаємодії користувача [31].

Загалом робота з анімаціями для гравця виглядала наступним чином:

– спершу перейти на об’єкт гравця (Player) у ієрархії;

– потім обрати розміщений об’єкт в ієрархії, з яким буде взаємодіяти гравець, наприклад – пістолет;

– після чого перейти в розділ «Animation» обраного пістолету та створити анімації з певними назвами;

– в кінці, для кожної анімації: обрати стан «Preview» та почати запис, під час якого змінювати позицію пістолету по потребі.

Робота з анімацією тримання зброї, а також вікно роботи з анімаціями зображено на рисунку 3.3.

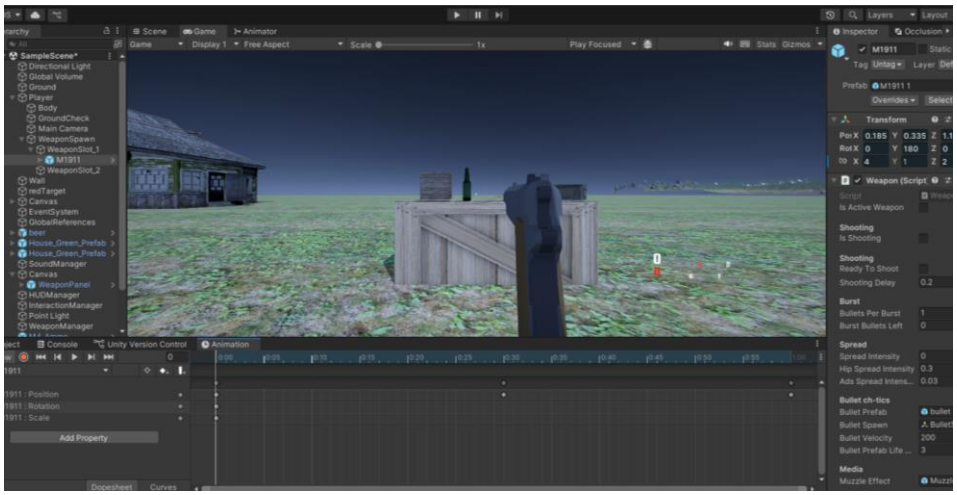


Рисунок 3.3 – Робота з анімацією Idle

Після проведеної над анімаціями роботи було ознайомлено із функціоналом вікна Animator та створено, по необхідності, певні тригери та булеві параметри для роботи коду з анімаціями, наприклад:

- RECOIL: тригер для анімації віддачі при звичайні стрільбі;
- RELOAD: тригер для анімації перезарядки;
- isADS: булевий параметр для стану прицілювання;
- RECOIL_ADS: тригер для анімації віддачі при прицільній стрільбі.

Нижче приведено приклад використання різних параметрів Animator.

Bool (переключає стан прицілювання):

```
animator.SetBool("isADS", true);
```

Trigger (активує анімацію віддачі):

```
animator.SetTrigger("RECOIL");
```

.

Ці параметри потім використовуються у Transitions (переходах) між станами, як умови для перемикання анімацій. Робота із анімаціїями, а також загалом з’єднанні анімації в відповідному вікні, зображені на рисунку 3.4.

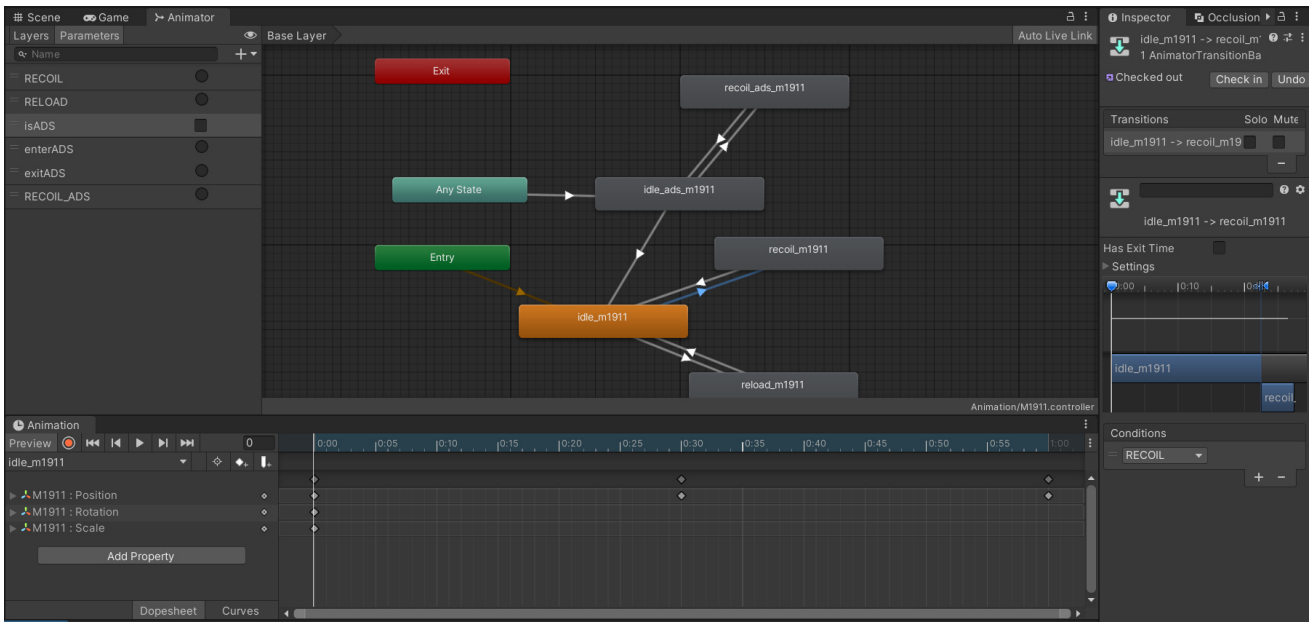


Рисунок 3.4 – Взаємодія з анімаціями в переходах

У Unity Animator параметри використовуються для керування переходами між анімаційними станами. Animator Controller дозволяє створювати логіку переходів за допомогою параметрів типу Float, Int, Bool і Trigger згаданих вище.

Стани відповідають окремим анімаціям. Переходи між ними створюються вручну через вікно Animator шляхом вибору стану → Make Transition → вибір цільового стану. Після створення переходу в Inspector можна задати параметри переходу, такі як:

- Has Exit Time: чи є потреба очікування завершення анімації;
- Transition Duration: про тривалість переходу;
- Conditions: умови, за яких здійснюється перехід.

Загалом умови переходу базуються на параметрах:

- Float і Int: використовуються для числових умов (>, <, ==, !=);
- Bool: для логічних умов (true, false);
- Trigger: для одноразових подій, скидається після спрацьовування.

Загалом по закінченню роботи над графічною частиною гри було отримано візуально оформлене ігрове середовище, наповнене різноманітними об'єктами та анімаціями ключових механік, що забезпечує базове візуальне сприйняття ігрового процесу. У результаті створено базове візуальне середовище гри з ключовими об'єктами та анімаціями.

3.3 Вимоги до технічних та програмних засобів

У процесі розробки програмного продукту особливу увагу слід приділяти врахуванню як технічних, так і програмних обмежень, адже саме вони можуть істотно вплинути на загальну працездатність, стабільність і зручність використання застосунку. У даному випадку дипломний проєкт було реалізовано з використанням сучасного ігрового рушія Unity, який забезпечує достатню гнучкість, стабільність і зручність при створенні тривимірних ігор.

Гра орієнтована на запуск на персональних комп'ютерах, що обумовило відповідні технічні рішення. Оскільки проєкт є тривимірним шутером із невисокими системними вимогами, загальні апаратні та програмні потреби залишаються на прийнятному рівні для більшості сучасних користувачів. Нижче представлено таблицю, в якій зазначено мінімальні та рекомендовані конфігурації системи, необхідні для забезпечення коректної, безперебійної та комфортної роботи гри.

Таблиця 3 – Вимоги

Характеристика	Мінімальні	Рекомендовані
Операційна система	Windows 7 (64-bit) або macOS 10.13	Windows 10 (64-bit) або macOS 10.15+
Процесор	Intel Core i3-4130 або AMD FX-6300	Intel Core i5-6400 або AMD Ryzen 5 1600
Оперативна пам'ять	4 GB	8 GB
Відеокарта	NVIDIA GeForce GTX 750 Ti або AMD R7 360	NVIDIA GeForce GTX 1060 або AMD Radeon RX 580
Місце на диску	5 GB вільного простору	5 GB вільного простору
Звукова карта	Сумісна з DirectX 10	Сумісна з DirectX 11

3.4 Тестування програмного забезпечення

Важливим етапом в розробці ПЗ є його тестування. Відповідно у процесі створення програмного забезпечення для даного проєкту, а також після завершення всіх етапів розробки, було проводено ретельне тестування, яке включало в себе кілька важливих видів: дослідницьке та unit.

Дослідницьке тестування – це підхід до перевірки ігрового програмного забезпечення, який поєднує етапи планування, дослідження та виконання тестів. Такий підхід забезпечує максимальну свободу дій для тестувальника, дозволяючи йому творчо вивчати поведінку ігрових механік у реалістичних умовах, не обмежуючись наперед визначеними тест-кейсами [32].

Unit-тестування – це тип автоматизованого тестування, який зосереджується на перевірці окремих одиниць програмної логіки в ізоляції від решти системи. У межах Unity-проєктів такими одиницями зазвичай виступають окремі компоненти, скрипти або методи класів, відповідальні за реалізацію конкретної функціональності гри [33].

Для реалізації unit-тестів у даному проєкті було використано фреймворк NUnit, який повністю підтримується Unity та дозволяє створювати прості, зрозумілі й повторювані перевірки. За допомогою цього інструменту виконувалася автоматична верифікація логіки скриптів – зокрема, їх правильна ініціалізація, коректне оновлення станів, обробка взаємодій та виконання внутрішніх обчислень [36].

Unit-тести запускалися в ізольованому середовищі, що дозволяло миттєво виявляти помилки на рівні логіки, не вдаючись до повного запуску сцени чи вручну відтворюваних сценаріїв. Це значно прискорювало цикл розробки та допомагало уникати регресій при подальших змінах у коді [39].

Першочергово, під час розробки продукту було застосовано дослідницьке тестування та unit-тестування. Перший вид тестування дозволив глибоко вивчити функціональність системи та виявити потенційні проблеми, різного роду баги,

тощо. В свою чергу – юніт-тестування проводилось для перевірки скриптів на працездатність та забезпечення їхньої коректної роботи.

Щоб надати приклад успішного виконання дослідницького тестування варто звернутись до попереднього пункту, а саме до розробки анімацій в ньому та їх інтеграцію в гру загалом – у скрипти, де анімації викликаються. Після інтегрування анімацій в код було знайдено баг, через якого анімація віддачі не спрацьовувала під час прицільної стрільби. Прицільна стрільба відбувалась наступним чином:

– «Гравець» натискає праву кнопку миші;
– «Пістолет» міняє позицію ближче до гравця;
– «Гравець» натискає ліву кнопку миші;
– «Пістолет» стріляє;
– «Гравець» відпускає праву кнопку миші;
– «Пістолет» запускає анімацію віддачі так, ніби ми тільки що стріляли.

Для виправлення цього багу була додана змінна isADS, щоб точно розуміти, коли гравець прицілюється, а коли ні та, відповідно, програвати для цього різні анімації зброї.

Приклад коду:

```
if (isADS)
    {animator.SetTrigger("RECOIL_ADS");}
else
    {animator.SetTrigger("RECOIL");}
```

Для прикладу проведення unit-тестування взято уже згаданий функціонал зброї, а саме скрипт Weapon.cs та допоміжні для тестування скрипти, без яких функціонал працювати не буде: WeaponManager.cs, SoundManager.cs.

Підготовка до виконання кожного тесту здійснювалася в методі SetUp, який автоматично викликався перед запуском кожного окремого тесту. У ньому створювався об’єкт зброї, додавалися необхідні компоненти, а також ініціалізувалися значення, потрібні для роботи функцій класу Weapon.

Першочергово було проведено перевірку початкової кількості патронів у магазині. Для цього протестовано, чи метод Awake() правильно ініціалізує кількість патронів у магазині, відповідно до заданого значення magazineSize. Очікувалося, що після виклику методу Awake() кількість патронів у змінній bulletsLeft дорівнюватиме значенню magazineSize. Це дозволяло впевнитися в правильності логіки початкової ініціалізації зброї.

Код тесту:

```
[Test]
public void Weapon_HasInitialMagazineBullets_WhenAwake()
{
    weapon.magazineSize = 10;
    weapon.Awake();
    Assert.AreEqual(10, weapon.bulletsLeft);
}
```

Другим тестом була перевірка повернення вектора напрямку при розрахунку розсіювання. Цей тест підтверджував, що метод CalculateDirectionAndSpread() коректно обчислює напрямок пострілу та повертає значення типу Vector3. Використання Assert.IsInstanceOf<Vector3> гарантувало, що функція не повертала null або некоректний тип, що особливо важливо для подальшого руху куль або ефектів у грі.

Код тесту:

```
[Test]
public void CalculateDirectionAndSpread_ReturnsVector3()
{
    Vector3 direction = weapon.CalculateDirectionAndSpread();
    Assert.IsInstanceOf<Vector3>(direction);
}
```

Тестування зменшення кількості патронів після пострілу було проведено для того, щоб перевірити на скільки зменшується кількість патронів у магазині після виклику методу FireWeapon().

До початку тесту кількість патронів дорівнювала 5. Після одного пострілу значення змінної bulletsLeft мало зменшитися на одиницю. Тест також демонстрував важливість повної ініціалізації об’єкта зброї перед викликом методу стрільби, зокрема – наявність об'єкта камери, префабу кулі, ефектів та звуку.

Код тесту:

```
[Test]
public void FireWeapon_DecreasesBulletsLeft()
{
    weapon.bulletPrefab = GameObject.CreatePrimitive(PrimitiveType.Cube);
    weapon.bulletPrefab.AddComponent<Rigidbody>();
    var spawn = new GameObject("BulletSpawn");
    weapon.bulletSpawn = spawn.transform;
    GameObject soundManagerGO = new GameObject("SoundManager");
    var soundManager = soundManagerGO.AddComponent<TestSoundManager>();
    SoundManager.Instance = soundManager;
    weapon.readyToShoot = true;
    weapon.currentShootingMode = ShootingMode.Single;
    var cameraGO = new GameObject("Camera");
    cameraGO.tag = "MainCamera";
    cameraGO.AddComponent<Camera>();
    weapon.muzzleEffect = new GameObject();
    weapon.muzzleEffect.AddComponent<ParticleSystem>();
    weapon.FireWeapon();
    Assert.AreEqual(4, weapon.bulletsLeft);
}
```

Перевірка правильності завершення перезаряджання: тест дозволяв переконатися, що метод ReloadCompleted() правильно поповнює магазин до максимальної кількості та зменшує відповідну кількість патронів у загальному запасі боєприпасів.

Загалом перевірено чи при наявності достатньої кількості патронів у загальному запасі (totalPistolAmmo), метод ReloadCompleted() додаватиме відсутні 5 патронів у магазин (bulletsLeft змінюється з 1 до 6), і відповідно зменшуватиме запас у WeaponManager.

Код тесту:

```
[Test]
public void ReloadCompleted_FillsMagazineCorrectly()
{
    weapon.weaponModel = WeaponModel.pistol1911;
    weapon.magazineSize = 6;
    weapon.bulletsLeft = 1;
    var wmGO = new GameObject("WeaponManager");
    var wm = wmGO.AddComponent<WeaponManager>();
    wm.totalPistolAmmo = 20;
    wm.weaponSlots = new System.Collections.Generic.List<GameObject>
          { weaponGO };
    wm.activeWeaponSlot = weaponGO;
    WeaponManager.Instance = wm;
    weapon.ReloadCompleted();
    Assert.AreEqual(6, weapon.bulletsLeft);
    Assert.AreEqual(14, wm.totalPistolAmmo);
}
```

У останньому тесті проводилась перевірка впливу розсіювання на напрямок пострілу, тобто, чи метод CalculateDirectionAndSpread() генерує вектор, відмінний від нульового, коли встановлено розсіювання. Тест демонстрував, що за ненульового значення spreadIntensity вектор пострілу змінювався відповідно до доданого шуму або відхилення.

Приведене юніт-тестування дозволило перевірити роботу гри, в тому числі і її основний функціонал – стрільбу. Такий підхід гарантував, що дії гравця у грі будуть працювати стабільно та передбачувано, в грі буде мінімум помилок.

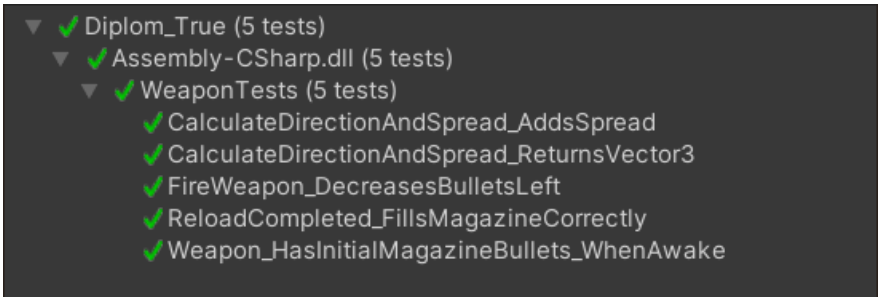


Рисунок 3.5 – Результати тестування модулю зброї

3.5 Висновки. Програмна реалізація та тестування продукту

Основна увага пункту була приділена реалізації стрільби як центрального елементу ігрового процесу. Втім, у рамках проєкту також впроваджено низку інших важливих функціональних складових, що забезпечують ігрову цілісність. У процесі розробки було створено повноцінну ігрову сцену, яка включає основні механіки, необхідні для базового функціонування шутера. Серед них реалізовано: стрільбу, пересування гравця, взаємодію з об'єктами середовища та ворогами.

Графічна частина адаптована до вибраного стилю: додано візуальні ефекти, накладено фільтри для симуляції VHS-естетики, використано стилізовані текстури й освітлення. Завершальним етапом стало тестування всіх реалізованих елементів. Відповідно до більшості життєвих циклів проектів було перевірено коректність виконання базових дій, узгодженість між програмною та візуальною частинами, а також стабільність роботи сцени в цілому.

ВИСНОВКИ

У межах кваліфікаційної роботи було реалізовано повноцінний проєкт із розробки гри-шутера з елементами виживання та хорору на базі двигуна Unity. Робота охоплювала всі етапи життєвого циклу програмного продукту: від постановки задачі, збору вимог і аналізу предметної області до проєктування архітектури, реалізації функціоналу, проведення тестування та формування рекомендацій для подальшого розвитку проєкту.

На етапі аналізу предметної області було вивчено особливості жанру шутера, його еволюцію та поєднання з іншими жанрами, зокрема хорором і виживанням. Дослідження показало, що поєднання цих жанрів дозволяє створити глибше занурення у гру, надає більше простору для експериментів з атмосферою та викликами для гравця. Крім того, було враховано сучасні тренди в ігровій індустрії, такі як зростання популярності ретро-візуальної стилістики VHS-хорору та аналогового жаху, які були успішно імплементовані в проєкт.

Було обґрунтовано вибір рушія Unity як оптимального середовища для реалізації проєкту завдяки його широкій функціональності, підтримці мови C#, активній спільноті та наявності великої кількості готових рішень, зокрема графічних і звукових ресурсів. Unity дозволив забезпечити мультиплатформену підтримку, масштабованість, а головне – швидкість розробки.

Окріп слідування ООП, архітектура гри місцями була побудована з урахуванням принципів Data-Oriented Design, що дозволило досягти високої продуктивності при роботі з великою кількістю об’єктів. Використання Scriptable Objects сприяло гнучкості налаштувань і зручності повторного використання коду. Реалізовано модульну структуру, яка дозволяє надалі розширювати проєкт без суттєвих змін у коді [38].

Програмна реалізація охопила розробку основних компонентів гри: руху гравця, системи стрільби, механіки підбору предметів, логіки супротивників, реалізації візуальних ефектів, звукового супроводу та взаємодії з середовищем.

Окрему увагу було приділено ігровій атмосфері: змінному освітленню, аудіоефектам, та ретро-естетиці. У грі реалізовано базовий штучний інтелект ворогів, що включає в себе реакцію на шум, переслідування та тактичні атаки.

Тестування проводилося на декількох рівнях: юніт-тестування для окремих модулів, дослідницьке тестування для перевірки ігрових сценаріїв та загальне тестування продуктивності. За результатами тестування було виявлено та усунено низку недоліків, зокрема пов'язаних із продуктивністю, коректністю обробки колізій та поведінкою ворогів у граничних ситуаціях.

Проєкт відповідає поставленим на початку завданням:

- проведено аналіз сучасного стану індустрії та вибраного жанру;
- визначено функціональні та нефункціональні вимоги до гри;
- розроблено технічну архітектуру з використанням сучасних патернів;
- реалізовано основні механіки гри;
- реалізована графічна сторона гри відповідно до сеттингу;
- проведено тестування і вдосконалення проєкту згідно результатів.

Підсумовуючи викладене, можна зробити висновок, що реалізований ігровий продукт є конкурентоспроможним у межах аматорських та інді-проєктів, демонструючи достатній рівень технічної та функціональної реалізації. Створене програмне рішення характеризується наявністю базової, проте стабільної геймплейної основи, що забезпечує цілісність користувацького досвіду й створює потенціал для подальшого масштабування. Зокрема, розробка може слугувати фундаментом для інтеграції нових рівнів, вдосконалення сюжетної лінії, розширення системи супротивників та загального поглиблення ігрової механіки.

Зазначена робота стала наочним прикладом комплексного, поетапного підходу до розробки ігрового програмного забезпечення, що, у свою чергу, підтверджує як актуальність обраної теми, так і потребу в подальших дослідженнях у цьому напрямку. Крім того, отримано ґрунтовне уявлення про повний цикл життєвого шляху програмного продукту — від початкової концепції до завершального етапу налагодження та оцінювання працездатності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Gamers by age: Gamer age distribution. Headphones Addict. 28.03.2024. Gaming. URL: https://headphonesaddict.com/gamer-demographics-statistics/ (дата звернення 01.01.2025).
2. Top 10 Gaming Companies. Surfe. Industries – Gaming. URL: https://www.surfe.com/industries/gaming/ (дата звернення 03.01.2025).
3. How Games Typically Get Built. The Pragmatic Engineer. 22.08.2023. URL: https://newsletter.pragmaticengineer.com/p/how-games-typically-get-built (дата звернення 05.01.2025).
4. Electronic shooter game | History & Examples. Britannica. Games, Hobbies & Recreational Activities. URL: https://www.britannica.com/topic/electronic-shooter-game (дата звернення 10.01.2025).
5. What are Horror Games? Acer corner. Gaming. 15.01.2023 URL: https://blog.acer.com/en/discussion/192/what-are-horror-games (дата звернення 19.01.2025).
6. Survival Video Game. European Studios. URL: https://www.europeanstudios.com/ encyclopedia/survival-video-game-genre/ (дата звернення 25.01.2025).
7. 9 кіл сетингу: як створити привабливий ігровий простір. VOKi games. Блог. URL: https://vokigames.com/ua/9-kil-setingu-yak-stvoriti-privablivij-igrovij-prostir/ (дата звернення 28.01.2025).
8. What is Analog Horror – History and Examples Explained. studiobinder. 15.03.2025. URL: https://www.studiobinder.com/blog/what-is-analog-horror-definition/ (дата звернення 06.02.2025).
9. Doom. DoomWiki. URL: https://doomwiki.org/wiki/Doom (дата звернення 09.02.2025).
10. F.E.A.R. Steam. URL: https://store.steampowered.com/app/21090/FEAR/ (дата звернення 14.02.2025).

11. Cry of Fear. Cry of Fear Wiki. URL: https://cry-of-fear.fandom.com/wiki/Cry_of_Fear (дата звернення 15.02.2025).

12. Functional vs. Non Functional Requirements. GeeksForGeeks. 08.01.2025. URL: https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/ (дата звернення 17.02.2025).

13. Best Unity Game Development Resources. ConstructG. URL: https://constructg.com/game-development-resources/ (дата звернення 21.02.2025).

14. Entity Component System. Unity. Manual. URL: https://docs.unity3d.com/Packages/ com.unity.entities@0.17/manual/index.html (дата звернення 24.02.2025).

15. Object-Oriented Programming (OOP) in Unity Game Development. Unity. Manual. URL: https://swapnilmore03.medium.com/object-oriented-programming-oop-in-unity-game-development-452486d1e4c5 (дата звернення 25.02.2025).

16. DOTS - Unity's Data-Oriented Technology Stack. Unity. URL: https://unity.com/dots (дата звернення 28.02.2025).

17. ScriptableObject. Unity. Manual. URL: https://docs.unity3d.com/Manual/class-ScriptableObject.html (дата звернення 01.03.2025).

18. IDEF – Integrated DEFinition Methods (IDEF). URL: https://www.idef.com (дата звернення 02.03.2025).

19. Як будувати UML-діаграми. Розбираємо три найпопулярніші варіанти. DOU. Блоги. 27.10.2022. URL: https://dou.ua/forums/topic/40575/ (дата звернення 03.03.2025).

20. IDEF – Integrated DEFinition Methods (IDEF). URL: https://www.idef.com (дата звернення 04.03.2025).

21. C#: що це за мова та де її використовують. r_d media. URL: https://robotdreams.cc/uk/blog/284-s-chto-eto-za-yazyk-i-gde-ego-ispolzuyut (дата звернення 06.03.2025).

22. C++ Tutorial. w3schools. URL: https://www.w3schools.com/cpp/ (дата звернення 08.03.2025).

23. Unity Real-Time Development Platform. Unity. URL: https://unity.com/ (дата звернення 09.03.2025).

24. MonoGame. URL: https://monogame.net/ (дата звернення 11.03.2025).

25. Godot Engine. URL: https://godotengine.org/ (дата звернення 12.03.2025).

26. Unity Version Control. Unity. URL: https://unity.com/solutions/version-control (дата звернення 14.03.2025).

27. GitHub. URL: https://github.com/ (дата звернення 16.03.2025).

28. Unity Asset Store: The Best Assets for Game Making. Unity. URL: https://assetstore.unity.com/?srsltid=AfmBOopokRg3sn6vxxC0u4CN0png9x84P6DGAkQen7L4DLQ7N0ZYOZDa (дата звернення 18.03.2025).

29. About Unity Test Framework. Unity. Manual. URL: https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/ index.html (дата звернення 20.03.2025).

30. Singleton in C#. Refactoring.Guru. Design Patterns. 10.03.2023. URL: https://refactoring.guru/design-patterns/singleton/csharp/example (дата звернення 22.03.2025).

31. Animator Controller. Unity. Mecanim Animation system URL: https://docs.unity3d.com/Manual//class-AnimatorController.html (дата звернення 24.03.2025).

32. Що таке Exploratory Testing? QualityAssuranceGroup. Publications. URL: https://qagroup.com.ua/publications/ what-is-exploratory-testing/ (дата звернення 28.03.2025).

33. Automated testing. Unity. Environment and tools. URL: https://docs.unity3d.com/Manual/testing-editortestsrunner.html (дата звернення 01.04.2025).

34. Data-oriented design. Solita. 04.08.2023. URL: https://www.solita.fi/blogs/data-oriented-design/ (дата звернення 03.04.2025)

35. ScriptableObject. Unity. Scripting API. URL: https://docs.unity3d.com/ScriptReference/ScriptableObject.html (дата звернення 07.04.2025).

36. Testing and quality assurance tips for Unity projects. Unity. URL: https://unity.com/how-to/testing-and-quality-assurance-tips-unity-projects (дата звернення 11.04.2025).

37. Optimizing Game Performance. Hackernoon. 23.06.2023. URL: https://hackernoon.com/optimizing-performance-and-boosting-fps-in-unity-games (дата звернення 15.04.2025).

38. Scriptable Objects in Unity. gamedevbeginner. Advanced Scripting. 27.06.2024. URL: https://gamedevbeginner.com/scriptable-objects-in-unity/ (дата звернення 26.04.2025).

39. Automated Game Testing. arxiv. 29.03.2021. URL: https://arxiv.org/abs/2103.15819 (дата звернення 30.04.2025).

40. Analog Horror. TVtropes. URL: https://tvtropes.org/pmwiki/pmwiki.php/Main/AnalogHorror (дата звернення 14.02.2025).

41. Кваліфікаційна робота : Методичні настанови для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 121 «Інженерія програмного забезпечення» / Л. П. Бедратюк, Г. І. Радельчук. Хмельницький : ХНУ, 2023. 60 с.

42. Borromeo N. A. Hands-On Unity 2022 Game Development: Learn to use the latest Unity 2022 features to create your first video game in the simplest way possible / Packt Publishing. 3-тє вид. Б. м. : Packt Publishing, 2022. 712 с. ISBN 1803236914.

43. Форкун Ю., Мартинюк В., Яшина О. Метод розробки та проектування архітектурної складової програмного застосунку. MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES. 2023. вип. 4. С. 93.

44. Архітектурні методи оптимізації швидкодії та відмовостійкості програмних застосунків / Ю. Форкун та ін. MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES. 2023. вип. 2. С. 196–201.

45. Порівняння програмних метрик для оцінки якості програмних продуктів / О. М. Яшина та ін. Вісник Хмельницького національного університету. 2021. вип. 5. С. 4.

Додаток А
(Презентація)

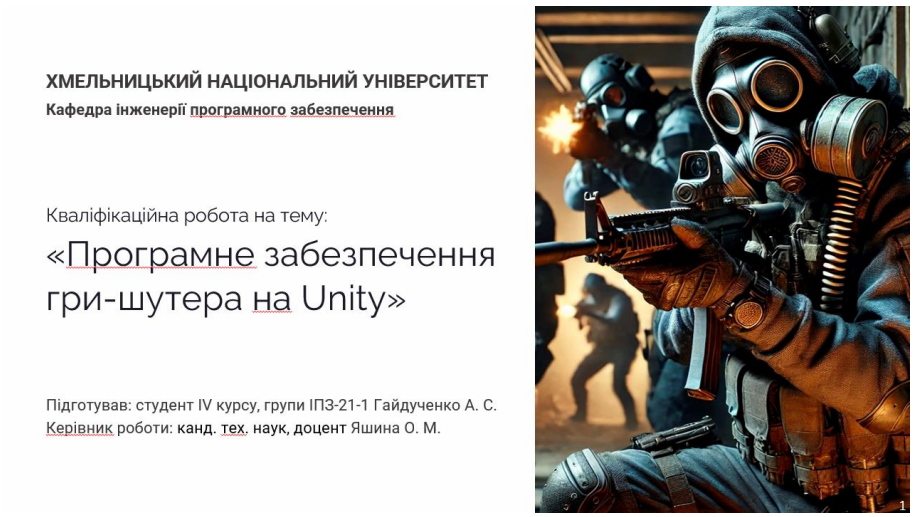


Рисунок А.1

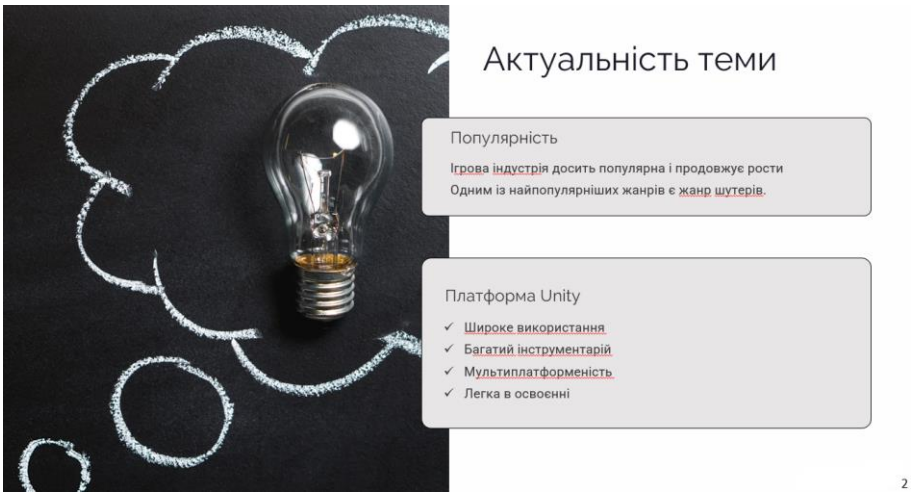


Рисунок А.2

Рисунок А.3

Рисунок А.4

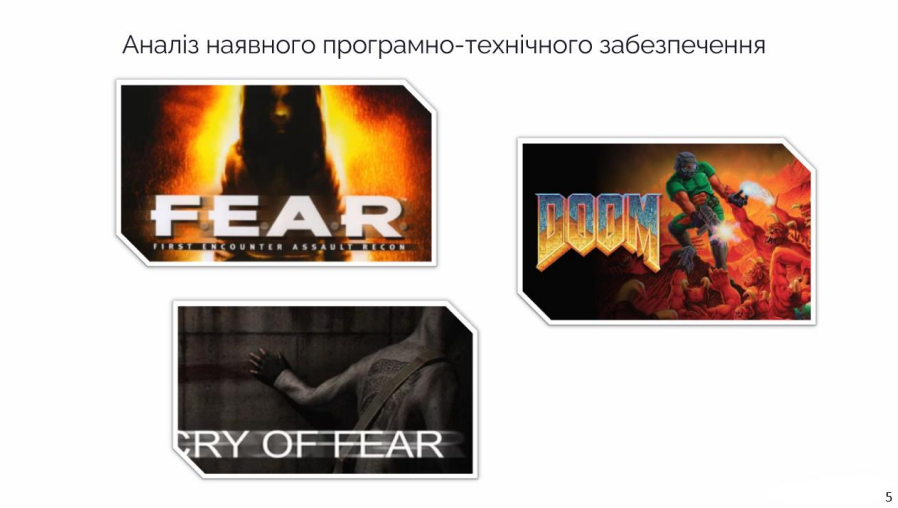


Рисунок А.5

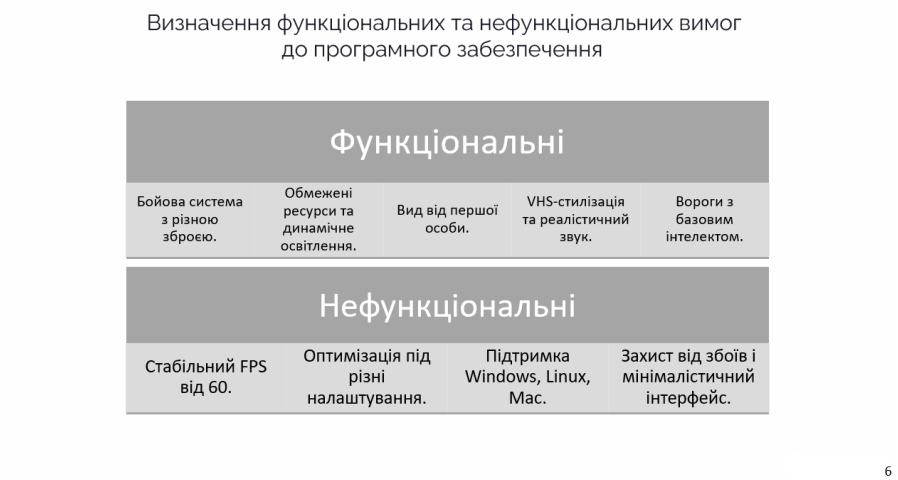


Рисунок А.6

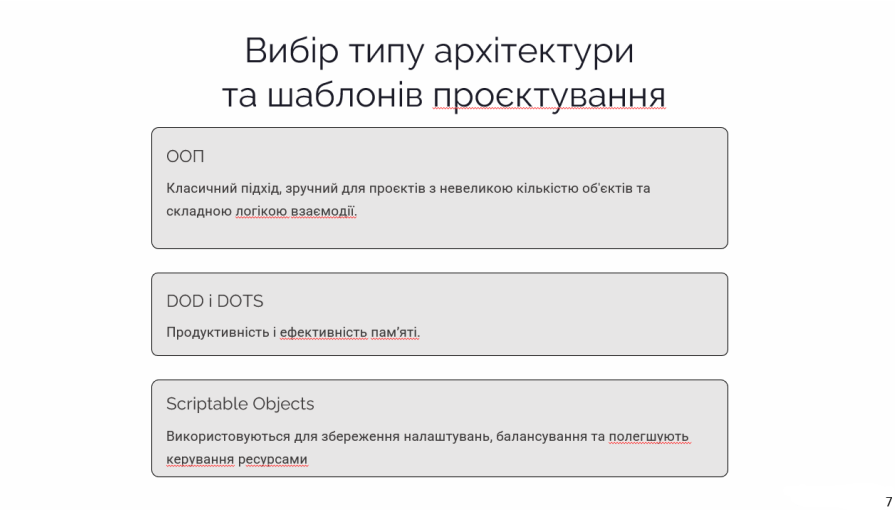


Рисунок А.7

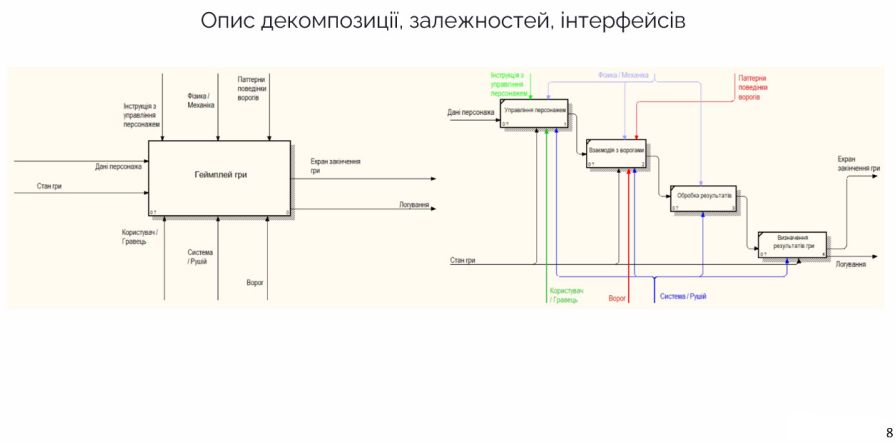


Рисунок А.8

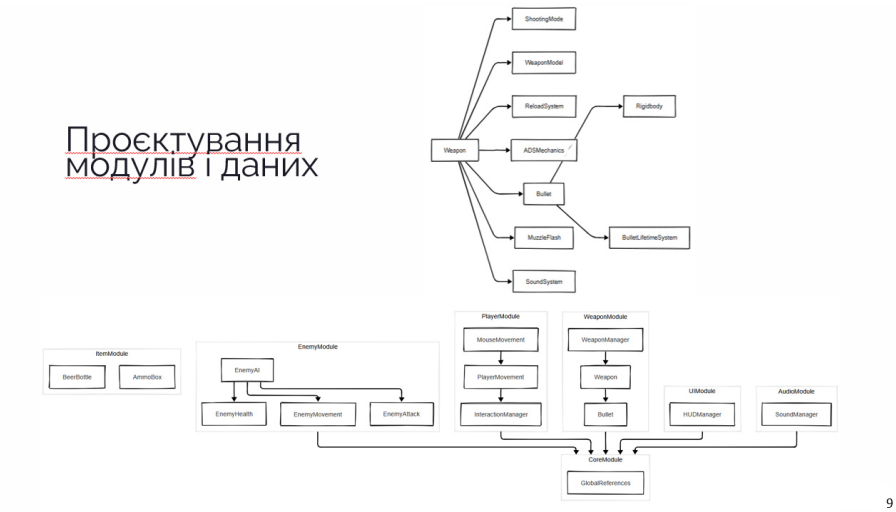


Рисунок А.9

10

Рисунок А.10

11

Рисунок А.11

Вимоги до технічного та програмного забезпечення

Операційна система	Windows 10 (64-bit) або macOS 10.15+
Процесор	Intel Core i5-6400 або AMD Ryzen 5 1600
Оперативна пам'ять (RAM)	8 GB
Відеокарта	NVIDIA GeForce GTX 1060 або AMD Radeon RX 580
Місце на диску	2 GB вільного простору
Звукова карта	Сумісна з DirectX 11

12

Рисунок А.12

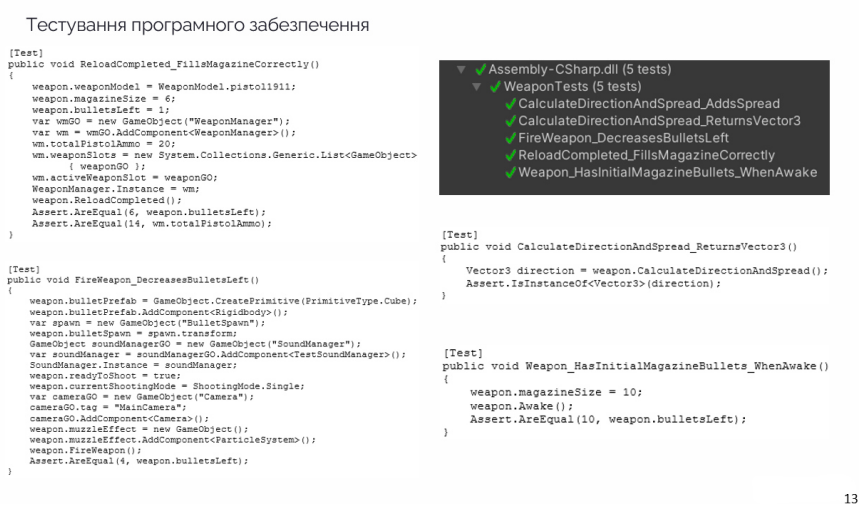


Рисунок А.13

Висновки

У межах кваліфікаційної роботи було розроблено прототип гри-шутера від першої особи. Було реалізовано базову систему управління персонажем, стрільбу, інтерфейс гравця, ефекти оточення та ворогів з базовою логікою поведінки.

Проєкт відповідає функціональним і нефункціональним вимогам, а також демонструє потенціал використання Unity для розробки повноцінних ігор FPS.

Проєкт має широкі перспективи розвитку. Можливе розширення ігрового світу новими рівнями та сюжетною лінією, удосконалення інтелекту ворогів з патрулюванням і реакцією на звук та світло, а також додавання мережевого режиму (кооператив або PvP). Нові типи зброї й ворогів забезпечать глибший геймплей. Завершальним етапом може стати публікація гри на платформах, як-от itch.io чи Steam.

14

Рисунок А.14

ДЯКУЮ ЗА УВАГУ

15

Рисунок А.15

Додаток Б
(Програмний код)

Bullet.cs

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

public class Bullet : MonoBehaviour
{
    private void OnCollisionEnter(Collision objectWeHit)
    {
        if (objectWeHit.gameObject.CompareTag("Target"))
        {
            print("hit " + objectWeHit.gameObject.name + " !");

            CreateBulletImpactEffect(objectWeHit);

            Destroy(gameObject);
        }

        if (objectWeHit.gameObject.CompareTag("Wall"))
        {
            print("hit a wall !");

            CreateBulletImpactEffect(objectWeHit);

            Destroy(gameObject);
        }

        if (objectWeHit.gameObject.CompareTag("Bottle"))
        {
            print("hit a Bottle !");

            objectWeHit.gameObject.GetComponent<BeerBottle>().Shatter();
        }
    }

    void CreateBulletImpactEffect (Collision objectWeHit)
    {
        ContactPoint contact = objectWeHit.contacts[0];

        GameObject hole = Instantiate(
            GlobalReferences.Instance.bulletImpactEffectPrefab,
            contact.point,
            Quaternion.LookRotation(contact.normal)

            );

        hole.transform.SetParent(objectWeHit.gameObject.transform);
    }
}
```

MouseMovement.cs

```
using UnityEngine;

public class MouseMovement : MonoBehaviour
{
```

```
    public float mouseSensitivity = 100f;

    float xRotation = 0f;
    float yRotation = 0f;

    public float topClamp = -90f;
    public float bottomClamp = 90f;

    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;
    }

    void Update()
    {
        float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity *
Time.deltaTime;
        float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity *
Time.deltaTime;

        xRotation -= mouseY;

        xRotation = Mathf.Clamp(xRotation, topClamp, bottomClamp);

        yRotation += mouseX;
        transform.localRotation = Quaternion.Euler(xRotation, yRotation, 0f);
    }
}
```

PlayerMovement.cs

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

public class PlayerMovement : MonoBehaviour
{
    private CharacterController controller;
    public float speed = 12f;
    public float gravity = -10f * 2;
    public float jumpHeight = 3f;
    public Transform groundCheck;
    public float groundDistance = 0.4f;
    public LayerMask groundMask;
    Vector3 velocity;
    bool isGrounded;
    bool isMoving;
    private Vector3 lastPosition = new Vector3(0f, 0f, 0f);
    void Start()
```

```
    {
        controller = GetComponent<CharacterController>();
    }
    void Update()
    {
        isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance,
groundMask);
        if (isGrounded && velocity.y < 0)
        {
            velocity.y = -2f;
        }
        float x = Input.GetAxis("Horizontal");
        float z = Input.GetAxis("Vertical");
        Vector3 move = transform.right * x + transform.forward * z;
        controller.Move(move * speed * Time.deltaTime);
        if (Input.GetButtonDown("Jump") && isGrounded)
        {
            velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity);
        }
        velocity.y += gravity * Time.deltaTime;
        controller.Move(velocity * Time.deltaTime);
        if (lastPosition != gameObject.transform.position && isGrounded ==
true)
        {
            isMoving = true;

        }
        else
        {
            isMoving = false;
        }

        lastPosition = gameObject.transform.position;
    }
}
```

Weapon.cs

```
using System.Collections;
using UnityEngine;

public class Weapon : MonoBehaviour
{
    public bool isActiveWeapon;
```

```
[Header("Shooting")]
public bool isShooting, readyToShoot;
bool allowReset = true;
public float shootingDelay = 2f;

[Header("Burst")]
public int bulletsPerBurst = 3;
public int burstBulletsLeft;

[Header("Spread")]
public float spreadIntensity;
public float hipSpreadIntensity;
public float adsSpreadIntensity;

[Header("Bullet ch-tics")]
public GameObject bulletPrefab;
public Transform bulletSpawn;
public float bulletVelocity = 100;
public float bulletPrefabLifeTime = 3f;

[Header("Media")]
public GameObject muzzleEffect;
internal Animator animator;

[Header("Spawn position")]
public Vector3 spawnPosition;
public Vector3 spawnRotation;

[Header("Ammo")]
public float reloadTime;
public int magazineSize, bulletsLeft;
public bool isReloading;

internal bool isADS;

public enum ShootingMode
{
    Single,
    Burst,
    Auto
}

public enum WeaponModel
{
    pistol1911,
    m4
}

public WeaponModel weaponModel;

void Update()
{
    if (isActiveWeapon)
    {
        if (Input.GetMouseButtonDown(1))
        {
            EnterADS();
        }
        if (Input.GetMouseButtonUp(1))
        {
            ExitADS();
        }
```

```
            GetComponent<Outline>().enabled = false;

            if (bulletsLeft == 0 && isShooting)
            {
                SoundManager.Instance.emptyMagazineSound1911.Play();
            }

            if (currentShootingMode == ShootingMode.Auto)
            {
                isShooting = Input.GetKey(KeyCode.Mouse0);
            }
            else if (currentShootingMode == ShootingMode.Single ||
currentShootingMode == ShootingMode.Burst)
            {
                isShooting = Input.GetKeyDown(KeyCode.Mouse0);
            }

            if (Input.GetKeyDown(KeyCode.R) && bulletsLeft < magazineSize &&
!isReloading && WeaponManager.Instance.CheckAmmoLeftFor(weaponModel) > 0)
            {
                Reload();
            }

            if (readyToShoot && isShooting && bulletsLeft > 0)
            {
                burstBulletsLeft = bulletsPerBurst;
                FireWeapon();
            }
        }
    }

    public ShootingMode currentShootingMode;

    internal void Awake()
    {
        readyToShoot = true;
        burstBulletsLeft = bulletsPerBurst;
        animator = GetComponent<Animator>();

        bulletsLeft = magazineSize;

        spreadIntensity = hipSpreadIntensity;
    }

    private void EnterADS()
    {
        animator.SetTrigger("enterADS");
        isADS = true;
        HUDManager.Instance.middleDot.SetActive(false);
        spreadIntensity = adsSpreadIntensity;
    }

    private void ExitADS()
    {
        animator.SetTrigger("exitADS");
        isADS = false;
        HUDManager.Instance.middleDot.SetActive(true);
        spreadIntensity = hipSpreadIntensity;
    }

    internal void FireWeapon()
    {
        bulletsLeft--;
```

```
        muzzleEffect.GetComponent<ParticleSystem>().Play();

        if (isADS)
        {
            animator.SetTrigger("RECOIL_ADS");
        }
        else
        {
            animator.SetTrigger("RECOIL");
        }

        SoundManager.Instance.PlayShootingSound(weaponModel);

        readyToShoot = false;

        Vector3 shootingDirection = CalculateDirectionAndSpread().normalized;

        GameObject bullet = Instantiate(bulletPrefab, bulletSpawn.position,
Quaternion.identity);

        bullet.transform.forward = shootingDirection;

        bullet.GetComponent<Rigidbody>().AddForce(shootingDirection *
bulletVelocity, ForceMode.Impulse);

        StartCoroutine(DestroyBulletTime(bullet, bulletPrefabLifeTime));

        if (allowReset)
        {
            Invoke("ResetShot", shootingDelay);
            allowReset = false;
        }

        // Burst Mode
        if (currentShootingMode == ShootingMode.Burst && burstBulletsLeft > 1)
        {
            burstBulletsLeft--;
            Invoke("FireWeapon", shootingDelay);
        }
    }

    private void Reload()
    {
        SoundManager.Instance.PlayReloadSound(weaponModel);
        animator.SetTrigger("RELOAD");

        isReloading = true;
        Invoke("ReloadCompleted", reloadTime);
    }

    internal void ReloadCompleted()
    {
        if (WeaponManager.Instance.CheckAmmoLeftFor(weaponModel) >
magazineSize)
        {
            bulletsLeft = magazineSize;
            WeaponManager.Instance.DecreaseTotalAmmo(bulletsLeft, weaponModel);
        }
        else
        {
            bulletsLeft = WeaponManager.Instance.CheckAmmoLeftFor(weaponModel);
            WeaponManager.Instance.DecreaseTotalAmmo(bulletsLeft, weaponModel);
```

```
        }

        isReloading = false;
    }

    private void ResetShot()
    {
        readyToShoot = true;
        allowReset = true;
    }

    public Vector3 CalculateDirectionAndSpread()
    {
        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);
        }

        Vector3 direction = targetPoint - bulletSpawn.position;

        float z = UnityEngine.Random.Range(-spreadIntensity, spreadIntensity);
        float y = UnityEngine.Random.Range(-spreadIntensity, spreadIntensity);

        return direction + new Vector3(0, y, z);
    }

    private IEnumerator DestroyBulletTime(GameObject bullet, float delay)
    {
        yield return new WaitForSeconds(delay);
        Destroy(bullet);
    }
}
```

WeaponManager.cs

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

public class WeaponManager : MonoBehaviour
{
    public static WeaponManager Instance { get; set; }

    public List<GameObject> weaponSlots;

    public GameObject activeWeaponSlot;

    [Header("Ammo")]
    public int totalRifleAmmo = 0;
    public int totalPistolAmmo = 0;

    private void Awake()
    {
        if (Instance != null && Instance != this)
```

```
        {
            Destroy(gameObject);
        }
        else
        {
            Instance = this;
        }
    }

    private void Start()
    {
        activeWeaponSlot = weaponSlots[0];
    }

    public void Update()
    {
        foreach (GameObject weaponSlot in weaponSlots)
        {
            if (weaponSlot == activeWeaponSlot)
            {
                weaponSlot.SetActive(true);
            }
            else
            {
                weaponSlot.SetActive(false);
            }
        }

        if (Input.GetKeyDown(KeyCode.Alpha1))
        {
            SwitchActiveSlot(0);
        }
        if(Input.GetKeyDown(KeyCode.Alpha2))
        {
            SwitchActiveSlot(1);
        }
    }

    public void PickUpWeapon(GameObject pickedUpWeapon)
    {
        AddWeaponIntoActiveSlot(pickedUpWeapon);
    }

    private void AddWeaponIntoActiveSlot(GameObject pickedUpWeapon)
    {
        DropCurrentWeapon(pickedUpWeapon);

        pickedUpWeapon.transform.SetParent(activeWeaponSlot.transform, false);

        Weapon weapon = pickedUpWeapon.GetComponent<Weapon>();

        pickedUpWeapon.transform.localPosition = new
Vector3(weapon.spawnPosition.x, weapon.spawnPosition.y,
weapon.spawnPosition.z);
        pickedUpWeapon.transform.localRotation =
Quaternion.Euler(weapon.spawnRotation.x, weapon.spawnRotation.y,
weapon.spawnRotation.z);

        weapon.isActiveWeapon = true;
        weapon.animator.enabled = true;
    }

    private void DropCurrentWeapon(GameObject pickedUpWeapon)
```

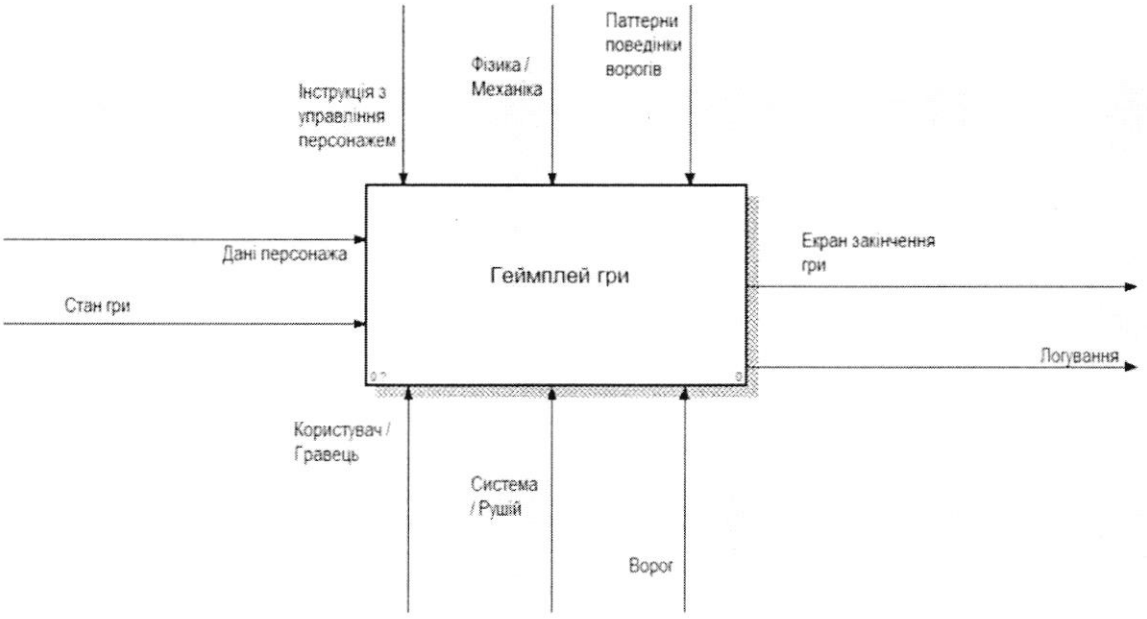
```
    {
        if (activeWeaponSlot.transform.childCount > 0)
        {
            var weaponToDrop =
activeWeaponSlot.transform.GetChild(0).gameObject;
            weaponToDrop.GetComponent<Weapon>().isActiveWeapon = false;
            weaponToDrop.GetComponent<Weapon>().animator.enabled = false;

            weaponToDrop.transform.SetParent(pickedUpWeapon.transform.parent);
            weaponToDrop.transform.localPosition =
pickedUpWeapon.transform.localPosition;
            weaponToDrop.transform.localRotation =
pickedUpWeapon.transform.localRotation;
        }
    }

    public void SwitchActiveSlot(int slotNumber)
    {
        if (activeWeaponSlot.transform.childCount > 0)
        {
            Weapon currentWeapon =
activeWeaponSlot.transform.GetChild(0).GetComponent<Weapon>();
            currentWeapon.isActiveWeapon = false;
        }

        activeWeaponSlot = weaponSlots[slotNumber];

        if (activeWeaponSlot.transform.childCount > 0)
        {
            Weapon newWeapon =
activeWeaponSlot.transform.GetChild(0).GetComponent<Weapon>();
            newWeapon.isActiveWeapon = true;
        }
    }

    internal void PickUpAmmo(AmmoBox ammo)
    {
        switch (ammo.ammoType)
        {
            case AmmoBox.AmmoType.PistolAmmo:
                totalPistolAmmo += ammo.ammoAmount;
                break;
            case AmmoBox.AmmoType.RifleAmmo:
                totalRifleAmmo += ammo.ammoAmount;
                break;
        }
    }

    internal void DecreaseTotalAmmo(int bulletsToDecrease, Weapon.WeaponModel
weaponModel)
    {
        switch (weaponModel)
        {
            case Weapon.WeaponModel.m4:
                totalRifleAmmo -= bulletsToDecrease;
                break;
            case Weapon.WeaponModel.pistol1911:
                totalPistolAmmo -= bulletsToDecrease;
                break;
        }
    }

    internal int CheckAmmoLeftFor(Weapon.WeaponModel weaponModel)
```

```
    {
        switch (weaponModel)
        {
            case Weapon.WeaponModel.m4:
                return WeaponManager.Instance.totalRifleAmmo;
            case Weapon.WeaponModel.pistol1911:
                return WeaponManager.Instance.totalPistolAmmo;
            default:
                return 0;
        }
    }
}
```

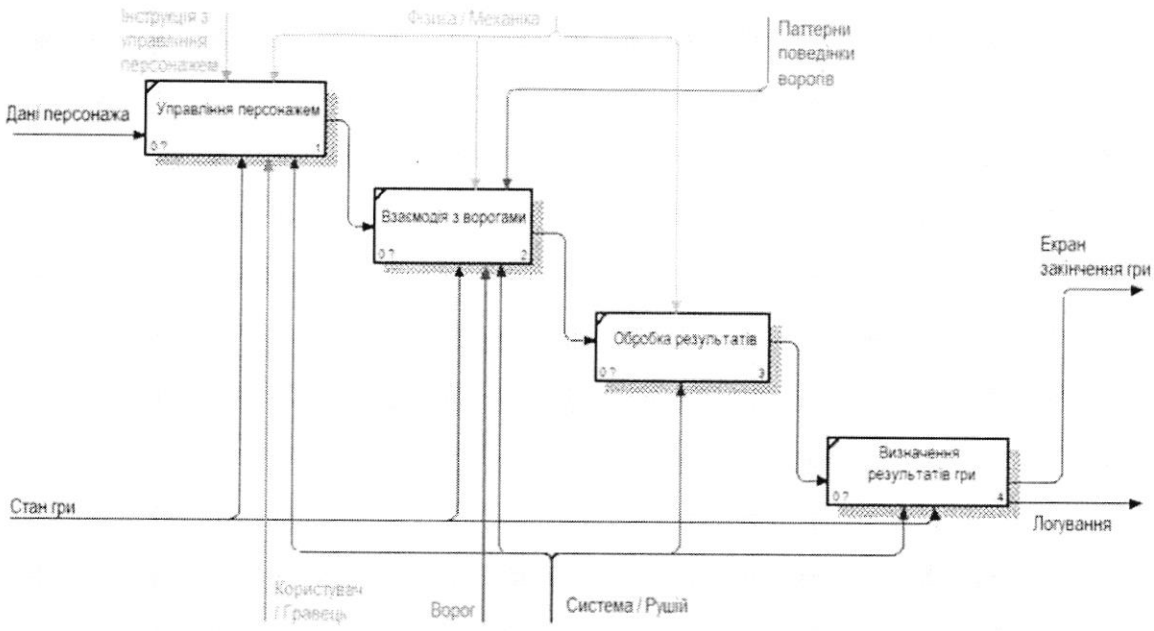
SoundManager.cs

```
using UnityEngine;
using static Weapon;

public class SoundManager : MonoBehaviour
{
    public static SoundManager Instance { get; set; }

    public AudioSource shootingChannel;

    public AudioClip PM1911Shot;
    public AudioSource reloadingSound1911;
    public AudioSource emptyMagazineSound1911;

    public AudioClip M4Shot;
    public AudioSource reloadingSoundM4;

    private void Awake()
    {
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
        }
        else
        {
            Instance = this;
        }
    }

    public virtual void PlayShootingSound(WeaponModel weapon)
    {
        switch (weapon)
        {
            case WeaponModel.pistol1911:
                shootingChannel.PlayOneShot(PM1911Shot); break;
            case WeaponModel.m4:
                shootingChannel.PlayOneShot(M4Shot); break;
        }
    }
    public void PlayReloadSound(WeaponModel weapon)
    {
        switch (weapon)
        {
            case WeaponModel.pistol1911:
                reloadingSound1911.Play(); break;
            case WeaponModel.m4:
                reloadingSoundM4.Play(); break;
        }
    }}
```

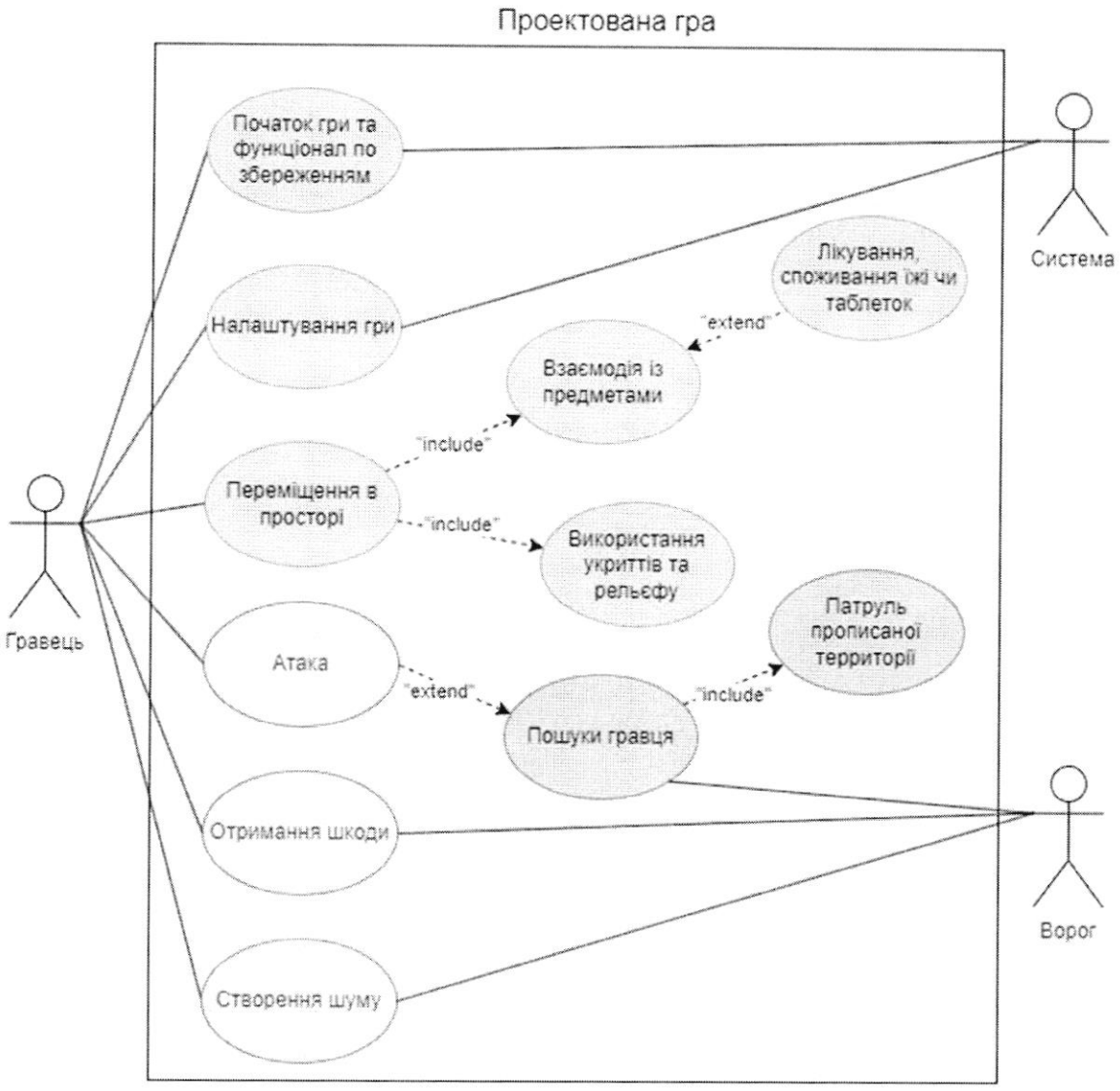
ГРАФІЧНА ЧАСТИНА



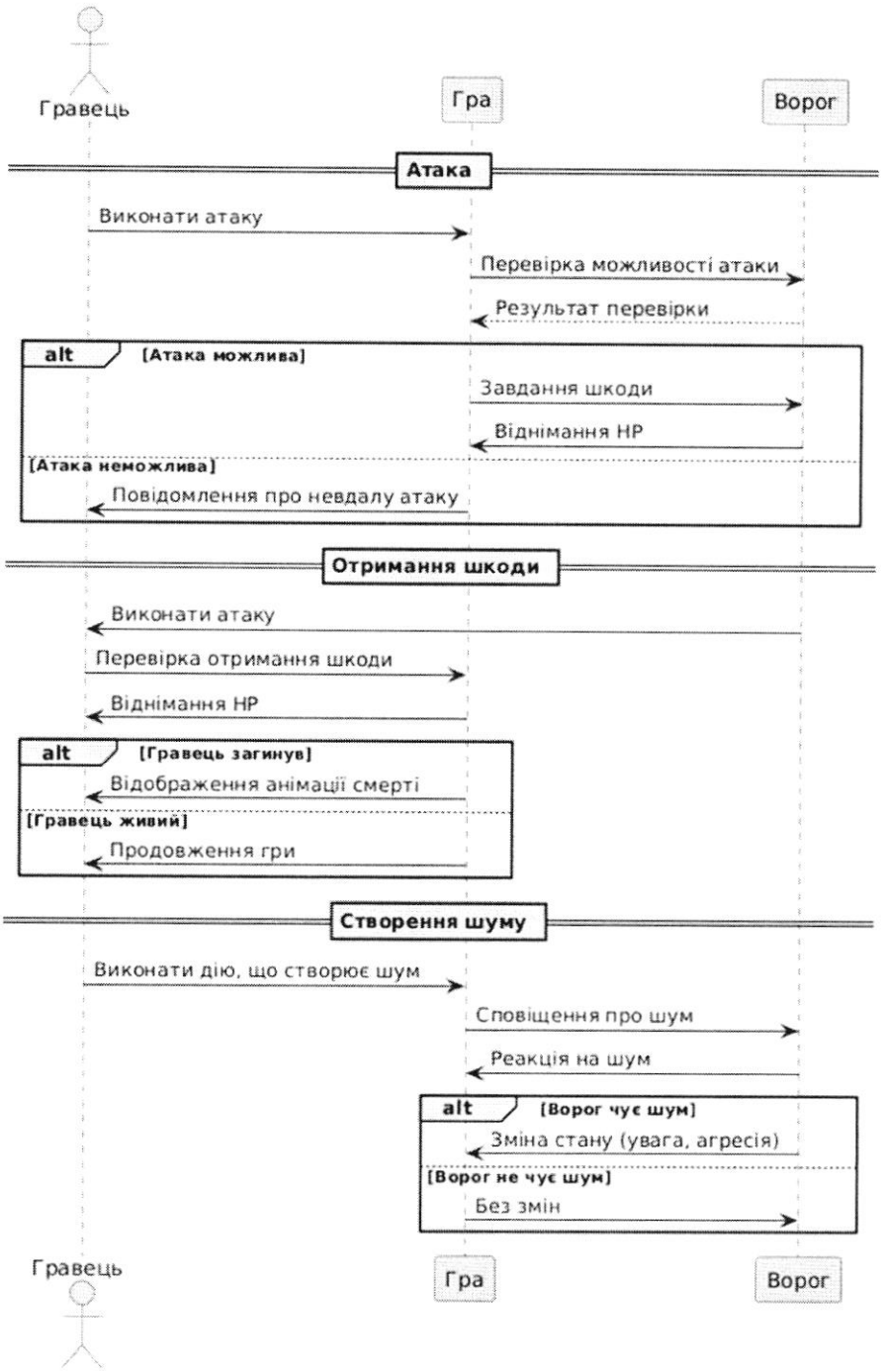
					КвРІПЗ.2101069.01.02.Е8			
					Програмне забезпечення гри-шутера на Unity Контекстна діаграма А-0 «Геймплей гри»	*Літ.*	*Маса.*	*Масштаб*
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*				
Розробив		Гайдученко А. С.		04.06		Аркуш 1	Аркушів 5	
Керівник		Яшина О. М.		04.06				
Н. Контр.		Форкун Ю. В.		04.06		ХНУ, ІПЗ-21-1		
Зав. каф.		Бедратюк Л. П.		04.06				



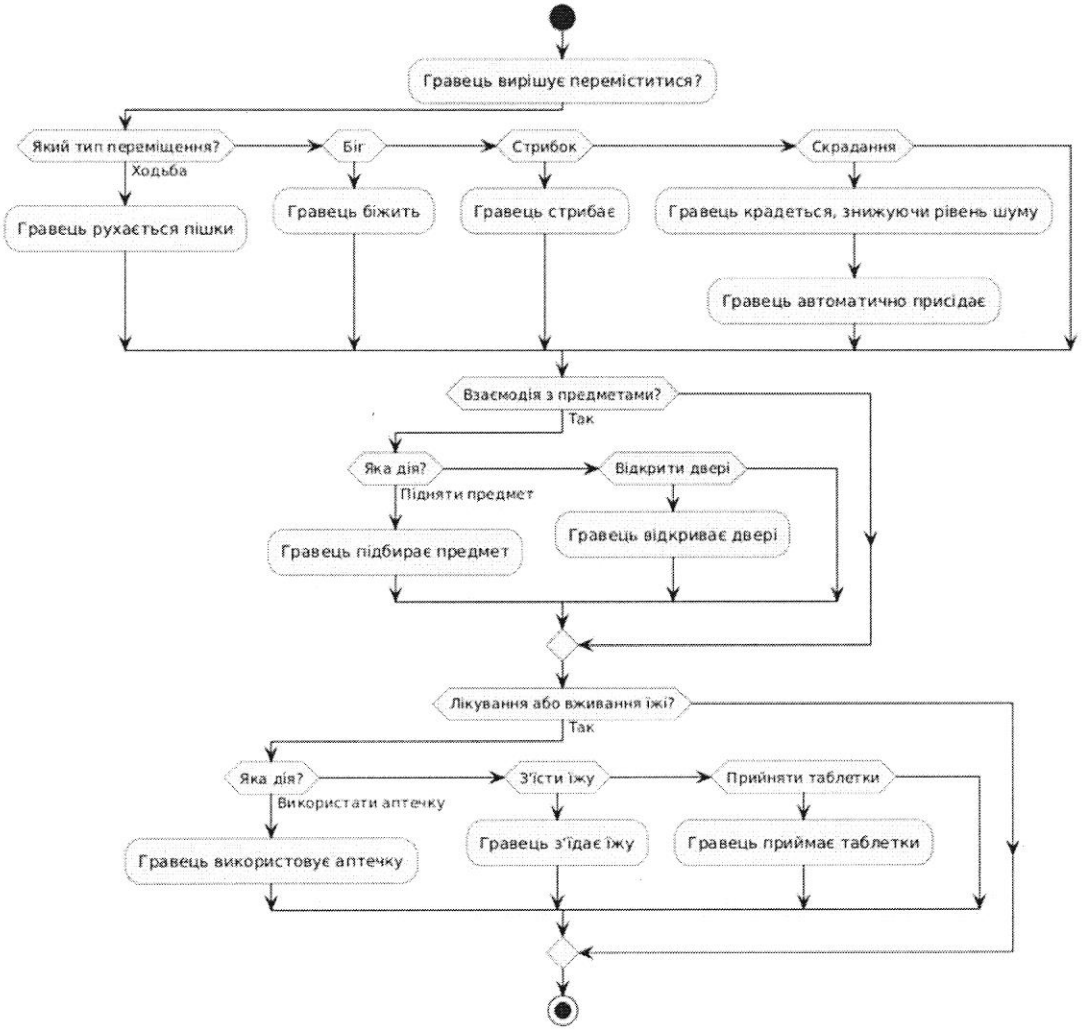
					КвРІПЗ.2101069.01.02.Е8			
					Програмне забезпечення гри-шутера на Unity Діаграма декомпозиції А-1 «Геймплей гри»	*Літ.*	*Маса.*	*Масштаб*
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*				
Розробив		Гайдученко А. С.		04.06		Аркуш 2	Аркушів 5	
Керівник		Яшина О. М.		04.06				
Н. Контр.		Форкун Ю. В.		04.06		ХНУ, ІПЗ-21-1		
Зав. каф.		Бедратюк Л. П.		05.06				



					КвРІПЗ.2101069.01.02.Е8			
					Програмне забезпечення гри-шутера на Unity	*Літ.*	*Маса.*	*Масштаб*
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*				
Розробив		Гайдученко А. С.		04.06				
Керівник		Яшина О. М.		04.06	Діаграма варіантів використання гри	Аркуш 3	Аркушів 5	
Н. Контр.		Форкун Ю. В.		04.06		ХНУ, ІПЗ-21-1		
Зав. каф.		Бедратюк Л. П.		05.06				



					КвРІПЗ.2101069.01.02.Е8		
					Програмне забезпечення гри-шутера на Unity Діаграма послідовностей сірих сценаріїв	*Літ.*	*Маса.*	*Масштаб*
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*				
Розробив		Гайдученко А. С.		04.06		Аркуш 4	Аркушів 5	
Керівник		Яшина О. М.		04.06				
Н. Контр.		Форкун Ю. В.		04.06		ХНУ, ІПЗ-21-1		
Зав. каф.		Бедратюк Л. П.		05.06				



					КвРІПЗ.2101069.01.02.Е8		
					Програмне забезпечення гри-шутера на Unity Блок схема для сценарію «Переміщення в просторі»	*Літ.*	*Маса.*	*Масштаб*
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*				
Розробив		Гайдученко А. С.		04.06		Аркуш 5	Аркушів 5
Керівник		Яшина О. М.		04.06			
Н. Контр.		Форкун Ю. В.		04.06		ХНУ, ІПЗ-21-1	
Зав. каф.		Бедратюк Л. П.		05.06			

СУПРОВІДНІ ДОКУМЕНТИ

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Гайдученка Артема Сергійовича
факультет ІТ, IV курс, група ІПЗ-21-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

29.05.2025
дата

підпис

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Гайдученко

Співавтор:

Назва: БКР_Програмне забезпечення гри-шутера на Unity

Науковий керівник:

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:2.7%

Коефіцієнт подібності 2:0.5%

Мікропробіли: 0

Заміна букв: 2

Інтервали: 0

Білі знаки: 2

Дата створення звіту: 2025-05-29 12:31:24.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата експерт

29.05.2025

Anti-Plagiarism (UA) v-15.281 Educational

The maximum coincidence with one document 2.0%

Dictionaries check: en_US, ru_RU, ua_UA. **Errors in the documents: 12%**

	Document		Sum coincidence on the DB	
ID: 242386 Title: БКР_Програмне забезпечення гри-шутера на Unity Added in a DB: 2025-05-29 Authors: Гайдученко Heads: Оксана ЯШИНА канд. техн. наук, доцент Consultants: Opponents:	Symbols	Lexemes	Symbols	Lexemes
	66767	1019	1994 (3%)	23 (2%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

освітнього ступеня «Бакалавр»

Дипломник Гайдученко Артем Сергійович
Тема Програмне забезпечення гри-шутера на Unity
Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 5 ; кількість сторінок записки 73 .
1.Короткий зміст пояснювальної записки та прийнятих рішень У кваліфікаційній роботі розглянуто створення гри в жанрі «шутер». Проведено аналіз предметної області, сформульовано вимоги, обґрунтовано архітектурні та технологічні рішення. Реалізовано продукт можливостями стрільби, взаємодією з ворогами, їх базову логіку поведінки. Проведено тестування, що підтвердило коректну роботу розробленого програмного забезпечення.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана відповідно до поставленого завдання та з дотриманням всіх вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи Розробка проєкту розпочалася з обґрунтування доцільності створення ігрової сцени в жанрі шутера з елементами виживання та хорору. Було визначено основні художні та технічні орієнтири, сформульовано вимоги до функціональності й візуального оформлення, зокрема щодо створення напруженої атмосфери та стилізації під естетику відеозапису. На наступному етапі виконано проєктування архітектури сцени з використанням можливостей рушія Unity. Реалізовано ключові ігрові механіки, включаючи переміщення, стрільбу, взаємодію з оточенням і поведінку противників. Візуальні елементи та ефекти постобробки були підібрані з урахуванням загальної стилістики та вимог до продуктивності. Завершальним етапом стало тестування сцени, під час якого перевірено стабільність роботи функціональних модулів, відповідність реалізованих рішень заданим вимогам і виконано базову оптимізацію. Результати засвідчили працездатність системи та досягнення запланованого ефекту візуального й ігрового досвіду.

4. Позитивні сторони роботи Проєкт демонструє сучасний підхід до створення інтерактивного ігрового середовища з елементами виживання. Візуальне оформлення виконано з урахуванням стилістичної цілісності. Реалізовано основні механіки взаємодії, що створюють напружену атмосферу. У роботі застосовано можливості рушія Unity на належному технічному рівні.

5. Негативні сторони роботи Обмеженість функціоналу: реалізовано лише базові механіки без поглибленої системи ігрової логіки чи прогресу. Не передбачено складних анімацій або сценаріїв штучного інтелекту. Візуальні ефекти могли б бути більш варіативними для глибшої імерсії.

6. Оцінка графічного оформлення та пояснювальної записки Графічне оформлення відповідає тематиці роботи та містить діаграми архітектури, послідовностей, компонентів тощо. Пояснювальна записка оформлена згідно вимог і має логічну структуру викладення матеріалу.

7. Відгук про кваліфікаційну роботу в цілому Кваліфікаційна робота є завершеною, цілісною та технічно грамотною. У пояснювальній записці чітко прослідковується логіка побудови рішення – від аналізу проблеми до її практичної реалізації. Застосовані сучасні технології та архітектурні підходи забезпечили створення хорошого продукту. Робота добре структурована та наповнена ілюстративним матеріалом.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана у повному обсязі, відповідає поставленій задачі та заслуговує на оцінку «відмінно».

РЕЦЕНЗЕНТ Лисенко Сергій Миколайович, доктор технічних наук, професор, заступник декана факультету інформаційних технологій з наукової та міжнародної роботи.

"4" червня 2025 р. _____
(підпис)

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Програмне забезпечення гри-шутера на Unity
Автор Гайдученко Артем Сергійович
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»
Рівень вищої освіти Перший (бакалаврський)
Спеціальність 121 «Інженерія програмного забезпечення»
Науковий керівник: Яшина Оксана Миколаївна, канд. техн. наук, доцент

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	**відповідає**
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті кваліфікаційної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, завдання, анотація, відомість документів), у структурі змісту, назвах розділів/підрозділів тощо, у назвах публікацій та у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посиланання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає 2,7%, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 5.06.2025

Завідувач кафедри _____ Підпис Леонід БЕДРАТЮК Ім'я, ПРІЗВИЩЕ

Гарант освітньої програми _____ Підпис Леонід БЕДРАТЮК Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи _____ Підпис Оксана ЯШИНА Ім'я, ПРІЗВИЩЕ