МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня бакалавра

на тему: «**Розробка ігрового додатку для PC з елементами взаємодії із оточенням в середовищі Unity»**

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Загребельний Євгеній Михайлович

Керівник: *доктор філософії з прикладної математики, старший викладач кафедри інформаційних технологій та аналітики даних*
Красюк Богдан Віталійович

Рецензент: *Розробник програмного забезпечення в ТОВ ОБНОВА-ЄВРОШОП, викладач кафедри інформаційних технологій та аналітики даних НаУОА*
Шведюк Володимир Володимирович

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ
Завідувач кафедри інформаційних технологій та аналітики даних _____
(проф., д.е.н. Кривицька О.Р.)
Протокол №10 від 28 травня 2025 р.

Острог, 2025

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: *Розробка ігрового додатку для PC з елементами взаємодії із оточенням в середовищі Unity*

Автор: *Загребельний Євгеній Михайлович*

Науковий керівник: *Красюк Богдан Віталійович, доктор філософії з прикладної математики, викладач, фахівець-практик*

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: *53 с., 1 рис., 1 табл., 1 додаток, 16 джерел.*

Ключові слова: *Unity, шутер від першої особи (FPS), геймдизайн, взаємодія з оточенням, ігрові механіки, штучний інтелект, UI, C#, розробка ігор.*

Короткий зміст праці:
Кваліфікаційна робота присвячена дослідженню процесу розробки інтерактивного ігрового додатку для персонального комп'ютера в середовищі Unity. У роботі проаналізовано особливості рушія Unity, методи реалізації механік руху персонажа, стрільби, стрибків, бігу по стінах, використання гака, а також систему взаємодії з оточенням. Реалізовано інтерфейс користувача, моделі об'єктів та систему руйнування за допомогою C# і Blender. Створено штучний інтелект для неігрових персонажів. Вивчено приклади успішних FPS-проєктів для визначення актуальних підходів. Результатом стала повнофункціональна прототипна гра з базовими елементами геймплею, що демонструє інтеграцію технічних рішень та дизайнерських принципів сучасної розробки ігор.

The qualification work is dedicated to the study of the development process of an interactive game application for personal computers using the Unity engine. The paper analyzes the features of the Unity engine and methods for implementing mechanics such as character movement, shooting, jumping, wall running, and grappling, as well as systems for interaction with the environment. The user interface, object models, and destruction system were developed using C# and Blender. Artificial Intelligence was created for non-playable characters. Examples of successful FPS projects were reviewed to identify relevant approaches. As a result, a fully functional prototype game with basic gameplay elements was created, demonstrating the integration of technical solutions and design principles of modern game development.

Зміст

ВСТУП

Рушій Unity є одним із провідних інструментів для розробки інтерактивних ігор завдяки своїй універсальності, гнучкості, широкому набору інструментів та кросплатформеності. Його використання дозволяє створювати як прості мобільні ігри, так і масштабні тривимірні проєкти з реалістичною графікою, фізикою та складними сценаріями взаємодії.

Актуальність обраної теми зумовлена стійкою популярністю ігрових додатків для персональних комп'ютерів (PC), що відзначаються високим рівнем геймплею, графіки та інтерактивності. Розробка таких продуктів вимагає глибокого розуміння особливостей ігрового середовища, ефективної інтеграції механік взаємодії із віртуальним оточенням, а також дотримання вимог до якості та продуктивності сучасних ігор.

Метою кваліфікаційної роботи є дослідження процесу розробки ігрового додатку для PC в середовищі Unity з інтеграцією елементів взаємодії користувача із оточенням. У ході виконання роботи буде спроєктовано та реалізовано гру, який демонструватиме базові ігрові механіки: систему руху, стрільби, ШІ а також інтерактивну взаємодію з об'єктами середовища.

Об’єктом дослідження є ринок ігрових додатків для ПК з елементами взаємодії з оточенням.

Предметом дослідження є проєктування та розробка ігрового додатку для ПК з елементами взаємодії користувача з оточенням у середовищі Unity.

Основні завдання, що необхідно вирішити для досягнення мети дослідження:

- Проаналізувати особливості середовища Unity для розробки інтерактивних ігрових додатків для PC.
- Дослідити сучасні підходи до реалізації механік взаємодії користувача з об'єктами середовища.

- Розробити концепцію ігрового додатку з визначенням ключових елементів геймплею та взаємодії.
- Реалізувати основні ігрові механіки у середовищі Unity.
- Провести тестування прототипу з оцінкою його функціональності, інтерактивності та продуктивності.

Результати дослідження та реалізації проєкту будуть корисними для подальших розробок ігрових додатків, орієнтованих на платформу PC, а також для впровадження сучасних рішень у сфері геймдеву.

РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Опис предметного середовища

Ігрова індустрія на сьогодні є однією з найдинамічніших сфер програмної розробки. Вона об'єднує інноваційні технології, художнє бачення, психологію користувача та інженерію. Серед найпопулярніших жанрів комп’ютерних ігор особливе місце займає шутер від першої особи (FPS – First Person Shooter), який забезпечує високий рівень занурення гравця у віртуальний світ за рахунок перспективи «очима персонажа».

Цей жанр характеризується тим, що гравець бачить ігровий світ очима персонажа, що забезпечує глибоке занурення у процес та високий рівень взаємодії з ігровим середовищем. Основний ігровий процес полягає у знищенні супротивників за допомогою різних видів вогнепальної або енергетичної зброї, проходженні місій, вирішенні тактичних або стратегічних завдань.

Ігри жанру FPS зазвичай орієнтовані на динамічний геймплей, що вимагає від гравця швидкої реакції, точності та вміння орієнтуватися в просторі. Камера розташовується таким чином, щоб імітувати погляд персонажа — у полі зору видно лише руки героя, зброю та навколишній простір, що створює ефект присутності. Такий підхід дозволяє підсилити емоційне сприйняття подій, адже все, що відбувається, ніби трапляється з гравцем безпосередньо.

Основні характеристики FPS-жанру включають:

- Камера від першої особи: відображення гри з точки зору персонажа, що підсилює ефект занурення;
- Система керування персонажем: зазвичай поєднує клавіатурне управління переміщенням і мишу для огляду та прицілювання;
- Озброєння та стрільба: ключовий елемент, що передбачає наявність різноманітної зброї з унікальними характеристиками (швидкострільність, точність, шкода, дальність тощо);

- Боєприпаси та ресурси: обмежена кількість патронів, аптечок, броні тощо, які гравець повинен знаходити або купувати;
- Інтерактивне середовище: можливість взаємодії з об'єктами — відкриття дверей, активація механізмів, використання укриттів;
- Штучний інтелект ворогів: поведінка NPC (non-player characters) змінюється в залежності від дій гравця, рівня складності чи сценарію гри;
- Ігрові місії та сценарій: зазвичай FPS має сюжетну складову або місії, які потрібно виконувати на кожному рівні (наприклад, знищити ворогів, врятувати заручників, дістатися до певного місця)

Ігри жанру FPS також значною мірою залежать від рівня інтерактивності та можливостей гравця взаємодіяти з навколишнім ігровим середовищем. На відміну від аркадних шутерів минулого, де середовище було статичним фоном для бойових дій, сучасні FPS активно впроваджують технології, що дозволяють створювати динамічний, реагуючий світ, який змінюється залежно від дій користувача.

Взаємодія з оточенням у FPS може мати різні форми:

- Фізична взаємодія: гравець може відкривати двері, активувати важелі, піднімати або пересувати об’єкти, користуватися технікою тощо. Такі елементи створюють ілюзію реальності та поглиблюють занурення у гру.
- Руйнування об’єктів: у багатьох сучасних іграх реалізовано часткове або повне знищення елементів оточення, таких як стіни, барикади, вікна. Це дозволяє змінювати тактичну ситуацію прямо під час бою.
- Використання укриттів: гравець може ховатися за об’єктами, що частково або повністю захищають від вогню противника. Часто така механіка поєднується з системою присідання, нахилу, стрільби з-за рогу.
- Взаємодія з NPC та елементами сценарію: у сюжетних FPS користувач може брати участь у діалогах, рятувати цивільних, супроводжувати союзників, відповідати за вибір сценарного шляху.

- Збирання ресурсів і предметів: підбирання зброї, боєприпасів, аптечок, ключів, квестових об'єктів також є частиною механіки взаємодії з ігровим середовищем.

1.2 Опис середовища розробки

У процесі створення ігрового додатку важливим етапом є вибір середовища розробки, яке відповідатиме технічним та функціональним вимогам проєкту. Для цього були проаналізовані сучасні інструменти розробки ігор, а також середовища для створення 3D-моделей та інтеграції їх у ігровий рушій. Вибір було здійснено з урахуванням таких факторів, як функціональні можливості, сумісність між інструментами, простота освоєння та підтримка спільноти.

1.2.1 Ігровий рушій

У процесі вибору ігрового рушія для розробки додатку було розглянуто два найпопулярніших рушії сучасності: Unity та Unreal Engine. Обидва є потужними інструментами для створення високоякісних ігор та інтерактивних додатків, мають великий функціонал, активну спільноту користувачів та підтримку новітніх технологій.

Unity — це кросплатформений ігровий рушій, що активно використовується для створення 2D та 3D ігор, віртуальної та доповненої реальності, а також симуляцій. Основні переваги Unity:

- Мультиплатформеність — підтримка понад 25 платформ, включаючи Windows, Mac, Linux, Android, iOS, PlayStation, Xbox.
- Гнучка система роботи з 2D та 3D графікою — Unity має інтегровані інструменти для побудови тривимірних сцен, а також для роботи з 2D іграми.
- Висока продуктивність та можливість оптимізації проєктів під різні платформи.
- Використання мови C# — одна з найпопулярніших мов програмування, що забезпечує легке освоєння та високу продуктивність.
- Unity Asset Store — велика бібліотека ресурсів, моделей, текстур, скриптів, що спрощує та прискорює розробку.

- Зручний UI-редактор та система анімацій.

Unreal Engine — потужний рушій від компанії Epic Games, що здебільшого орієнтований на створення графічно насичених AAA-проєктів. Основні переваги Unreal Engine:

- Фотореалістична графіка завдяки рушію Lumen та Nanite — дозволяє досягати вражаючої деталізації та реалістичності сцен.
- BluePrint Visual Scripting — система візуального програмування, яка дозволяє створювати ігрову логіку без написання коду.
- Вбудована система управління фізикою Chaos, яка дозволяє реалізовувати складні фізичні симуляції.
- Використання мови програмування C++, що дозволяє досягти високої продуктивності та гнучкості, однак вимагає більш високого рівня підготовки розробника.

Тепер порівняємо обидва рушії, звернувши увагу на ключові характеристики, які можуть вплинути на вибір між ними:

Таблиця 1.1

Порівняльна таблиця ігрових рушіїв

Характеристика	Unity	Unreal Engine
Тип ігор	2D, 3D, VR/AR, мобільні	3D, VR, AAA-проєкти, симуляції
Мова програмування	C#	C++, Blueprints
Графічні можливості	Високі	Дуже високі
Крива навчання	Помірна	Висока
Підтримка платформ	Більше 25 платформ	ПК, консолі, VR
Система фізики	Вбудована PhysX	Chaos Physics
Інтеграція сторонніх ресурсів	Unity Asset Store	Unreal Marketplace
Продуктивність	Висока для середніх проєктів	Висока для великих проєктів

Як результат після аналізу можливостей обох рушіїв для реалізації кваліфікаційної роботи було обрано Unity. Це рішення було прийнято з урахуванням кількох ключових факторів. По-перше, Unity має простий інтерфейс і легкий вхід для початківців та середніх розробників, що є важливим для освітнього проєкту. По-друге, цей рушій оптимально підходить для розробки невеликих та середніх проєктів завдяки зручному циклу розробки, тестування та деплою, що дозволяє ефективно реалізувати гру з інтерактивною взаємодією з оточенням. Крім того, використання мови C# спрощує освоєння порівняно з C++, дозволяючи зосередитися на реалізації функціоналу гри, а не на складних аспектах управління пам'яттю. Останнім, але не менш важливим фактором є широка підтримка Unity та наявність великої кількості навчальних матеріалів, що сприяють швидкому освоєнню і реалізації задуму.

1.2.2 Моделювання

Для створення 3D-моделей та анімацій у рамках кваліфікаційної роботи було обрано Blender. Це потужне середовище для 3D-моделювання, яке активно використовується як професіоналами, так і початківцями завдяки своїй гнучкості, широкому набору інструментів та безкоштовності.Основні переваги мови C# для розробки в Unity:

Основні причини вибору Blender для моделювання:

- Безкоштовність та доступність:

  Blender є відкритим програмним забезпеченням з безкоштовною ліцензією, що дає змогу використовувати його без будь-яких фінансових обмежень. Це особливо важливо для освітніх проєктів, де бюджет часто є обмеженим, а можливість використовувати професійний інструмент безкоштовно є значною перевагою.

- Потужні інструменти для 3D-моделювання та анімації:

  Blender надає широкий спектр інструментів для створення 3D-моделей, текстурування, рендерингу та анімації. Завдяки своїм потужним можливостям для роботи з полігонами, кривими та іншими об'єктами, Blender є ідеальним інструментом для створення складних 3D-сцен та персонажів. Можливості для створення анімацій, від простих рухів до складних механізмів, дозволяють створювати інтерактивні елементи для ігор з високим рівнем деталізації.

- Інтеграція з Unity:

  Blender має зручну інтеграцію з Unity, що дозволяє експортувати моделі та анімації з Blender прямо в Unity для подальшого використання в ігровому процесі. Це значно спрощує робочий процес, дозволяючи швидко тестувати та вносити зміни в моделі без необхідності виконувати складні процедури імпорту чи конвертації.

- Широка підтримка спільноти та навчальних матеріалів:

  Blender має велику і активну спільноту користувачів, що забезпечує доступ до безлічі туторіалів, форумів та інших ресурсів, які значно полегшують процес навчання та освоєння програмного забезпечення. Крім того, наявність численних безкоштовних і платних плагінів та додатків розширює функціональність Blender.
- Можливості для створення фізичних симуляцій:

  Blender підтримує потужні інструменти для симуляції фізичних явищ, таких як рідини, дим, частинки та тверді тіла. Ці можливості можуть бути використані для створення реалістичних ефектів у грі, що взаємодіють з навколишнім середовищем.

1.2.3 Середовище написання коду

Для розробки програмного забезпечення в рамках кваліфікаційної роботи було обрано Visual Studio як основне середовище розробки. Visual Studio є потужним і багатофункціональним інструментом для створення різноманітних типів програм, включаючи ігри, веб-додатки, мобільні програми та інші програмні рішення.

Причини вибору Visual Studio:

- Підтримка мови C#:

  Visual Studio надає повну підтримку мови програмування C#, яка є основною мовою для розробки в середовищі Unity. Це забезпечує зручність і ефективність програмування, оскільки всі інструменти, плагіни та відлагоджувачі інтегровані безпосередньо в середовище.
- Інтеграція з Unity:

  Visual Studio безпосередньо інтегрується з Unity, що дозволяє зручно писати код для гри, а також здійснювати його тестування і налагодження в одному середовищі. Завдяки цьому процес розробки стає більш зручним та швидким.

- Потужний відлагоджувач:

  Visual Studio надає потужний відлагоджувач, який дозволяє детально досліджувати виконання програми, ставити точки зупину, переглядати значення змінних та виконувати код по кроках. Це значно спрощує процес виправлення помилок і оптимізації програм.

- Зручний інтерфейс та підтримка розширень:

  Visual Studio має зручний та інтуїтивно зрозумілий інтерфейс, який дозволяє розробнику легко переходити між різними частинами проекту. Крім того, середовище підтримує безліч розширень та плагінів, які можуть бути додані для полегшення роботи, таких як інструменти для профілювання, версійний контроль, інтеграція з іншими середовищами та багато інших.

- Підтримка багатьох платформ:

  Visual Studio підтримує розробку для різних платформ, включаючи Windows, macOS, Android, iOS та інші. Це дозволяє створювати додатки, які можуть працювати на різних пристроях і операційних системах, зокрема для ігор на платформі Unity.

- Широка документація та спільнота:

  Visual Studio має величезну кількість документації та активну спільноту розробників, що значно полегшує навчання та допомагає вирішити можливі технічні питання. Окрім того, багато онлайн-ресурсів, відеоуроків і форумів допомагають освоїти всі можливості цього середовища.

1.3 Огляд наявних аналогів

Для ефективного проєктування власного ігрового додатку важливо ознайомитися з уже існуючими рішеннями в обраній предметній області. У цьому розділі розглянуто приклади популярних ігор у жанрі шутер від першого лиця (FPS), які мають розвинену систему інтерактивності та взаємодії з ігровим середовищем. Аналіз таких аналогів

дозволяє краще зрозуміти поточні тенденції, вимоги до функціоналу, а також інноваційні підходи в дизайні ігрових механік.

1.3.1 Half-Life 2 (2004, Valve)

Half-Life 2 є однією з найвпливовіших ігор у жанрі FPS. Гра зробила справжню революцію завдяки впровадженню розширеної фізики та інноваційної механіки взаємодії з об'єктами в ігровому світі. Ігровий рушій Source дозволив реалізувати складну модель поведінки предметів, використання гравітаційної гармати (Gravity Gun) стало символом взаємодії з оточенням.

Гравець може переміщувати предмети, будувати барикади, запускати об'єкти в супротивників, відкривати або зламувати двері. Усе це стало можливим завдяки вбудованим фізичним законам. Значну роль у грі відіграють також сценарні події, що залежать від дій користувача. Оточення в грі не лише реалістичне, але й логічно вплетене в сюжет.

Геймплей у Half-Life 2 вирізняється плавним поєднанням бойових сцен, головоломок, платформінгу та елементів дослідження, що робить її зразковим прикладом балансу між стрілянням та взаємодією з ігровим світом.

1.3.2 Call of Duty (серія, Infinity Ward / Treyarch / Sledgehammer Games)

Серія Call of Duty, яка розпочалась у 2003 році, зробила акцент на інтенсивний, кінематографічний бойовий досвід. В іграх використовуються сценарні скрипти, які чітко керують подіями та просуванням сюжету. Взаємодія з оточенням обмежена і зазвичай реалізована через підказки: натискання кнопки для відкриття дверей, посадка в техніку, взаємодія з вибраними NPC.

З часом у серію додавали елементи руйнування середовища, використання дронів, укриттів, нічного бачення, а також інтерактивних місій зі стелсом. Наприклад, в Modern Warfare (2019) гравець має змогу проводити зачистки кімнат, приймати рішення у реальному часі, сканувати територію тепловізором.

Попри обмежену свободу дій, Call of Duty створює відчуття активної участі у великомасштабному конфлікті завдяки точній постановці сцен, глибокому озвученню та швидкому темпу гри.

1.3.3 Killzone (серія, Guerrilla Games)

Killzone — серія науково-фантастичних шутерів від першого лиця, створена для платформ PlayStation. Вона вирізняється темною, похмурою атмосферою, складними візуальними рішеннями та використанням потужної зброї. Killzone акцентує увагу на тактичному бою і використанні укриттів. Особливо зручно реалізовано систему стрільби з-за кутів, що впливає на тактичне мислення гравця.

Killzone 2 та 3 демонструють реалістичну фізику, анімовану реакцію ворогів на постріли, руйнування об’єктів, а також часткову взаємодію з елементами оточення — відкриття дверей, активація консолей, використання стаціонарної зброї.

Ігрова механіка відзначається інерційністю рухів, що створює більш відчутну присутність у просторі. Також гра реалізує взаємодію з напарниками та використання скриптів у бойових ситуаціях, які підсилюють атмосферу військового конфлікту.

1.3.4 BLACK (2006, Criterion Games)

Гра BLACK, розроблена для PlayStation 2 та Xbox, стала культовою завдяки надзвичайному акценту на звуковому супроводі, деталізації зброї та масштабному руйнуванню середовища. Гравець отримував справжнє тактильне відчуття від стрільби — кожен постріл супроводжувався ефектами, а елементи середовища (вікна, двері, паркани) активно руйнувалися.

Взаємодія з оточенням у BLACK не була складною, але реалізація візуального впливу гравця на середовище зробила гру унікальною. Наприклад, ворогів можна було

вразити не лише напряму, а й завдяки руйнуванню навколишніх конструкцій або викликанню вибухів через навколишні об'єкти.

BLACK стала однією з перших ігор, де стрільба виглядала і звучала "вагомо", що значно вплинуло на подальший розвиток FPS-ігор з акцентом на реалізм бою та візуальні ефекти.

Висновки до розділу 1

Розробка ігрового додатку вимагає вибору з багатьох різноманітних інструментів та середовищ розробки. Для реалізації даного проєкту було обрано Unity, яке забезпечує зручний інтерфейс та оптимальні можливості для створення ігор середнього рівня складності з інтерактивною взаємодією з оточенням. Порівняно з іншими рушіями, Unity є більш доступним для початківців та розробників середнього рівня завдяки простоті освоєння та підтримці широкого спектру платформ.

Для реалізації моделей було обрано Blender. для моделювання в рамках розробки ігрового додатку є обґрунтованим завдяки його багатофункціональності та відкритості. Це середовище дозволяє ефективно створювати 3D-моделі, текстури та анімації, які можуть бути безпроблемно інтегровані в Unity. Blender забезпечує високий рівень деталізації та реалізму, що важливо для створення інтерктивного середовища в грі. Враховуючи безкоштовний доступ до цього інструменту, а також його широку спільноту користувачів та документацію, Blender є ідеальним вибором для розробників, які прагнуть досягти професійних результатів з обмеженими витратами.

РОЗДІЛ 2. РЕАЛІЗАЦІЯ БАЗОВИХ МЕХАНІК

2.1 Реалізація механіки руху персонажа

Для забезпечення природного та динамічного пересування персонажа у грі була реалізована система руху, що включає кілька ключових механік. Основною метою є створення гнучкої та реалістичної поведінки персонажа з урахуванням фізики, рельєфу та взаємодії з оточенням.

2.1.1 Базове переміщення

Реалізація переміщення персонажа базується на обробці вхідних даних через систему введення (InputManager) та фізичних взаємодіях за допомогою компонента Rigidbody (див. Лістинг 2.1). Рух здійснюється шляхом додавання сили у відповідному напрямку, залежно від положення камери та орієнтації персонажа.

```
public void Movement(bool move)
{

    //Extra gravity
    if (!IsPlayerOnSlope() || (IsPlayerOnSlope() && rb.velocity.y < 0))
rb.AddForce(Vector3.down * Time.deltaTime * 10);

    //Find actual velocity relative to where player is looking
    Vector2 relativeVelocity = FindVelRelativeToLook();
    float xRelativeVelocity = relativeVelocity.x, yRelativeVelocity =
relativeVelocity.y;

    //Counteract sliding and sloppy movement
    FrictionForce(InputManager.x, InputManager.y, relativeVelocity);

    //If speed is larger than maxspeed, cancel out the input so you don't go over max
speed
    if (InputManager.x > 0 && xRelativeVelocity > currentSpeed || InputManager.x < 0
&& xRelativeVelocity < -currentSpeed) InputManager.x = 0;
    if (InputManager.y > 0 && yRelativeVelocity > currentSpeed || InputManager.y < 0
&& yRelativeVelocity < -currentSpeed) InputManager.y = 0;

    float multiplier = (!grounded) ? controlAirborne : 1;
    float multiplierV = (!grounded) ? controlAirborne : 1;

    float multiplier2 = (weaponController.weapon != null) ?
weaponController.weapon.weightMultiplier : 1;

    if (rb.velocity.sqrMagnitude < .02f) rb.velocity = Vector3.zero;

    if (!move)
    {
```

```
        if (grounded) rb.velocity = Vector3.zero;
        return;
    }
    if (isCrouching && rb.velocity.magnitude >= crouchSpeed &&
!allowMoveWhileSliding) return;

    if (IsPlayerOnSlope())
    {
        rb.AddForce(GetSlopeDirection() * acceleration * Time.deltaTime * multiplier
/ multiplier2);

        rb.useGravity = false;

        if (rb.velocity.y > 0) rb.AddForce(Vector3.down * 70, ForceMode.Force);
    }
    else
    {
        rb.AddForce(orientation.transform.forward * InputManager.y * acceleration *
Time.deltaTime * multiplier * multiplierV / multiplier2);
        rb.AddForce(orientation.transform.right * InputManager.x * acceleration *
Time.deltaTime * multiplier / multiplier2);
    }
}
```

Лістинг 2.1 Реалізація базового переміщення.

Джерело: Автор

Основні особливості реалізації:

- **Додаткова гравітація:**

  Щоб уникнути некоректної поведінки персонажа на схилах або в повітрі, застосовуються додаткові сили тяжіння, що гарантують стабільний контакт із землею.

- **Контроль руху на схилах:**

  Перевіряється, чи перебуває персонаж на похилій поверхні. У випадку перебування на схилі використовується спеціальний метод для визначення напрямку руху по похилій поверхні, з метою коректного додавання сили у цьому напрямку. Також вимикається стандартна гравітація Rigidbody для уникнення некоректної взаємодії з фізикою Unity.

- **Обмеження максимальної швидкості:**

  Для запобігання перевищенню максимально дозволеної швидкості здійснюється перевірка відносної швидкості руху по кожній з осей. При досягненні ліміту введення блокується.

- **Протидія ковзанню:**

  Використовується спеціальний метод, який аналізує напрямок та швидкість руху персонажа, після чого додає силу, спрямовану проти ковзання. Це дозволяє мінімізувати небажане ковзання по поверхнях та забезпечити більш контрольований рух.
- **Корекція руху у повітрі:**

  Якщо персонаж перебуває у повітрі, вводяться корекційні коефіцієнти, що знижують ефективність управління, імітуючи менший контроль над рухом.
- **Вплив ваги зброї:**

  У разі, якщо персонаж тримає зброю, враховується додатковий коефіцієнт, що знижує швидкість та прискорення персонажа залежно від ваги обраної зброї.

Завдяки такій структурі та реалізації механік досягається плавне, фізично обґрунтоване переміщення персонажа, яке адаптується до різних ситуацій: рух по рівній поверхні, схилах, у повітрі та під час взаємодії з елементами оточення.

2.1.2 Стрибки

Механіка стрибка реалізована у файлі PlayerMovement.cs (див. Лістинг 2.2), що є центральним скриптом управління рухом персонажа. Метод Jump() забезпечує як базовий стрибок, так і більш складні варіанти, такі як стрибок від стін та напрямлені стрибки.

```
public void Jump()
        {
            if (!allowJump || jumpCount <= 0) return;

            jumpCount--;
            readyToJump = false;
            hasJumped = true;
            cancellingGrounded = false;

            if (doubleJumpResetsFallDamage) stats.height = transform.position.y;

            //Add jump forces
            if (wallRunning) // When we wallrun, we want to add extra side forces
            {
                rb.velocity = new Vector3(rb.velocity.x, 0, rb.velocity.z);
                rb.AddForce(transform.up * upwardsWallJumpForce);
```

```
                rb.AddForce(wallNormal * normalWallJumpForce, ForceMode.Impulse);
            }
            else
            {
                rb.AddForce(Vector3.up * jumpForce * 1.5f, ForceMode.Impulse);
                rb.AddForce(normalVector * jumpForce * 0.5f, ForceMode.Impulse);
                // Handle directional jumping
                if (!grounded && directionalJumpMethod != DirectionalJumpMethod.None
&& maxJumps > 1 && !wallOpposite)
                {
                    if (Vector3.Dot(rb.velocity, new Vector3(InputManager.x, 0,
InputManager.y)) > .5f)
                        rb.velocity = rb.velocity / 2;
                    if (directionalJumpMethod == DirectionalJumpMethod.InputBased) //
Input based method for directional jumping
                    {
                        rb.AddForce(orientation.right * InputManager.x *
directionalJumpForce, ForceMode.Impulse);
                        rb.AddForce(orientation.forward * InputManager.y *
directionalJumpForce, ForceMode.Impulse);
                    }
                    if (directionalJumpMethod ==
DirectionalJumpMethod.ForwardMovement) // Forward Movement method for directional
jumping, dependant on orientation
                        rb.AddForce(orientation.forward * Mathf.Abs(InputManager.y) *
directionalJumpForce, ForceMode.VelocityChange);
                }

                //If jumping while falling, reset y velocity.
                if (rb.velocity.y < 0.5f)
                    rb.velocity = new Vector3(rb.velocity.x, 0, rb.velocity.z);
                else if (rb.velocity.y > 0)
                    rb.velocity = new Vector3(rb.velocity.x, rb.velocity.y / 2,
rb.velocity.z);
            }

            //staminaLoss
            if (usesStamina) stamina -= staminaLossOnJump;


            SoundManager.Instance.PlaySound(sounds.jumpSFX, 0, 0, false, 0);
            Invoke(nameof(ResetJump), jumpCooldown);
        }
```

Лістинг 2.2 Реалізація стрибків.

Джерело: Автор

Основні особливості реалізації:

- **Перевірка умов для стрибка:**

  Першочергово виконується перевірка дозволу на стрибок та наявності доступних стрибків. Якщо параметр allowJump відключений або кількість залишкових

стрибків (jumpCount) дорівнює нулю, дія стрибка блокується. Після виконання стрибка кількість доступних стрибків зменшується на одиницю.

- **Стрибок від стін (Wall Jump):**

  Якщо персонаж знаходиться у стані бігу по стіні (wallRunning), здійснюється специфічна обробка стрибка. Вертикальна складова швидкості обнуляється, після чого додаються сили вертикального відштовхування (upwardsWallJumpForce) та бокового відштовхування від стіни (normalWallJumpForce). Це дозволяє виконувати акробатичні трюки, розширюючи можливості пересування в складних умовах.

- **Звичайний стрибок:**

  У випадках, коли персонаж знаходиться на землі або не виконує біг по стіні, додається стандартна вертикальна сила стрибка (jumpForce). Також враховується нормаль поверхні для коректної взаємодії з похилими площинами. Якщо активовано функцію напрямленого стрибка, додаються додаткові сили у напрямку руху, що визначається станом керування (InputManager).

- **Корекція вертикальної швидкості:**

  Для забезпечення природної динаміки руху, у разі стрибка під час падіння, вертикальна швидкість обмежується. Якщо персонаж має високу швидкість по осі Y, вона зменшується для уникнення неприродного прискорення або сповільнення.

- **Витрати витривалості:**

  При активній системі витривалості (usesStamina) після виконання стрибка витрачається визначена кількість витривалості (staminaLossOnJump), що додає глибини до геймплейних механік та змушує гравця більш обережно використовувати стрибки.

- **Звукові ефекти.**

  Для посилення відчуття динаміки та глибини ігрового процесу під час стрибка відтворюється відповідний звуковий ефект через систему SoundManager.

- **Скидання стану після стрибка:**

Після виконання стрибка активується відлік часу (jumpCooldown), після завершення якого відбувається скидання стану та дозволяється виконати наступний стрибок.

Завдяки такій структурі механіки стрибка, гравець отримує можливість виконувати складні комбінації рухів, що робить геймплей більш захоплюючим та багатогранним.

2.1.3 Ковзання та присідання

Ковзання активується під час присідання на великій швидкості (див. Лістинг 2.3).

```
public void StartCrouch()
        {
            if (!allowCrouch) return;
            isCrouching = true;

            if (crouchCancelMethod == CrouchCancelMethod.FullStop)
                currentSpeed = crouchSpeed;

            if (rb.velocity.magnitude >= walkSpeed && grounded && allowSliding &&
!hasJumped)
            {
                events.OnSlide.Invoke();

                rb.AddForce(orientation.transform.forward * slideForce);

                if (usesStamina) stamina -= staminaLossOnSlide;
            }
        }
```

Лістинг 2.3 Реалізація ковзання та присідання.

Джерело: Автор

Основні особливості реалізації:

- **Перевірка умов для ковзання:**

  Виконується перевірка дозволу на присідання (allowCrouch). У разі дозволу змінна isCrouching приймає значення true, і якщо швидкість персонажа перевищує walkSpeed, він перебуває на землі (grounded), дозволено ковзання (allowSliding), а також не було здійснено стрибка (hasJumped), відбувається активація ковзання.

- **Активація ковзання:**

Під час активації викликається подія OnSlide, а до персонажа додається сила у напрямку його руху (slideForce), що створює ефект ковзання.

- **Витрати витривалості:**

  Якщо увімкнено використання витривалості (usesStamina) за потреби, при ковзанні витрачається staminaLossOnSlide.

2.1.4 Біг по стінах

Метод WallRun() (див. Лістинг 2.4) реалізує механіку руху персонажа по вертикальних стінах, додаючи динамічності та вертикальності геймплею.

```
private void WallRun()
        {
            wallNormal = wallRight ? wallRightHit.normal : wallLeftHit.normal;
            wallDirection = Vector3.Cross(wallNormal, transform.up).normalized * 10;

            if ((orientation.forward - wallDirection).magnitude > 
(orientation.forward + wallDirection).magnitude) wallDirection = -wallDirection;

            if (OppositeVectors() < -.5f) StopWallRun();

            if (cancelWallRunMethod == CancelWallRunMethod.Timer)
            {
                wallRunTimer -= Time.deltaTime;
                if (wallRunTimer <= 0)
                {
                    rb.AddForce(wallNormal * stopWallRunningImpulse, 
ForceMode.Impulse);
                    StopWallRun();
                }
            }

            if (!wallRunning) StartWallRunning();

            rb.useGravity = useGravity;

            if (useGravity && rb.velocity.y < 0) rb.AddForce(transform.up * 
wallrunGravityCounterForce, ForceMode.Force);
            rb.velocity = Vector3.ClampMagnitude(rb.velocity, maxWallRunSpeed);

            if (!(wallRight && InputManager.x < 0) && !(wallLeft && InputManager.x > 
0)) rb.AddForce(-wallNormal * 100, ForceMode.Force);

            rb.AddForce(wallDirection, ForceMode.Force);
        }
```

Лістинг 2.4 Реалізація бігу по стінах.

Основні особливості реалізації:

- **Визначення напряму бігу:**

  Обчислюється нормаль стіни (wallNormal) та напрям бігу по стіні (wallDirection) шляхом перехресного добутку нормалі та вектора вгору (transform.up). У разі, якщо напрям протилежний орієнтації гравця, напрям бігу інвертується.

- **Перевірка умов припинення бігу по стіні:**

  Якщо вектори напрямку руху та орієнтації персонажа надто протилежні (OppositeVectors() < -0.5), викликається метод StopWallRun(), який припиняє біг по стіні.

- **Таймер обмеження тривалості:**

  Якщо активовано метод завершення за таймером (CancelWallRunMethod.Timer), час бігу по стіні поступово зменшується (wallRunTimer). Коли таймер досягає нуля, персонаж відштовхується від стіни з імпульсом (stopWallRunningImpulse), після чого біг по стіні припиняється.

- **Гравітація:**

  Гравітація може бути увімкнена або вимкнена в залежності від налаштувань (useGravity). Якщо персонаж втрачає висоту під час бігу по стіні, додається сила протидії гравітації (wallrunGravityCounterForce), яка дозволяє зменшити вплив гравітації і підтримувати стабільний рух по стіні.

- **Обмеження швидкості:**

  Швидкість персонажа обмежується до максимальної, встановленої для бігу по стіні (maxWallRunSpeed), щоб уникнути неконтрольованого прискорення.

- **Корекція руху до стіни:**

  Якщо персонаж не рухається в напрямку відповідної стіни (правої або лівої), до нього прикладається сила, що притискає його до стіни (у напрямку протилежному нормалі стіни), щоб уникнути передчасного падіння.

- **Додавання сили руху:**

  Постійно додається сила у напрямку бігу по стіні для підтримання руху.

2.1.5 Гак

Метод StartGrapple() (див. Лістинг 2.5) реалізує функціонал використання гака для швидкого переміщення персонажа до заданої точки. Це додає динамічності до геймплею, дозволяючи гравцю долати значні відстані або швидко уникати небезпеки.

```
private void StartGrapple()
        {
            if (!grappleEnabled) return;
            RaycastHit hit;

            if (Physics.Raycast(weaponController.mainCamera.transform.position,
weaponController.mainCamera.transform.forward, out hit, maxGrappleDistance,
whatIsGround))
            {
                grappling = true;
                grapplePoint = hit.point;


                joint = gameObject.AddComponent<SpringJoint>();
                joint.autoConfigureConnectedAnchor = false;
                joint.connectedAnchor = grapplePoint;

                float distanceFromAnchor = Vector3.Distance(this.transform.position,
grapplePoint);

                joint.maxDistance = distanceFromAnchor * 0.8f;
                joint.minDistance = distanceFromAnchor * 0.25f;

                joint.spring = grappleSpringForce;
                joint.damper = grappleDamper;
                joint.massScale = 4.5f;

                events.OnStartGrapple?.Invoke();// Perform custom methods
                if (grappleSounds.startGrappleSFX)
                    SoundManager.Instance.PlaySound(grappleSounds.startGrappleSFX, 0,
0, false, 0);
            }
        }
```

Лістинг 2.5 Реалізація гаку.

Джерело: Автор

Основні особливості реалізації:

1. **Активація механіки**

   Перевіряється, чи включений гак через змінну grappleEnabled. Якщо гак недоступний, метод завершиться без дій. Використовується Raycast, щоб визначити точку зачеплення (grapplePoint) в межах максимальної дистанції (maxGrappleDistance).

2. **Ініціалізація гака**

   Створюється компонент SpringJoint, який забезпечує фізичний зв’язок між персонажем і точкою зачеплення (grapplePoint). Встановлюються параметри з'єднання:

   - maxDistance: максимальна відстань, на якій персонаж може залишатися прикріпленим.
   - minDistance: мінімальна відстань, до якої персонаж може наблизитися.
   - spring: сила натягування троса.
   - damper: амортизація, що зменшує коливання троса.
   - massScale: вплив маси персонажа на поведінку троса.

3. **Аудіоефекти та події**

   Відтворюється звук початку використання гака через SoundManager. Викликаються події з events.OnStartGrapple, які можуть бути використані для налаштування додаткових дій, таких як візуальні ефекти або інші дії, що супроводжують активацію механіки.

2.2 Реалізація зброї

Реалізація механіки зброї базується на взаємодії кількох класів і компонентів, таких як WeaponController, Weapon_SO, WeaponIdentification, і Bullet. Основні функції включають стрільбу, прицілювання, перезарядку, анімацію та роботу з додатковими ефектами, такими як віддача та сліди куль.

2.2.1 Основні компоненти зброї

1. **WeaponController**

   Основний клас, який керує всіма аспектами роботи зброї:

   - Перемикання між зброєю.
   - Відтворення анімацій.
   - Стрільба (хітскан або снаряд).
   - Перезарядка.

2. **Weapon_SO**

   ScriptableObject, що зберігає параметри зброї:

   - Тип стрільби (хітскан, снаряд, ближній бій).
   - Швидкість кулі, час між пострілами, розмір магазину, ефекти вибухів тощо.
   - Налаштування прицілювання, FOV, та віддачі.

3. **WeaponIdentification**

   Прив'язується до префаба зброї і дозволяє використовувати її в грі:

   - Точка вильоту кулі (FirePoint).
   - Сумісні модифікації, такі як глушники, приціли тощо.

4. **Bullet**

   Реалізація кулі, яка враховує траєкторію, шкоду, гравітацію, вибухи та ефекти взаємодії з іншими об'єктами.

2.2.2 Реалізація стрільби

Метод хітскан стрільби забезпечує миттєве виявлення попадання снаряда в об'єкти (наприклад, для снайперських гвинтівок або пістолетів). Реалізація цього методу здійснюється за допомогою використання Physics.Raycast для визначення попадання кулі в об'єкт (див. Лістинг 2.6).

```
private void HitscanShot()
        {
            events.OnShoot.Invoke();
            if (resizeCrosshair && UIController.instance.crosshair != null)
UIController.instance.crosshair.Resize(weapon.crosshairResize * 100);

            Transform hitObj;

            //This defines the first hit on the object
            Vector3 dir = FPS_ProjectUtilities.GetSpreadDirection(spread,
mainCamera);
            Ray ray = new Ray(mainCamera.transform.position, dir);

            if (Physics.Raycast(ray, out hit, weapon.bulletRange, hitLayer))
            {
                float dmg = damagePerBullet * stats.damageMultiplier;
                Hit(hit.collider.gameObject.layer, dmg, hit, true);
                hitObj = hit.collider.transform;

                if (hit.transform.TryGetComponent<Rigidbody>(out Rigidbody rb))
                {
                    rb.AddForceAtPosition(ray.direction * 10, hit.point,
ForceMode.Impulse);
                }

                //Handle Penetration
                Ray newRay = new Ray(hit.point, ray.direction);
                RaycastHit newHit;

                if (Physics.Raycast(newRay, out newHit, penetrationAmount, hitLayer))
                {
                    if (hitObj != newHit.collider.transform)
                    {
                        float dmg_ = damagePerBullet * stats.damageMultiplier *
weapon.penetrationDamageReduction;
                        Hit(newHit.collider.gameObject.layer, dmg_, newHit, true);
                    }
                }

                // Handle Bullet Trails
                if (weapon.bulletTrail == null) return;

                foreach (var p in firePoint)
                {
                    TrailRenderer trail = Instantiate(weapon.bulletTrail, p.position,
Quaternion.identity);

                    StartCoroutine(SpawnTrail(trail, hit));
```

```
                }
            }
        }
```

Лістинг 2.6 Реалізація хітскан стрільби.

Джерело: Автор

Опис механіки:

4. **Ініціалізація Raycast:**
    - Напрямок стрільби визначається за допомогою головної камери персонажа (mainCamera).
    - Для автоматичної зброї використовується функція модифікації напрямку стрільби з врахуванням розльоту кулі (Spread) для забезпечення реалістичності.

- **Визначення попадання:**
    - За допомогою Physics.Raycast перевіряється, чи потрапила куля в об'єкт в межах визначеного діапазону (bulletRange).
    - Якщо попадання відбулося, викликається метод Hit(), що застосовує шкоду до об'єкта та ініціює відповідні візуальні та фізичні ефекти.
- **Ефекти кулі:**
    - Для відображення траєкторії кулі використовуються сліди кулі (Bullet Trails), що додаються через компоненти типу TrailRenderer.
    - Додатково можуть бути використані візуальні ефекти, такі як частинки чи ефекти крові, залежно від типу об'єкта, що був вражений.
- **Додаткові дії:**
    - При стрільбі викликаються події через events.OnShoot, що дозволяє відслідковувати статистику, відтворювати анімації чи звукові ефекти, пов'язані з пострілами.

2.2.3 Реалізація анімацій та перезарядки

Процес перезарядки залежить від типу механізму перезарядки, наприклад, стандартного перезарядження або перегріву зброї. Система використовує анімації для візуалізації цього процесу, а також управляє кількістю набоїв в магазині (див. Лістинг 2.7).

```
private IEnumerator DefaultReload()
        {

            // Play reload sound
            SoundManager.Instance.PlaySound(id.bulletsLeftInMagazine == 0 ?
weapon.audioSFX.emptyMagReload : weapon.audioSFX.reload, .1f, 0, true, 0);
            reloading = true;
            yield return new WaitForSeconds(.001f);

            // Play animation
            FPS_ProjectUtilities.PlayAnim("reloading",
inventory[currentWeapon].GetComponentInChildren<Animator>());

            // Wait reloadTime seconds, assigned in the weapon scriptable object.
            yield return new WaitForSeconds(reloadTime);

            // Reload has finished
            events.OnFinishReload.Invoke();

            canShoot = true;
            if (!reloading) yield break;

            // Run custom event
            events.OnReload.Invoke();

            reloading = false;

            // Set the proper amount of bullets, depending on magazine type.
            if (!weapon.limitedMagazines) id.bulletsLeftInMagazine = id.magazineSize;
            else
            {
                if (id.totalBullets > id.magazineSize) // You can still reload a full
magazine
                {
                    id.totalBullets = id.totalBullets - (id.magazineSize -
id.bulletsLeftInMagazine);
                    id.bulletsLeftInMagazine = id.magazineSize;
                }
                else if (id.totalBullets == id.magazineSize) // You can only reload a
single full magazine more
                {
                    id.totalBullets = id.totalBullets - (id.magazineSize -
id.bulletsLeftInMagazine);
                    id.bulletsLeftInMagazine = id.magazineSize;
                }
                else if (id.totalBullets < id.magazineSize) // You cant reload a
whole magazine
                {
```

```
                    int bulletsLeft = id.bulletsLeftInMagazine;
                    if (id.bulletsLeftInMagazine + id.totalBullets <=
id.magazineSize)
                    {
                        id.bulletsLeftInMagazine = id.bulletsLeftInMagazine +
id.totalBullets;
                        if (id.totalBullets - (id.magazineSize - bulletsLeft) >= 0)
id.totalBullets = id.totalBullets - (id.magazineSize - bulletsLeft);
                        else id.totalBullets = 0;
                    }
                    else
                    {
                        int ToAdd = id.magazineSize - id.bulletsLeftInMagazine;
                        id.bulletsLeftInMagazine = id.bulletsLeftInMagazine + ToAdd;
                        if (id.totalBullets - ToAdd >= 0) id.totalBullets =
id.totalBullets - ToAdd;
                        else id.totalBullets = 0;
                    }
                }
            }
        }
```

Лістинг 2.7 Реалізація анімацій та перезарядкі

Джерело: Автор

Опис механіки:

- **Ініціалізація перезарядки:**

  Коли розпочинається перезарядка, система перевіряє, чи є патрони в магазині. Якщо магазин порожній, відтворюється звук для порожнього магазину, а якщо ні — звук для заповненого магазину. Після цього активується анімація перезарядки через компонент Animator.

- **Час перезарядки:**

  Після запуску анімації система чекає заданий час перезарядки, який визначається через параметр reloadTime. Цей час затримує виконання кроків, поки не завершиться процес перезарядки.

- **Завершення перезарядки:**

  Після завершення затримки викликається подія, яка вказує на завершення перезарядки, і відновлюється можливість стрільби. Система також коригує кількість набоїв у магазині згідно з доступними патронами, в залежності від типу магазину (обмежений чи не обмежений).

- **Особливості для обмежених магазинів:**

Якщо магазин обмежений, кількість набоїв в магазині оновлюється таким чином, щоб не перевищити ємність магазину, в залежності від того, скільки набоїв залишилось у запасі. Якщо патронів недостатньо для заповнення магазину, залишок додається до наявних набоїв у магазині, а решта зберігається в запасі.

Висновки до розділу 2

У результаті реалізації механік руху та стрільби для персонажа було досягнуто високої рівності та гнучкості у поведінці персонажа під час взаємодії з навколишнім середовищем.

Механіка руху забезпечує реалістичну взаємодію з навколишнім середовищем завдяки таким механізмам, як ковзання, біг по стінах, і використання гака для переміщення. Ці елементи вносять в гру динамічні моменти та дозволяють гравцеві швидко адаптуватися до змінюваних умов гри.

Стрільба була реалізована через механізми хітскан та снаряди, що дозволяє варіювати стиль гри та надає кожному виду зброї свою унікальність. Хітскан дозволяє досягти високої точності при використанні снайперських гвинтівок, тоді як снарядна стрільба дозволяє використовувати більш складні траєкторії та ефекти.

Анімація та перезарядка додають візуальні ефекти, що покращують занурення у гру. Застосування різних анімацій для процесу перезарядки підвищує реалістичність та інтуїтивну зрозумілість дій гравця, забезпечуючи плавність і безперешкодність ігрового процесу.

Реалізація всіх цих механік дозволила створити інтерактивний та динамічний ігровий процес, який адаптується до вимог гравця та дозволяє глибше зануритись у світ гри.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ВЗАЄМОДІЇ З СЕРЕДОВИЩЕМ

У цьому розділі розглядається процес реалізації механік взаємодії з навколишнім середовищем у контексті розробки ігрової системи. Оскільки сучасні відеоігри значною мірою зосереджуються на інтерактивності, особливо на взаємодії гравця з різними об’єктами в ігровому світі, важливою частиною є правильна реалізація таких аспектів, як інтерфейс користувача, елементи взаємодії та штучний інтелект (ШІ).

3.1 Реалізація інтерфейсу

Реалізація інтерфейсу користувача (UI) забезпечує зручний доступ до інформації про стан гравця, зброю, взаємодію з об’єктами, а також візуалізацію прогресу геймплею. Клас UIController відповідає за управління елементами інтерфейсу, такими як панель здоров'я, боєзапас, індикатор взаємодії та відображення досвіду.

Рис. 3.1. Скріншот з інтерфейсом гри

Джерело: Автор

3.1.1 Основні компоненти інтерфейсу

Інтерфейс користувача в ігровому проєкті виконує функцію візуалізації станів гравця, відображення поточних показників, а також забезпечення інтерактивності з об’єктами ігрового середовища. Клас UIController відповідає за динамічне оновлення та управління усіма ключовими елементами інтерфейсу.

1. **Система індикації здоров’я та захисту**

Візуалізація рівня здоров’я та щита реалізована за допомогою компонентів Slider і TextMeshProUGUI. В залежності від налаштувань (barHealthDisplay та numericHealthDisplay), інтерфейс може відображати дані у вигляді шкали або числових значень. У разі отримання шкоди або відновлення здоров’я, кольори індикатора (healthStatesEffect) змінюються відповідно до типу впливу — наприклад, використовується damageColor при ушкодженні та healColor при лікуванні. Також передбачено поступове згасання ефекту з використанням параметра fadeOutTime.

2. **Відображення боєприпасів та стану зброї**

Графічні елементи bulletsUI та magazineUI демонструють поточну кількість патронів у магазині та загальний боєзапас. Іконка активної зброї (currentWeaponDisplay) оновлюється згідно з вибраною моделлю зброї. Для зброї з механікою перегріву реалізовано окремий індикатор (overheatUI), який змінює свій рівень заповнення залежно від ступеня нагріву.

3. **Взаємодія з об'єктами середовища**

Компоненти interactUI, interactText і interactUIProgressDisplay забезпечують зворотний зв’язок при наближенні до об’єктів, з якими можна взаємодіяти. Текст динамічно оновлюється на основі назви або типу об'єкта. Індикатор прогресу (interactUIProgressDisplay) дозволяє візуально спостерігати за ходом дії, що виконується (наприклад, відкриття дверей, підняття предмета тощо).

4. **Додаткові елементи інтерфейсу**

1. **Індикація влучань**: При влучанні у противника активується hitmarker, який відтворює візуальний ефект, що залежить від зони ураження (наприклад, головний постріл).
2. **Killfeed**: Список останніх вбивств (killfeedContainer) оновлюється автоматично при кожному успішному знищенні противника.

5. **Інші функції**

Інтерфейс також охоплює механіку ривка гравця (dash), інспекцію зброї та взаємодію з елементами інвентарю. Всі відповідні елементи інтерфейсу зберігаються у відповідних контейнерах, які оновлюються через події (event-driven model), що реалізовані за допомогою делегатів та підписок на події (UIEvents).

3.1.2 Інтеграція з FPSController

Інтерфейс користувача інтегрується з основною системою управління гравцем через FPSController, забезпечуючи постійний обмін даними між станом персонажа та відображенням актуальної інформації в елементах інтерфейсу. Це дозволяє оперативно відображати інформацію про здоров’я, боєзапас, стан зброї, взаємодію з об’єктами та інші показники, що впливають на геймплей.

Основні етапи інтеграції:

1. **Підключення до гравця:**

   Інтеграція реалізується шляхом отримання посилання на компонент PlayerMovement, через який отримуються основні дані про стан персонажа, зокрема поточне здоров’я, витривалість, боєзапас, стан зброї та інші параметри.

2. **Оновлення інтерфейсу в реальному часі:**
   - У класі UIController використовується метод Update(), що відповідає за постійне оновлення стану інтерфейсу відповідно до актуальних даних.

- Відображення даних здійснюється за рахунок регулярного отримання інформації з інших систем гри, зокрема зі систем здоров’я, боєзапасу, зброї та взаємодії з об’єктами.

3. **Події інтерфейсу (UI Events):**

   Для забезпечення динамічної взаємодії між компонентами інтерфейсу використовується система подій UIEvents, яка дозволяє реагувати на зміни стану в грі. Наприклад:

   - Використовується подія UIEvents.onHealthChanged, що оновлює відображення панелі здоров’я при отриманні або втраті шкоди.
   - Подія UIEvents.onBulletsChanged використовується для своєчасного оновлення інформації про боєзапас на інтерфейсі.

Також слід зазначити особливості реалізації:

1. **Модульність:**

   Архітектура інтерфейсу побудована таким чином, що дозволяє легко змінювати або замінювати окремі елементи UI без необхідності зміни загальної структури системи.

2. **Гнучкість відображення:**

   - Система підтримує роботу як з текстовими, так і з графічними елементами інтерфейсу.
   - Інтерфейс адаптується до різних стилів гри, забезпечуючи зручність сприйняття інформації незалежно від ігрових умов.

3. **Система подій:**

   Використання подієвої моделі спрощує обмін даними між окремими компонентами, зменшуючи залежність між ними, що сприяє підвищенню гнучкості та масштабованості проекту.

Завдяки такій інтеграції інтерфейс користувача забезпечує гравця актуальною інформацією про стан гри в реальному часі, що позитивно впливає на зручність та динамічність геймплею.

3.2 Реалізація елементів взаємодії

Елементи взаємодії додають інтерактивності до ігрового середовища, забезпечуючи гравцю можливість впливати на об'єкти, такі як ящики, вибухові бочки або спеціальні механізми (наприклад, стрибкові платформи). Реалізація таких об'єктів ґрунтується на модульному підході, який дозволяє використовувати базові класи і розширювати їх під конкретні завдання.

3.2.1 Реалізація руйнування об’єктів

Клас Destructible (див. Лістинг 3.1) є базовим шаблоном для реалізації об’єктів, що підлягають знищенню у віртуальному середовищі гри. Його основна мета — надати універсальний інтерфейс для взаємодії ігрових об’єктів зі системою пошкоджень, що дозволяє повторно використовувати код у межах різних типів об’єктів (наприклад, ящиків, бочок, будівель, техніки тощо).

```
public class Destructible : MonoBehaviour, IDamageable
    {
        float health;
        [Tooltip("Initial health of the destructible"), SerializeField] protected
float maxHealth;

        [Tooltip("Instantiate something cool when the object is destroyed"),
SerializeField]
        protected GameObject lootInside;

        [SerializeField] protected AudioClip destroyedSFX;

        // Set health
        private void Start() => health = maxHealth;

        private void Update()
        {
            // Handle destruction
            if (health <= 0) Die();
        }

        // Handle damage, have in mind that this is also IDamageable
        public void Damage(float damage, bool isHeadshot) => health -= damage;
```

```
        public virtual void Die()
        {
            if (lootInside != null) Instantiate(lootInside, transform.position,
Quaternion.identity);
            Destroy(this.gameObject);
        }
    }
```

Лістинг 3.1 Реалізація руйнування об’єктів

Джерело: Автор

Основним параметром класу є змінна maxHealth, що визначає початковий (максимальний) рівень здоров’я об’єкта. У процесі ініціалізації поточне значення здоров’я об'єкта встановлюється відповідно до цього параметра. Надалі кожного разу, коли об’єкт отримує шкоду, його здоров’я зменшується. Цей процес реалізується шляхом виклику спеціального методу, який відповідає інтерфейсу IDamageable, що забезпечує уніфікований спосіб нанесення шкоди незалежно від конкретного типу об’єкта.

У разі досягнення або перевищення критичного порогу (тобто, коли значення здоров’я стає меншим або дорівнює нулю), виконується процедура знищення об’єкта. Вона реалізована в методі Die, який може бути перевизначений у похідних класах для розширення функціональності. У базовій реалізації цей метод відповідає за фізичне видалення об’єкта зі сцени, а також може ініціювати появу додаткових ефектів, зокрема генерацію лута — візуального або ігрового елементу, що може бути зібраним гравцем (наприклад, бонуси, ресурси, патрони тощо). Для цього використовується попередньо визначений об’єкт lootInside, який при знищенні створюється у позиції знищеного об’єкта.

Також передбачена можливість відтворення звукового ефекту у момент знищення (destroyedSFX), що посилює візуально-аудіальне сприйняття події та покращує загальний ігровий досвід.

Клас побудований з урахуванням принципів об’єктно-орієнтованого програмування, зокрема наслідування та інкапсуляції, що дає змогу розробникам легко розширювати базову поведінку під потреби конкретного ігрового процесу. Наприклад,

успадковуючи Destructible, можна реалізувати спеціалізовані об'єкти, які при знищенні змінюють оточення, викликають нові події або впливають на поведінку інших персонажів.

3.2.2 Реалізація об’єктів взаємодії

У межах ігрової механіки, крім загального класу Destructible, реалізовано низку спеціалізованих об'єктів, які мають індивідуальну поведінку при знищенні або взаємодії з гравцем. Ці об'єкти не лише урізноманітнюють геймплей, а й виступають засобами зворотного зв’язку, що підсилюють відчуття фізичної присутності в ігровому світі.

1. Ящики (Crate)

Ящики є простими статичними об’єктами, які можна знищити в результаті впливу гравця або інших факторів (наприклад, пострілу чи вибуху). Вони наслідують базовий клас Destructible та мають такі особливості:

- **Генерація луту**: Після знищення об’єкта з нього може випадати лут — корисні предмети, які може підібрати гравець (боєприпаси, ресурси, монети тощо).
- **Візуальні ефекти**: Знищення супроводжується появою уламків або інших декоративних об’єктів, що підвищує візуальну достовірність події.
- **Звуковий супровід**: Додатковий звуковий ефект створює відчуття фізичної взаємодії та покращує загальне занурення в ігровий процес.

2. Вибухові бочки (ExplosiveBarrel)

Бочки з вибухівкою виконують подвійну функцію: є як об’єктом середовища, так і джерелом вторинної шкоди. Їхня поведінка визначається рядом особливостей:

- **Вибухова хвиля**: У момент знищення генерується імпульс фізичної сили, який впливає на всі фізичні об’єкти у визначеному радіусі. Це дозволяє, наприклад, відштовхнути інші об’єкти або повалити ворогів.
- **Шкода за радіусом**: Усі об’єкти в зоні ураження отримують шкоду, яка зменшується пропорційно до відстані від епіцентру вибуху, що імітує реалістичну модель фізичного впливу.

- **Візуальна складова**: Вибух супроводжується ефектами частинок (дим, вогонь) та зміною візуального стану об'єкта (наприклад, обгоріла поверхня, зникнення моделі).

3. Стрибкові платформи (JumpPad)

На відміну від знищуваних об’єктів, платформи типу JumpPad не руйнуються, але виконують роль механічної взаємодії з гравцем, дозволяючи йому змінювати позицію у просторі:

- **Вертикальне та горизонтальне прискорення**: Платформи можуть задавати напрямок руху гравця при торканні, що реалізується через додавання сили до компонента фізики (Rigidbody) персонажа.
- **Гнучкість налаштувань**: Параметри сили поштовху, напрямку та тривалості дії можуть бути змінені вручну або адаптовані до рівня складності.
- **Різні режими орієнтації**: Напрямок поштовху може базуватися на локальній осі платформи, напрямку погляду персонажа або його поточному руху, що забезпечує адаптивну поведінку.

3.3 Реалізація ворогів та ШІ

Однією з ключових складових сучасних інтерактивних ігор є наявність ворогів, які не лише створюють перешкоди для гравця, а й значно збагачують ігровий процес завдяки динамічності, виклику та елементу непередбачуваності. Для цього необхідно реалізувати ефективну систему штучного інтелекту (ШІ), яка дозволяє ігровим агентам — ворогам — автономно функціонувати у віртуальному середовищі, реагуючи на дії гравця, зміну обстановки та внутрішні стани.

У межах даного проєкту ШІ противника реалізовано через поєднання компонентів EnemyAI та AI, що базуються на використанні вбудованих засобів Unity, зокрема NavMeshAgent, Animator, Raycasting та обробці колізій. Така архітектура дозволяє створити ворогів, які вміють пересуватись, виявляти гравця у полі зору, відкривати

вогонь, атакувати у ближньому бою, змінювати стани (наприклад, патрулювання, пошук, атака), а також динамічно реагувати на отримання шкоди.

Для моделі ворогів було взято тестову модель персонажа[1].

3.3.1 Реалізація поведінки

Поведінка ігрового ворога визначає його реакції на дії гравця, навколишнє середовище та внутрішні стани. У даному проєкті реалізовано систему поведінки на основі скінченного автомата станів (Finite State Machine), яка забезпечує логічно структуроване та передбачуване перемикання між основними режимами діяльності ворога. Такий підхід дозволяє створити адаптивну та реалістичну модель взаємодії з ігровим світом.

У структурі класу AI поведінка ворога поділяється на три основні стани:

- **Idle (Очікування)** — базовий стан, у якому ворог не бачить гравця та або перебуває у спокої, або виконує запрограмовані дії (патрулювання по маршруту, випадкове переміщення або повна нерухомість). Конкретна поведінка в цьому стані визначається параметром MoveMode, який включає варіанти Waypoints (переміщення за точками), Random (випадковий рух у межах заданого радіусу) та Idle (статична позиція).
- **Search (Пошук)** — активується, коли ворог втрачає візуальний контакт із гравцем, але має підозру щодо його присутності. У цьому стані агент виконує випадкові переміщення в межах області пошуку (searchRadius) і тимчасово збільшує кут огляду, якщо активовано параметр increaseSightOnAttack. Поведінка триває до завершення таймера searchTimer, після чого ворог повертається у стан очікування.
- **Attack (Атака)** — активується, коли гравець виявлений у межах поля зору (FOV) та досяжний без перешкод. Подальші дії залежать від типу ворога (EnemyType): для Shooter використовується атака на відстані, а для Melee — зближення з ціллю і ближній бій. Під час атаки агент пересувається до гравця, орієнтується на нього та змінює анімаційні стани залежно від дистанції та активності.

Перехід між станами здійснюється динамічно у методі Update() відповідно до умов видимості гравця (canSeePlayer) та внутрішніх таймерів. У разі втрати контакту в режимі атаки ворог автоматично переходить у стан пошуку. Це забезпечує природну зміну поведінки та створює ілюзію «розумної» реакції на зміну ситуації (див. Лістинг 3.1).

```
    private void Update()
    {
        if(aiType == AIType.NPC)
        {
            IdleNPCState();
        }

        if(aiType == AIType.Enemy)
        {
            switch (aiState)
            {
                case State.Idle:
                    EnemyIdleState();
                    break;

                case State.Search:
                    EnemySearchState();
                    break;

                case State.Attack:
                    EnemyAttackState();
                    break;
            }

            if (aiState == State.Attack && !canSeePlayer)
                aiState = State.Search;
        }
    }
```

Лістинг 3.1 Реалізація переходу між станами.

Джерело: Автор

Ключовою перевагою реалізованої системи є її гнучкість — завдяки використанню перерахувань (enum) та модульних функцій кожен тип ворога може мати власну поведінкову модель, при цьому загальна логіка взаємодії зберігає цілісність. Такий підхід дозволяє не лише урізноманітнити геймплей, але й розширювати систему новими типами ворогів без необхідності зміни існуючого коду.

3.3.2 Реалізація виявлення гравця

Виявлення гравця є одним із ключових аспектів системи штучного інтелекту ворога. Поведінка ворога до моменту виявлення гравця може змінюватися залежно від

обраного типу: це може бути патрулювання визначеним маршрутом, випадкове переміщення між точками або перебування на одному місці.

Для реалізації руху маршрутом використовується список координатних точок (вейпойнтів), які задаються у вигляді дочірніх об'єктів на сцені. Вороги можуть циклічно переміщатися між цими точками, створюючи ефект патрулювання. Крім того, реалізована можливість зациклення маршруту, завдяки чому після досягнення останньої точки ворог повертається до першої, забезпечуючи безперервний рух.

У варіанті випадкового патрулювання ворог вибирає наступну точку з наявного списку випадковим чином, що створює більш непередбачувану та динамічну поведінку. Такий підхід ускладнює передбачення руху ворога гравцем, що підвищує загальний рівень складності гри.

Також передбачено варіант повної відсутності руху, коли ворог залишається на місці до моменту виявлення гравця. Цей режим особливо корисний для ворогів, які охороняють певну ділянку або очікують появи гравця в конкретній точці.

Під час перебування в одному з вищезазначених станів (руху або очікування) ворог постійно перевіряє навколишній простір на предмет наявності гравця. Для цього використовується відповідний алгоритм виявлення — наприклад, перевірка дистанції до гравця, напрямку руху та, за потреби, додаткових умов, таких як наявність перешкод. При виявленні гравця ворог переходить у стан переслідування або атаки, де змінюється як його анімація, так і логіка дій.

Такий підхід забезпечує гнучкість в налаштуванні різних типів ворогів із різною поведінкою, що сприяє розширенню ігрового досвіду та більшій різноманітності у проходженні.

3.3.3 Реалізація атаки

Поведінка нападу ворога активується, коли гравець потрапляє в зону дії, що визначена параметром дистанції атаки. Напад на гравця здійснюється через перевірку відстані між ворогом та гравцем. Якщо ця відстань менша або дорівнює заданому

радіусу атаки і гравець знаходиться в полі зору ворога, він зупиняється та переходить у бойовий режим.

Коли ворог переходить у бойовий режим, запускається відповідна анімація через систему Animator, яка активує тригер атаки. Після завершення анімації, за допомогою спеціальної події (animation event), викликається метод Hitscan() або MeleeDamage() для нанесення шкоди.

У методі Hitscan() (див. Лістинг 3.2) перевіряється, чи потрапив снаряд або атака в об'єкт з тегом "Player". Якщо так, викликається метод Damage() на компоненті PlayerStats на об'єкті гравця, і завдається шкода із заздалегідь визначеним значенням.

```
    void Hitscan()
    {
        Vector3 direction = CalculateSpread(transform.forward, spreadAmount);

        Ray ray = new Ray(firePoint.position, direction);

        RaycastHit hit;
        if (Physics.Raycast(ray, out hit, Mathf.Infinity))
        {
            TrailRenderer trail = Instantiate(bulletTrail, firePoint.position,
Quaternion.identity);
            StartCoroutine(SpawnTrail(trail, hit));

            CheckLayer(hit.collider.gameObject.layer, hit);

            if (hit.transform.gameObject.CompareTag("Player"))
            {

hit.transform.gameObject.GetComponent<PlayerStats>().Damage(damageAmount, false);
            }
        }
        else
        {
            if (hit.collider != null)
            {
                TrailRenderer trail = Instantiate(bulletTrail, firePoint.position,
Quaternion.identity);

                StartCoroutine(SpawnTrail(trail, ray.GetPoint(100)));
            }
        }

        objectSoundManager.PlaySound(fireClip, 0, 0, true, 0);
    }
```

Лістинг 3.2 Реалізація пострілів ворога.

Джерело: Автор

У методі MeleeDamage() (див. Лістинг 3.3) перевіряється, чи є гравець в радіусі атаки ворога. Якщо так, то завдається шкода із заздалегідь визначеним значенням.

```
    public void MeleeDamage()
    {
        if (enemyType == EnemyType.Melee && inRange)
        {

player.transform.gameObject.GetComponent<PlayerStats>().Damage(damageAmount, false);
        }
    }
```

Лістинг 3.3 Реалізація ударів ворога.

Джерело: Автор

До атак додано розкид (див. Лістинг 3.4), що дозволяє атакувати з випадковим відхиленням від ідеальної траєкторії. Це додає елемент непередбачуваності до бою, змушуючи гравця постійно реагувати на можливі варіації напрямку атак. Розкид визначається випадковим відхиленням від початкового напрямку в межах заданого кута.

```
private Vector3 CalculateSpread(Vector3 direction, float angle)
{
    // Calculate random angles within the spread range
    float spreadX = Random.Range(-angle, angle);
    float spreadY = Random.Range(-angle, angle);

    // Apply the spread to the direction
    direction = Quaternion.Euler(spreadX, spreadY, 0) * direction;

    return direction;
}
```

Лістинг 3.5 Реалізація розкиду куль.

Джерело: Автор

Частота атак базується на тривалості бойової анімації. Вона виконується щоразу, коли гравець залишається в межах радіусу атаки, а система анімації дозволяє повторити атаку після завершення попередньої.

Цей підхід дозволяє швидко запускати бойові дії без необхідності додаткової логіки затримок, але ставить технічні обмеження на тривалість анімаційних циклів. З належним налаштуванням тривалості анімації, можна уникнути надмірного навантаження на гравця.

Завдяки розкиду атак система стає більш динамічною та реалістичною, додаючи різноманіття в бою та змушуючи гравця постійно бути насторожі.

3.3.4 Реалізація отримання шкоди

У грі реалізована система отримання шкоди ворогами (див. Лістинг 3.6), яка включає використання щита та здоров'я для визначення впливу пошкоджень.

```
public virtual void Damage(float dmg, bool isHeadshot)
{
    float damage = Mathf.Abs(dmg);
    float oldDmg = damage;

    if (damage <= shield)
    {
        shield -= damage;
        if (shieldSlider != null)
shieldSlider.GetComponent<Animator>().Play("UIDamageShieldEnemy");
    }
    else
    {
        damage = damage - shield;
        shield = 0;
        health -= damage;

        if (shieldSlider != null && health >= 0)
shieldSlider.GetComponent<Animator>().Play("UIDamageShieldEnemy");
        if (healthSlider != null && health >= 0)
healthSlider.GetComponent<Animator>().Play("UIDamageEnemy");
    }

    events.OnDamaged.Invoke();
    UIEvents.onEnemyHit?.Invoke(isHeadshot);

    if (!showUI) return;
    GameObject popup = Instantiate(damagePopUp, transform.position,
Quaternion.identity);
    if (oldDmg / Mathf.FloorToInt(oldDmg) == 1)
        popup.transform.GetChild(0).GetComponent<TMP_Text>().text =
oldDmg.ToString("F0");
    else
        popup.transform.GetChild(0).GetComponent<TMP_Text>().text =
oldDmg.ToString("F1");
    float xRand = Random.Range(-xVariation, xVariation);
    popup.transform.position = popup.transform.position + new Vector3(xRand, 0, 0);
}
```

Лістинг 3.6 Реалізація отримання шкоди

Джерело: Автор

Процес обробки отриманої шкоди виглядає наступним чином:

- **Перевірка шкоди:**

Спочатку шкода перетворюється в позитивне значення за допомогою **Mathf.Abs(dmg)**, щоб уникнути негативних значень шкоди. Цей крок гарантує, що шкода завжди буде коректною.

- **Шкода по щиту:**

  Якщо значення отриманої шкоди менше або рівне залишковому щиту ворога, шкода буде поглинена щитом, і його значення зменшується. Якщо залишковий щит не вистачає для поглинання всієї шкоди, залишок шкоди переходить до здоров'я ворога.

- **Шкода по здоров'ю:**

  Якщо шкоди більше, ніж залишок щита, залишковий шкоди віднімається від здоров'я ворога, а щит знищується (встановлюється в 0). Якщо залишок здоров'я ворога стає меншим або рівним нулю, ворог вмирає.

- **Події та реакції**
  - Після того як шкода була нанесена, викликається подія **events.OnDamaged.Invoke()**.
  - Якщо шкода була завдана у голову (хедшот), викликається подія **UIEvents.onEnemyHit?.Invoke(isHeadshot)**, що активує специфічний ефект для хедшотів.

Висновки до розділу 3

У ході реалізації бойової поведінки ворога було впроваджено низку механік, які дозволяють створити динамічну ігрову ситуацію, що вимагає активної реакції з боку гравця. Основними елементами цієї поведінки стали виявлення гравця, перехід у бойовий режим, атака з анімаційною підтримкою та обробка нанесеної шкоди.

Виявлення гравця здійснюється через перевірку дистанції між ворогом та гравцем, а також наявності прямої видимості. Це дозволяє уникнути ситуацій, у яких ворог реагує

на гравця через стіни або інші перешкоди. Перехід у бойовий режим активує відповідну анімацію атаки, синхронізовану з механікою нанесення шкоди.

Важливою особливістю реалізації стало застосування розкиду під час атак. Цей підхід дозволяє уникнути максимальної точності ворога, додаючи випадковість у траєкторію пострілу або напрямок атаки. Таким чином, ворог не завжди влучає точно в ціль, навіть якщо гравець знаходиться в межах досяжності. Це створює більш природну та динамічну бойову ситуацію, у якій гравець має шанс уникнути шкоди завдяки активному маневруванню. Впровадження розкиду також підвищує загальну реалістичність боїв і робить ворога менш передбачуваним, що позитивно впливає на геймплейний досвід.

ЗАГАЛЬНІ ВИСНОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

У результаті виконання кваліфікаційної роботи на тему «Розробка комп'ютерної гри з елементами взаємодії з оточенням в Unity» були досягнуті поставлені мета і завдання, що дозволило вирішити важливу задачу – розробити комп'ютерну гру, яка поєднує функціональні можливості сучасного рушія Unity з технологіями інтерактивної взаємодії користувача з елементами віртуального середовища. Проведене дослідження дозволяє сформулювати наступні загальні висновки.

По-перше, у процесі виконання роботи було здійснено аналіз теоретичних засад розробки комп'ютерних ігор та їх ролі в сучасному цифровому суспільстві. Розглянуто основні принципи побудови ігрових світів, механіки взаємодії користувача з віртуальним середовищем та технологічні аспекти реалізації інтерактивних можливостей. Установлено, що на сучасному етапі розвитку комп'ютерних ігор особливо актуальними є аспекти підвищення реалізму, інтерактивності та емоційного залучення гравців.

По-друге, було проведено огляд і аналіз можливостей ігрового рушія Unity, що є одним із найпопулярніших інструментів для створення ігор різних жанрів. Встановлено, що Unity має розвинені засоби для створення тривимірного середовища, інтеграції фізичних законів, налаштування поведінки об'єктів, а також забезпечує гнучкі інструменти для написання сценаріїв та налаштування взаємодії з оточенням. Проаналізовано структуру проєктів у Unity, основні компоненти (GameObject, Rigidbody, Collider, скрипти), які були застосовані у практичній частині роботи.

По-третє, на основі аналізу предметної області була розроблена концепція гри, в якій передбачено різноманітні механіки взаємодії гравця з елементами середовища, такі як взаємодія з об'єктами, переміщення предметів, натискання кнопок, відкриття дверей тощо. Особливу увагу було приділено питанням створення реалістичного фізичного

оточення та впровадження механік, що дозволяють гравцеві відчувати вплив на середовище гри.

У практичній частині було безпосередньо реалізовано комп'ютерну гру, в якій успішно інтегровані розроблені механіки взаємодії. Створене тривимірне середовище включає інтерактивні об'єкти, можливість маніпуляції предметами, використання фізики для виконання дій над об'єктами та інших ефектів. Для забезпечення зазначених функціональних можливостей були створені та застосовані скрипти мовою C# із використанням API Unity. Розробка відзначається оптимізацією ігрового процесу з урахуванням продуктивності та зручності користувача.

В процесі розробки гри були застосовані принципи об'єктно-орієнтованого програмування, використано шаблони проектування, що дозволило зробити код гри більш структурованим, гнучким та масштабованим.

Таким чином, поставлена мета кваліфікаційної роботи була досягнута. Отримані результати мають практичне значення та можуть бути використані для подальшої розробки ігор із більш складною інтерактивністю, впровадження VR-елементів, розширення функціоналу проєкту з урахуванням новітніх технологічних тенденцій. Розробка показала перспективність використання Unity для створення ігор із підвищеною інтерактивністю, що дозволяє значно урізноманітнити ігровий процес та забезпечити кращу взаємодію користувача з віртуальним світом.

У подальшій діяльності доцільно поглибити вивчення методів створення більш складних сценаріїв взаємодії, застосування штучного інтелекту для об'єктів оточення, інтеграції мультиплеєрних можливостей, а також дослідити можливості адаптації гри під різні платформи (PC, мобільні пристрої, VR).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sketchfab. Character Y Bot [Електронний ресурс] – Режим доступу: https://sketchfab.com/3d-models/character-y-bot-de266ad4998447a89eb694c7cce5066d (дата звернення: 02.04.2025 р.)
2. Unity Technologies. Unity Manual [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/Manual/index.html (дата звернення: 02.04.2025 р.).
3. Unity Technologies. Unity Scripting API [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/ScriptReference/index.html (дата звернення: 02.04.2025 р.).
4. Blender Foundation. Blender Documentation [Електронний ресурс]. – Режим доступу: https://www.blender.org/documentation/ (дата звернення: 03.04.2025 р.).
5. Holistic3D. Learn Unity Engine [Електронний ресурс]. – Режим доступу: https://www.holistic3d.com (дата звернення: 03.04.2025 р.).
6. Microsoft. C# Programming Guide [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 03.04.2025 р.).
7. ShaderLab Documentation [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/Manual/SL-Shader.html (дата звернення: 03.04.2025 р.).
8. Post Processing Stack v2 Documentation [Електронний ресурс]. – Режим доступу: https://github.com/Unity-Technologies/PostProcessing (дата звернення: 05.04.2025 р.).
9. Digital Tutors. Unity Game Development Tutorials [Електронний ресурс]. – Режим доступу: https://www.digitaltutors.com/ (дата звернення: 05.04.2025 р.).
10. GameDev.net. Game Development Community [Електронний ресурс]. – Режим доступу: https://www.gamedev.net/ (дата звернення: 08.04.2025 р.).
11. Unity Technologies. Unity Learn [Електронний ресурс]. – Режим доступу: https://learn.unity.com/ (дата звернення: 08.04.2025 р.).
12. Udemy. Unity C# Online Course [Електронний ресурс]. – Режим доступу: https://www.udemy.com/course/unity-online-course/ (дата звернення: 09.04.2025 р.).
13. Stfalcon Academy. Розробка ігор на Unity [Електронний ресурс]. – Режим доступу: https://kids.stfalcon.com/courses/unity (дата звернення: 14.04.2025 р.).
14. ArtCraft CG School. Створення ігор на Unity [Електронний ресурс]. – Режим доступу: https://artcraft.net.ua/courses/stvorennya-igor-na-unity (дата звернення: 14.04.2025 р.).
15. ITC.ua. Вчимося створювати ігри на Unity та Unreal Engine: кращі курси з C# і C++ [Електронний ресурс]. – Режим доступу: https://itc.ua/ua/articles/vchymosya-

stvoryuvaty-igry-na-unity-ta-unreal-engine-krashhi-kursy-z-c-j-c/ (дата звернення: 14.04.2025 р.).
16. ITVDN. Новий відео курс Unity Стартовий - вивчай розробку ігор з нуля [Електронний ресурс]. – Режим доступу: https://itvdn.com/ua/news/article/new-course-unity-starter-u (дата звернення: 18.04.2025 р.).