

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Гра платформер в жанрі метроїдванія»

Виконав студент групи ІПЗ-20-4
ГАННУЩЕНКО Олександр Васильович

Керівник роботи:
ГОДЛЕВСЬКИЙ Юрій Олександрович

Рецензент:
ВЛАСЕНКО Олег Васильович

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Завідувач кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«26» лютого 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ГАННУЩЕНКО Олександр Васильович**

Керівник роботи: **ГОДЛЕВСЬКИЙ Юрій Олександрович**

Тема роботи: **«Гра платформер в жанрі Метроїдванія»**,

затверджена Наказом закладу вищої освіти від «23» лютого 2024 р., №74-с

Вихідні дані для роботи: **Об'єктом дослідження** є процес реалізації ігрового проєкту в контексті Unity. **Предметом дослідження** є розробка програмного коду для «Гри платформер в жанрі метроїдванія».

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Годлевський Ю.О.	26.02.2024р.	26.02.2024р.
2	Годлевський Ю.О.	26.02.2024р.	26.02.2024р.
3	Годлевський Ю.О.	26.02.2024р.	26.02.2024р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з додатку «Гра платформер в жанрі метроїдванія» та також пояснювальної записки. Пояснювальна записка містить 60 сторінок, 27 ілюстрацій та 1 таблицю.

Метою роботи є розробки є гра платформер в жанрі метроїдванія, яка включає головне меню, налаштування гри, систему збереження та завантаження гри; ігрові рівні з різноманітними ворогами; основні ігрові механіки такі як пересування, ривок, стрибок, переكات; бойову систему для персонажу; систему здоров'я; стейт машину для ворогів; адміністративну панель для налаштувань гри.

В роботі визначено основні завдання на розробку проекту, проаналізовано аналоги, поставлено завдання на розробку проекту, обгрунтовано вибір засобів та інструментів розробки.

Ключові слова: UNITY, ІГРОВИЙ ДОДАТОК, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, MICROSOFT VISUAL STUDIO, ZENJECT, PATTERN, CLASS, C# , МЕТРОІДВАНІЯ, STATEMACHINE

					ІПЗ.КР.Б – 121 – 24 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	Гра платформер в жанрі метроїдванія	Літ.	Арк.	Аркушів
Розроб.		Ганнущенко О.В						
Керівник		Годлевський Ю.О					3	82
Н. контр.		Тичина О.П				Житомирська політехніка, група ІПЗ-20-4		
Зав. каф.		Вакалюк Т.А						

ABSTRACT

The final bachelor's thesis consists of the application "Platformer game in the genre of metroidvania" and an explanatory note. The explanatory note contains 60 pages, 27 illustrations and 1 tables.

The purpose of the work is to develop a platformer game in the genre of metroidvania, which includes the main menu, game settings, game saving and loading system; game levels with various enemies; basic game mechanics such as movement, advancement, jumping, rolling; combat system for the character; health system; state machine for enemies; administrative panel for game settings.

The paper defines the main tasks for the project development, analyzes analogs, sets tasks for the project development, and justifies the choice of development tools and instruments.

Keywords: UNITY, GAME APPLICATION, SOFTWARE, MICROSOFT VISUAL STUDIO, ZENJECT, PATTERN, 3D, CLASS, C#, MANAGER, METROIDVANIA.

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ІЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		4

ЗМІСТ

РЕФЕРАТ.....	4
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ.....	9
1.1 Постановка задачі	9
1.2 Аналіз аналогів програмного продукту.....	12
1.3 Обґрунтування вибору інструментальних засобів	20
Висновки до першого розділу	24
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	25
2.1 Визначення варіантів використання та об'єктно орієнтованої та структури системи.....	25
2.2 Фреймворк Zenjест та його використання під час розробки гри	32
2.3 Основні етапи реалізації гри.....	34
2.3.1 Реалізація механік гравця	35
2.3.2 Реалізація системи здоров'я	42
2.3.2 Реалізація стейт машини	44
Висновки до другого розділу	48
РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ	50
3.1 Порядок встановлення та налаштування системи.....	50
3.2 Інтерфейс та порядок роботи з програмним додатком.....	50
3.3 Тестування роботи програмного додатку	56
Висновки до третього розділу	59
ВИСНОВКИ	60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	61
ДОДАТКИ.....	63

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		5

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ПЗ – програмне забезпечення.

AI – штучний ігровий інтелект.

ООП – одна з парадигм програмування, яка розглядає програму як множину «об'єктів», що взаємодіють між собою.

HUD - (Head-up-Display). -

SOLID – це набір принципів ООП для покращення дизайну та підтримання коду (Single Responsibility Principle, Open/Close Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle)

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		6

ВСТУП

Актуальність теми розробки проєкту «Гра платформер в жанрі метроїдванія» обумовлена низкою аспектів, які відображають сучасні тенденції та потреби в індустрії відеоігор. Метроїдванія залишається, популярним серед гравців завдяки своїм унікальним механікам та глибокому ігровому процесу. Ігри цього жанру поєднують різноманітні елементи як платформінг, динамічний бій, нелінійне прогресія тощо. Вивчення та вдосконалення цих механік є важливим напрямком у розробці сучасних ігор, що дозволяє створювати більш захоплюючий сетинг.

Розробка ігрового проєкту забезпечує цінний практичний досвід у вирішенні задач програмування, дизайну та тестування. В наслідок чого це не тільки сприяє розвитку професійних навичок, але й надає можливість застосувати теоретичні знання на практиці. Практичний досвід у розробці гри дозволяє краще зрозуміти процес створення ігор.

Таким чином, розробка проєкту «Гра платформер в жанрі метроїдванія» є актуальною темою як з точки зору програмування, застосування інформаційних технологій, а також з точки зору культурного впливу. Це дозволяє створювати цікаві ігрові продукти сприяючи розвитку професійних навичок.

Метою кваліфікаційної роботи є створення програмного коду, що включає реалізацію основних ігрових механік в жанрі метроїдванія.

Завдання кваліфікаційної роботи включає такі етапи:

1. Розробити функціональний ігровий проєкт, який демонструє всі основні механіки жанру метроїдванія.
2. Дослідження та реалізація найкращих практик ігрового дизайну для створення захоплюючого ігрового досвіду.
3. Створити інтерактивний та захоплюючий ігровий світ, включаючи різноманітні рівні, забезпечити баланс складності для персонажа.
4. Провести тестування гри для виявлення та виправлення помилок.
Оптимізувати продуктивність гри для забезпечення плавного ігрового процесу та створити документацію про процес розробки.

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ док.ум.	Підпис	Дата		7

Об’єктом дослідження є процес реалізації ігрового проєкту в контексті Unity.

Предметом дослідження є розробка програмного коду для «Гри платформер в жанрі метроїдванія». В середовищі програмування Microsoft Visual Studio на мові програмування C#, використовуючи ігровий рушій Unity.

Методи дослідження: для розробки було гри, було використано такі методи як використання фреймворку Zenject, використання патерну State, PoolOfObjects, Шаблоний метод та технології програмування C#, дизайн ігрових механік, анімації та тестування.

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		8

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Постановка задачі

Метою кваліфікаційної роботи є створення продукту, що включає реалізацію основних ігрових механік, дизайн рівнів, штучного інтелекту ворогів[1] та оптимізація продуктивності.

Основною метою розробки гри в жанрі метроїдванія є розробка механік. Виділяються такі основні механіки[2] як:

1. Пересування. Створення базового пересування, що включає : переكات для ухилення від атак ворогів, стрибок для подолання вертикальних перешкод та досягнення певних місць локацій, а також ривок, що дозволяє гравцеві долати короткі дистанції, що корисно для ухилень від атак або для швидкого пересування між елементами левел дизайну.

2. Система здоров'я. Гравець має обмежену кількість здоров'я, які можуть бути відновлені шляхом збирання певних предметів. Втрата здоров'я призведе до смерті і повернення на останню збережену позицію. Дана механіка забезпечить відчуття небезпеки й створює додаткові виклики у ігровому світі.

3. Шкода від зіткнень. Дана механіка пов'язана з системою здоров'я. У разі зіткнення з ворогом або в наслідок довгого падіння гравець отримує шкоду, що зменшує його здоров'я. Це змушує гравця бути обережним та уникати зайвих зіткнень.

4. Види зброї. Одним з ключових аспектів ігор даного жанру є різні види зброї, що дозволить гравцю обрати власний підхід в залежності від складності ворога і в тому ж випадку надає змогу виконувати як ближні атаки так і атаки на відстані.

5. Різновид ворогів. В іграх даного жанру є важливим мати різну кількість ворогів, що мають різну поведінку залежно від ситуації, що змушує гравця шукати підхід для усунення перешкоди.

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		9

6. Збереження прогресу[3]. Один з важливих аспектів є збереження прогресу у спеціальних місцях, що дозволяє уникнути втрати досягнень у разі поразки та повертатися до досліджень попередніх місць.

7. Навігація по рівнях. Можливість досліджувати різні рівні гри, приховані області, зустріч з неігровими персонажами та збирання корисних предметів.

У рамках кваліфікаційної роботи потрібно врахувати вимоги та розробити основні механіки гри, дизайн рівнів, користувацький інтерфейс, побудувати архітектуру класів. Для досягнення поставленої задачі потрібно виконати такі етапи:

Перший етап. Розробити основні механіки гри:

1. Розробити базове пересування персонажа, що включає перекати, стрибок, ривок .

2. Реалізувати систему здоров'я, яка забезпечує відновлення здоров'я шляхом збирання певних предметів, та повернення на останню збережену позицію у разі смерті.

3. Впровадити механіку шкоди від зіткнень з ворогами внаслідок довгого падіння.

4. Перевірити правильність роботи та взаємодії основних механік.

Другий етап. Створення бойової системи гравця:

1. Реалізувати логіку комбо атак.

2. Розробити два типи зброї.

3. Реалізувати перезарядження зброї.

4. Перевірити на зразкових ворогах

5. Протестувати зроблені етапи.

Третій етап. Створення системи ворогів та налаштування їх на сцені:

1. Розробити машину станів для ворога за допомогою патерна State, що допоможе контролювати та стани та задавати необхідну логіку.

2. Реалізувати машину станів для ближнього та дальнього типів ворогів.

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				10
Змн.	Арк.	№ докум.	Підпис	Дата		

3. Розробити логіку бойової системи для кожного з типів ворогів .
 4. Додати логіку здоров'я для ворогів, а також відображення отриманої шкоди.
 5. Під'єднати спрайтовані анімації та вказати умови переходу.
 6. Протестувати зроблені етапи.
- Четвертий етап. Створити UI – користувацький інтерфейс[4] для гри:
1. Розробити UI елементи для відображення здоров'я, зброї та важливої інформації
 2. Впровадити керування користувацьким інтерфейсом, забезпечуючи зручність використання.
 3. Провести тестування користувацького інтерфейсу на різних стадіях гри.
- П'ятий етап. Дизайн ігрових рівнів:
1. Створити загальну концепцію дизайну гри, включаючи стилістику та тематику.
 2. Розробити елементи рівнів[5], як платформи та інші елементи.
 3. Впровадити механіки взаємодії гравця з середовищем, як отримання шкоди від зіткнення, падіння.
 4. Перевірити локації та взаємодію з її елементами.
- Шостий етап. Інтеграція усіх компонентів:
1. Створити основну ігрову сцену, що об'єднає всі розроблені компоненти
 2. Створити ворогів ближнього та дального типу, налаштувати та розмістити місця патрулювання.
 3. Інтегрувати користувацький інтерфейс та персонажу до основної сцени .
 4. Реалізувати перехід між елементами інтерфейсу.
 5. Провести повне тестування гри для виявлення та виправлення помилок.

		О.В Ганнуценко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		11

1.2 Аналіз аналогів програмного продукту

Аналіз існуючих програмних продуктів у жанрі метроїдванія дозволяє виявити ключові механіки, перед тим як розпочати створювати ігровий додаток. Розглянемо детально кілька популярних ігор цього жанру, визначаючи їхні сильні та слабкі сторони.

«Hollow Knight» [6] є одним з найпопулярніших представників жанру метроїдванія, завдяки своїй шлибокій атмосфері та складному ігровому процесу.

Гравець занурюється у величезний підземний світ, досліджуючи його різноманітні локації і знаходячи навички для боротьби з ворогами, що допомагають подолати числені перешкоди. Гра пропонує гравцям не тільки захоплюючий ігровий процес, але і багату історію, що розкривається завдяки взаємодії з персонажами і оточенням, що відрізняє її від інших представників жанру. (рис.1.1).



Рисунок 1.1 – Малюнок перед початком гри «Hollow Knight»

Переваги механік гри:

Перше, що привертає увагу в Hollow Knight – це глибина та різноманітність бойової системи. Гравець має змогу використовувати різні атаки, стрибки, ухилення та спеціальні прийоми (рис. 1.2). Бойова система включає в себе основні удари, які можна комбінувати з іншими здібностями, такими як заклинання Vengeful Spirit, що випускає заряд енергії вперед, або Desolate Dive, яке завдає шкоди ворогам у зоні навколо героя під час стрибка. Ці заклинання дозволяють створювати різноманітні стратегії бою та адаптуватися до різних ситуацій.

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		12

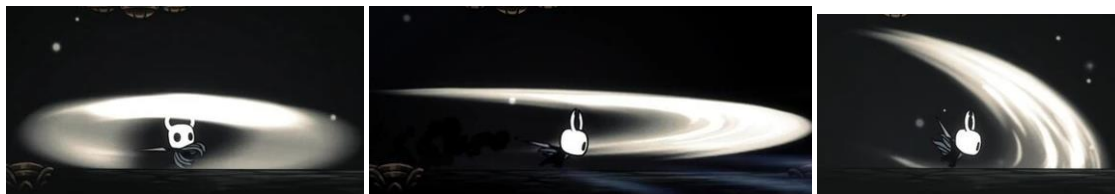


Рисунок 1.2 – Спеціальні прийоми «Hollow Knight»

Ще однією цікавою механікою є система заклинань(рис.1.3), яка дозволяє персоналізувати здібності персонажа. Можна знайти або придбати талісмани, які змінюють різні параметри, такі як швидкість атаки, захист та спеціальні здібності. Наприклад, заклинання "Шаманський камінь" збільшує шкоду, заподіяну заклинанням, а "швидкий удар" збільшує швидкість атаки. Це надає ігровому процесу глибину, підлаштовувати персонажа під його власний стиль гри і спонукає гравців експериментувати з різними конфігураціями атракціонів.



Рисунок 1.3 – Усі заклинання в «Hollow Knight»

Варто також відзначити механізм дослідження світу. Ігровий світ створений за принципом блочного дизайну, що означає поділ ігрового простору на окремі секції, кожна з яких має свій візуальний ряд, і це дозволяє створити враження безперервного і деталізованого світу. Ключовим фактором є те, що гравці можуть повертатися до місць, де вони вже були, з новими навичками та відкривати нові шляхи та секрети. Дизайн заохочує гравців постійно досліджувати та відкривати нові зони.

Також механізм випадкової генерації ворогів і предметів заслуговує на увагу. Вороги в грі мають різні атаки та дії, які гравець повинен адаптувати до кожного типу ворогів. Наприклад, деякі вороги мають броню, яку можна пробити лише певними атаками, або вони атакують у певному ритмі, який потрібно коригувати. Це зробить всі битви цікавими і напруженими. Гра також

ключає багато битв з босами, кожен з яких вимагає унікального підходу та стратегії.(рис 1.4)



Рисунок 1.4 – Приклад босів «Hollow Knight»

Недоліки в механіках гри:

Незважаючи на велику кількість позитивних моментів, є й деякі недоліки. Один з них полягає в тому, що складність гри висока. Це може засмутити недосвідчених гравців і може швидко призвести до втрати інтересу через постійні поразки. Через відсутність чітких інструкцій та підказок гравцям може бути важко зрозуміти, куди йти і що робити далі, що ускладнює проходження гри. Це може призвести до плутанини та уповільнення прогресу в грі.

Ще одним недоліком є специфічна однорідність противника на деяких ділянках. У грі присутні різні типи супротивників, але з часом їх дії можуть стати занадто передбачуваними. Це дозволяє знизити складність для досвідчених гравців, які швидко адаптуються до повторюваних атак. Деяким гравцям також може здатися, що система чарів дещо обмежена, оскільки деякі чари можуть бути значно сильнішими за інші. Це створює дисбаланс, змушуючи гравця вибирати лише найкращі чари та залишаючи інших гравців без нагляду.

Враховуючи ці аспекти, можна зробити висновок, що Hollow Knight потребує певного вдосконалення, щоб залучити більшу аудиторію, але також варто зазначити, що глибока механіка та захоплюючий геймплей роблять гру набагато простішою в освоєнні.

Наступна гра яка була проаналізована це «Ori»

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				14
Змн.	Арк.	№ докум.	Підпис	Дата		

«Ori and the Blind Forest» [7] – це метроїдванія, що виділяється своєю чудовою графікою, емоційним сюжетом та плавним геймплеєм. Гравець керує духом лісу на ім'я Орі, який повинен зібрати сили та врятувати свій дім від загибелі.

Метою гри «Ori the blind Forest» є забезпечення гравця захопливим досвідом дослідження та розвитку персонажа, типового для жанру метроїдванія. Це досягається через поєднання плавного управління, складної бойової системи та розгалуженої системи умінь, яка дозволяє гравцю постійно вдосконалювати свого персонажа, відкривати нові здібності та досліджувати нові території.
(рис.1.5)



Рисунок 1.5 – Малюнок гри «Ori the Blind Forest»

Переваги механік гри:

Однією з ключових переваг Ori the Blind Forest є її плавна і динамічна система управління. Орі володіє високою мобільністю, що дозволяє йому здійснювати подвійні стрибки, лазити по стінах і плавати. Ця мобільність робить геймплей захопливим і різноманітним, дозволяючи гравцеві досліджувати світ гри у своєму темпі. Система умінь дозволяє гравцеві поступово відкривати нові здібності, такі як Bash (використання ворогів або снарядів для стрибка) або Stomp (потужний удар по землі), що додає глибини ігровому процесу.

		О.В. Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				15
Змн.	Арк.	№ докум.	Підпис	Дата		



а)

б)

Рисунок 1.6 – а) спеціальна здібність Bash, б) стани переміщення в грі «Ori the Blind Forest»

Різноманітні вороги в «Ori the Blind Forest» також є важливим елементом гри. Ворогів у грі можна поділити на кілька типів, кожен з яких має унікальні патерни поведінки та атаки. Наприклад, деякі вороги атакують з відстані, використовуючи снаряди, тоді як інші намагаються підійти ближче і завдати шкоди в ближньому бою. Є також вороги, які здатні літати, що додає ще більше різноманітності та викликів у боях. Це змушує гравця адаптуватися до різних ситуацій та використовувати різні стратегії для перемоги над ворогами (рис. 1.7).

Hopper		Sunken Glades, Spirit Caverns, Hollow Grove, Black Root Burrows, Thornfelt Swamp, Ginso Tree, Misty Woods
Darkwing		Valley of the Wind, Sorrow Pass, Misty Woods, Hollow Grove, Black Root Burrows
Crawler		Sunken Glades, Spirit Caverns, Black Root Burrows

Рисунок 1.7 – Приклад ворогів.

Механіка збирання енергії та збереження прогресу також є важливою складовою гри. Орі може збирати енергію з різних джерел, що дозволяє використовувати спеціальні прийоми та створювати точки збереження у будь-якому місці гри. Це надає гравцю можливість стратегічно планувати свої дії та забезпечує гнучкість у збереженні прогресу. Така система збереження є

важливою для жанру метроїдванія, оскільки вона дозволяє гравцям більш стратегічно підходити до управління ресурсами та планувати свої дії.

Недоліки механік гри:

Лінійний сюжет є одним із основних недоліків Ori the Blind Forest. Це обмежує можливості повторного проходження гри. Гравці, які шукають більше свободи дій і численних варіантів розвитку подій, можуть відчути певний дискомфорт через лінійність сюжету. Лінійність може знижувати інтерес до повторного проходження гри, що є важливим аспектом для багатьох шанувальників жанру метроїдванія.

Хоча система Soul Link, яка дозволяє гравцям створювати контрольні точки в будь-якому місці, є інноваційною, вона також може бути недоліком водночас. Якщо гравець не створює контрольні точки достатньо часто, це може призвести до втрати значного прогресу після смерті.

Гра не завжди чітко пояснює, як працюють певні механіки або що саме потрібно робити в певних ситуаціях, що може викликати у гравців розгубленість та фрустрацію.

Хоча керування у грі зазвичай дуже плавне, деякі гравці відзначають, що в певних ситуаціях воно може бути недостатньо точним, що призводить до незручностей, особливо під час складних платформених секцій.

«Blasphemous» [8] – це метроїдванія, яка вирізняється своєю похмурою атмосферою, глибокою релігійною символікою та складними бойовими механіками

Метою " Blasphemous" є проходження серії складних рівнів, кожен з яких наповнений ворогами, пастками та головоломками. Гравець повинен використовувати свої бойові навички та магічні здібності, щоб перемогти могутніх босів та знайти реліквії, які допоможуть йому вдосконалити свого персонажа і досягти кінцевої мети гри – розкрити таємниці Cvstodia та здолати потужних супротивників. (рис.1.8).

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				17
Змн.	Арк.	№ докум.	Підпис	Дата		



Рисунок 1.8 – Малюнок гри «Blasphemous»

Переваги механік гри:

Однією з основних переваг Blasphemous є її бойова система, яка поєднує швидкі атаки, ухилення та використання спеціальних прийомів. Гравець має доступ до різних типів зброї, кожна з яких має свої унікальні властивості та стилі бою. Наприклад, меч «The Mea Culpa» може бути покращений завдяки сльозам спокути, щоб завдавати більше шкоди або відкривати нові комбінації атак рис(1.9). Така різноманітність зброї та можливість її вдосконалення забезпечують глибину бойового процесу та стимулюють гравця експериментувати з різними стилями гри.



Рисунок 1.9 – Ігрова валюта «Blasphemous»

Унікальною особливістю гри є система покарань за смерть. Коли гравець помирає, він залишає за собою слід провини, який знижує максимальну кількість його мани. Це додає додаткового виклику та стимулює гравця бути обережним під час боїв та досліджень. Щоб позбутися покарання, гравець повинен повернутися до місця своєї смерті і зібрати залишену провину, що робить кожне повернення до минулого місця події напруженим і ризикованим.

		О.В. Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				18
Змн.	Арк.	№ докум.	Підпис	Дата		

Світ Blasphemous складається з багатьох різних зон, кожна з яких має свій унікальний дизайн, ворогів та головоломки. Гравець поступово відкриває нові здібності, які дозволяють йому досліджувати раніше недоступні області. Наприклад, здібності, що дозволяють проходити крізь вогонь або долати великі відстані за допомогою подвоєння стрибка, відкривають нові можливості для дослідження світу. Це стимулює гравця постійно повертатися до вже досліджених зон, щоб знайти нові секрети та бонуси. Гра заохочує ретельне дослідження та експерименти, що забезпечує постійний інтерес і мотивацію для продовження гри.(рис 1.10)



Рисунок 1.10 – Відкриття нових спеціальних можливостей в «Blasphemous»

Недоліки механік гри:

Потенційним недоліком є незрозуміла система карт. Карта не завжди дає чіткі підказки щодо наступного місця призначення або важливих точок інтересу. Це може призвести до зайвих блудів по локаціях. Відсутність деталей на карті може не показувати всі доступні секрети або важливі деталі, які можуть допомогти у просуванні гри. Це може ускладнювати процес дослідження і вимагати від гравця більше часу та зусиль для розуміння, куди йти далі(рис. 1.11).

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		19

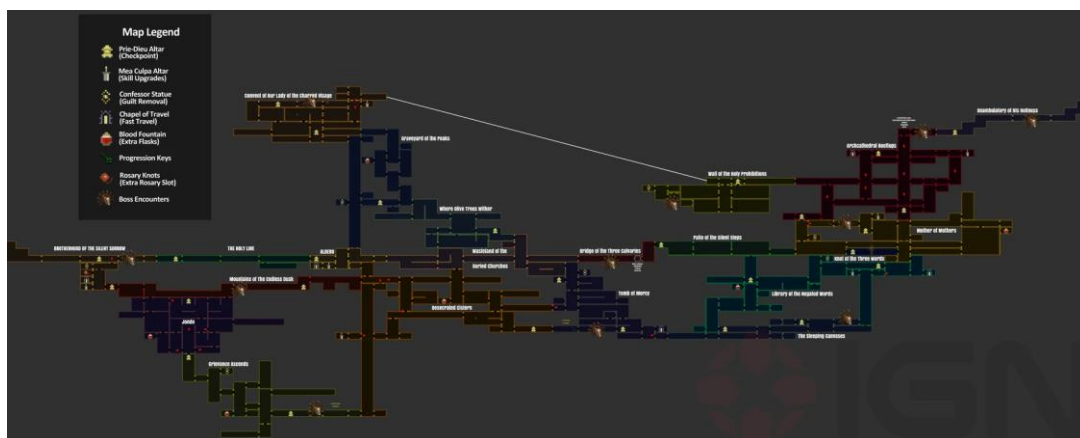


Рисунок 1.11 – Карта «Blasphemous»

Сюжет та лор гри також мають свої недоліки. Історія гри подається через випадкові зустрічі, діалоги та предмети, що можуть бути легко пропущені гравцем. Це робить сюжет важким для розуміння без додаткових зусиль. Деякі аспекти сюжету і світу гри можуть здаватися незрозумілими без глибокого занурення в лор, що може вимагати читання додаткових матеріалів або дослідження. Повторюваність ворогів і завдань також може знижувати інтерес до гри. Деякі вороги мають схожі атаки та тактики, що може зробити бої з ними менш цікавими після декількох годин гри. Побічні завдання можуть здаватися однотипними і не приносити великої різноманітності у геймплей. Це може знижувати мотивацію гравця до виконання додаткових місій та завдань.

Також технічні проблеми, з якими стикаються гравці, також додають недоліків гри. У перші тижні після релізу деякі гравці стикалися з багами, такими як збої при збереженні гри, візуальні глюки або зависання. Гра може мати проблеми з продуктивністю на різних платформах, що може вплинути на плавність геймплею. Це може створювати додаткові труднощі для гравців та негативно впливати на загальний ігровий досвід.

1.3 Обґрунтування вибору інструментальних засобів

Unity[9] є одним з найпопулярніших ігрових рушіїв для розробки ігор, який використовується як професійними розробниками, так і інді-розробниками. Даний рушій підтримує широкий спектр платформ, включаючи Windows, macOS, Android, iOS, Linux, а також консолі (Playstation, Xbox, Nintendo Switch) і веб платформи. Саме тому це робить рушій універсальним, знизуючи поріг

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				20
Змн.	Арк.	№ докум.	Підпис	Дата		

входження від розробників початківців до професійної команди розробників.

Переваги Unity:

Unity має одну з головних переваг, а саме активну спільноту розробників, яка пропонує численні ресурси, такі як офіційні та неофіційні уроки, форуми, документацію та відео на YouTube. Це значно полегшує процес навчання та вирішення технічних питань. Unity надає інтуїтивно зрозумілий графічний інтерфейс користувача, який дозволяє легко створювати ігрові об'єкти, налаштовувати сцени та компоненти, що знижує бар'єр входу для новачків і дозволяє швидко прототипувати ідеї.

Unity також пропонує широкий спектр 2D та 3D проєктів, включаючи фізичні симуляції, систему часток, анімацію, обробку звуку, мережеві функції тощо. Також Unity безкоштовний і має всі функціональні можливості, необхідні для розробки та випуску гри. Це важливо для інди-розробників та студентів, які не мають певного бюджету.

Однак Unity має свої недоліки. Він підходить для більшості ігрових проєктів, але якщо ви розробляєте дуже велику і складну гру з високоякісною графікою, продуктивність двигуна може бути обмежена. У таких випадках можуть виникнути проблеми з оптимізацією та продуктивністю. Unity доступний у безкоштовній версії, але комерційні проєкти з високим рівнем доходу повинні придбати ліцензію Unity Pro. Це дуже дорого і може стати величезним фінансовим тягарем для невеликих студій та незалежних розробників. Через велику кількість функцій та універсальність проєкти Unity можуть вимагати значної оптимізації, особливо для мобільних платформ, і можуть вимагати від розробників додаткових зусиль для забезпечення стабільної роботи гри на різних пристроях.

Порівняння Unity з іншими рушіями:

Якщо порівнювати Unity з іншими рушіями, то варто відзначити, що Unreal Engine є сильним конкурентом, особливо в області розробки ігор з високоякісною графікою. Unreal Engine використовує мову програмування C++ та власну мову blueprint для візуального програмування, надаючи чудові графічні можливості та

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		21

інструменти, але через складність C++ процес навчання стає складнішим.

Будучи безкоштовним рушієм з відкритим вихідним кодом, Godot також підтримує розробку 2D і 3D ігор і використовує свою власну скриптову мову GDScript, який простий в освоєнні. Хоча Godot відомий своєю простотою та швидкістю розробки, він не обов'язково має всі функції та підтримку, які пропонує Unity. Студія GameMaker спеціалізується на розробці 2D-ігор і використовує мову програмування gml, що робить його ідеальним для 2D-проектів завдяки своїй простоті та зручності використання, але він обмежує можливості 3D-розробки і не є кросплатформенним на тому ж рівні, що і Unity.

Вибір Microsoft Visual Studio як середовища розробки:

Microsoft Visual Studio[10] обрано як основне середовище розробки для проектів на Unity з кількох причин. Ця платформа забезпечує розробникам потужні інструменти для написання, налагодження та тестування коду, що суттєво підвищує продуктивність та якість розробки програмного забезпечення. Та має інтеграцією з Unity зповною підтримкою мови C#, Visual Studio стає оптимальним вибором для створення ігор, забезпечуючи усіма необхідними інструментами для створення якісного коду.

Переваги Microsoft Visual Studio:

Однією з головних переваг Visual Studio є його інтеграція з Unity. Середовище розробки підтримує основну мову програмування Unity, що забезпечує такими можливостями як, автозаповнення коду, підсвічування синтаксису та інструменти налагодження, що робить процес написання коду та пошуку помилок більш ефективно.

Ще однією важливою перевагою є інтеграція з іншими інструментами Microsoft, такими як Azure та GitHub, забезпечує легке керування проектами, збереження версій коду та в комплексних проектах забезпечує співпрацю з іншими членами команди.

Visual Studio також відома своєю потужною системою розширень. Розробники можуть додавати різноманітні плагіни та розширення, що покращують функціональність середовища розробки та дозволяють

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				22
Змн.	Арк.	№ докум.	Підпис	Дата		

налаштовувати його під свої потреби. Це робить Visual Studio гнучким інструментом, який може бути адаптований для будь-якого проекту.

Ще однією важливою перевагою Visual Studio є його інтеграція з іншими інструментами Microsoft, такими як Azure, GitHub та інші. Це дозволяє розробникам легко керувати своїми проектами, зберігати версії коду та співпрацювати з іншими членами команди.

Одна незважаючи на численні переваги, Visual Studio має низку недоліків. Перш за все це вимогливість до ресурсів комп'ютера, середовище розробки може споживати значну кількість оперативної пам'яті та процесорного часу, особливо при роботі з великими проєктами або використанні великої кількості розширень.

Порівняння Visual Studio з іншими середовищами розробки:

Серед основних конкурентів Visual Studio можна виділити JetBrains Rider. Цей інструмент також є популярним середовищем розробки для C#, має швидкий інтерфейс та потужні інструменти для кодування. Проте, Rider є комерційним продуктом з високою вартістю, що може бути перешкодою для незалежних розробників або невеликих студій.

Visual Studio має безкоштовну версію (Visual Studio Community), яка надає більшість необхідних функцій для розробки ігор на Unity. Це робить його більш доступним для широкого кола розробників. Крім того, Visual Studio має більш тісну інтеграцію з Unity, що забезпечує кращу підтримку та зручність у використанні.

Іншою альтернативою є Visual Studio Code, легша версія Visual Studio, яка також підтримує розширення та має гарну інтеграцію з Unity через відповідні плагіни. Visual Studio Code менш вимогливе до ресурсів, але може не мати всіх можливостей і потужних інструментів для налагодження, які доступні у повній версії Visual Studio.

Ще однією альтернативою є MonoDevelop, яке довгий час було основним середовищем розробки для Unity на macOS і Linux. MonoDevelop менш потужне порівняно з Visual Studio і має менш інтуїтивний інтерфейс, але воно є безкоштовним і також підтримує роботу з C#.

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		23

Microsoft Visual Studio, є обґрунтованим завдяки його потужним інструментам для написання та налагодження коду, гнучкості та інтеграції з іншими інструментами розробки. Незважаючи на деякі недоліки, Visual Studio залишається одним з найкращих інструментів для розробки ігор.

Висновки до першого розділу

У першому розділі було поставлено завдання та визначено основні механіки гри в жанрі метроїдванія: пересування персонажа, систему здоров'я, шкоду від зіткнень, різні види зброї, ворогів, збереження прогресу та навігацію по рівнях. Мета роботи полягає у створенні продукту, що включає реалізацію цих механік, дизайн рівнів, штучний інтелект ворогів та оптимізацію продуктивності.

Аналіз трьох відомих ігор жанру метроїдванія («Hollow Knight», «Ori and the Blind Forest», «Blasphemous») дозволив виявити їхні сильні та слабкі сторони. У «Hollow Knight» акцент зроблено на глибокій бойовій системі та системі заклинань. В «Ori and the Blind Forest» особлива увага приділена плавній системі управління та мобільності персонажа. «Blasphemous» вирізняється системою покарань за смерть та детальним дизайном.

Обґрунтовано вибір Unity як основного рушія для розробки гри, зважаючи на його широкі можливості для створення 2D і 3D ігор, підтримку багатьох платформ та активну спільноту розробників.

Також обґрунтовано вибір Microsoft Visual Studio як середовища розробки, завдяки його потужним інструментам для написання, налагодження та тестування коду, інтеграції з Unity

		О.В. Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		24

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Визначення варіантів використання та об'єктно орієнтованої та структури системи

У цій частині кваліфікаційної роботи розглядається аналіз вимог до програмного продукту з акцентом на ідентифікацію функціональних та нефункціональних вимог. Він також описує різні варіанти використання. Для забезпечення чіткого розуміння структури системи додається діаграма сутностей.

Аналіз вимог передбачає ретельне вивчення потреб та очікувань користувачів щодо функціональності та характеристик гри платформера в жанрі метроїдванія. Основною метою аналізу вимог є з'ясування, того що потрібно реалізувати в додатку. Додатково були зібрані функціональні та не функціональні вимоги.

1. Функціональні вимоги:
2. Базове переміщення
3. Перекат
4. Стрибок
5. Ривок
6. Система здоров'я
7. Відображення здоров'я
8. Використання зброї
9. Перезарядка зброї.
10. Постріл
11. Отримання шкоди
12. Зберігання процесу

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		25

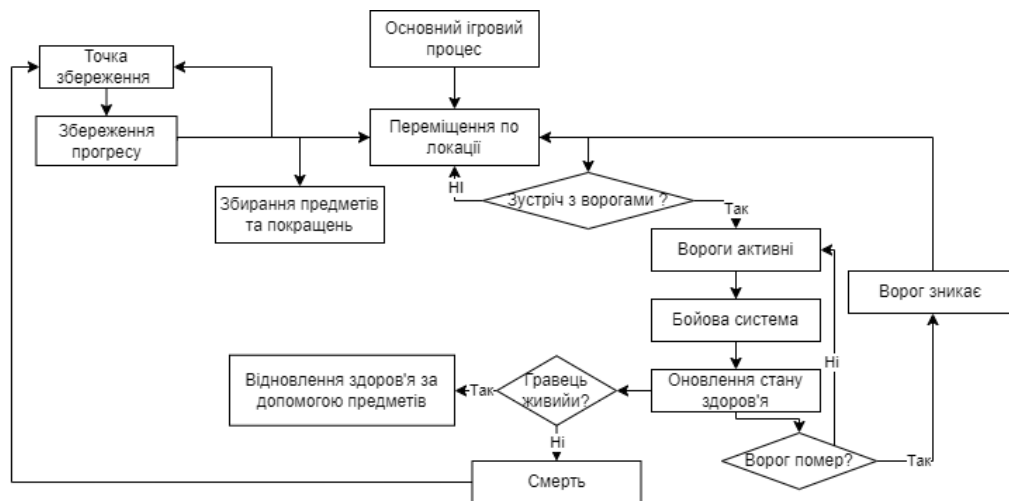


Рисунок 2.1 – Діаграма ігрового циклу

Перед тим як розпочати досліджувати ігровий процес, необхідно запустити клієнт гри та в головному меню натиснути кнопку нова гра, після чого відбудеться перехід з головного меню до гри.

На діаграмі ігрового циклу (рис. 2.1) показано сам цикл гри. Спочатку користувач має доступ до переміщення по локації та досліджувати рівні гри. Під час дослідження локації гравець може зустріти точки збереження, де можна зберегти свій прогрес. Це є важливою частиною гри, оскільки збереження прогресу дозволяє гравцеві повернутися до гри з того місця, де він залишився, навіть після загибелі персонажа.

Після збереження гравець продовжує дослідження локації, збираючи предмети. У процесі переміщення гравець може зустрітися з ворогами. Якщо відбувається зустріч з ворогами, то активізується бойова система. Під час бойової системи гравець бориться з ворогами, використовуючи свою зброю. Бойова система включає різні стратегії, що залежить від кількості ворогів та типів. Під час битви гравець наносить або отримує шкоду, в наслідок чого відбувається оновлення стану здоров'я гравця. В разі якщо гравець залишається живим після битви, або гравець втік, він продовжує переміщення по локації. У випадку загибелі гравця, гра завершується й починається з останнього збереження. Якщо ж ворог помирає під час битви, він зникає з локації, даючи гравцеві можливість продовжити свій шлях.

Також користувач має можливість відновлювати своє здоров'я за

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		26

допомогою предметів, зібраних під час переміщення по локації. Це будуть лікувальні предмети, які відновлюють частину або все здоров'я персонажу. Відновлення здоров'я дозволяє гравцеві бути готовим до наступних зіткнень з ворогами і продовжувати свій шлях.

Таким чином, ігровий цикл включає в себе постійне переміщення по локаціях, збір предметів, збереження прогресу, участь у битвах з ворогами, відновленням здоров'я та можливою загибеллю як гравця так і ворогів. Це дозволяє створити захоплюючий і динамічний процес гри.

Діаграма варіантів використання (Use Case Diagram) (рис. 2.2). Вона вказує на те, які дії гравець може робити під час усієї гри.

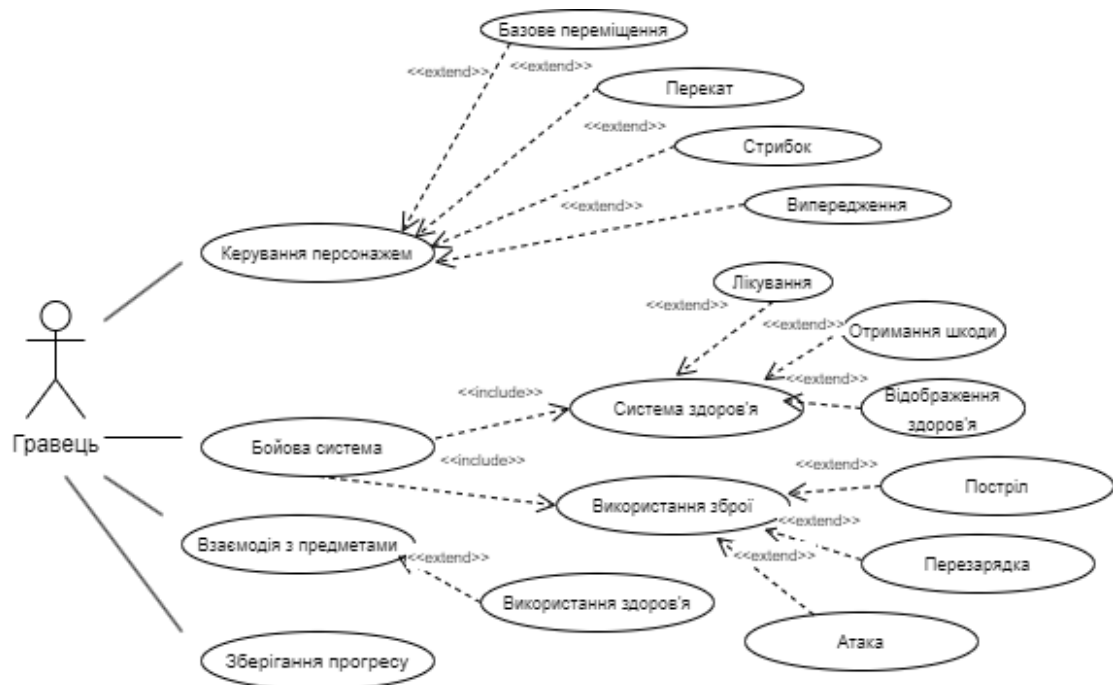
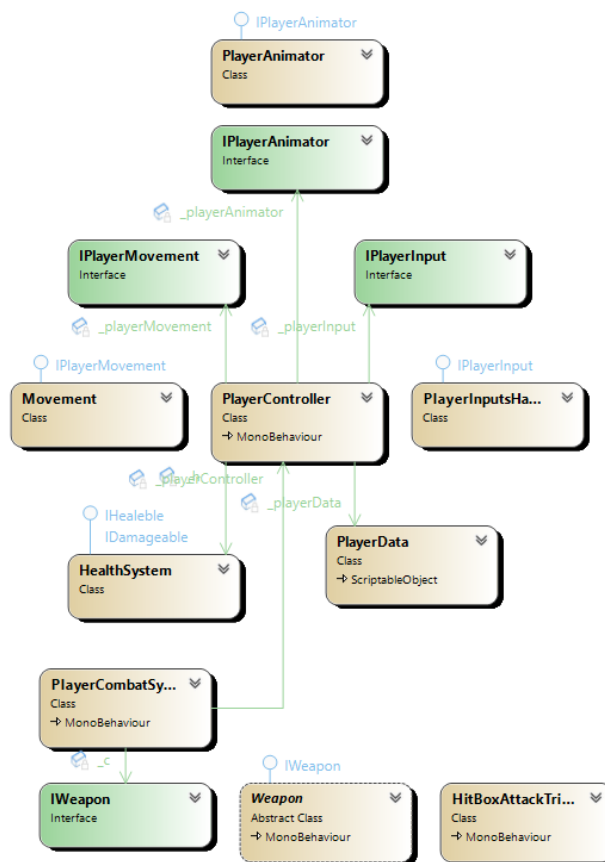


Рисунок 2.2 – Діаграма варіантів використання

Основними механіками є переміщення, що включає в себе ривок, стрибок та перекач, які можуть бути використаними для ухилення від атак. Збереження прогресу дозволяє гравцеві фіксувати досягнутий етап гри в спеціальних точках збереження, щоб потім продовжити гру з цього місця. Збирання предметів є важливою частиною гри, оскільки предмети надають гравцеві нові можливості та покращення, що допомагають у подальших пригодах. Бойова система включає арсенал з двох типів зброї, що дозволяє гравцеві вибирати оптимальний варіант для кожної ситуації, а також комбо атаки для ближнього бою. Оновлення

Також були створені діаграми класів. Діаграми класів використовуються для демонстрації мінливості, успадкування та зв'язків між різними класами.



Діаграма допомагає візуалізувати структуру системи та показує, як різні класи взаємодіють один з одним. На діаграмі (рис. 2.3) показано діаграму класу, де відображено взаємозв'язки один з одним.

1. `PlayerController` – є головним класом, який координує дії гравця. Що по собі діє як передавач між іншими класами, робить запуск методів в потрібні моменти, наприклад такі як пересування, стрибок, ривок, переكات чи взаємодія зі здоров'ям гравця.

		О.В Ганнущенко			ІІЗ.КР.Б – 121 – 24 – ІЗ	Арк.
		Ю.О. Годлевський				28
Змн.	Арк.	№ докум.	Підпис	Дата		

2. `PlayerInputsHandler` – є відповідальним за обробку вводу від гравця. Він реалізує інтерфейс `IplayerInput` і забезпечує отримання вводу руху та також обробляє зміни напрямку спрайту гравця залежно від напрямку вводу.

3. `PlayerAnimator` – відповідає за управління анімаціями та використовує компонент `Animator` для зміни станів, приймаючи булеві значення від компонентів гри і передає їх в аніматор `Unity`, забезпечуючи відповідність між діями гравця і відображенням цих дій на екрані.

4. `PlayerData` – зберігає налаштування гравця, такі як швидкість руху, сила стрибка, тривалість та сила ривок, параметри здоров'я та інші важливі характеристики. Це дозволяє налаштувати поведінку гравця відповідно до його можливостей і параметрів, визначених в грі.

5. `Movement` – відповідає за пересування гравця, та реалізує інтерфейс `IPlayerMovement`. Він реалізує логіку для переміщення, стрибків, випереджень, перекатів, перевірок, а також оброблює гравітацію і зміни позиції гравця.

6. `PlayerCombatSystem` – відповідає за бойову систему. Він взаємодіє з поточною зброєю та також забезпечує зміну поточної, дозволяючи виконувати атаки в бік ворогів на основі вводу.

7. `HealthSystem` – є універсальною системою управління здоров'я, яка може бути використана як гравцем, так і ворогами. З боку гравця, може відновлювати своє здоров'я та отримувати шкоду. Після кожної зміни здоров'я викликається подія зміни здоров'я що дозволяє оновлювати відображення здоров'я на екрані.

Наступною є діаграма класів ворогів (рис. 2.8), в ній також вказана взаємодія між класами, а також показана реалізація `StateMachine`[11].

Цей патерн часто використовується для реалізації стейт машин, де поведінка об'єкта змінюється динамічно в залежності від його стану, забезпечуючи більш чистий і підтримуваний код шляхом ізоляції логіки станів в окремих класах.

Використовуючи фабричний метод[12], для створення екземплярів станів дозволяє відокремити логіку створення станів від основної логіки машини станів, спрощуючи додавання нових станів і зміну існуючих.

Шаблонний метод[13] – це патерн проектування, який визначає основну структуру алгоритму в методі, дозволяючи підкласам реалізувати деякі з його кроків. Іншими словами, він надає загальну основу алгоритму в базовому класі, а конкретні кроки цього алгоритму деталізуються в похідних класах.

Діаграма наочно показує структуру взаємозв'язків між станами і переходами, що полегшує розуміння логіки роботи ворогів, що дозволяє краще планувати і реалізовувати нові функції.

На основі створеної діаграми розгортання для Unity проекту (рис. 2.5), можна побачити ключові компоненти, такі як Unity Editor та локальний комп'ютер, де проект буде розгорнуто.

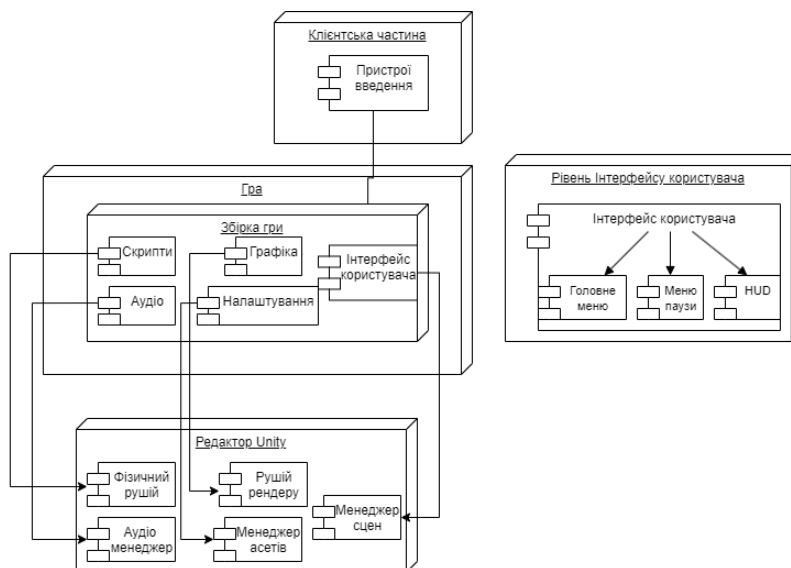


Рисунок 2.5 – Діаграма розгортання

Діаграма розгортання демонструє як різні компоненти проекту Unity взаємодіють один з одним на різних рівнях, починаючи від пристроїв введення до білду гри та компонентів Unity. Дана діаграма складається з трьох частин, такі

як клієнтська частина, гра та редактор Unity. На клієнтському рівні знаходяться пристрої введення такі як, клавіатура, миша. Рівень гри включає основні модулі, які забезпечують функціонування гри. Збірка гри містить усі необхідні компоненти для ігрового процесу. Редактор Unity використовується для розробки гри, створення оточення, налаштування фізичних і рендеринг компонентів, а також налаштування звукових ефектів. Рівень інтерфейсу користувача включає в себе головне меню, меню паузи та HUD (Head-up-Display), які відображаються за допомогою модуля інтерфейсу користувача у збірці гри і взаємодіють з пристроями вводу для отримання команд від гравця.

2.2 Фреймворк Zenject та його використання під час розробки гри

Zenject[14] є потужним фреймворком ін'єкції залежностей, спеціально розробленим для середовища Unity. Цей інструмент дозволяє розробникам створювати більш модульні, тестовані та масштабовані проекти, значно полегшуючи управління залежностями між різними компонентами гри.

Основні концепції Zenject включають:

- Контейнер (Container)
- Прив'язки (Bindings)
- Інсталери (Installers)

Контейнер є центральним елементом Zenject. Він відповідає за реєстрацію типів, створення екземплярів об'єктів, вирішення залежностей та керування їх життєвим циклом. Контейнер служить своєрідним сховищем, де зберігаються всі залежності та правила їх створення.

Ключові компоненти Unity для роботи з Zenject:

- SceneContext
- MonoInstaller

SceneContext - це компонент Unity, який ініціалізує та керує контейнером Zenject для конкретної сцени. MonoInstaller - спеціальний тип інсталера, який може бути доданий до GameObject у сцені, дозволяючи налаштовувати залежності безпосередньо з інспектора Unity.

Zenject підтримує кілька методів ін'єкції залежностей:

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		32

- Конструкторна ін'єкція
- Ін'єкція в поле
- Ін'єкція методом

Використання Zenject надає ряд суттєвих переваг:

- Зменшення зв'язаності між компонентами
- Покращення тестованості коду
- Спрощення рефакторингу та підтримки
- Централізоване управління залежностями
- Підтримка модульної архітектури

Процес налаштування проекту з Zenject включає імпорт фреймворку, додавання SceneContext до сцен, створення та налаштування інсталерів, конфігурацію залежностей через прив'язки та використання ін'єкції в класах.

Zenject надає потужний інструментарій для управління залежностями в проектах Unity, дозволяючи створювати більш гнучкі, масштабовані та легко підтримувані ігрові проекти. Впровадження Zenject у робочий процес розробки може значно підвищити якість коду та ефективність розробки в довгостроковій перспективі. Цей фреймворк особливо корисний у великих проектах, де управління залежностями стає складним завданням. Він допомагає дотримуватися принципів SOLID, особливо принципу інверсії залежностей, що призводить до створення більш модульного та розширюваного коду.

Одним із прикладів використання Zenject є, впровадження залежностей гравця та створення його на сцені. Для того щоб це зробити, необхідно створити клас який наслідується від MonoInstaller. Безпосередньо в методі InstalBindings ми впроваджуємо залежності гравця. Для того щоб створити гравця на сцені, необхідно вказати префаб та точку появи гравця. Префаб – це попердньо налаштований об'єкт Unity, який зберігається як шаблон для повторного використання.

```
public class PlayerInstaller : MonoInstaller
{
    [Header("Preferences")]
    [SerializeField] private GameObject _playerPrefab;
    [SerializeField] private Transform _spawnPosition;
```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		33

```

public override void InstallBindings()
{
    var player = Container.
        InstantiatePrefabForComponent<PlayerController>(
            _playerPrefab,
            _spawnPosition.position,
            Quaternion.identity,
            null);

    Container.
        Bind<PlayerController>().
        FromInstance(player).
        AsSingle();
}
}

```

Для отримання реалізації гравця, вороги мають спеціальний метод з атрибутом Inject, який дозволяє отримати його реалізацію.

```

[Inject]
private void Construct(PlayerController player)
{
    _player = player;
}

```

Таким чином класи, що отримують залежність через Zenject, мають доступ до класів без необхідності створювати їх самостійно. Що в свою чергу позбавляє жорстких посилань між класами.

2.3 Основні етапи реалізації гри

Починаючи процес програмування на Unity, важливо мати чітке уявлення про концепцію гри, її функції та основні характеристики. Програміст повинен розуміти, як ігрові механіки будуть реалізовані та які компоненти необхідні для їх впровадження. Наприклад, якщо гра містить складну фізику або взаємодію об'єктів, потрібно визначити, які конкретні системи Unity, такі як Rigidbody[15] або Collider[16], будуть використовуватися для досягнення необхідної функціональності.

Для створення та інтеграції ігрових механік у Unity можна скористатися різноманітними засобами і техніками. Unity використовує мову програмування C#, яка дозволяє створювати скрипти для контролю поведінки ігрових об'єктів. Процес починається з організації проекту[17], що включає створення структурованих папок для зберігання скриптів, ігрових моделей, сцен і плагінів. Якщо необхідні додаткові плагіни чи бібліотеки, їх потрібно завантажити та додати до проекту. Потім скрипти додаються до відповідних ігрових об'єктів

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		34

через інтерфейс Unity Editor. Це включає не лише додавання скриптів, але й налаштування всіх необхідних компонентів, таких як Rigidbody для фізики або Animator для анімації, і конфігурацію їхніх параметрів.

Ключовим аспектом роботи з Unity є забезпечення коректної роботи всіх налаштованих компонентів. Це означає, що після створення скриптів та їх інтеграції з ігровими об'єктами, потрібно провести ретельне тестування для перевірки функціональності та виявлення можливих помилок. Важливо також налаштувати самі скрипти в Unity Editor, щоб забезпечити їх правильне виконання. Цей процес включає налаштування властивостей скриптів, визначення зв'язків між об'єктами та коректне реагування на події в грі. Таким чином, програмування на Unity — це комплексний і багатоетапний процес, що вимагає ретельного планування, налаштування та тестування для створення функціональної та стабільної гри

2.3.1 Реалізація механік гравця

Насамперед для початку було вирішено реалізувати клас PlayerController один із ключових компонентів гри, оскільки він відповідальний за зберігання важливих компонентів та забезпечення функціональності гравця. Цей клас діє як контролер, який координує та об'єднує різні аспекти управління гравцем, включаючи пересування, анімації, введення користувача та стан здоров'я. Це також дозволяє уникнути дублювання коду та покращити організацію проекту, розділяючи відповідальність між різними класами та компонентами.

Перелік змінних які необхідні для налаштування гравця:

```
private IPlayerInput _playerInput;
private IPlayerAnimator _playerAnimator;
private IPlayerMovement _playerMovement;
private SpriteRenderer _spriteRenderer;
private HealthSystem _healthSystem;
private BoxCollider _boxCollider;
private Rigidbody _rb;
public HealthSystem HealthSystem {get {return _healthSystem;}}
public IPlayerInput Input {get {return _playerInput;}}
public IPlayerAnimator Animator {get {return _playerAnimator;}}
public PlayerData PlayerData { get { return _playerData; } }
private Camera _camera;
[SerializeField] private PlayerData _playerData;
[SerializeField] private UnityEvent Events;
private bool _wasGrounded;
```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		35

Для можливості доступу до компонентів контролеру їх було проініціалізовано в класі.

```
private void Awake()
{
    _playerInput = new PlayerInputsHandler();
    _camera = GetComponentInChildren<Camera>();
    _rb = GetComponent<Rigidbody>();
    _playerMovement = new Movement(_rb,
    transform, GetComponentInChildren<Animator>(), _playerData);
    _playerAnimator = new PlayerAnimator(GetComponentInChildren<Animator>());
    _spriteRenderer = GetComponentInChildren<SpriteRenderer>();
    _animationActions = GetComponentInChildren<AnimationActions>();
    //Health
    _healthSystem = new HealthSystem(_playerData.maxHealth);
}
```

Система вводу відповідає за зчитування команд користувача і передачу її до системи управління, що забезпечує зв'язок між гравцем і грою. Для того щоб отримати кращий ігровий досвід, було обрано newInputSystem[18]. Для реалізації нової системи вводу був використаний згенерований Unity Input System клас PlayerInputs. Відповідно PlayerInputs було ініціалізовано PlayerInputsHandler що реалізує інтерфейс гравця IplayerInput.

PlayerInputsHandler забезпечує зчитування і обробку команд гравця. Використання як окремого класу для обробки вводу дозволяє легко змінювати або розширювати функціональність. Тому в конструкторі відбувається ініціалізація нової системи вводу й на основі згенерованого конфігураційного файлу активуємо.

```
public PlayerInputsHandler()
{
    _playerInputs = new PlayerInputs();
    _playerInputs.Enable();
}
```

Це забезпечує готовність системи вводу до роботи після створення екземпляру класу.

Також метод HandleFlip відповідає за зміну напрямку спрайта персонажа залежно від напрямку руху.

```

public void HandleFlip(Vector2 input,SpriteRenderer spriteRenderer)
{
    float horizontalInput = input.x;
    Quaternion changeRotation = spriteRenderer.transform.rotation;
    if (horizontalInput < 0)
    {
        changeRotation.y = 180f;
        spriteRenderer.gameObject.transform.rotation = changeRotation;
    }
    else if (horizontalInput > 0)
    {
        changeRotation.y = 0f;
        spriteRenderer.gameObject.transform.rotation = changeRotation;
    }
}

```

В методі MovementUpdate відбувається оброблення введення від гравця, керування пересуванням, анімаціями, та елементами пересування як ривок, переكات, стрибок.

```

public void MovementUpdate()
{
    if(_playerInput.GetJumping() && _wasGrounded )
    {
        _playerMovement.Jump();
        _playerAnimator.SetJumping(true);
    }
    Vector3 dashdirection = new Vector3(input.x, 0, input.y);
    if(_playerInput.GetDash() && dashdirection.magnitude >
        0f && _playerMovement.CanDash())
    {
        _playerMovement.Dash(dashdirection);
        _playerAnimator.SetDashing(true);
    }
    else
    {
        _playerAnimator.SetDashing(false);
    }
    Vector3 rollDirection = new Vector3(input.x, 0f, input.y);
    if(_playerInput.GetRoll() && rollDirection.magnitude > 0f)
    {
        _playerMovement.Roll(rollDirection);
        _playerAnimator.SetRolling(true);
    }
    else {_playerAnimator.SetRolling(false);}
}

```

Для того щоб отримувати швидкий доступ до даних було реалізовано клас PlayerData. PlayerData – клас який зберігає ігрові налаштування гравця та забезпечує доступ до цих налаштувань.

Клас пересування Movement, охоплює в собі основні раніше згадані механіки стрибок, переكات, ривок та пересування. Цей клас відповідає лише за пересування, залишаючи інші аспекти управління іншим класам .

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		37

```
[CreateAssetMenu(fileName = "PlayerData", menuName = "PlayerData", order = 0)]
public class PlayerData : ScriptableObject {
    [Header("Attack")]
    public float damage = 25f;
    [Header("Player Movement")]
    public float speed = 20f;
    public float jumpForce = 20f;
    public float dashDistance = 5f;
    public float dashCooldown = 2f;
    public float dashForce = 50f;
    public float rollForce = 10f;
    public float rollDuration = 0.5f;
    public float acceleration = 10f;
    public float deceleration = 8f;
    public float gravity = 9.8f;
    public float groundDistance = 1f;
    public LayerMask groundMask;
    [Header("Player Health")]
    public float maxHealth = 100f;
}
```

Метод Jump відповідає за реалізацію стрибка гравця. Щоб запобігти можливості багаторазових стрибків було використано метод IsGrounded()

```
public void Jump()
{
    if (_isGrounded)
    {
        _rigidbody.AddForce(Vector3.up * _playerData.jumpForce, ForceMode.Impulse);
    }
}
```

Якщо гравець знаходиться на землі, тоді додається сила вгору до компоненту Rigidbody, в наслідок чого відбувається підйом.

Метод Dash відповідає за реалізацію ривок гравця. Він додає силу, нормалізує напрямок руху, після чого множиться на силу ривок, забезпечуючи швидке прискорення за короткий рух.

```
public void Dash(Vector3 dashDirection)
{
    if (!_isDashing && _dashCooldownTimer <= 0f)
    {
        _isDashing = true;
        _dashDirection = dashDirection.normalized;
        _dashTimer = 0f;
        _dashDistance = _playerData.dashDistance;
        _rigidbody.AddForce(_dashDirection *
            playerData.dashForce, ForceMode.Impulse);
    }
}
```

Нормалізація напрямку ривок означає, що напрямок вектора зберігається, але його довжина стає рівною. Це робиться для того, щоб уривок завжди мав однакову силу незалежно від початкової швидкості або напрямку руху гравця.

```
public void UpdateDash()
{
    if (_isDashing)
    {
        _dashTimer += Time.deltaTime;
        if (_dashTimer >= _playerData.dashDistance / _playerData.dashForce)
        {
            Debug.Log("Dash Is End");
            _isDashing = false;
            _dashTimer = 0f;

            _dashCooldownTimer = _playerData.dashCooldown;
        }
    }
    else if (_dashCooldownTimer > 0f)
        _dashCooldownTimer -= Time.deltaTime;
}
```

Також використовується таймер для контролю тривалості ривок і його відновлення. Коли час ривок завершується починається відновлення.

Пережат працює аналогічно ривку, але з меншою силою та тривалістю, що додає реалізм та варіативність в механіки ухилення гравця. Time.deltaTime – це час, що пройшов між двома кадрами, і він допомагає робити рухи та анімації у грі плавними і однаковими на всіх комп'ютерах, незалежно від швидкості їх роботи.

Згідно з варіантами використання (рис. 2.1), до гравця також відноситься бойова система, яка включає систему здоров'я, використання поточної зброї та вибір нової, вибір нової зброї відбувається шляхом підписування до події скролу миші, в наслідок чого встановлюється нова поточна зброя та має методи використання зброї.

```
using UnityEngine;

public class PlayerCombatSystem : MonoBehaviour
{
    private IWeapon _currentWeapon;
    private PlayerController _playerController;

    private void Start() {
        _playerController = GetComponent<PlayerController>();
    }

    private void Update() {
        if (_playerController.Input.GetAttack()) {
            _currentWeapon.UseWeapon();
        }
    }
}
```

```

    }
}

public void SetCurrentWeapon(IWeapon weapon) {
    if (_currentWeapon != null) {
        _currentWeapon.IsActive = false;
        _currentWeapon.ExitWeapon();
    }

    _currentWeapon = weapon;
    _currentWeapon.IsActive = true;

    Debug.Log("Current Weapon: " + _currentWeapon.ToString());
}
}

```

Для реалізації зброї було створено інтерфейс IWeapon, який у свою чергу реалізує клас Weapon. Інтерфейс IWeapon визначає властивості та основні методи, які повинна мати зброя. Це дозволяє легше розробляти, тестувати та розширювати систему. Крім того, класи можуть реалізовувати кілька інтерфейсів, що дає можливість комбінувати різні набори поведінок.

```

public interface IWeapon
{
    float Damage { get; }
    float AttackInterval { get; }
    bool IsActive { get; set; }

    public void UseWeapon();
    public void ExitWeapon();
}

```

Абстрактний клас Weapon реалізовує інтерфейс IWeapon та надає базові властивості. Класи що наслідуються від Weapon реалізують конкретну поведінку зброї. Наприклад, зброя ближнього типу BladeWeapon. При натисканні кнопки удару, відбувається перевірка того чи можемо нанести удар, підписуємося на подію яка перевіряє зіткнення зброї. Також перевіряємо чи не дійшли до останньої комбо – атаки, якщо дійшли починаємо з початку. Наступним кроком перевизначаємо поточну анімацію атаки на анімацію з комбо.

```

public override void UseWeapon()
{
    if (_canAttack && IsActive)
    {
        SlowPlayer();
        SubscribeToAction();

        _isInCombo = true;
        _lasComboTime = 0f;

        if (_comboID == _comboAttacks.Count)
        {
            _comboID = 0;
        }
    }
}

```

```

        _canAttack = false;
        _playerAnimator.runtimeAnimatorController = _comboAttacks[_comboID];
        _playerAnimator.Play("Attack", 0, 0f);
        _comboID++;

        Invoke(nameof(ResetCanAttack), _attackInterval);
    }
}

```

Ми отримуємо компонент HitBoxAttackTrigger в якому є подія котра повертає об'єкт з яким відбулося зіткнення. Метод CheckHitBoxTarget підписується на дану подію й в момент виклику події отримуємо об'єкт та відбувається перевірка чи відбулося зіткнення з ворогом. Якщо зіткнення відбулося наносимо шкоду по ворогу.

```

private void CheckHitBoxTarget(GameObject target)
{
    StateMachine enemy;

    if (target.TryGetComponent<StateMachine>(out enemy))
    {
        Debug.Log("Player attack enemy");
        enemy.Components.Health.DecreaseHealth(_owner.PlayerData.damage);
    }
}

```

Підпис на подію виконується в методі SubscribeToAction. У цьому методі відбувається перевірка, якщо немає підпису на подію то відбувається перевірка.

Було також реалізовано зброю дального типу, що забезпечує комплексну логіку стрільби та перезарядки пістолета, а також дозволяє гравцеві взаємодіяти з ігровим середовищем без перешкод. Коли гравець використовує зброю викликається метод UseWeapon, Відбувається перевірка чи зброя не здійснює перезаряджання, відбувається також перевірка кількості куль у магазині, якщо кількість куль більше нуля тоді можемо здійснити постріл.

```

public override void UseWeapon()
{
    base.UseWeapon();

    if (_backToIdleAction == null)
    {
        _backToIdleAction = TryBackToIdle;
        _owner.AnimationActions.OnEndAnimation = _backToIdleAction;
    }

    if (!_isReloading)
    {
        if (_storeCount != 0 && _canShooting)
        {
            ShotFromGun();
        }
        else if (_storeCount == 0)
        {

```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		41

```

        StorageReloadingProcces();
    }
}

Коли відбувається постріл ми зменшуємо кількість куль у магазині на
одну, відключаємо можливість руху для гравця, змінюємо анімацію на анімацію
пострілу та встановлюємо затримку для дозволу наступного пострілу
викликаючи метод ResetCanShooting
private void ShotFromGun()
{
    _storeCount--;
    _canShooting = false;
    _isShooting = true;
    _owner.CanMoving = false;

    _playerAnimator.runtimeAnimatorController = _attack0V;
    _playerAnimator.Play("Attack", 0, 0f);

    CreateAndSetUpBullet();

    Invoke(nameof(ResetCanShooting), _attackInterval);
}

```

Процес перезарядки відбувається коли магазин порожній. Вказуємо що магазин порожній, змінюємо анімацію на очікування якщо гравець не стріляє в цей момент.

2.3.2 Реалізація системи здоров'я

Система здоров'я в проекті «Гра платформер в жанрі метроїдванія» відіграє ключову роль. Кожен ігровий об'єкт, який може взаємодіяти з гравцем, може впливати на його здоров'я.

Клас HealthSystem призначений для управління рівнем здоров'я гравця у грі. Він включає механізми для встановлення максимального та поточного рівня здоров'я, збільшення та зменшення здоров'я, а також виклику подій при зміні здоров'я або смерті гравця. Це дозволяє легко контролювати стан здоров'я персонажа і реагувати на його зміни під час ігрового процесу.

У класі є два приватних поля: `_maxHealth`, яке зберігає максимальне здоров'я, та `_currentHealth`, яке зберігає поточний рівень здоров'я. Властивості `CurrentHealth` і `MaxHealth` надають доступ до цих полів для інших частин програми. Вони забезпечують можливість тільки читання, що гарантує, що значення цих полів можуть бути змінені тільки методами цього класу.

```
private float _maxHealth;
```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		42

```
private float _currentHealth;
public float CurrentHealth {get {return _currentHealth;}}
public float MaxHealth {get {return _maxHealth;}}
```

Для обробки змін у стані здоров'я використовуються два делегати: HealthChanged та DeathEvent. Вони визначають типи подій, які можуть бути викликані при зміні здоров'я і смерті відповідно. Відповідні події OnHealthChanged та OnDeath оголошені як публічні, щоб інші частини програми могли підписатися на них і виконувати відповідні дії у випадку змін здоров'я або смерті гравця.

```
public void IncreaseHealth(float value)
{
    _currentHealth = Mathf.Clamp(_currentHealth + value, 0, _maxHealth);
    OnHealthChanged?.Invoke();
    Debug.Log("Player current health is: " + _currentHealth);
}
public void DecreaseHealth(float value)
{
    _currentHealth = Mathf.Clamp(_currentHealth - value, 0, _maxHealth);
    OnHealthChanged?.Invoke();
    if(_currentHealth <= 0)
    {
        Debug.Log("Player is Dead");
        OnDeath?.Invoke();
    }
}
}
```

Конструктор класу приймає значення максимального здоров'я і встановлює його як початкове значення для поточного здоров'я. Це означає, що при створенні об'єкта HealthSystem, гравець матиме повне здоров'я. Метод IncreaseHealth збільшує поточний рівень здоров'я на задану величину, обмежуючи його максимумом, і викликає подію OnHealthChanged. Метод DecreaseHealth зменшує поточний рівень здоров'я на задану величину, обмежуючи його мінімумом, і також викликає подію OnHealthChanged. Якщо після зменшення здоров'я його рівень досягає нуля, викликається подія OnDeath, що сигналізує про смерть гравця.

```
public HealthSystem(float maxHealth)
{
    _maxHealth = maxHealth;
    _currentHealth = maxHealth;
}
```

Цей клас забезпечує чітку структуру для управління здоров'ям персонажа, полегшуючи інтеграцію змін у здоров'ї в загальну логіку гри та надаючи

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		43

розробникам можливість швидко реагувати на критичні події, такі як смерть персонажа.

2.3.2 Реалізація стейт машини

Машина станів стала наступним кроком в розробці гри. Машина станів – це модель обчислення яку використовують як послідовність станів. Кожен стан описує поточну поведінку, а переходи визначають з якого стану необхідно здійснити перехід в інший.

Для реалізації StateMachine в Unity було створено інтерфейс IState, який визначав основні методи в кожному стані.

```
using UnityEngine;
```

```
public interface IState
{
    void Enter(StateMachine owner);
    void Update(StateMachine owner);
    void Exit(StateMachine owner);
}
```

Далі було реалізовано абстрактний стан State, що включає логіку входу, оновлення та виходу з стану та також список переходів, які можуть бути виконані зі стану.

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

```
public abstract class State : IState
{
    protected List<Transition> _transitions = new List<Transition>();

    public State(List<Transition> transitions)
    {
        _transitions = transitions;
    }

    public virtual void Enter(StateMachine owner) { }
    public virtual void Update(StateMachine owner)
    {
        foreach (var transition in _transitions)
        {
            if (transition.IsCanTransition(owner))
            {
                owner.ChangeState(transition.GetTransitionState());
            }
        }
    }
    public virtual void Exit(StateMachine owner) { } } }
```

Управління поточними станами здійснює клас StateMachine. Даний клас містить список станів і методи для зміни оновлення станів. Для прикладу ми розглянемо стан патрулювання.

Клас RangePatrollingState реалізує логіку стану патрулювання для ворога в грі, яка базується на системі станів. Цей клас спадкується від базового класу State і визначає поведінку ворога, коли він перебуває у стані патрулювання.

Конструктор класу приймає список переходів List<Transition> і передає їх до базового конструктора State. Це дозволяє керувати можливими переходами між станами, коли ворог змінює свою поведінку в залежності від певних умов.

Метод Enter викликається при вході в стан патрулювання. Тут відбувається ініціалізація основних компонентів, таких як PatrolWayPoints та NavMeshAgent[19]. Встановлюється початкова швидкість руху ворога, визначена в налаштуваннях ворога, і задається перша точка патрулювання. Аніматор ворога оновлюється для відображення стану патрулювання, а також виконується початковий переворот персонажа в напрямку першої точки патрулювання.

```
public override void Enter(StateMachine owner)
{
    Debug.Log("In Patrolling State");
    base.Enter(owner);
    _wayPoints = owner.GetComponent<PatrolWayPoints>();
    _agent = owner.GetComponent<NavMeshAgent>();
    _agent.isStopped = false;
    owner.Components.Animator.SetBool("isPatrolling", true);
    _path = _wayPoints.GetWayPoints();
    _agent.speed = owner.Components.EnemySettings.MoveSpeed;
    _agent.SetDestination(_path[_wpId]);
    owner.Components.EnemyFlip.FlipCharacter(_path[_wpId]);
}
```

Метод Update викликається на кожному кадрі гри, коли ворог перебуває у стані патрулювання. Якщо ворог досягне поточної точки патрулювання, перевіряється, чи він досяг своєї цілі за допомогою методу ReachedDestination. Якщо ціль досягнута, оновлюється ідентифікатор точки патрулювання _wpId, і задається нова цільова точка для руху. Ворог також перевертається в напрямку нової точки патрулювання.

```
public override void Update(StateMachine owner)
{
    base.Update(owner);
    if (ReachedDestination())
```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				45
Змн.	Арк.	№ док.ум.	Підпис	Дата		

```

{
    UpdateWayPointID();
    SetDestination(owner);
    owner.Components.EnemyFlip.FlipCharacter(_path[_wpId]);
}
}

```

Метод SetDestination визначає нову цільову точку для NavMeshAgent з використанням поточної точки патрулювання _wpId. Він перетворює двовимірні координати точки на тривимірні, які використовуються агентом для навігації.

Метод ReachedDestination перевіряє, чи агент досяг своєї цільової точки, порівнюючи відстань, що залишилася до цілі, з мінімальною відстанню зупинки агента.

```

private bool ReachedDestination()
{
    Debug.Log("Reached Destination");
    return _agent.remainingDistance <= _agent.stoppingDistance ;
}

```

Метод UpdateWayPointID відповідає за оновлення індексу поточної точки патрулювання _wpId. Якщо ворог рухається вперед по списку точок і досягає кінця, напрямок змінюється на зворотний, і ворог починає рухатися у зворотному напрямку. Це забезпечує безперервний рух по патрульному маршруту.

```

private void UpdateWayPointID()
{
    if (_isForward)
    {
        _wpId++;
        if(_wpId >= _path.Count)
        {
            _wpId = _path.Count - 1;
            _isForward = false;
        }
    }
    else
    {
        _wpId--;
        if(_wpId < 0)
        {
            _wpId = 1;
            _isForward = true;
        }
    }
}

```

Метод Exit викликається при виході зі стану патрулювання. Він зупиняє агент та оновлює анімацію, щоб відобразити, що ворог більше не патрулює.

```

public override void Exit(StateMachine owner)
{
    owner.Components.Animator.SetBool("isPatrolling", false);
    _agent.isStopped = true;
    base.Exit(owner);
}

```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		46

Таким чином, клас RangePatrollingState забезпечує логіку переміщення ворога по заздалегідь визначеному маршруту патрулювання, враховуючи коректну зміну напрямку руху та відповідну анімацію. Це дозволяє ворогам виконувати основну функцію патрулювання, що підвищує реалістичність і динамічність ігрового процесу.

Кожен стан має переходи, за

```
protected override void InitializeStates()
{
    Transition isInjuredTransition = new IsInjured_T(StatesEnum.Injured, this);

    _states[StatesEnum.Idle] = new RangeIdleState(new List<Transition>
    {
        new InSightRange_T(StatesEnum.Chasing, this),
        isInjuredTransition
    });
}
```

Умови переходу між станами визначаються класом Transition. Розглянемо приклад переходів з стану спокою до стану переслідування. Цей перехід відбувається, коли ворог виявляє гравця в полі зору. Це може бути корисно для ворогів, які повинні починати переслідування, як тільки гравець опиняється в зоні видимості.

```
public class InSightRange_T : Transition
{
    private IsTargetFOW _isTargetFOW;
    public InSightRange_T(StatesEnum transitionState, StateMachine machine) :
    base(transitionState)
    {
        _isTargetFOW = new IsTargetFOW(machine);
    }
    public override StatesEnum GetTransitionState()
    {
        return _transitionState;
    }

    public override bool IsCanTransition(StateMachine machine)
    {
        var playerPosition = machine.Components.Player.transform.position;
        var enemyPosition = machine.transform.position;
        float distanceToPlayer = Vector3.Distance(enemyPosition, playerPosition);
        var attackRange = machine.Components.EnemySettings.AttackRange;
        float viewDistance = machine.Components.EnemySettings.ViewDistance;

        if (distanceToPlayer < viewDistance && distanceToPlayer > attackRange)
        {
            Debug.Log("Playe in view distance !");
            if (_isTargetFOW.isPlayerVisible())
            {
                return true;
            }
            return false;
        }
        return false;
    }
}
```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				47
Змн.	Арк.	№ ДОКУМ.	Підпис	Дата		

}

Конструктор переходу `InSightRange_T` приймає поточний стан та стей машину як параметри. А `IsCanTransition` визначає, чи гравець знаходиться в зоні видимості ворога. Якщо відстань між ворогом та гравцем менше або дорівнює радіусу зору, то повертаємо істину. В стейт машині виглядає наступним чином перехід з стану спокою, де вказана умов `InSightRange_T` й перехід в інший стан.

```
protected override void InitializeStates()
{
    Transition isInjuredTransition = new IsInjured_T(StatesEnum.Injured, this);

    _states[StatesEnum.Idle] = new RangeIdleState(new List<Transition>
    {
        new InSightRange_T(StatesEnum.Chasing, this),
        isInjuredTransition
    });
}

ChangeState(StatesEnum.Patroling);
```

Таким чином визначається з якого стану в який ми можемо перейти та при якій умові.

Висновки до другого розділу

У другому розділі кваліфікаційної роботи бакалавра детально розглянуто основні етапи проектування та розробки програмного забезпечення гри в жанрі метроїдванія. Було визначено варіанти використання та об'єктно-орієнтовану структуру системи, а також проаналізовано фреймворк Zenject та його використання під час розробки гри.

Основні етапи реалізації гри включають розробку механік гравця, таких як пересування, стрибок, ривок і переكات, а також бойової системи та системи здоров'я. Було приділено увагу реалізації ворогів та їхньої штучної поведінки, використовуючи StateMachine патерн.

У цьому розділі представлено діаграми класів та варіантів використання, що ілюструють структуру системи та взаємодію між її компонентами. Діаграми демонструють механіки гри, включаючи бойову систему, збирання предметів, збереження прогресу та відновлення здоров'я гравця.

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		48

Цей розділ забезпечує розуміння того, як реалізовані основні функції гри, як налаштовані взаємодії між об'єктами та як використовуються програмні патерни для забезпечення гнучкості та розширюваності системи

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		49

РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ

3.1 Порядок встановлення та налаштування системи

«Гра платформер в жанрі метроїдванія» — це проект, який було створено за допомогою ігрового рушія Unity. Після завершення розробки, гру можна буде розмістити на різноманітних веб-сайтах, де гравці зможуть завантажити і встановити її на свій комп'ютер. Версія гри буде доступна для операційних систем Windows.

Для того, щоб додаток працював без проблем та не було малої кількості кадрів за секунду в грі, потрібно мати такі мінімальні вимоги до системи:

- Операційна система: Windows 10.
- 4 Гб оперативної пам'яті.
- Графічний процесор з підтримкою DirectX 10 та 2гб відео пам'яті
- intel core i3 9100f
- 100 мб вільно місця

Рекомендовані вимоги системи:

- Операційна система: Windows 10 і нижче.
- 4 Гб оперативної пам'яті.
- Графічний процесор з підтримкою DirectX 11 та 8гб відео пам'яті
- intel core i5-9400f
- 100 мб вільно місця

3.2 Інтерфейс та порядок роботи з програмним додатком

При запуску гри на користувача чекає стилізоване головне меню,

В меню гравець може, почати нову гру натиснувши на кнопку «Play» або вийти з гри натиснувши кнопку «Exit» (рис. 3.1).

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				50
Змн.	Арк.	№ док.ум.	Підпис	Дата		

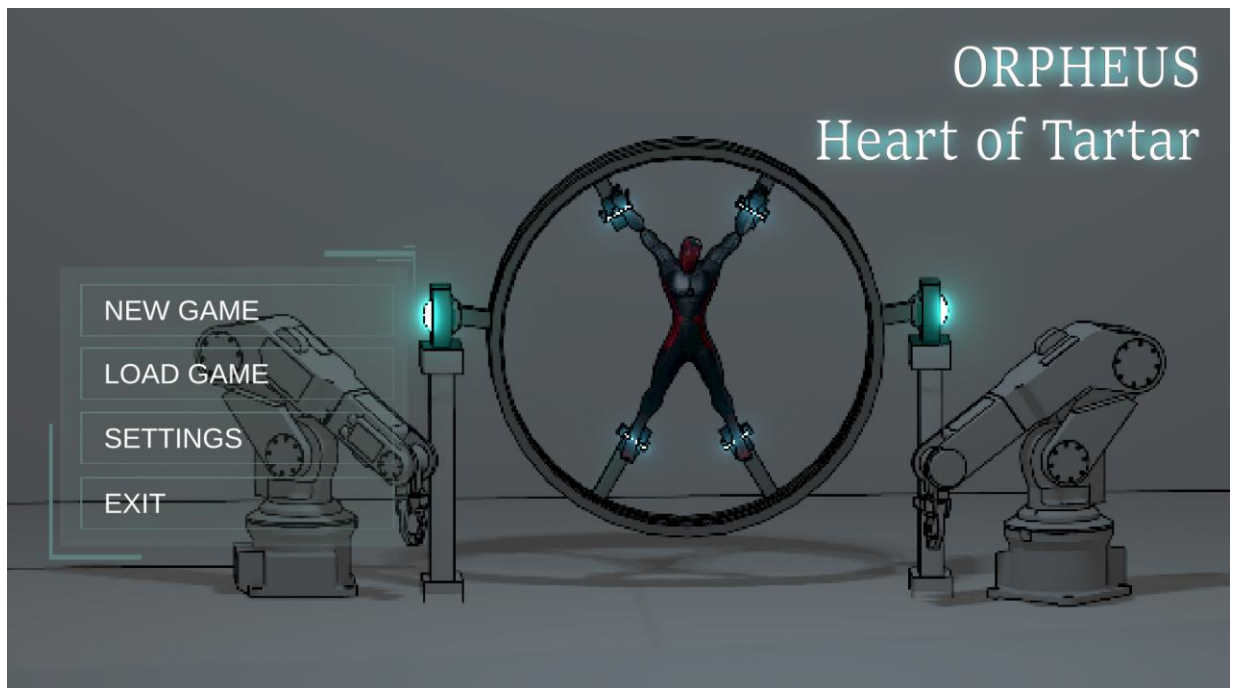


Рисунок 3.1 – Головне меню гри

Після натискання кнопки «Exit» відображається панельне вікно, яке просить підтвердження вийти з гри, дане вікно має дві опції, підтвердити вихід з гри та відхилити.(рис. 3.2)

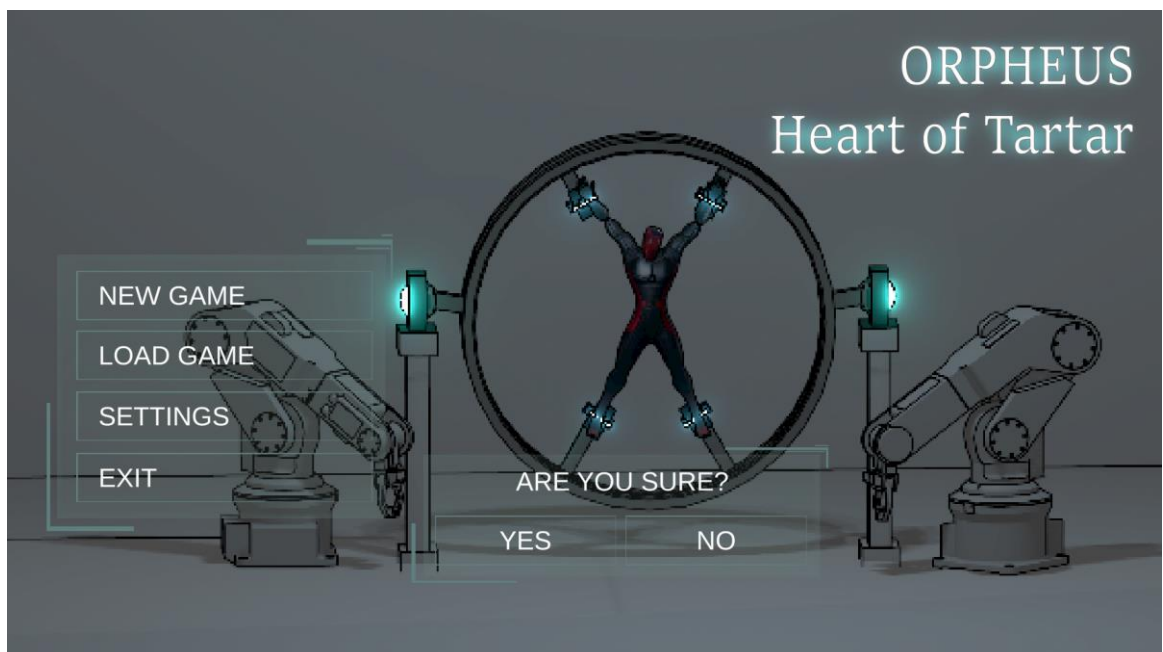


Рисунок 3.2 – Головне меню гри, підтвердження виходу

Після натискання на кнопку «New Game» відображається анімоване вікно переходу з головного меню на сцену під час якого завантажується сцена(рис. 3.3).

ORPHEUS

Heart of Tartar

Рисунок 3.3– Вікно переходу

Після завершення завантаження, гравець з'являється на сцені де він може почати пересуватися та досліджувати локацію.



Рисунок 3.4 – Головна сцена.

Також зліва зверху можливо додатково побачити стан здоров'я гравця.(рис.3.5)

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				52
Змн.	Арк.	№ докум.	Підпис	Дата		



Рисунок 3.5 – Панель здоров'я гравця

Під час пересування гравець може використовувати різні здібності такі як ривок, що спрямовує гравця швидко вперед, та можна побачити відповідну анімацію.(рис. 3.6)

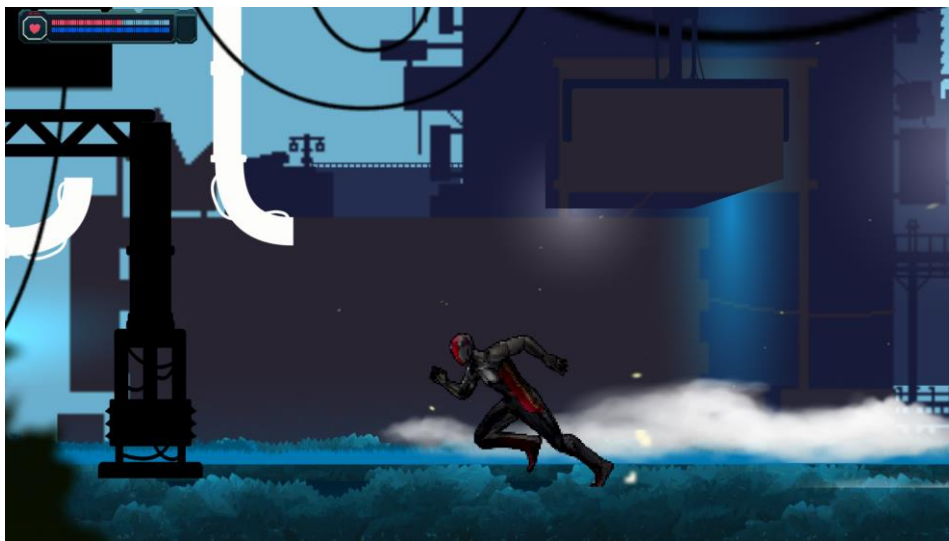


Рисунок 3.6 – Застосування ривку

Також є можливість виконувати переكات на відповідну клавішу, що дозволяє в певний моменти уникнути атаки ворога(рис.3.7)



Рисунок 3.7 – Застосування перекаату

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				53
Змн.	Арк.	№ докум.	Підпис	Дата		

Окрім того гравець може здійснювати стрибок при натисканні клавіші пробілу. (рис.3.8)

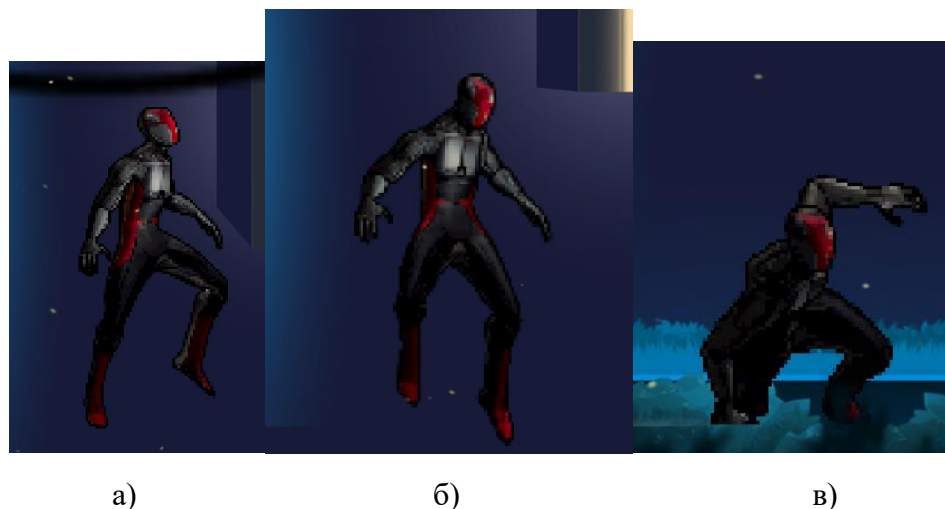


Рисунок 3.8 – а) анімація стрибку; б) анімація падіння; в) анімація приземлення

Після зустрічі з ворогом, гравець може увійти в бій з ворогом застосовуючи різні види зброї й наносити та отримувати шкоду.(рис 3.9)

Кожна зброя має також свої характеристики як от шкода та анімації. Використання ближнього типу зброї має комбо з відповідних трьох анімацій.

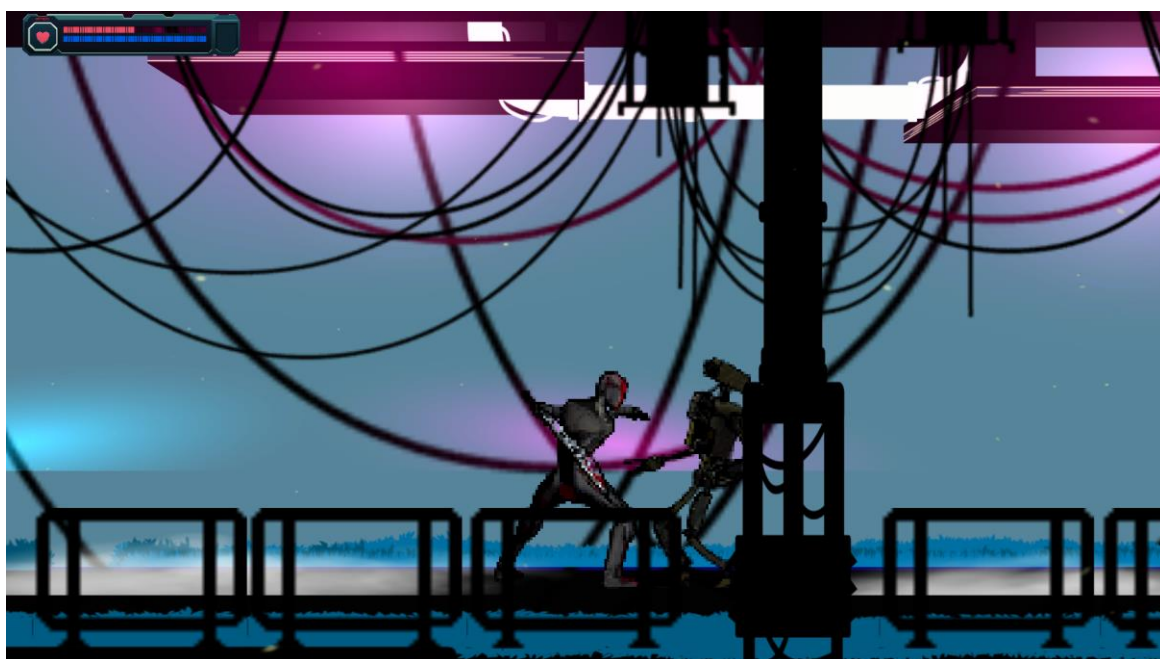


Рисунок 3.9 – Нанесення урону

На колесо миші гравець має змогу обрати дальню зброю,з якої він може наносити шкоду на середній. Повторне натискання змінить на іншу. (рис 3.10)

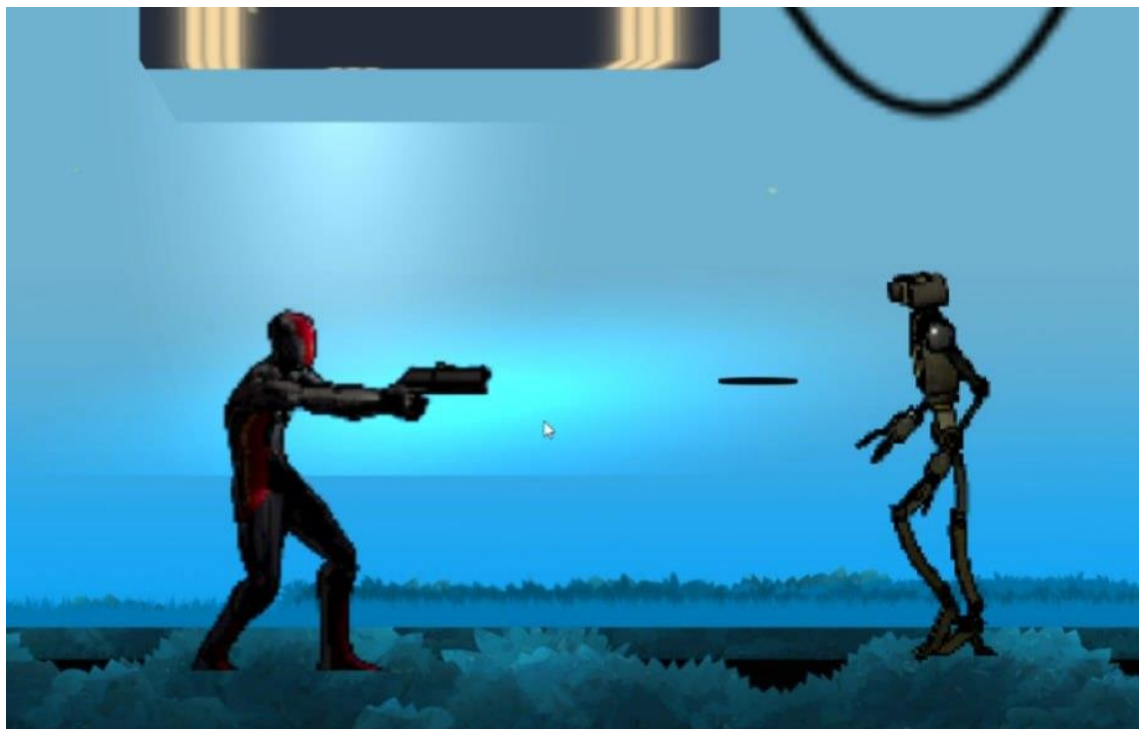


Рисунок 3.10 – Нанесення урону з дальньої зброї

Якщо кулі в магазині закінчилися то гравець перезаряджає зброю.(рис.3.11)

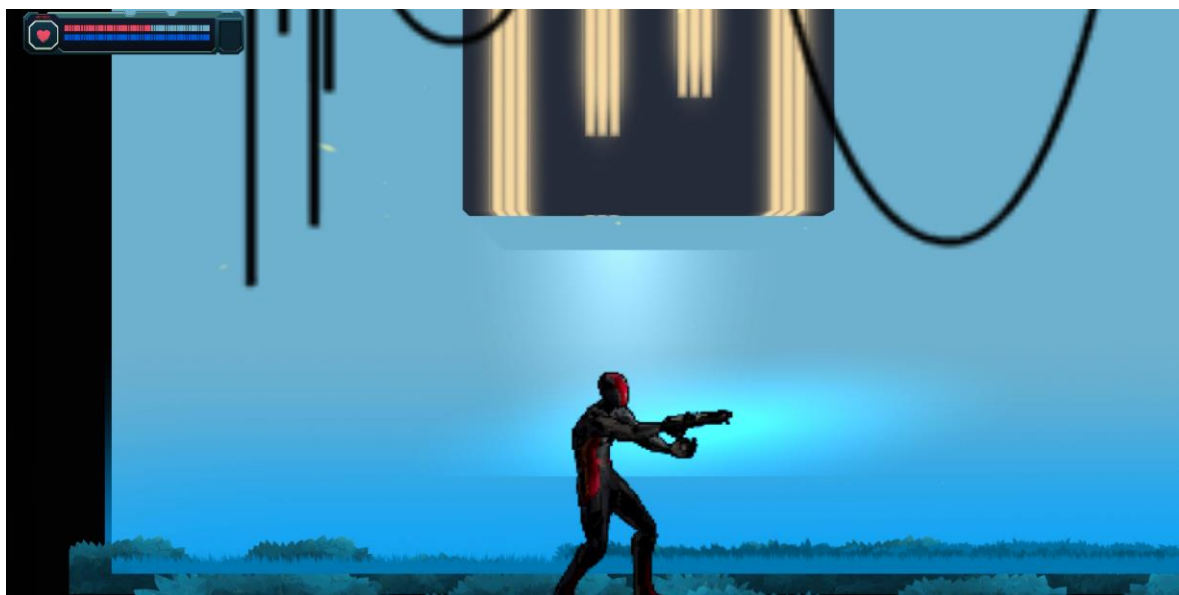


Рисунок 3.11 – Перезаряджання зброї

Після смерті гравця, персонаж з'являється на останній точці збереження та з новими ворогами.

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				55
Змн.	Арк.	№ докум.	Підпис	Дата		

3.3 Тестування роботи програмного додатку

Проведено тестування[20] усього програмного продукту, за для виявлення багів та помилок. Усі пункти тестування вказані в таблиці – таблиця 3.1

Таблиця 3.1

Таблиця тестування

Назва тестування	Вхідні дані	Очікувані дані	Отримані дані
Вхід в гру	Запуск гри	Показ головного меню	Як і очікувалося
Натискання кнопки "Play"	Натискання кнопки "Play"	Перехід до екрану завантаження або відразу до ігрової сцени	Як і очікувалося
Натискання кнопки "Exit"	Натискання кнопки "Exit"	Вихід з гри	Як і очікувалося
Перехід в ігрову сцену	Натискання кнопки "Start" або після завантаження	Перехід до ігрової сцени	Як і очікувалося
Екран завантаження	Запуск гри, натискання "Play"	Показ екрану завантаження, індикатор прогресу	Як і очікувалося
Пересування гравця	Натискання клавіш для руху	Гравець пересувається в обраному напрямку	Як і очікувалося
Здійснення стрибку	Натискання кнопки стрибка	Гравець здійснює стрибок	Як і очікувалося
Здійснення ривку	Натискання кнопки ривку	Гравець здійснює ривок вперед або в обраному напрямку	Як і очікувалося
Здійснення перекачу	Натискання кнопки перекачу	Гравець здійснює перекачу	Як і очікувалося
Вибір зброї	Вибір зброї з інвентарю	Зміна поточної зброї на обрану	Як і очікувалося

Застосування ближньої зброї	Натискання кнопки атаки ближньою зброєю	Гравець атакує ближньою зброєю	Як і очікувалося
Застосування дальньої зброї	Натискання кнопки атаки дальньою зброєю	Гравець атакує дальньою зброєю	Як і очікувалося
Робота анімацій	Виконання будь-яких дій	Відповідні анімації виконуються без затримок або збоїв	Як і очікувалося
Пересування ворогів	Гравець підходить до ворога	Ворог починає рухатися	Як і очікувалося
Знаходження гравця	Гравець потрапляє в зону зору ворога	Ворог помічає гравця і реагує	Як і очікувалося
Переслідування гравця	Гравець віддаляється від ворога	Ворог переслідує гравця	Як і очікувалося
Атакування гравця	Ворог знаходиться в межах атаки	Ворог здійснює атаку на гравця	Як і очікувалося
Патрулювання	Ворог знаходиться без гравця в зоні видимості	Ворог патрулює територію	Як і очікувалося
Віднімання здоров'я ворога	Атака ворога гравцем	Ворог втрачає частину здоров'я	Як і очікувалося
Віднімання здоров'я гравця	Атака гравця ворогом	Гравець втрачає частину здоров'я	Як і очікувалося
Смерть гравця	Гравець втрачає все здоров'я	Гравець помирає, показується екран смерті або гра перезавантажується	Як і очікувалося
Смерть ворога	Ворог втрачає все здоров'я	Ворог помирає, показується анімація смерті або ворог зникає	Як і очікувалося

Перезавантаження сцени	Повернення до попереднього екрану або рестарт гри	Гра перезавантажує поточну сцену або починає з останньої точки збереження	Як і очікувалося
------------------------	---	---	------------------

Під час тестування гри було виявлено велику кількість багів, що впливали на загальний ігровий процес. Одним із перших виявлених багів була проблема з пересуванням гравця. Під час руху гравець іноді міг застрягти в певних місцях на карті, що унеможливлювало подальше просування по рівню. Це створювало значні незручності для гравців і вимагало негайного виправлення. Проблему було вирішено шляхом коригування колайдерів, зменшення швидкості пересування гравця та налаштування сили стрибка, що дозволило уникнути застрягання в об'єктах.

Ще одна значна проблема стосувалася екрану завантаження. Після натискання кнопки "Play" гра іноді зависала на екрані завантаження, або цей екран не з'являвся взагалі. Це створювало враження, що гра не реагує на дії гравця, і могло призвести до того, що гравці просто закривали її. Для виправлення цієї проблеми було оптимізовано процес завантаження ресурсів та додано індикатор прогресу, який точно відображав хід завантаження.

Також були виявлені баги зі смертю гравця. У випадках, коли рівень здоров'я гравця досягав нуля, гра не завжди коректно обробляла цей стан. Іноді гравець міг продовжувати рухатися або виконувати дії навіть після смерті. Це суттєво порушувало логіку гри та впливало на ігровий досвід. Проблему було вирішено шляхом додавання додаткових перевірок на стан здоров'я та виклику події смерті, що забезпечило коректну обробку стану смерті гравця.

Проблеми виникали і з атакуванням гравця ворогами. Вороги іноді не могли правильно визначити місце знаходження гравця і не атакували його, навіть перебуваючи в межах досяжності. Це суттєво знижувало складність гри і робило

		О.В. Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		58

її менш цікавою для гравців. Виправлення включало покращення логіки пошуку та переслідування гравця, що забезпечило коректну поведінку ворогів і підвищило загальний рівень виклику у грі.

Після виправлення зазначених проблем гра стала більш стабільною та приємною для користувачів. Кожне з виправлень було ретельно протестовано, щоб забезпечити відсутність нових багів та стабільну роботу всіх механік гри. Це дозволило підвищити якість ігрового процесу та покращити загальне враження від гри.

Висновки до третього розділу

У третьому розділі дипломної роботи було ретельно розглянуто процес встановлення, налаштування та тестування гри «Гра платформер в жанрі метроїдванія». Описані системні вимоги допомагають забезпечити користувачам безперешкодний доступ та оптимальне використання ресурсів комп'ютера, зокрема встановлено мінімальні та рекомендовані параметри для комфортної гри.

Детально представлено інтерфейс та ігровий процес, зокрема етапи від старту гри до навігації по ігрових рівнях. Зокрема, акцент зроблено на користувацькі взаємодії, такі як вибір головних опцій у меню, механіки пересування персонажа та бойові взаємодії. Ці елементи були візуалізовані та підкріплені знімками екранів, що значно покращує зрозумілість опису ігрового процесу.

Процес тестування виявив ряд технічних недоліків, які були виправлені для забезпечення стабільності та покращення користувацького досвіду. Описані баги та їх вирішення підтверджують важливість комплексного підходу до тестування ігрових додатків. Виправлення помилок дозволило покращити якість гри, забезпечити її стабільність та зробити ігровий процес цікавішим і залучаючим для кінцевого користувача.

		О.В.Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О.Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		59

ВИСНОВКИ

В ході роботи над кваліфікаційною роботою було поставлено завдання та визначено основні механіки гри «Гра платформер в жанрі метроїдванія». Основними механіками було обрано пересування, ривок, стрибок, переكات, а також розробка системи ворогів із застосуванням StateMachine та NavMeshAgent. Проведено аналіз та порівняння відомих ігор у жанрі метроїдванія, розглянуто їхні сильні та слабкі сторони, що дозволило виділити ключові аспекти, які потрібно було реалізувати у власному проекті.

Було проведено аналіз ігрових рушіїв та середовищ розробки. В результаті проведеного аналізу було обрано ігровий рушій Unity та середовище розробки Microsoft Visual Studio. Цей вибір був обґрунтований широкими можливостями Unity для створення 2D-ігор, зручністю інтеграції з Visual Studio та доступністю великої кількості навчальних матеріалів. Було також описано використання фреймворку Zenject для реалізації принципів dependency injection, створено створено систему зберігань по точкам збереження.

Описано основні етапи реалізації гри. Було описано функціонал та процес гри, включаючи дії, які може виконувати гравець, такі як пересування, ривок, стрибок та переكات. Було продемонстровано результати тестування, описано знайдені баги та способи їх виправлення. У результаті отримано готовий ігровий проект під назвою «Гра платформер в жанрі метроїдванія», який демонструє рівень якості та відповідає вимогам жанру.

		О.В. Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				60
Змн.	Арк.	№ докум.	Підпис	Дата		

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке штучний інтелект: історія, види та складові – [Електронний ресурс]: <https://gigacloud.ua/blog/navchannja/scho-take-shtuchnij-intelekt-istorija-vidi-ta-skladovi>
2. Video Game Mechanics: Beginner`s Guide, Tips, and Ideas
<https://gamedesignlounge.com/video-game-mechanics-for-beginners/>
3. Saving the Day: Save Systems in Games складові – [Електронний ресурс]:
<https://www.gamedeveloper.com/design/saving-the-day-save-systems-in-games>
4. User interface design and implementation in Unity складові – [Електронний ресурс]: <https://unity.com/resources/user-interface-design-and-implementation-in-unity>
5. Metroid-like World Design – [Електронний ресурс]:
<https://nikles.it/2016/game-design/metroidvania-metroid-like-world-design/>
6. Hollow Knight – [Електронний ресурс]:
https://store.steampowered.com/app/367520/Hollow_Knight/
7. Ori and the Blind Forest – [Електронний ресурс]:
https://store.steampowered.com/app/387290/Ori_and_the_Blind_Forest_Definitive_Edition/
8. Blasphemous – [Електронний ресурс]:
<https://store.steampowered.com/app/774361/Blasphemous/>
9. Unity - [Електронний ресурс]: <https://docs.unity.com/>
10. Microsoft Visual Studio – [Електронний ресурс]: Official Documentation
<https://learn.microsoft.com/en-us/visualstudio/windows/?view=vs-2022>
11. State Design Patern - Unity Manual – [Електронний ресурс]:
https://sourcemaking.com/design_patterns/state
12. Фабричний метод – [Електронний ресурс]:
<https://www.bestprog.net/uk/2023/09/06/patterns-factory-method-pattern-implementation-of-the-structure-the-simplest-case-ua/>

		О.В. Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				61
Змн.	Арк.	№ докум.	Підпис	Дата		

13. Шаблонний метод – [Електронний ресурс]: <https://abitap.com/3-4-shablonnyj-metod-template-method/>
14. Zenject: Dependency Injection Framework for Unity3D – [Електронний ресурс]: <https://github.com/modesttree/Zenject>
15. Unity - Scripting API: Rigidbody – [Електронний ресурс]: <https://docs.unity3d.com/ScriptReference/Rigidbody.html>
16. Unity - Scripting API: Collider – [Електронний ресурс]: <https://docs.unity3d.com/ScriptReference/Collider.html>
17. How to structure your Unity projects <https://gamedevbeginner.com/how-to-structure-your-unity-project-best-practice-tips/>
18. Input System – [Електронний ресурс]: <https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/index.html>
19. Unity – Scripting API: NavMesh – [Електронний ресурс]: <https://docs.unity3d.com/ScriptReference/AI.NavMesh.html>
20. Game Testing – [Електронний ресурс]: <https://artoftesting.com/game-testing>

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				62
Змн.	Арк.	№ докум.	Підпис	Дата		

ДОДАТКИ

		О.В. Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		63

Файл PlayerController

```

using System.Collections;
using UnityEngine;
using UnityEngine.Events;

public class PlayerController : MonoBehaviour
{
    private IPlayerInput _playerInput;
    private IPlayerAnimator _playerAnimator;
    private IPlayerMovement _playerMovement;
    private SpriteRenderer _spriteRenderer;
    private HealthSystem _healthSystem;
    private BoxCollider _boxCollider;
    private Rigidbody _rb;
    private AnimationActions _animationActions;

    public HealthSystem HealthSystem {get {return _healthSystem;}}
    public AnimationActions AnimationActions {get {return _animationActions;}}
    public IPlayerInput Input {get {return _playerInput;}}
    public IPlayerAnimator Animator {get {return _playerAnimator;}}
    public IPlayerMovement Movement {get {return _playerMovement;}}
    public PlayerData PlayerData { get { return _playerData; } }
    //Костиль переробить
    private Camera _camera;

    [SerializeField] private PlayerData _playerData;
    [SerializeField] private UnityEvent Events;

    private bool _wasGrounded;
    private bool _canMoving = true;

    public bool CanMoving {get {return _canMoving;} set { _canMoving = value; } }

    private void Awake()
    {
        _playerInput = new PlayerInputsHandler();
        _camera = GetComponentInChildren<Camera>();
        _rb = GetComponent<Rigidbody>();
        _playerMovement = new Movement(_rb,
        transform, GetComponentInChildren<Animator>(), _playerData);
        _playerAnimator = new PlayerAnimator(GetComponentInChildren<Animator>());
        _spriteRenderer = GetComponentInChildren<SpriteRenderer>();
        _animationActions = GetComponentInChildren<AnimationActions>();
        //Health
        _healthSystem = new HealthSystem(_playerData.maxHealth);
    }

    private void Start()
    {
        _boxCollider = GetComponent<BoxCollider>();
    }

    private void Update()
    {
        if (_canMoving) {
            MovementUpdate();
            _playerMovement.Update();
            bool isGrounded = _playerMovement.CheckGround();
            if (isGrounded && !_wasGrounded) {
                OnLanding();
            }
            _wasGrounded = isGrounded;
        }
    }
}

```

		О.В Ганнущенко			ІІЗ.КР.Б – 121 – 24 – ІІЗ	Арк.
		Ю.О. Годлевський				64
ЗМН.	Арк.	№ ДОКУМ.	Підпис	Дата		

```

    }
    else {
        _rb.velocity = Vector3.zero;
    }
}

public void MovementUpdate()
{
    Vector2 input = _playerInput.GetMovement();
    HandleFlip(input);
    _playerMovement.Move(input);
    float positionChange = _playerMovement.GetPositionChange();
    _playerAnimator.SetSpeed(positionChange);
    if(_playerInput.GetJumping() && _wasGrounded )
    {
        _playerMovement.Jump();
        _playerAnimator.SetJumping(true);
    }
    Vector3 dashdirection = new Vector3(input.x, 0, input.y);
    if(_playerInput.GetDash() && dashdirection.magnitude > 0f &&
    _playerMovement.CanDash())
    {
        _playerMovement.Dash(dashdirection);
        _playerAnimator.SetDashing(true);
    }
    else
    {
        _playerAnimator.SetDashing(false);
    }
    Vector3 rollDirection = new Vector3(input.x, 0f, input.y);
    if(_playerInput.GetRoll() && rollDirection.magnitude > 0f)
    {
        _playerMovement.Roll(rollDirection);
        _playerAnimator.SetRolling(true);
    }
    else {_playerAnimator.SetRolling(false);}
}

public void OnLanding()
{
    _playerAnimator.SetJumping(false);
    _playerAnimator.SetFalling(false);
}

public void HandleFlip(Vector2 input)
{
    float horizontalInput = input.x;
    Quaternion changeRotation = _spriteRenderer.transform.rotation;
    if (horizontalInput<0)
    {
        changeRotation.y = 180f;
        _spriteRenderer.gameObject.transform.rotation = changeRotation;
    }
    else if(horizontalInput>0)
    {
        changeRotation.y = 0f;
        _spriteRenderer.gameObject.transform.rotation = changeRotation;
    }
}

//Health
public void ApplyDamage(float damageAmount)
{
    _healthSystem.DecreaseHealth(damageAmount);
}

public void ApplyHealing(float healAmount)
{

```

```

        _healthSystem.IncreaseHealth(healAmount);
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.GetComponent<StateMachine>())
        {
            var stateMachine = collision.gameObject.GetComponent<StateMachine>();
            var contactY = collision.contacts[0].point.y - 1.2f;
            var playerLowerY = transform.position.y - ((_boxCollider.size.y * 0.5f));

            if (contactY > collision.transform.position.y) {
                if (contactY < playerLowerY) {
                    stateMachine.enabled = false;
                    Debug.Log("Player jump to enemy head");
                    Vector3 direction = Vector3.zero;
                    float rotationY = _spriteRenderer.transform.rotation.y;

                    Debug.Log("Rotation Y: " + rotationY);

                    if (rotationY == 1) {
                        direction = Vector3.right;
                    }
                    else if (rotationY == 0) {
                        direction = Vector3.left;
                    }

                    _rb.AddForce(Vector3.up * 20f, ForceMode.Impulse);
                    _rb.AddForce(direction * 30f, ForceMode.Impulse);

                    StartCoroutine(EnableStateMachine(stateMachine));
                }
            }

            Debug.Log($"PlayerCollideWith c: {contactY}");
            Debug.Log($"PlayerCollideWith P: {playerLowerY}");
        }
    }

    IEnumerator EnableStateMachine(StateMachine machine) {
        yield return new WaitForSeconds(0.5f);
        machine.enabled = true;
    }
}

```

Файл Movement:

```

using UnityEngine;

public class Movement : IPlayerMovement
{
    private Animator _animator;
    private Rigidbody _rigidbody;
    private Transform _playerTransform;
    private PlayerData _playerData;
    private BoxCollider _collider;

    private Vector3 _targetPosition;
    private Vector3 _dashDirection;
    private Vector3 _rollDirection;
    private Vector3 _previousPosition;

    private bool _isGrounded;
    private bool _isJumping;
    private bool _isFalling;
    private bool _isDashing;
}

```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ІЗ	Арк.
		Ю.О. Годлевський				
ЗМН.	Арк.	№ ДОКУМ.	Підпис	Дата		66

```

private bool _isRolling;
private bool _isCanRolling = false;
private bool _IsJumped;

private float _dashDistance;
private float _dashTimer;
private float _dashCooldownTimer;
private float _rollingLeftTime = 0f;

private Vector3 _rollingColliderSize = Vector3.zero;
private Vector3 _defaultColliderSize = Vector3.zero;

public Movement(Rigidbody rigidbody, Transform transform, Animator animator,
PlayerData playerData)
{
    _rigidbody = rigidbody;
    _playerTransform = transform;
    _animator = animator;
    _playerData = playerData;

    if (_collider == null)
    {
        _collider = _playerTransform.GetComponent<BoxCollider>();

        _defaultColliderSize = _collider.size;
        _rollingColliderSize = _collider.size;
        _rollingColliderSize.y /= 2;
    }
}

public void Update()
{
    Vector3 currentVelocity = _rigidbody.velocity;
    Vector3 targetVelocity = _targetPosition;
    targetVelocity.y = currentVelocity.y;
    if (!_isDashing)
    {
        _rigidbody.velocity = Vector3.Lerp(currentVelocity, targetVelocity,
_playerData.acceleration * Time.deltaTime);
    }
    else if (!_isRolling)
    {
        _rigidbody.velocity = Vector3.Lerp(currentVelocity, targetVelocity,
_playerData.acceleration * Time.deltaTime);
    }

    ApplyGravity();
    UpdateDash();
    UpdateRoll();
    UpdateFalling();
    if (_isGrounded && _isJumping) _isJumping = false;
    _previousPosition = _playerTransform.position;
}

public void Move(Vector2 input)
{
    Vector3 movementDirection = new Vector3(input.x, 0, input.y).normalized;
    movementDirection = _playerTransform.TransformDirection(movementDirection);
    _targetPosition = movementDirection * _playerData.speed;
}

public float GetPositionChange()
{
    return _targetPosition.magnitude;
}

public void Jump()

```

		О.В Ганнущенко			ІІЗ.КР.Б – 121 – 24 – ІІЗ	Арк.
		Ю.О. Годлевський				67
ЗМН.	Арк.	№ докум.	Підпис	Дата		

```

{
    if (_isGrounded)
    {
        _rigidbody.AddForce(Vector3.up * _playerData.jumpForce,
ForceMode.Impulse);
        _isGrounded = false;
        _isJumping = true;
        _isFalling = false;
        _animator.SetBool("isJumping", _isJumping);
        _animator.SetBool("isFalling", false);
    }
}

public void UpdateFalling()
{
    if (!_isGrounded)
    {
        if (_playerTransform.position.y < _previousPosition.y)
        {
            _isJumping = false;
            _isFalling = true;
            _animator.SetBool("isJumping", false);
            _animator.SetBool("isFalling", true);
        }
    }
    else
    {
        _isJumping = false;
        _isFalling = false;
        _animator.SetBool("isJumping", false);
        _animator.SetBool("isFalling", false);
    }
}

public void Dash(Vector3 dashDirection)
{
    if (!_isDashing && _dashCooldownTimer <= 0f)
    {
        _isDashing = true;
        _dashDirection = dashDirection.normalized;
        _dashTimer = 0f;
        _dashDistance = _playerData.dashDistance;
        _rigidbody.AddForce(_dashDirection * _playerData.dashForce,
ForceMode.Impulse);
    }
}

public void UpdateDash()
{
    if (_isDashing)
    {
        _dashTimer += Time.deltaTime;
        if (_dashTimer >= _playerData.dashDistance / _playerData.dashForce)
        {
            Debug.Log("Dash Is End");
            _isDashing = false;
            _dashTimer = 0f;

            _dashCooldownTimer = _playerData.dashCooldown;
        }
    }
    else if (_dashCooldownTimer > 0f)
        _dashCooldownTimer -= Time.deltaTime;
}

public bool CanDash()
{
    return !_isDashing && _dashCooldownTimer <= 0f;
}

```

```

    }

    public void Roll(Vector3 rollDirection)
    {
        if (!_isRolling)
        {
            Debug.Log("Roll start");
            _isRolling = true;

            Vector3 rollinColliderCenter = Vector3.zero;
            _collider.size = _rollingColliderSize;
            rollinColliderCenter.y = _collider.size.y / 2;
            _collider.center -= rollinColliderCenter;

            _rollDirection = rollDirection.normalized;
            _rollingLeftTime = _playerData.rollDuration;

            _rigidbody.AddForce(_rollDirection * _playerData.rollForce,
ForceMode.Impulse);
        }
    }

    public void UpdateRoll()
    {
        if (_isRolling)
        {
            _rollingLeftTime -= Time.deltaTime;
            if (_rollingLeftTime <= 0f)
            {
                _isRolling = false;
                Debug.Log("Rolling end");

                Vector3 defaultColliderCenter = _collider.center;
                defaultColliderCenter.y = 0f;
                _collider.center = defaultColliderCenter;
                _collider.size = _defaultColliderSize;
            }
        }
    }

    public void ApplyGravity()
    {
        if (!_isGrounded)
        {
            Vector3 gravity = Vector3.down * _playerData.gravity;
            _rigidbody.AddForce(gravity, ForceMode.Acceleration);
        }
    }

    public bool CheckGround()
    {
        Ray ray = new Ray(_playerTransform.position, Vector3.down);
        RaycastHit hit;
        Debug.DrawRay(ray.origin, ray.direction * _playerData.groundDistance,
Color.red);
        if (Physics.Raycast(ray, out hit, _playerData.groundDistance,
_playerData.groundMask))
        {
            _isGrounded = true;
            _isJumping = false;
            return true;
        }
        else
        {
            _isGrounded = false;
        }
    }

```

```

        return false;
    }
}
}

```

Файл PlayerData

```

using UnityEngine;

[CreateAssetMenu(fileName = "PlayerData", menuName = "PlayerData", order = 0)]
public class PlayerData : ScriptableObject {
    [Header("Attack")]
    public float damage = 25f;
    [Header("Player Movement")]
    public float speed = 20f;
    public float jumpForce = 20f;
    public float dashDistance = 5f;
    public float dashCooldown = 2f;
    public float dashForce = 50f;
    public float rollForce = 10f;
    public float rollDuration = 0.5f;
    public float acceleration = 10f;
    public float deceleration = 8f;
    public float gravity = 9.8f;
    public float groundDistance = 1f;
    public LayerMask groundMask;
    [Header("Player Health")]
    public float maxHealth = 100f;
}

```

Файл PlayerCombatSystem

```

using UnityEngine;

public class PlayerCombatSystem : MonoBehaviour
{
    private IWeapon _currentWeapon;
    private PlayerController _playerController;

    private void Start() {
        _playerController = GetComponent<PlayerController>();
    }

    private void Update() {
        if (_playerController.Input.GetAttack()) {
            _currentWeapon.UseWeapon();
        }
    }

    public void SetCurrentWeapon(IWeapon weapon) {
        if (_currentWeapon != null) {
            _currentWeapon.IsActive = false;
            _currentWeapon.ExitWeapon();
        }

        _currentWeapon = weapon;
        _currentWeapon.IsActive = true;

        Debug.Log("Current Weapon: " + _currentWeapon.ToString());
    }
}

```

Файл IWeapon

```
public interface IWeapon
{
    float Damage { get; }
    float AttackInterval { get; }
    bool IsActive { get; set; }

    public void UseWeapon();
    public void ExitWeapon();
}
```

Файл Weapon

```
using UnityEngine;

public abstract class Weapon : MonoBehaviour, IWeapon
{
    [Header("Common Settings")]
    [SerializeField] protected float _damage;
    [SerializeField] protected float _attackInterval;

    public float Damage => _damage;
    public float AttackInterval => _attackInterval;

    public bool IsActive { get; set; }

    protected PlayerController _owner;

    protected virtual void Start()
    {
        if (GetComponentInParent<PlayerController>())
        {
            _owner = GetComponentInParent<PlayerController>();
            Debug.Log("Owner was finded");
        }
    }

    public virtual void UseWeapon() { }

    public virtual void ExitWeapon() { }
}
```

Файл BladeWeapon

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

public class BladeWeapon : Weapon
{
    [Header("Blade Weapon Settings")]
    [SerializeField] private float _exitTime = 1f;
    [SerializeField] private List<AnimatorOverrideController> _comboAttacks;

    private int _comboID = 0;

    private bool _canAttack = true;
    private bool _isInCombo = false;
    private bool _slowMovement = false;

    private Action<GameObject> _attackTriggerAction;
```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				71
ЗМН.	АРК.	№ ДОКУМ.	Підпис	Дата		

```

private HitBoxAttackTrigger _attackTrigger;
private Animator _playerAnimator;
private float _lasComboTime = 0f;

protected override void Start()
{
    base.Start();
    _playerAnimator = _owner.GetComponentInChildren<Animator>();
    GetAttackTrigger();
}

private void Update()
{
    if (_isInCombo)
    {
        _lasComboTime += Time.deltaTime;

        if (_lasComboTime > _exitTime)
        {
            _comboID = 0;
            _isInCombo = false;
            _slowMovement = false;
            _owner.PlayerData.speed *= 4f;

            _playerAnimator.Play("Player_Idle");
        }
    }
}

public override void UseWeapon()
{
    if (_canAttack && IsActive)
    {
        SlowPlayer();
        SubscribeToAction();

        _isInCombo = true;
        _lasComboTime = 0f;

        if (_comboID == _comboAttacks.Count)
        {
            _comboID = 0;
        }

        _canAttack = false;
        _playerAnimator.runtimeAnimatorController = _comboAttacks[_comboID];
        _playerAnimator.Play("Attack", 0, 0f);
        _comboID++;

        Invoke(nameof(ResetCanAttack), _attackInterval);
    }
}

private void ResetCanAttack()
{
    _canAttack = true;
}

private void GetAttackTrigger()
{
    if (_owner.GetComponentInChildren<HitBoxAttackTrigger>() != null)
    {
        _attackTrigger = _owner.GetComponentInChildren<HitBoxAttackTrigger>();
        _attackTrigger.Action = CheckHitBoxTarget;
    }
}

```

```

    }

    SubscribeToAction();
}

private void CheackHitBoxTarget(GameObject target)
{
    StateMachine enemy;

    if (target.TryGetComponent<StateMachine>(out enemy))
    {
        Debug.Log("Player attack enemy");
        enemy.Components.Health.DecreaseHealth(_owner.PlayerData.damage);
    }
}

private void SlowPlayer()
{
    if (!_slowMovement)
    {
        _owner.PlayerData.speed /= 4f;
        _slowMovement = true;
    }
}

private void OnDestroy()
{
    _attackTrigger.TriggerAction -= _attackTriggerAction;
}

public override void ExitWeapon()
{
    if (_attackTrigger.TriggerAction != null) {
        _attackTrigger.TriggerAction -= _attackTriggerAction;
    }
}
}

```

Файл PistolWeapon

```

using UnityEngine;

public abstract class Weapon : MonoBehaviour, IWeapon
{
    [Header("Common Settings")]
    [SerializeField] protected float _damage;
    [SerializeField] protected float _attackInterval;

    public float Damage => _damage;
    public float AttackInterval => _attackInterval;

    public bool IsActive { get; set; }

    protected PlayerController _owner;

    protected virtual void Start()
    {
        if (GetComponentInParent<PlayerController>())
        {
            _owner = GetComponentInParent<PlayerController>();
            Debug.Log("Owner was finded");
        }
    }
}

```

```

        public virtual void UseWeapon() { }

        public virtual void ExitWeapon() { }
    }

```

Файл WeaponManager

```

using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class WeaponsManager : MonoBehaviour
{
    private List<Weapon> _weapons = new List<Weapon>();
    private PlayerCombatSystem _combatSystem;
    private PlayerController _player;

    private int _currentWeaponID = 0;
    public float _weelValue;

    private void Start() {
        _weapons = GetComponents<Weapon>().ToList<Weapon>();

        if (GetComponentInParent<PlayerCombatSystem>()) {
            _combatSystem = GetComponentInParent<PlayerCombatSystem>();
            _combatSystem.SetCurrentWeapon(_weapons[_currentWeaponID]);
        }

        _player = GetComponentInParent<PlayerController>();
    }

    private void Update() {
        _weelValue = _player.Input.GetMouseScroll();

        if (_weelValue > 0) {
            _currentWeaponID++;
            if (_currentWeaponID > _weapons.Count - 1) {
                _currentWeaponID = 0;
            }

            _combatSystem.SetCurrentWeapon(_weapons[_currentWeaponID]);
        } else if (_weelValue < 0) {
            _currentWeaponID--;
            if (_currentWeaponID < 0) {
                _currentWeaponID = _weapons.Count - 1;
            }

            _combatSystem.SetCurrentWeapon(_weapons[_currentWeaponID]);
        }
    }
}

```

Файл IDamageable

```

public interface IDamageable
{
    public void DecreaseHealth(float damage);
}

```

Файл IHealeble

```

using System.Collections;
using System.Collections.Generic;

```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
ЗМН.	Арк.	№ ДОКУМ.	Підпис	Дата		74

```

using UnityEngine;

public interface IHealeble
{
    public void IncreaseHealth(float amount);
}

Файл HealthSystem

using UnityEngine;

public class HealthSystem : IHealeble, IDamageable
{
    private float _maxHealth;
    private float _currentHealth;
    public float CurrentHealth {get {return _currentHealth;}}
    public float MaxHealth {get {return _maxHealth;}}
    public delegate void HealthChanged();
    public delegate void DeathEvent();
    public event HealthChanged OnHealthChanged;
    public event DeathEvent OnDeath;

    public HealthSystem(float maxHealth)
    {
        _maxHealth = maxHealth;
        _currentHealth = maxHealth;
    }
    public void IncreaseHealth(float value)
    {
        _currentHealth = Mathf.Clamp(_currentHealth + value, 0, _maxHealth);
        OnHealthChanged?.Invoke();
        Debug.Log("Player current health is: " + _currentHealth);
    }
    public void DecreaseHealth(float value)
    {
        _currentHealth = Mathf.Clamp(_currentHealth - value, 0, _maxHealth);
        OnHealthChanged?.Invoke();
        if(_currentHealth <= 0)
        {
            Debug.Log("Player is Dead");
            OnDeath?.Invoke();
        }
    }
}

```

Файл IState

```

using UnityEngine;

public interface IState
{
    void Enter(StateMachine owner);
    void Update(StateMachine owner);
    void Exit(StateMachine owner);
}

```

Файл StatesEnum

```

public enum StatesEnum
{
    Idle,
    Attack,
    Chasing,
    Patrolling,
    LostTarget,
    Injured,
}

```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				
Змн.	Арк.	№ докум.	Підпис	Дата		75

```

    Dead
}

```

Файл State

```

using System.Collections.Generic;

public abstract class State : IState
{
    protected List<Transition> _transitions = new List<Transition>();

    public State(List<Transition> transitions)
    {
        _transitions = transitions;
    }

    public virtual void Enter(StateMachine owner) { }
    public virtual void Update(StateMachine owner)
    {
        foreach (var transition in _transitions)
        {
            if (transition.IsCanTransition(owner))
            {
                owner.ChangeState(transition.GetTransitionState());
            }
        }
    }
    public virtual void Exit(StateMachine owner) { }
}

```

Файл StateMachine

```

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public abstract class StateMachine : MonoBehaviour
{
    protected EnemyComponents _components;
    protected PlayerController _playerController;
    protected IState _currentState;
    protected Dictionary<StatesEnum, IState> _states;
    public EnemyComponents Components { get { return _components; } }

    protected virtual void Start()
    {
        _states = new Dictionary<StatesEnum, IState>();
        InitializeStates();
    }

    protected virtual void Update()
    {
        if (_currentState != null)
        {
            Debug.Log($"Current enemy state: {_currentState}");
            _currentState.Update(this);
        }
    }
}

```

```

    public void ChangeState(StatesEnum state)
    {
        TransitionToState(state);
    }

    protected void TransitionToState(StatesEnum stateName)
    {
        if (_currentState != null)
        {
            _currentState.Exit(this);
        }

        _currentState = _states[stateName];
        _currentState.Enter(this);
    }

    protected abstract void InitializeStates();
}

```

Файл Transition

```

using UnityEngine;

public abstract class Transition
{
    protected StatesEnum _transitionState;
    protected Transition(StatesEnum transitionState)
    {
        _transitionState = transitionState;
    }

    public abstract bool IsCanTransition(StateMachine machine); // чи умова виконана
    public abstract StatesEnum GetTransitionState(); // в який стан переходимо
}

```

Файл EnemyComponents

```

using UnityEngine;
using UnityEngine.AI;

public class EnemyComponents
{
    private EnemySettingsSO _enemySettings;
    private PlayerController _playerController;
    private StateMachine _currentSM;
    private NavMeshAgent _agent;
    private PatrolWayPoints _patrolWayPoints;
    private Transform _transform;
    private Timer _timer;
    private HealthSystem _health;
    private Animator _animator;
    private AnimationActions _actions;
    private SpriteRenderer _spriteRenderer;
    private EnemyFlip _enymyFlip;

    public EnemySettingsSO EnemySettings { get { return _enemySettings; } }
    public PlayerController Player { get { return _playerController; } }
    public StateMachine StateMachine { get { return _currentSM; } }
    public NavMeshAgent Agent { get { return _agent; } }
}

```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				77
ЗМН.	Арк.	№ докум.	Підпис	Дата		

```

public PatrolWayPoints WayPoints { get { return _patrolWayPoints; } }
public Transform Transform { get { return _transform; } }
public Timer Timer { get { return _timer; } }
public HealthSystem Health { get { return _health; } }
public Animator Animator { get { return _animator; } }
public AnimationActions AnimationActions { get { return _actions; } }
public SpriteRenderer SpriteRenderer { get { return _spriteRenderer; } }
public EnemyFlip EnemyFlip { get { return _enmyFlip; } }

public EnemyComponents(EnemySettingsSO enemySettingsSO, PlayerController
playerController, StateMachine currentSM)
{
    _enemySettings = enemySettingsSO;
    _playerController = playerController;
    _currentSM = currentSM;
    _health = new HealthSystem(_enemySettings.MaxHelth);
    GetComponents();
}
private void GetComponents()
{
    _agent = _currentSM.GetComponent<NavMeshAgent>();
    _transform = _currentSM.GetComponent<Transform>();
    _timer = _currentSM.GetComponent<Timer>();
    _animator = _currentSM.GetComponentInChildren<Animator>();
    _actions = _currentSM.GetComponentInChildren<AnimationActions>();
    _spriteRenderer = _currentSM.GetComponentInChildren<SpriteRenderer>();
    _enmyFlip = new EnemyFlip(_transform, _playerController.transform,
_spriteRenderer);
    if (_timer == null )
    {
        _timer = _currentSM.gameObject.AddComponent<Timer>();
    }
}
}

```

Файл EnemySettingsSO

```

using UnityEngine;

[CreateAssetMenu(fileName = "NewEnemySettings", menuName = "EnemySettings/Settings")]
public class EnemySettingsSO : ScriptableObject
{
    [Header("Settings")]
    public float MaxHelth = 100f;

    [Header("Movement")]
    public float ChasingRange = 20f;
    public float AttackRange = 10f;
    public float ViewDistance = 10f;
    public float ViewAngle = 60f;
    public float MoveSpeed = 12f;
    public float RotateSpeed = 20f;
    public float WaitDuration = 2f;
    public float AttackInterval = 2f;
    public float AttackDamage = 20f;
}

```

Файл RangeEnemy_SM

```

using System.Collections.Generic;
using UnityEngine;

public class RangeEnemy_SM : StateMachine
{
    [SerializeField] private GameObject _projectile;
}

```

		О.В Ганнущенко			ІПЗ.КР.Б – 121 – 24 – ІІЗ	Арк.
		Ю.О. Годлевський				
ЗМН.	АРК.	№ ДОКУМ.	Підпис	Дата		78

```

[SerializeField] private EnemySettingsSO _settings;
[SerializeField] private PlayerController _playerControllerRE;
public GameObject Projectile { get { return _projectile; } }
protected override void Start()
{
    _components = new EnemyComponents(_settings, _playerControllerRE, this);
    base.Start();
}

protected override void InitializeStates()
{
    Transition isInjuredTransition = new IsInjured_T(StatesEnum.Injured, this);

    _states[StatesEnum.Idle] = new RangeIdleState(new List<Transition>
    {
        new InSightRange_T(StatesEnum.Chasing, this),
        isInjuredTransition
    });
    _states[StatesEnum.Attack] = new RangeAttackState(new List<Transition>
    {
        new OutOfAttackRange_T(StatesEnum.Chasing),
        new InAttackRange_T(StatesEnum.Attack, this),
        isInjuredTransition
    });
    _states[StatesEnum.Injured] = new RangeInjuredState(new List<Transition>
    {
        new InSightRange_T(StatesEnum.Chasing, this),
        new InAttackRange_T(StatesEnum.Attack, this),
        new OutOfSightRange_T(StatesEnum.LostTarget),
        new IsDead_T(StatesEnum.Dead)
    });
    _states[StatesEnum.Dead] = new RangeDeathState(new List<Transition> { });
    _states[StatesEnum.Chasing] = new RangeChasingState(new List<Transition>
    {
        new InAttackRange_T(StatesEnum.Attack, this),
        new OutOfSightRange_T(StatesEnum.LostTarget),
        isInjuredTransition
    });
    _states[StatesEnum.LostTarget] = new RangeLostTargetState(new List<Transition>
    {
        new InSightRange_T(StatesEnum.Chasing, this),
        new WaitTransition_T(StatesEnum.Patrolling),
        isInjuredTransition
    });
    _states[StatesEnum.Patrolling] = new RangePatrollingState(new List<Transition>
    {
        new InSightRange_T(StatesEnum.Chasing, this),
        new InAttackRange_T(StatesEnum.Patrolling, this),
        isInjuredTransition
    });
    ChangeState(StatesEnum.Patrolling);
}
}

```

Файл InSightRange_T

```
using UnityEngine;
```

```
public class InSightRange_T : Transition
```

		О.В Ганнущенко			ІІЗ.КР.Б – 121 – 24 – ПЗ	Арк.
		Ю.О. Годлевський				79
ЗМН.	Арк.	№ ДОКУМ.	Підпис	Дата		

```

{
    private IsTargetFOW _isTargetFOW;
    public InSightRange_T(StatesEnum transitionState, StateMachine machine) :
base(transitionState)
    {
        _isTargetFOW = new IsTargetFOW(machine);
    }
    public override StatesEnum GetTransitionState()
    {
        return _transitionState;
    }

    public override bool IsCanTransition(StateMachine machine)
    {
        var playerPosition = machine.Components.Player.transform.position;
        var enemyPosition = machine.transform.position;
        float distanceToPlayer = Vector3.Distance(enemyPosition, playerPosition);
        var attackRange = machine.Components.EnemySettings.AttackRange;
        float viewDistance = machine.Components.EnemySettings.ViewDistance;

        if (distanceToPlayer < viewDistance && distanceToPlayer > attackRange)
        {
            Debug.Log("Playe in view distance !");
            if (_isTargetFOW.isPlayerVisible())
            {
                return true;
            }
            return false;
        }
        return false;
    }
}

```

Файл OutOfSightRange

```

using UnityEngine;

public class OutOfSightRange_T : Transition
{
    public OutOfSightRange_T(StatesEnum transitionState) : base(transitionState)
    {
    }
    public override StatesEnum GetTransitionState()
    {
        return _transitionState;
    }

    public override bool IsCanTransition(StateMachine machine)
    {
        var playerPosition = machine.Components.Player.transform.position;
        var enemyPosition = machine.transform.position;
        float distanceToPlayer = Vector3.Distance(enemyPosition, playerPosition);
        float viewDistance = machine.Components.EnemySettings.ViewDistance;
        if (distanceToPlayer > viewDistance)
        {
            return true;
        }
        return false;
    }
}

//!_isTargetFOW.isPlayerVisible()

```

Файл RangeChasingState

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

public class RangeChasingState : State
{
    public RangeChasingState(List<Transition> transitions) : base(transitions){}
    private NavMeshAgent _agent;
    public override void Enter(StateMachine owner)
    {
        Debug.Log("In Chasing State");
        _agent = owner.GetComponent<NavMeshAgent>();
        owner.Components.Agent.isStopped = false;
        _agent.speed = owner.Components.EnemySettings.MoveSpeed;
        owner.Components.Animator.SetBool("isChasing", true);

        owner.Components.EnemyFlip.FlipCharacter(owner.Components.Player.transform.position);
    }

    public override void Update(StateMachine owner)
    {
        base.Update(owner);
        Vector3 playerPosition = owner.Components.Player.transform.position;
        owner.Components.Agent.SetDestination(playerPosition);
    }

    public override void Exit(StateMachine owner)
    {
        owner.Components.Agent.isStopped = true;
        owner.Components.Animator.SetBool("isChasing", false);
    }
}
```

Файл RangePatrollingState

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

public class RangePatrollingState : State
{
    private PatrolWayPoints _wayPoints;
    private NavMeshAgent _agent;
    private List<Vector2> _path = new List<Vector2>();
    private int _wpId = 0;
    private bool _isForward = true;
    private float _rotateSpeed;

    public RangePatrollingState(List<Transition> transitions) : base(transitions)
    {
    }

    public override void Enter(StateMachine owner)
    {
        Debug.Log("In Patrolling State");
        base.Enter(owner);
        _wayPoints = owner.GetComponent<PatrolWayPoints>();
        _agent = owner.GetComponent<NavMeshAgent>();
        _agent.isStopped = false;
        owner.Components.Animator.SetBool("isPatrolling", true);
        _path = _wayPoints.GetWayPoints();
        _agent.speed = owner.Components.EnemySettings.MoveSpeed;
    }
}
```

		О.В Ганнущенко			ІІЗ.КР.Б – 121 – 24 – ІІЗ	Арк.
		Ю.О. Годлевський				81
ЗМН.	АРК.	№ ДОКУМ.	Підпис	Дата		

```

        _agent.SetDestination(_path[_wpId]);
        owner.Components.EnemyFlip.FlipCharacter(_path[_wpId]);
    }
    public override void Update(StateMachine owner)
    {
        base.Update(owner);
        if (ReachedDestination())
        {
            UpdateWayPointID();
            SetDestination(owner);
            owner.Components.EnemyFlip.FlipCharacter(_path[_wpId]);
        }
    }
    private void SetDestination(StateMachine owner)
    {
        Vector2 currentPoint = _path[_wpId];
        _agent.SetDestination(new Vector3(currentPoint.x, currentPoint.y, 0f));
    }
    private bool ReachedDestination()
    {
        Debug.Log("Reached Destination");
        return _agent.remainingDistance <= _agent.stoppingDistance ;
    }
    private void UpdateWayPointID()
    {
        if (_isForward)
        {
            _wpId++;
            if(_wpId >= _path.Count)
            {
                _wpId = _path.Count - 1;
                _isForward = false;
            }
        }
        else
        {
            _wpId--;
            if(_wpId < 0)
            {
                _wpId = 1;
                _isForward = true;
            }
        }
    }
    private bool IsValidID()
    {
        return _wpId > 0 && _wpId < _path.Count -1;
    }
    public override void Exit(StateMachine owner)
    {
        owner.Components.Animator.SetBool("isPatrolling", false);
        _agent.isStopped = true;
        base.Exit(owner);
    }
}

```