

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

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 122 «Комп'ютерні науки»
(освітня програма «Комп'ютерна графіка та розробка ігор»)
на тему:

«Розробка поведінки ворогів та окремих механік для гри в
жанрах Metroidvania і Soulslike»

Виконав студент групи КН-20-1:
Іщук Олександр Юрійович

Керівник роботи:
МАРЧУК Галина Вікторівна

Рецензент:
КОРОТУН Ольга Володимирівна

Житомир – 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК
СПЕЦІАЛЬНІСТЬ 122 «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук
Марина ГРАФ
«25 січня»2024 р.

ЗАВДАННЯ

на кваліфікаційну роботу освітнього ступеня «бакалавр»
за спеціальністю 122 «Комп'ютерні науки»
(освітня програма «Комп'ютерна графіка та розробка ігор»)

Здобувач вищої освіти: ІЩУК Олександр Юрійович
Керівник роботи: МАРЧУК Галина Вікторівна
Тема роботи: «Розробка поведінки ворогів та окремих механік для гри у жанрах
Metroidvania і Soulslike»,
затверджена наказом закладу вищої освіти від «25 січня»2024 р., №22/с
Термін здачі закінченої роботи: «06» червня 2024 р.
Вихідні дані роботи (зазначається предмет і об'єкт дослідження): розробка поведінки
ворогів використовуючи ігровий двигун Unity для їх реалізації для гри у жанрі
метроїдванія.

Консультанти з кваліфікаційної роботи із зазначенням розділів, що їх стосуються

Розділ	Консультант	Дата	
		Завдання видав	Завдання прийняв
1			
2			
3			

Календарний план

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі. Пошук, огляд, аналіз доступного інструментарію та аналогічного ПЗ.	11.11.2023 – 12.12.2023	Виконано
2	Створення схем та структури додатку.	15.01.2024 – 05.02.2024	Виконано
3	Розробка програмної частини.	10.02.2024 – 20.05.2024	Виконано
4	Тестування.	21.05.2024 – 03.06.2024	Виконано
5	Попередній захист.	06.06.2024	Виконано

Здобувач вищої освіти Олександр ІЩУК
Керівник Галина МАРЧУК

РЕФЕРАТ

Кваліфікаційна робота складається з програмного додатку гри та пояснювальної записки

Пояснювальна записка до кваліфікаційної роботи складається з реферату, вступу, основної частини, загального висновку, списку використаної літератури та додатку.

Текстова частина викладена на 69 сторінках друкованого тексту.

Пояснювальна записка має 13 сторінок додатку. Список використаних джерел містить 20 найменувань, і займає дві сторінки.

В роботі наведено 29 рисунків та 6 таблиць. Загальний обсяг роботи – 84 сторінки.

Ключові слова: ГРА, METROIDVANIA, SOULSLIKE, UNITY, 2D, ПЛАТФОРМЕР, STATE MASHINE, URP, ІГРОВИЙ ІНТЕЛЕКТ.

					КН.КР.Б – 122 – 24 – ПЗ							
Змн.	Арк.	№ докум.	Підпис	Дата								
Розроб.		Іщук О.Ю.			«Розробка поведінки ворогів та окремих механік для гри в жанрах Metroidvania і Soulslike» Пояснювальна записка			Літ.		Арк.	Аркушів	
Керівник		Марчук Г.В.								2	84	
Реценз.		Коротун О.В.						Житомирська політехніка, група КН-20-1				
Н. Контр.												
Затверд.		Граф М.С.										

ABSTRACT

The graduation thesis of the bachelor consists of a software application of the game and an explanatory note.

An explanatory note to the final thesis consists of an abstract, introduction, main part, general conclusion, a list of used literature and an appendix.

The text part is laid out on 69 pages of printed text.

The explanatory note has 13 pages of annexes. The list of used sources contains 20 names and occupies two pages.

The work contains 29 figures and 6 table. The total volume of work is 84 pages.

Keywords: GAME, METROIDVANIA, SOULSLIKE, UNITY, 2D, PLATFORMER, STATE MASHINE, URP, GAME INTELLIGENCE.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		4

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ	
ВИРІШЕННЯ ЗАДАЧІ	9
1.1 Постановка задачі	9
1.2 Аналіз існуючого програмного забезпечення за тематикою кваліфікаційної роботи	10
1.3 Аналіз та вимоги до програмного забезпечення	14
Висновки до першого розділу	16
РОЗДІЛ 2. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	17
2.1 Проектування на основі UML	17
2.2 Розробка архітектури поведінки ворогів	28
2.3 Створення системи збереження та завантаження даних	32
2.4 Реалізація програмної частини State Mashine	38
2.5 Розгляд станів для State Mashine	45
2.6 Налаштування аніматора та ворога	53
Висновки до другого розділу	58
РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО	
ТЕСТУВАННЯ	60
3.1 Робота State mashine босів локацій	60
3.2 Тестування роботи додатку	63
Висновки до третього розділу	67
ВИСНОВКИ	68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	69
ДОДАТКИ	71

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ПП – програмний продукт.

ПЗ – програмне забезпечення.

FSM – Finite-state machine (Скінченний автомат).

UML – Unified Modeling Language.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		6

ВСТУП

Актуальність теми: Обрана тема є дуже актуальною в сучасному світі, де ігрова індустрія переживає вражаючий розквіт. Протягом останнього десятиріччя спостерігається експоненційне зростання популярності ігор серед різних груп користувачів. Широкий доступ до ігрових продуктів, включаючи їхню доступність на мобільних пристроях, зробив гру всеосяжною і частково визначальною частиною сучасної культури.

Ігри вже давно вийшли за межі простої розваги і стали важливим засобом вираження та спілкування. Вони не просто розважають, але і створюють свої власні світи з індивідуальними правилами і викликами. У цьому контексті ігровий інтелект стає ключовим компонентом, оскільки він визначає поведінку ворогів та навколишнього світу, впливаючи на загальний ігровий досвід користувачів. Такий інтелект дозволяє створити глибокий, захоплюючий та реалістичний світ, в якому гравці можуть відчувати себе частиною захоплюючої пригоди. Тому вивчення та розробка ігрового інтелекту є важливим етапом розвитку геймдевелопменту в наш час.

Метою кваліфікаційної роботи є розробка ігрового інтелекту для ворогів для ігрового додатку, використовуючи State Machine на рушії Unity, що забезпечить реалістичну та динамічну поведінку персонажів, підвищить загальний рівень складності гри та покращить ігровий досвід користувачів.

Об'єктом дослідження є технології створення ігрового інтелекту в Unity, що дозволять створювати складні та керовані стани поведінки для персонажів у грі, забезпечуючи більш реалістичну взаємодію та адаптивність ворогів до дій гравця.

Предметом дослідження є розробка ігрового інтелекту для ворогів у грі з використанням State Machine в Unity, з метою створення адаптивних і реалістичних поведінкових моделей, що покращують ігровий процес та підвищують рівень взаємодії з користувачем.

В рамках теми кваліфікаційної роботи було опубліковано тези:

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		7

Марчук Г. В., Іщук О.Ю. Ігровий інтелект. Технологічний розвиток та вплив на геймінг. Тези XIV Міжнародної науково-технічної конференції «Інформаційно-комп'ютерні технології – 2024», 28 – 29 березня 2024 року. Житомир : «Житомирська політехніка», 2024. С.99-100.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		8

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Постановка задачі

Основною метою розробки поведінки ворогів у будь-якому жанрі ігор є створення виклику та унікального досвіду для кожного гравця. Ігровий інтелект забезпечує занурення у віртуальний світ, а у випадку правильної реалізації створює цікаві та неповторні ігрові ситуації.

Також важливою частиною будь-якого додатку є збереження і завантаження прогресу. Саме це і необхідно реалізувати як окремі механіки, що пов'язані не лише з ворогами, але і іншими аспектами програми.

Підсумовуючи, задача розробки включає:

1. *Написання скелету State Mashine:*
 - Детальне вивчення технології State Machine.
 - Створення структури State Machine.
 - Реалізація розробленої структури.
2. *Розробка станів для ворога:*
 - Створення концептуальної схеми роботи ворога.
 - Написання відповідних станів, відповідно до розробленої схеми.
 - Перевірка ефективності та коректності роботи станів.
3. *Налаштування аніматора:*
 - Налаштування аніматора для взаємодії із State Machine.
4. *Розробка окремих механік:*
 - Розробка системи збережень і завантаження.

Ці завдання включають в себе створення основи для ігрового інтелекту, розробку конкретних станів для кожного типу ворога та налаштування анімаційного процесу. Також налічує розробку окремих налаштовуваних снарядів та системи збереження завантаження. Цей комплексний підхід передбачає реалізацію усіх необхідних механік. Аналіз особливостей жанрів

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		9

Metroidvania і Soulslike врахований для досягнення максимальної відповідності створюваного ігрового інтелекту специфіці.

1.2 Аналіз існуючого програмного забезпечення за тематикою кваліфікаційної роботи

У даному розділі буде проведено аналіз існуючого програмного забезпечення, яке використовується для розробки ігрового інтелекту та механік в іграх жанрів Metroidvania і Soulslike. Ці жанри характеризуються складними системами бою, нелінійною прогресією та високим рівнем складності, що вимагає від гравця стратегічного мислення та адаптивності. Розглянемо найпоширеніші інструменти та підходи, які використовуються в індустрії для досягнення таких цілей.

Hollow Knight [1] (рис. 1.1) гра Hollow Knight є яскравим прикладом жанру Metroidvania, де поведінка ворогів відіграє важливу роль у формуванні геймплею. Ворогів характеризують різні патерни атак та рухів, які вимагають від гравця вивчення та адаптації. Для реалізації такої поведінки в Unity використовуються комбінації State Machine, що дозволяє створювати реалістичні та складні П-моделі [2].

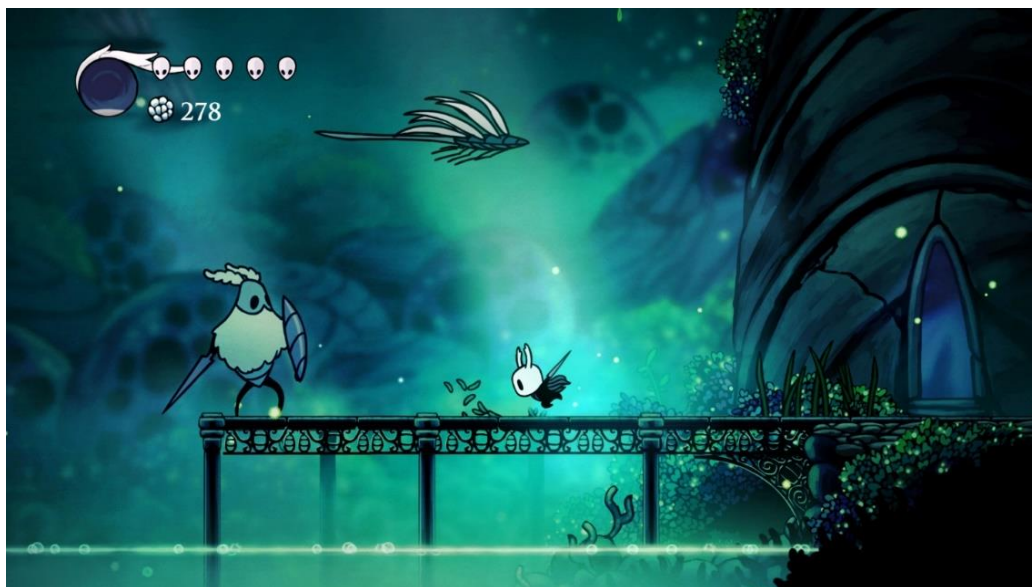


Рисунок 1.1 - Битва з ворогом в грі Hollow Knight

Dark Souls [3] (рис. 1.2) серія ігор, що є засновником жанру Soulslike, відомого своїм високим рівнем складності та глибокими механіками бою.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		10

Вороги в Dark Souls мають складні алгоритми поведінки, що включають розпізнавання атак гравця, ухилення та контратаки. Одним з ключових елементів є переслідування гравця: вороги активно реагують на присутність гравця в певній області і починають його переслідувати, намагаючись зменшити відстань і завдати удару. Область переслідування визначається певним радіусом навколо ворога, в якому він здатен виявити гравця та почати рух до нього.



Рисунок 1.2 - Битва з босом в грі Dark Souls

Боси в Dark Souls відрізняються своєю унікальною поведінкою та складністю. Кожен бос має власний набір атак і тактик, які гравець повинен вивчити та адаптуватися до них. Боси часто мають декілька фаз бою, де кожна наступна фаза стає складнішою і вимагає нових підходів для перемоги. Це додає глибини та варіативності в ігровий процес, змушуючи гравця постійно вдосконалювати свої навички та стратегії. Загалом, складні алгоритми поведінки ворогів і босів у Dark Souls роблять гру захоплюючою та викликовою, що є однією з причин її популярності серед гравців.

Blasphemous [4] (рис. 1.3) гра, що поєднує елементи обох жанрів, представляючи складні та жорстокі механіки бою в стилі soulslike, а також нелінійне дослідження світу, характерне для Metroidvania. Поведінка ворогів у Blasphemous є однією з ключових складових гри, і їхні дії часто включають

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		11

кілька фаз з різними патернами атак, що вимагає від гравця швидкої реакції та стратегічного підходу.



Рисунок 1.3 - Битва з босом в грі Blasphemous

Реалізація ворогів та босфайтів у іграх жанрів Metroidvania і Soulslike "Hollow Knight"

Звичайні вороги. В грі Hollow Knight вороги мають різноманітні патерни атак та рухів, що вимагає від гравця постійної адаптації та вивчення їхньої поведінки. Вороги можуть бути як наземними, так і повітряними, використовувати різні види атак: ближній бій, дистанційні атаки, стрибки та інше. Звичайні вороги діляться на кілька типів:

- Патрулюючи вороги рухаються по визначеному маршруту та атакують гравця при наближенні.
- Агресивні вороги активно шукають гравця та атакують при виявленні.
- Підстерігаючі ховаються та атакують несподівано, коли гравець наближається.

Босфайти. Боси в Hollow Knight зазвичай мають кілька фаз бою. Кожна фаза відрізняється новими атаками та зміненою поведінкою. Фази можуть включати:

- Перша фаза: базові атаки та прості патерни рухів.
- Друга фаза: додаються нові атаки або змінюються існуючі патерни, підвищується агресивність.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		12

– Третя фаза: бос може використовувати найпотужніші атаки, змінювати швидкість та тактику бою.

Dark Souls

Звичайні вороги. У серії ігор Dark Souls вороги мають складні алгоритми поведінки. Вони можуть реагувати на дії гравця, використовувати різні тактики в бою та переслідувати гравця. Типи ворогів включають:

- Патрулюючи вороги рухаються по маршруту і атакують гравця при наближенні.
- Агресивні вороги активно шукають гравця і нападають при виявленні.
- Стрільці атакують з відстані, використовуючи луки або магію.
- Підстерігаючі ховаються і атакують несподівано, коли гравець наближається.

Босфайти. Боси в Dark Souls відомі своєю складністю та унікальними механіками. Босфайти зазвичай складаються з кількох фаз:

- Перша фаза: бос використовує базові атаки та прості патерни.
- Друга фаза: додаються нові атаки, підвищується агресивність, можуть змінюватися патерни.
- Третя фаза та далі: бос може отримувати нові здібності, змінювати тактику, підвищувати швидкість атак.

Blasphemous

Звичайні вороги. У грі Blasphemous вороги поєднують елементи обох жанрів, представляючи складні та жорстокі механіки бою. Вони можуть мати кілька фаз з різними патернами атак:

- Патрулюючи вороги слідує за маршрутом і атакують гравця при виявленні.
- Агресивні вороги активно переслідують і атакують гравця.
- Стрільці атакують з відстані, використовуючи зброю або магію.
- Підстерігаючі ховаються та атакують несподівано, коли гравець наближається.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		13

Босфайти. Босфайти в Blasphemous відрізняються декількома фазами бою:

- Перша фаза: базові атаки та патерни.
- Друга фаза: додаються нові атаки, змінюються існуючі патерни, підвищується агресивність.
- Третя фаза та далі: бос може використовувати нові здібності, змінювати тактику, збільшувати швидкість та силу атак.

За результатами аналізу можна зробити наступні висновки: звичайні вороги в іграх жанрів Metroidvania та Soulslike мають складні алгоритми поведінки, які включають різноманітні патерни атак та рухів. Вони діляться на кілька типів залежно від їхніх дій та тактики. Босфайти зазвичай складаються з декількох фаз, де кожна наступна фаза додає нові виклики та атаки, що вимагає від гравця постійної адаптації та вдосконалення своєї стратегії. Такий підхід дозволяє створити глибокий та викликовий ігровий досвід, що є ключовим елементом у жанрах Metroidvania та Soulslike.

1.3 Аналіз та вимоги до програмного забезпечення

Для реалізації проєкту вибрано рушій Unity 2D, оскільки це середовище надає високий рівень візуальності та легко реалізовує геймплейні аспекти, що важливо для жанрів Metroidvania і Soulslike. Застосування FSM (англ. Finite State Machine) [5-7] обрано для досягнення логічного та структурованого управління поведінкою ворогів. В рамках функціональних вимог визначено необхідність реалізації руху ворогів, їхньої атаки та переходів між станами. З урахуванням нефункціональних вимог, таких як ефективність, гнучкість та зручність використання, визначено, що Unity 2D та FSM найкраще відповідають потребам проєкту. Загальний підхід до аналізу та визначення вимог до програмного забезпечення допомагає у подальшій розробці інтелекту ворогів у відповідності до жанрових вимог і специфікацій (Таблиця 1.1).

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		14

Вимоги до програмного забезпечення

Тип вимоги	Назва	Опис
Функціональна	Рух ворогів	вороги повинні мати можливість рухатися в заданих межах гри. Реалізація різних типів рухів, таких як патрулювання та переслідування гравця.
Функціональна	Атака ворогів	визначення атак ворогів залежно від їхнього стану та відстані до гравця. Можливість змінювати тип атаки ворогів для різноманітності геймплею.
Функціональна	Перехід між станами	вороги повинні мати FSM для визначення різних станів, таких як рух, атака, спостереження та патрулювання. Забезпечення зручного переходу між цими станами в залежності від зовнішніх подій, таких як виявлення гравця чи зіткнення із перешкодою.
Нефункціональні	Ефективність	використання Unity 2D та FSM повинно забезпечувати швидку реакцію ворогів на динамічні зміни в грі та гладкий геймплей. Порівняно з іншими середовищами розробки, Unity 2D відзначається великою оптимізацією та ефективністю, що робить його ідеальним вибором для створення ігор у жанрах Metroidvania і Soulslike
Нефункціональні	Гнучкість та легкість налаштування	Unity 2D дозволяє легко налаштовувати візуальні та фізичні параметри, тоді як FSM забезпечить можливість зміни станів та переходів без необхідності втручання у вихідний код. Такий підхід забезпечить оперативну адаптацію гри під змінні вимоги та швидку ітерацію розробки.

Продовження Таблиці 1.1

Нефункціональні	Легкість розширення	завдяки використанню Unity 2D та FSM, система легко розширюється. Додавання нових видів ворогів, станів або типів рухів вимагає мінімальних змін в існуючих компонентах, що полегшує процес розробки та підтримки.
Нефункціональні	Зручність використання	Unity 2D забезпечує інтуїтивний інтерфейс для візуалізації та редагування об'єктів, а FSM дозволяє визначати та регулювати стани та переходи за допомогою зрозумілого та простого інтерфейсу. Це забезпечить зручність та ефективність у використанні системи у процесі розробки гри.

Висновки до першого розділу

Було чітко сформульовано практичне завдання – розробка поведінки ворогів та механік збереження і завантаження для гри в жанрах Metroidvania і Soulslike. Було здійснено аналіз вимог до ігрового інтелекту, що послужило основою для подальшого проектування. Було проведено аналіз та визначення вимог до програмного забезпечення.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		16

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Проектування на основі UML

Діаграма активностей (англ. Activity diagram) - це графічний інструмент моделювання, що дозволяє відображати послідовності дій та процеси у системі. Використання її поширене для моделювання різних видів діяльності, таких як бізнес-процеси, алгоритми та робочі потоки. Діаграма активності показує послідовність дій, які виконуються у системі та їх переходи.

Основні елементи діаграми активності включають:

1. Дії (Actions): прямокутники представлені з заокругленими кінцями. Вони показують виконання конкретних дій у системі. Наприклад, "створити заявку" або "показати повідомлення".

2. Рішення (Decisions): представлені ромбами. Це можна використовувати для переключення між різними шляхами або альтернативами під час проведення процесу.

3. З'єднувальні стрілки (Transitions): це вказує на перехід між різними діями або станами. Вони можуть мати форму стрілок або ліній з позначенням напрямку.

4. Паралельність (Concurrent Actions): користуються спеціальними лініями або пунктирними стрілками, щоб показати одночасне виконання кількох дій.

5. Початок і завершення (Start and End Nodes): перший етап показується за допомогою великого кільця або стрілки, тоді як останній етап позначений кільцем з умисно намальованою "X" або ж прапорцем.

В даному випадку буде розглянуто діаграму активностей ворога - собаки, та його станів (рис. 2.1). Діаграма активностей відображає послідовність дій, які ворог - собака здійснює в різних ігрових ситуаціях. Це допомагає розібратися в поведінці ворога під час гри та зрозуміти, як він реагує на зміни в середовищі або дії гравця.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		17

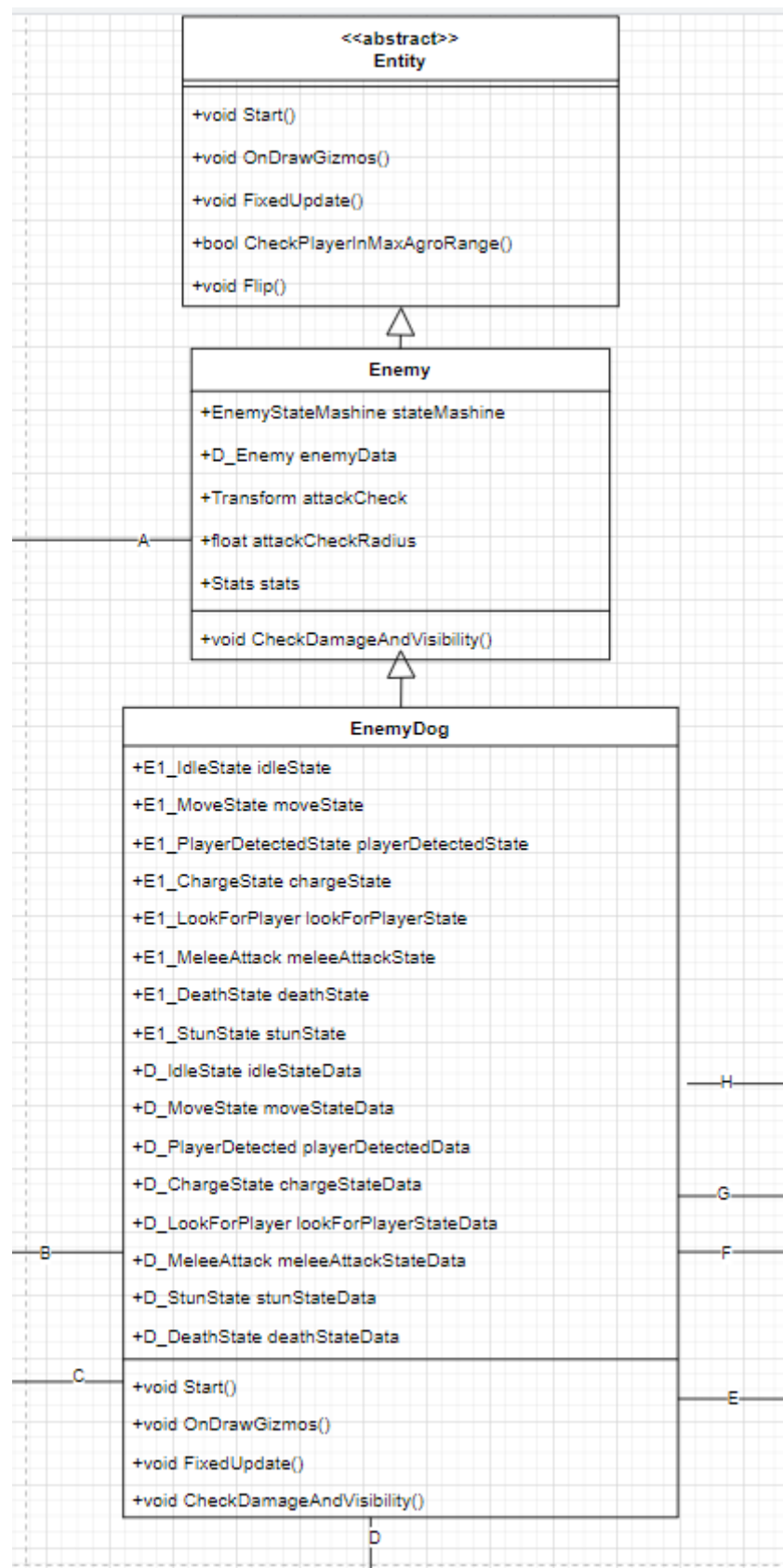


Рисунок 2.2 - Діаграма класів ворога собаки

Конкретний тип ворога представлений класом EnemyDog, який успадковується від Enemy. EnemyDog має конкретні стани (states) ворога, такі

як: E1_IdleState, E1_MoveState, E1_PlayerDetectedState, E1_ChargeState, E1_LookForPlayer, E1_MeleeAttack, E1_DeathState. Ці стани відповідають різним поведінковим моделям ворога. Клас EnemyDog також містить дані для кожного стану, такі як: D_IdleState idleStateData, D_MoveState moveStateData, D_PlayerDetected playerDetectedData, D_ChargeState chargeStateData, D_LookForPlayer lookForPlayerStateData, D_MeleeAttack meleeAttackStateData, D_DeathState deathStateData. Методи класу включають Start(), OnDrawGizmos(), FixedUpdate(), CheckDamageAndVisibility() та інші.

Для управління станами ворога використовується клас EnemyStateMashine. Цей клас містить методи Initialize(EnemyState startingState) для ініціалізації початкового стану та ChangeState(EnemyState newState) для зміни стану. Це дозволяє гнучко керувати переходами між різними станами ворога.

Абстрактний клас EnemyState служить базою для всіх станів ворога. Він містить методи - Enter(), Exit(), LogicUpdate(), PhysicsUpdate() та DoChecks(), які визначають основну логіку та перевірки для кожного стану. Конкретні стани для ворога типу EnemyDog включають: E1_IdleState, E1_MoveState, E1_PlayerDetectedState, E1_ChargeState, E1_LookForPlayer, E1_MeleeAttack, E1_DeathState. Кожен з цих класів має методи - DoChecks(), Enter(), Exit(), LogicUpdate(), PhysicsUpdate(), а деякі з них мають додаткові методи, специфічні для відповідного стану, наприклад, FinishAttack() для E1_MeleeAttack.

Загальні стани, такі як: ChargeState, DeathState, IdleState, LookForPlayerState, MeleeAttackState, MoveState, PlayerDetectedState, AttackState - можуть використовуватися різними типами ворогів. Вони містять специфічні дані та методи, що відповідають за логіку поведінки в даному стані. Наприклад, клас ChargeState містить дані D_ChargeState stateData, які включають параметри chargeSpeed та chargeTime. Інші стани, як-от DeathState, IdleState, LookForPlayerState, мають відповідні дані, такі як D_DeathState stateData, D_IdleState stateData, D_LookForPlayer stateData.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		20

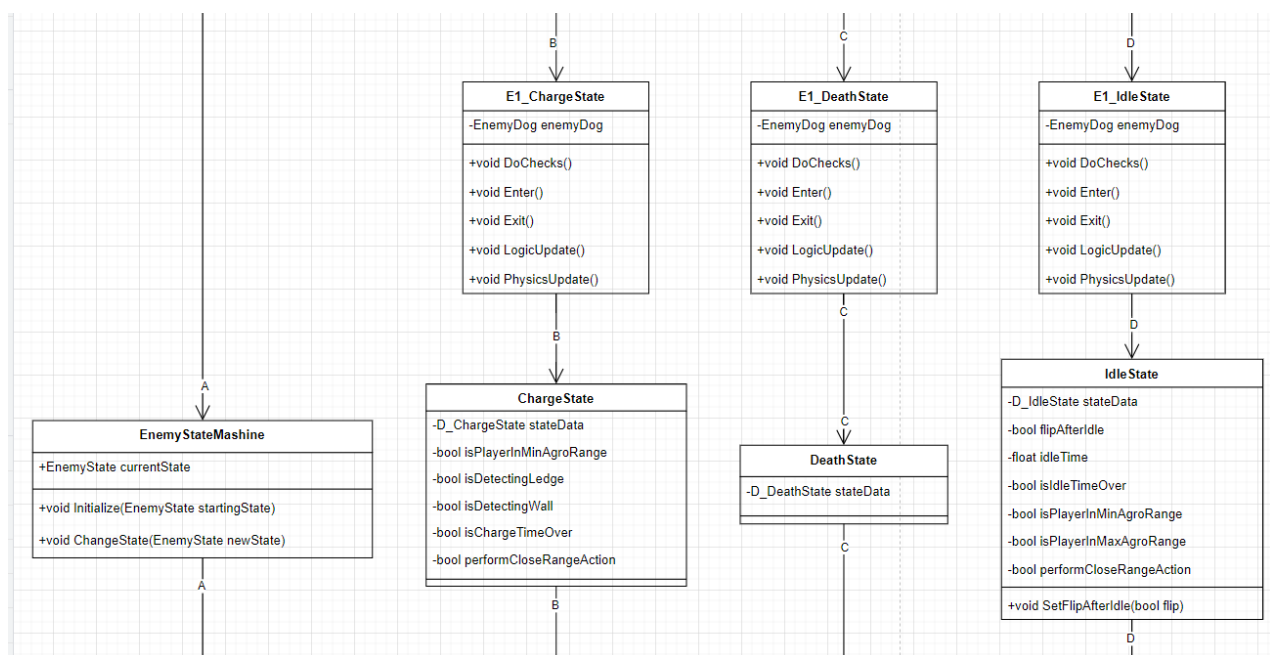


Рисунок 2.3 - Продовження діаграми класів ворога собаки

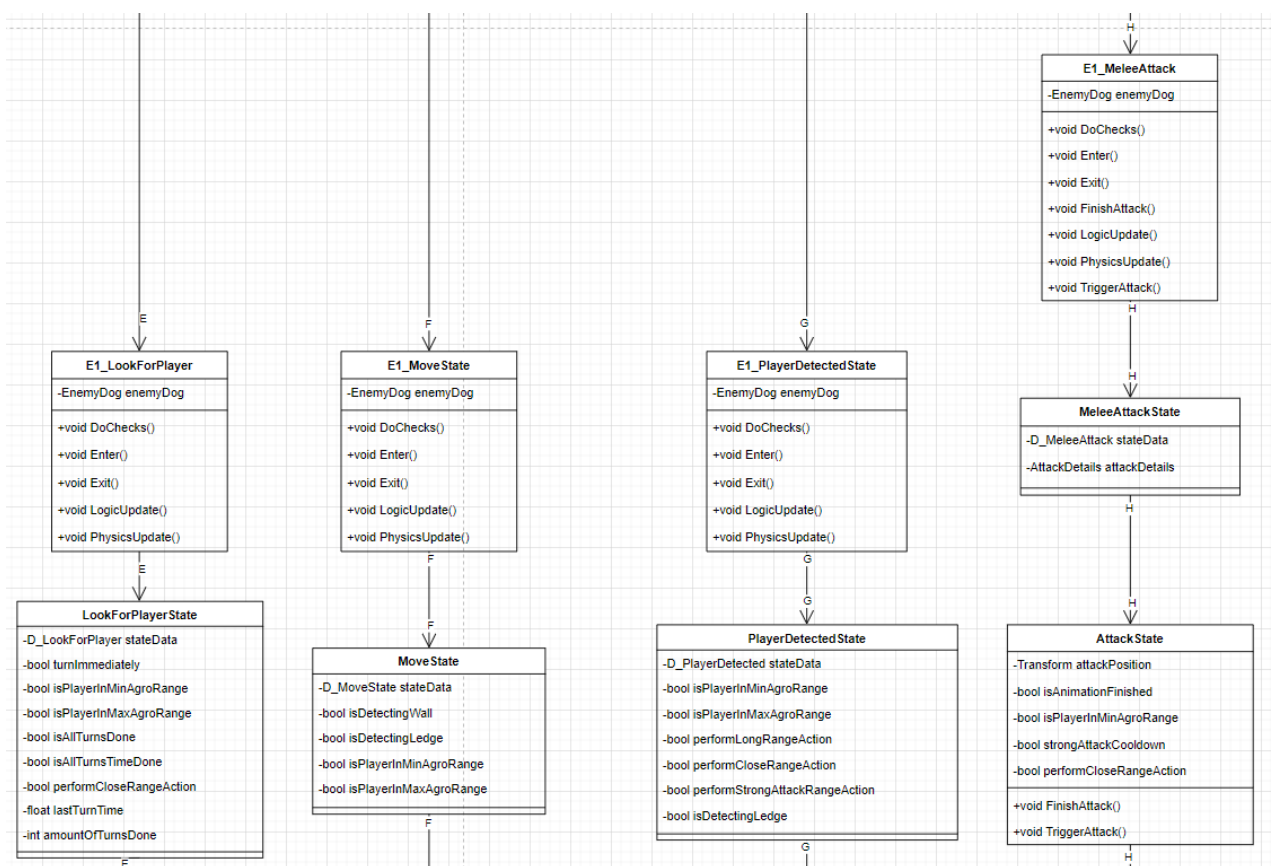


Рисунок 2.4 - Продовження діаграми класів ворога собаки

Класи «D_State» - містять дані для відповідних станів ворогів. Наприклад, D_ChargeState містить параметри chargeSpeed та chargeTime, а D_Enemy включає атрибути maxHealth, damageHopSpeed, enemyAttackDamage,

maxAgroDistance, minAgroDistance, groundCheckRadius, stunResistance, stunRecoveryTime, closeRangeAction-Distance, GameObject hitParticle, LayerMask whatIsPlayer.

Додатково, клас Stats використовується для зберігання статистики ворога, такої як поточне здоров'я (currentHealth) та статус пошкодження (damaged). Клас AttackDetails використовується для деталей атаки ворога.

Взаємозв'язки між класами включають успадкування, агрегацію та композицію, а також використання. Наприклад, Enemy успадковується від Entity, EnemyDog успадковується від Enemy, а станові класи мають посилання на відповідні об'єкти. Це дозволяє ефективно організувати логіку гри та полегшує розробку та підтримку коду.

Ця діаграма класів дозволяє зрозуміти структуру системи ворогів у грі, їх стани та взаємодію між ними.

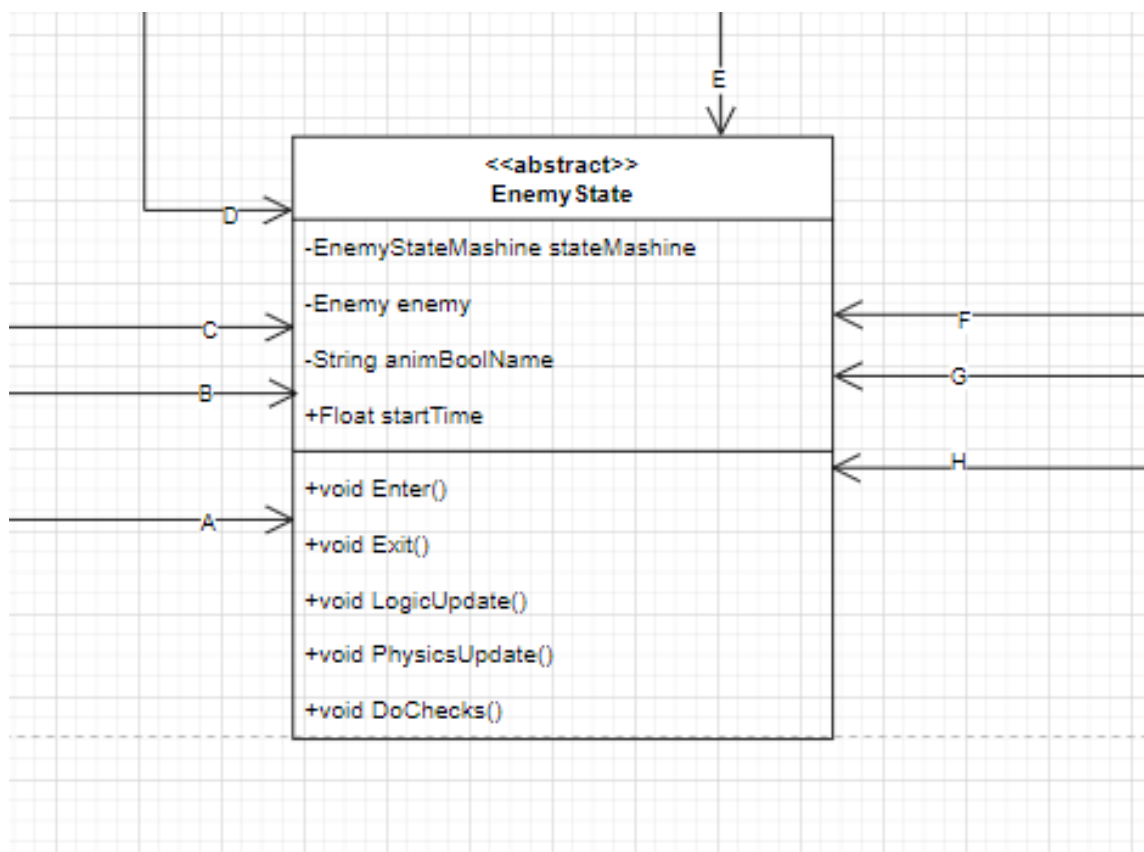


Рисунок 2.5 - Продовження діаграми класів ворога собаки

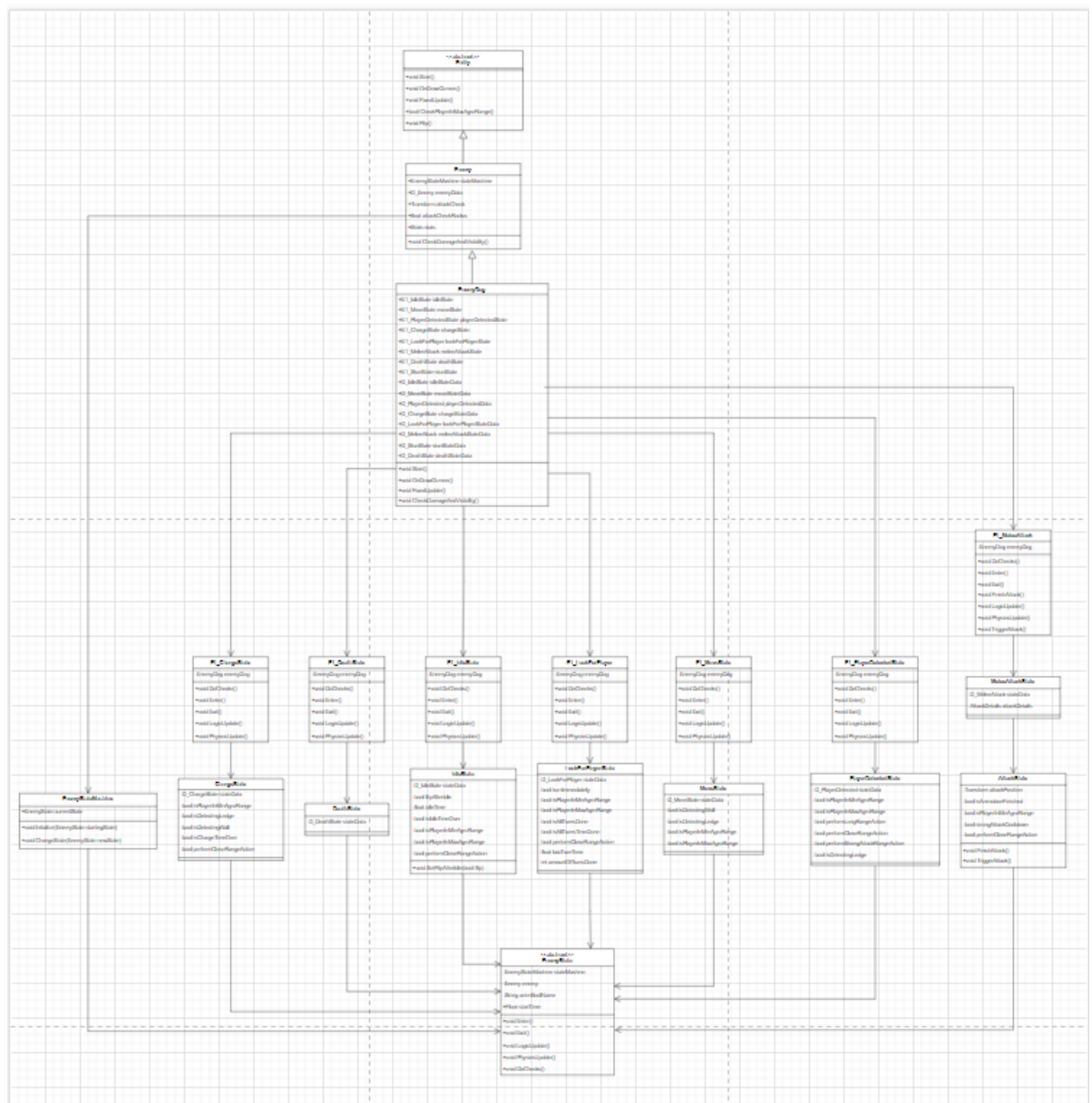


Рисунок 2.6 - Загальний вигляд діаграми класів ворога собаки

Діаграма компонентів є одним з видів діаграм UML, яка використовується для відображення організації та взаємодії між різними компонентами системи. Вона показує, як частини системи взаємодіють між собою та з якими інтерфейсами вони пов'язані. Це корисно для моделювання системи на високому рівні абстракції, де компоненти можуть бути фізичними або логічними частинами системи.

На діаграмі рисунок 2.7 ми розглядаємо структуру, що моделює компоненти об'єкта ворога (GameObject) у грі та їх взаємодію. Центральним елементом є GameObject Ворог, який містить декілька компонентів для

забезпечення різних функцій. GameObject Ворог включає компонент Alive, який відповідає анімації, тобто за визначення стану ворога (яку анімацію використовувати залежно від стану).

Компонент Alive містить кілька підкомпонентів. WallCheck відповідає за перевірку наявності стін і використовується для визначення того, чи зіткнувся ворог зі стіною, що важливо для управління рухом ворога та запобігання його зіткненню зі стінами. GroundCheck відповідає за перевірку наявності землі під ворогом, що важливо для визначення, чи знаходиться ворог на землі або в повітрі, і впливає на його рух та інші дії.

PlayerCheck використовується для перевірки наявності гравця поблизу. Цей компонент дозволяє виявляти гравця та визначати відповідні дії ворога, такі як атака або переслідування. MeleeAttackCheck визначає можливість ближньої атаки, перевіряючи, чи знаходиться гравець у радіусі атаки ворога, що дозволяє ворогу виконувати ближні атаки.

Компоненти взаємодіють один з одним для забезпечення комплексної поведінки ворога. Наприклад, WallCheck може передавати інформацію компоненту GameObject про те, що ворог зіткнувся зі стіною, що може змусити його змінити напрям руху. GroundCheck визначає, чи ворог знаходиться на землі, що може впливати на його здатність стрибати або падати. PlayerCheck дозволяє ворогу виявляти гравця та відповідним чином реагувати, наприклад, переслідувати його або готуватися до атаки. MeleeAttackCheck перевіряє, чи знаходиться гравець у радіусі ближньої атаки, дозволяючи ворогу виконувати атаки у відповідний момент.

Ця діаграма компонентів допомагає розробникам зрозуміти, як різні частини об'єкта ворога взаємодіють між собою. Вона дозволяє виявити залежності між компонентами та забезпечити їх правильну інтеграцію в загальну систему. Також діаграма сприяє виявленню можливих місць для оптимізації та поліпшення коду, полегшуючи підтримку та розширення функціональності системи в майбутньому.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		24

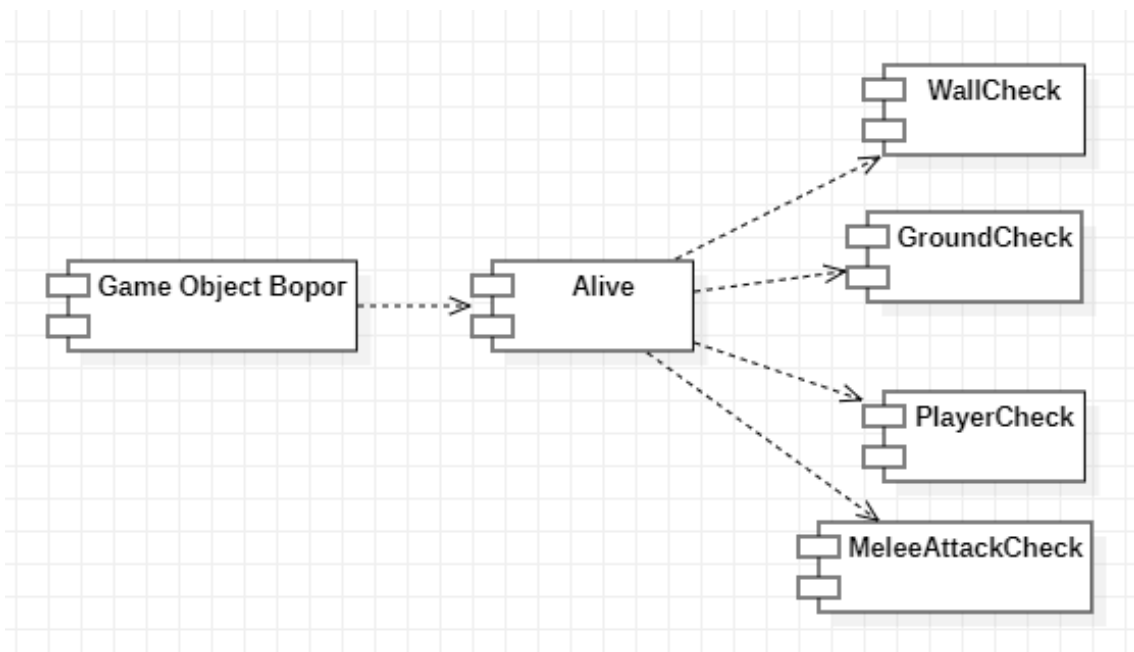


Рисунок 2.7 - Діаграма компонентів ворога близького бою

Діаграма розгортання є важливим інструментом UML, що використовується для візуалізації фізичного розташування компонентів системи на апаратних вузлах. Вона показує, як програмні компоненти розгортаються в реальному середовищі та взаємодіють між собою. Розглянемо діаграму розгортання для гри, розробленої на Unity 2D, яка складається з чотирьох основних блоків.

Блок 1: User

Цей блок представляє користувача, який взаємодіє з грою через модуль *user*. Модуль *user* забезпечує інтерфейс для введення даних, таких як клавіатура, миша або контролер, і відображає вихідну інформацію на екрані. Користувач використовує цей модуль для запуску гри, керування персонажами та взаємодії з ігровим світом.

Блок 2: PC App

Цей блок представляє додаток, який працює на персональному комп'ютері. В ньому знаходиться модуль *PC Application*, який відповідає за виконання основної логіки гри. Всередині модуля *PC Application* знаходиться модуль *game.exe*, який є виконуваним файлом гри. Модуль *game.exe* містить всі необхідні інструкції та дані для запуску і виконання гри на ПК. Він

взаємодіє з операційною системою, обробляє введення користувача та відображає графіку і звук.

Блок 3: Unity Engine

Цей блок представляє ігровий рушій Unity 2D, який забезпечує основні функціональні можливості для розробки і виконання гри. *Unity Engine* складається з кількох модулів:

- *Environment Module*: відповідає за створення та управління ігровим середовищем, включаючи фон, ландшафт, об'єкти та їх розташування.
- *Physics Module*: забезпечує фізичну симуляцію в грі, таку як гравітація, зіткнення та рух об'єктів. Хоча це 2D гра, фізичний модуль все одно відіграє ключову роль в обробці взаємодій між об'єктами.
- *Components Module*: містить різні компоненти, які можна додавати до ігрових об'єктів, щоб надати їм певні функції, такі як рендеринг, анімація, скрипти тощо.
- *Audio Module*: відповідає за обробку звуку в грі, включаючи фонову музику, звукові ефекти та голосові команди.

Блок 4: Enemies

Цей блок представляє ворогів у грі і містить два основні модулі: Basic Enemy AI і Boss AI. Модуль *Basic Enemy AI* відповідає за базову штучну інтелекцію ворогів. Він включає наступні під-модулі:

- *Environment Interaction*: відповідає за взаємодію ворогів з ігровим середовищем. Наприклад, вороги можуть обходити перешкоди, реагувати на зміни в оточенні та адаптувати свою поведінку відповідно до ситуації.
- *Combat System*: відповідає за бойову систему ворогів, включаючи атаки, захист і інші бойові дії. Цей модуль визначає, як вороги атакують гравця та реагують на його дії.

Модуль *Boss AI* відповідає за штучну інтелекцію босів у грі. Він включає наступні підмодулі:

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		26

– *Environment Interaction*: відповідає за взаємодію босів з ігровим середовищем, аналогічно до базових ворогів, але може мати більш складні алгоритми та реакції.

– *Combat System*: відповідає за бойову систему босів, яка є більш складною і розвиненою, ніж у базових ворогів. Вона може включати спеціальні атаки, унікальні стратегії та поведінку, що робить босів важкими противниками для гравця.

Взаємозв'язки між блоками

Користувач взаємодіє з *PC App* через модуль *user*, використовуючи інтерфейс для керування грою. *PC App*, в свою чергу, взаємодіє з *Unity Engine* через модуль *game.exe*, який використовує різні модулі Unity для забезпечення функціонування гри. *Unity Engine* надає базові функції для гри, включаючи обробку середовища, фізики, компонентів і звуку. Вороги, представлені в блоці *Enemies*, використовують можливості *Unity Engine* для реалізації своєї поведінки в грі, включаючи взаємодію з середовищем і бойові дії.

Ця діаграма розгортання допомагає зрозуміти, як різні частини системи взаємодіють між собою на фізичному рівні, забезпечуючи чітке уявлення про розташування та взаємозв'язки між компонентами гри, створеної на Unity 2D.

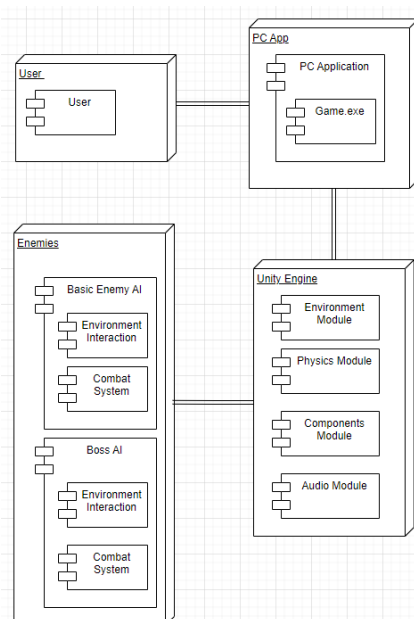


Рисунок 2.8 - Діаграма розгортання

2.2 Розробка архітектури поведінки ворогів

На цьому етапі основний акцент робиться на створенні алгоритму з використанням концепції State Machine [8], яка є потужним інструментом для моделювання станів та переходів у системі. Спочатку визначається тип ворога, який буде розроблятися, і в даному випадку вирішено було розробити ворога на основі собаки або вовка. Для цього типу ворога створюється відповідна схема поведінки, яка буде базою для розробки станів та їх взаємодії. Машина станів використовується для моделювання складної поведінки ворога, в якій керування відбувається за різних умов та дій гравця. На основі цієї архітектури створюється схема поведінки ворога, яка описує всі стани та переходи між ними, детально визначаючи поведінку в різних ситуаціях. Ця модель дозволяє точно визначити стани та переходи, спрощуючи розуміння та управління поведінкою в грі. State Machine є потужним інструментом для моделювання різних станів та переходів між ними у системі. Основна мета полягає в створенні логічної моделі, яка керуватиме поведінкою ворогів у грі.

Визначимо ворога для якого будемо розробляти стани, було вирішено розробити ворога на подібні собаки або вовка, отож він повинен мати можливість: стояти, ходити, бачити гравця, атакувати гравця, переслідувати гравця, шукати гравця, бути приголомшеним та повинен мати здатність помирати. На основі вище оголошених потреб була створена відповідна схема поведінки яка буде підґрунтям для розроблення станів та їх взаємодії (рис. 2.1).

Схема поведінки ворога блиського бою, реалізована за допомогою State Machine, описується наступним чином:

1. Move State:

- Вхід в цей стан визначає початок руху ворога.
- Якщо ворог зіткнувся зі стіною або виступом, він переходить у стан

Idle State, де залишається протягом заданого часу.

2. Idle State:

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		28

- Ворог зупиняється та очікує в цьому стані певний зазначений час.

3. Player Detected State:

- Якщо гравець знаходиться в області видимості ворога, ворог переходить у цей стан.

- Є два варіанти розвитку подій:

- Якщо гравець знаходиться в радіусі атаки, ворог атакує його.
- Якщо гравець знаходиться далеко, ворог переходить у стан

Chase Player State, намагаючись наздогнати його.

4. Chase Player State:

- Ворог активно переслідує гравця, намагаючись його догнати.

5. Look for Player State:

- Коли гравець виходить з зони видимості ворога, ворог переходить у цей стан.

- В цей період ворог розглядає оточення, намагаючись знайти гравця.

- Якщо гравець знаходиться, ворог повертається до стану Player Detected State; в іншому випадку, він активує Move State.

Ця State Machine моделює складну поведінку ворога, яка залежить від різних умов і дій гравця. Такий підхід дозволяє точно визначати стани та переходи між ними, що полегшує розуміння та управління поведінкою в грі чи іншій сценарії. Можна переглянути на рисунку 2.9.

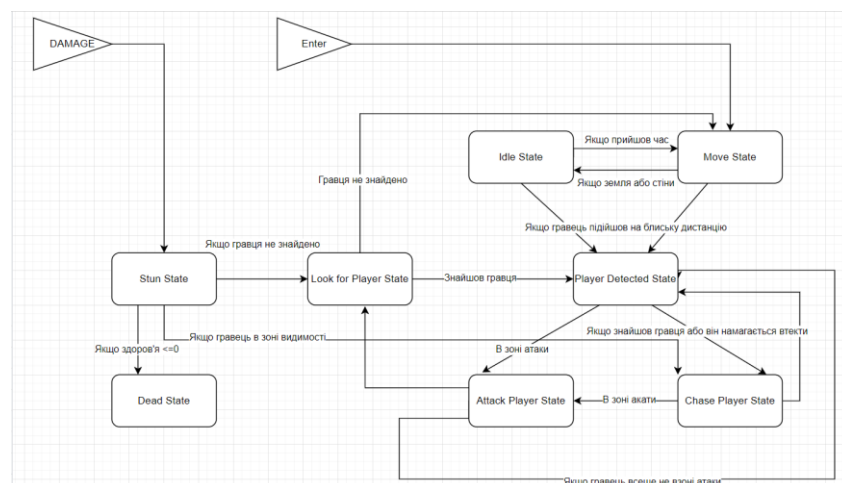


Рисунок 2.9 - Схема поведінки ворога близького бою для реалізації State Machine

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		29

Також буде розроблено ворогів дальнього бою до прикладу візьмемо лучника. Для нього потрібна окрема модель поведінки, отож визначемо, що потрібно лучнику він має: ходити, стояти, бачити гравця, відпригувати від гравця, атакувати гравця з близької дистанції, атакувати з дальньої дистанції, шукати гравця та померати з такими можливостями ворог дальнього бою вийде цікавим та жвавим.

На основі вище оголошених потреб була створена відповідна схема поведінки яка буде підґрунтям для розроблення станів та їх взаємодії.

Схема поведінки ворога блиського бою, реалізована за допомогою State Machine, описується наступним чином:

Move State:

- Вхід в цей стан визначає початок руху ворога.
- Якщо ворог зіткнувся зі стіною або виступом, він переходить у стан Idle State, де залишається протягом заданого часу.

Idle State:

- Ворог зупиняється та очікує в цьому стані певний зазначений час.

Melee Attack:

- Якщо гравець знаходиться у блиській дистанції до ворога і при цьому стан Dodge знаходиться на перезарядці.
- Якщо гравець різко втратився з поля зору ворог переходить в Look for Player.
- Якщо гравець розпочинає втікати від ворогу але все ще в полі зору він переходить в стан Range Attack.

Range Attack:

- Якщо ворог замінив гравця й він знаходиться на дільній дистанції до ворога
- Якщо гравець втік з радіусу видимості ворог переходить в Look for Player.
- Якщо гравець знаходиться блисько, ворог переходить у стан Dodge.

Dodge:

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		30

2.3 Створення системи збереження та завантаження даних

Ця система збереження використовує Unity PlayerPrefs [9-10] для збереження та завантаження даних у форматі JSON. Ось докладний опис того, як вона працює.

SaveManager — це статичний клас, який містить методи для збереження та завантаження даних. Він використовує Unity JsonUtility [11] для серіалізації та десеріалізації об'єктів в JSON-формат, а потім зберігає їх у PlayerPrefs.

Метод *Save<T>*

– **T**: генеричний параметр, що дозволяє використовувати цей метод для збереження будь-якого типу даних.

– **key**: ключ, за яким буде збережено дані у PlayerPrefs.

– **saveData**: дані, які потрібно зберегти.

```
public static void Save<T>(string key, T saveData)
{
    string JSONDataString = JsonUtility.ToJson(saveData, true);
    PlayerPrefs.SetString(key, JSONDataString);
}
```

У цьому методі `JsonUtility.ToJson(saveData, true)` серіалізує об'єкт `saveData` у JSON-рядок. Параметр `true` означає, що JSON буде відформатовано для зручності читання. `PlayerPrefs.SetString(key, JSONDataString)` зберігає цей JSON-рядок у PlayerPrefs під заданим ключем.

Метод *Load<T>*

– **T**: генеричний параметр, який вказує тип даних, що будуть завантажуватися.

– **key**: ключ, за яким зберігаються дані у PlayerPrefs.

```
public static T Load<T>(string key) where T : new()
{
    if (PlayerPrefs.HasKey(key))
    {
        string loadedString = PlayerPrefs.GetString(key);
        return JsonUtility.FromJson<T>(loadedString);
    }
    else
        return new T();
}
```

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		32

}

У цьому методі `PlayerPrefs.HasKey(key)` перевіряє, чи є збережені дані за заданим ключем. Якщо дані існують, `PlayerPrefs.GetString(key)` завантажує JSON-рядок за цим ключем. `JsonUtility.FromJson<T>(loadedString)` десеріалізує JSON-рядок у об'єкт типу `T` і повертає його. Якщо даних за заданим ключем не існує, метод повертає новий об'єкт типу `T`.

Переваги та Недоліки

Переваги:

- Простота використання: система дозволяє легко зберігати та завантажувати дані.
- Гнучкість: використання генериків дозволяє зберігати та завантажувати дані будь-якого типу.
- Легкість у налаштуванні: використання `JsonUtility` дозволяє легко серіалізувати та десеріалізувати дані.

Недоліки:

- Обмеження `PlayerPrefs`: `PlayerPrefs` краще підходить для збереження невеликих обсягів даних. Для великих обсягів даних або складних структур варто використовувати інші методи збереження, наприклад, файли або бази даних.
- Відсутність шифрування: дані зберігаються у відкритому вигляді. Для чутливих даних варто додати шифрування.
- Обмежена підтримка типів: `JsonUtility` не підтримує всі типи даних, такі як словники (`Dictionary`). Для цього можуть знадобитися сторонні бібліотеки, такі як `Newtonsoft.Json`.

Ця система збереження є простою та ефективною для базових потреб, але для складніших задач може знадобитися розширення її можливостей. Для розширення базової системи збереження буде додано численні класи даних, що представляють різні аспекти ігрового стану. Кожен з цих класів серіалізується в JSON і зберігається за допомогою `SaveManager`.

Основні компоненти системи збереження:

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		33

1. Основний клас збереження (*SaveManager*)

```
public static class SaveManager
{
    public static void Save<T>(string key, T saveData)
    {
        string JSONDataString = JsonUtility.ToJson(saveData, true);
        PlayerPrefs.SetString(key, JSONDataString);
    }

    public static T Load<T>(string key) where T : new()
    {
        if (PlayerPrefs.HasKey(key))
        {
            string loadedString = PlayerPrefs.GetString(key);
            return JsonUtility.FromJson<T>(loadedString);
        }
        else
            return new T();
    }
}
```

Розширені класи даних для збереження:

Клас MainSettings

```
[System.Serializable]
public class MainSettings
{
    public bool _isFullScreen;
    public int _currentQualityLevel;
    public float _volumeValue;
    public int _width;
    public int _height;
    public bool[] _isOwned;

    public MainSettings()
    {
        _isFullScreen = true;
        _volumeValue = 0.5f;
    }
}
```

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		34

```

        _currentQualityLevel = 2;
        _width = 1920;
        _height = 1080;
    }
}

```

Цей клас зберігає основні налаштування гри, такі як повноекранний режим, рівень якості, гучність, роздільну здатність екрана і список володінь.

Клас CharaStatistic

```

[System.Serializable]
public class CharaStatistic
{
    public int _maxFlask;
    public int _currentFlask;
    public float _currentHealth;
    public float _maxHealth;
    public float _currentMana;
    public float _maxMana;
    public float _currentUlt;

    public CharaStatistic()
    {
        _maxFlask = 3;
        _currentFlask = _maxFlask;
        _maxMana = 100;
        _maxHealth = 200;
        _currentUlt = 0;
        _currentMana = 0;
        _currentHealth = _maxHealth;
    }
}

```

Цей клас зберігає статистику персонажа, включаючи максимальну і поточну кількість фляг, здоров'я, ману і ультимативну енергію.

Інші класи даних (MoneyPlayer, PlayerPos, WeaponSave, AbilitySave, CharmsSave, TalantsSave, ItemSave, LostMoneySave, LostStatusSave, StatueSave, CameraActivity, BossSave, DoorSave, MapSave, IconsSave, NPCDialogues)

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		35

аналогічно зберігають специфічні аспекти ігрового стану. Кожен клас має відповідні поля і конструктори для ініціалізації цих полів.

Використання системи збереження

Розглянемо приклад збереження та завантаження даних фляг для гри.

Створення ключа збереження

```
private string saveKey = "flasks";
```

Це визначає ключ, за яким будуть зберігатися дані в PlayerPrefs.

Завантаження даних

Метод Load використовується для завантаження даних про фляги з PlayerPrefs.

```
public void Load()
{
    var data = SaveManager.Load<SaveData.CharaStatistic>(saveKey); // Завантажуємо дані з PlayerPrefs
    currentFlasks = data._currentFlask; // Встановлюємо поточну кількість фляг
    maxFlasks = data._maxFlask; // Встановлюємо максимальну кількість фляг
    // Очищуємо всі зображення
    for (int i = 0; i < flaskGUI.images.Length; i++)
    {
        flaskGUI.ClearImage(i); // Викликаємо метод для очищення зображення фляги
    }
    // Заповнюємо зображення відповідно до кількості поточних фляг
    for (int i = 0; i < currentFlasks; i++)
    {
        flaskGUI.FillImage(i); // Викликаємо метод для заповнення зображення фляги
    }
}
```

Пояснення методу завантаження

- Завантаження даних: використовується SaveManager.Load для завантаження даних збереженого об'єкта SaveData.CharaStatistic з ключем saveKey.
- Встановлення поточних і максимальних значень фляг: дані завантажуються і зберігаються в змінних currentFlasks і maxFlasks.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		36

– Очищення зображень: очищуємо всі зображення фляг у графічному інтерфейсі (flaskGUI).

– Заповнення зображень: відповідно до кількості поточних фляг заповнюємо відповідні зображення.

Отримання даних для збереження

Метод GetData використовується для підготовки даних до збереження.

```
private SaveData.CharaStatistic GetData()
{
    var data = new SaveData.CharaStatistic()
    {
        _currentFlask = currentFlasks, // Записуємо поточну кількість фляг
        _maxFlask = maxFlasks, // Записуємо максимальну кількість фляг
    };
    return data; // Повертаємо підготовлений об'єкт даних
}
```

Пояснення методу отримання даних

– Створення об'єкта CharaStatistic: створюємо новий об'єкт SaveData.Chara-Statistic.

– Запис даних: встановлюємо значення для currentFlasks та maxFlasks.

– Повернення даних: повертаємо підготовлений об'єкт даних для подальшого збереження.

Загальна структура збереження та завантаження

1. Збереження

Коли необхідно зберегти дані, викликається метод Save, який використовує GetData для отримання актуальних даних.

```
public void Save()
{
    SaveManager.Save(saveKey, GetData()); // Зберігаємо дані за допомогою SaveManager
}
```

2. Завантаження

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		37

Коли необхідно завантажити дані, викликається метод Load, який завантажує дані та оновлює інтерфейс відповідно до збережених значень.

```
public void Load()
{
    var data = SaveManager.Load<SaveData.CharaStatistic>(saveKey); // Завантажуємо дані
    currentFlasks = data._currentFlask; // Встановлюємо поточну кількість фляг
    maxFlasks = data._maxFlask; // Встановлюємо максимальну кількість фляг
    // Очищуємо всі зображення
    for (int i = 0; i < flaskGUI.images.Length; i++)
    {
        flaskGUI.ClearImage(i); // Очищення зображень
    }
    // Заповнюємо зображення відповідно до кількості поточних фляг
    for (int i = 0; i < currentFlasks; i++)
    {
        flaskGUI.FillImage(i); // Заповнення зображень
    }
}
```

Ця система дозволяє ефективно зберігати та завантажувати дані гри, використовуючи PlayerPrefs для збереження стану гри у форматі JSON. Це забезпечує зручний спосіб збереження різних аспектів ігрового стану, таких як кількість фляг, статистика персонажа та інші дані.

2.4 Реалізація програмної частини State Mashine

Реалізація програмної частини State Machine (машини станів) у грі є ключовим елементом, що дозволяє керувати поведінкою ворогів та інших персонажів. Використання машини станів дає змогу розділити логіку поведінки на окремі стани, що спрощує розробку, налагодження та підтримку коду. Машина станів дозволяє чітко визначити, які дії повинні виконуватись у кожному конкретному стані, а також умови переходу між станами. Це забезпечує більш структурований підхід до програмування поведінки персонажів, сприяє модульності коду та полегшує його розширення новими функціональними можливостями.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		38

У цьому розділі ми розглянемо детальну реалізацію програмної частини State Machine для ворога у грі, включаючи ключові класи EnemyStateMashine та EnemyState. Описані класи демонструють, як можна ефективно керувати різними станами ворога та переходами між ними.

Клас EnemyStateMashine призначений для управління станами ворога. Ця класова структура реалізує основну логіку переходу між різними станами ворога. Основні елементи класу включають поточний стан, методи ініціалізації та зміни станів.

Властивість currentState:

```
public EnemyState currentState { get; private set; }
```

Ця властивість зберігає поточний стан ворога. Властивість має тільки метод доступу (getter) для зовнішніх класів, що забезпечує захищеність і контроль над станами.

1. Method Initialize:

```
public void Initialize(EnemyState startingState)
{
    currentState = startingState;
    currentState.Enter();
}
```

Метод ініціалізації задає початковий стан ворога і викликає метод Enter цього стану. Це дозволяє виконати початкову логіку для заданого стану.

2. Method ChangeState:

```
public void ChangeState(EnemyState newState)
{
    currentState.Exit();
    currentState = newState;
    currentState.Enter();
}
```

Метод для зміни стану. Спочатку викликається метод Exit поточного стану для виконання логіки виходу з цього стану, потім змінюється стан на

новий, після чого викликається метод Enter нового стану для виконання логіки входу.

Клас EnemyState є базовим класом для всіх конкретних станів ворога. Він забезпечує шаблон для створення конкретних станів з відповідною логікою. Основні елементи класу включають методи для входу в стан, оновлення логіки, виходу зі стану та оновлення фізичних властивостей.

Конструктор та методи

1. Конструктор:

```
public EnemyState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName)
{
    this.stateMashine = stateMashine;
    this.enemy = enemy;
    this.animBoolName = animBoolName;
}
```

Конструктор приймає параметри stateMashine, enemy та animBoolName і ініціалізує відповідні поля. stateMashine — це посилання на станкову машину ворога, enemy — це посилання на об'єкт ворога, а animBoolName — це назва булевого параметра анімації [12-13].

2. Метод Enter:

```
public virtual void Enter()
{
    startTime = Time.time;
    enemy.anim.SetBool(animBoolName, true);
    DoChecks();
}
```

Метод Enter виконує логіку, яка повинна бути виконана при вході в стан. Він зберігає час входу, активує відповідну анімацію і виконує початкові перевірки за допомогою методу DoChecks.

3. Метод LogicUpdate:

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

Метод `LogicUpdate` виконує оновлення логіки, яке викликається кожен кадр. Цей метод перевизначається в похідних класах для реалізації специфічної логіки.

4. Метод *Exit*:

```
public virtual void Exit()
{
    enemy.anim.SetBool(animBoolName, false);
}
```

Метод `Exit` виконує логіку, яка повинна бути виконана при виході зі стану. Він деактивує відповідну анімацію.

5. Метод *PhysicsUpdate*:

```
public virtual void PhysicsUpdate()
{
    DoChecks();
}
```

Метод `PhysicsUpdate` викликається для оновлення фізичних властивостей ворога. Він також викликає метод `DoChecks` для перевірки умов.

6. Метод *DoChecks*:

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

Метод `DoChecks` виконує різні перевірки (наприклад, чи знаходиться гравець у зоні досяжності ворога). Цей метод перевизначається в похідних класах для реалізації специфічних перевірок.

Розглянемо детальний приклад реалізації стану ворога, що використовується для його руху — клас `MoveState`. Цей клас є частиною системи машин станів, яка керує поведінкою ворога у грі. Також розглянемо похідний клас `E9_MoveState`, який спеціалізується для конкретного типу ворога — `EnemyAssassin`.

Клас *MoveState*

Клас `MoveState` наслідує від базового класу `EnemyState` і додає логіку для перевірки стану ворога під час руху. Цей клас також містить методи для

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		41

входу в стан, виходу з нього, оновлення логіки і фізичних властивостей ворога.

Опис методів класу MoveState

```
public class MoveState : EnemyState
{
    protected D_MoveState stateData;
    protected bool isDetectingWall;
    protected bool isDetectingLedge;
    protected bool isPlayerInMinAgroRange;
    protected bool isPlayerInMaxAgroRange;

    public MoveState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
        D_MoveState stateData) : base(stateMashine, enemy, animBoolName)
    {
        this.stateData = stateData;
    }
```

- *Конструктор*: ініціалізує станову машину, об'єкт ворога та анімаційну змінну. Приймає додатковий параметр stateData, який містить дані про стан.

```
public override void DoChecks()
{
    base.DoChecks();

    isPlayerInMaxAgroRange = enemy.CheckPlayerInMaxAgroRange();
    isDetectingLedge = enemy.IsGroundDetected();
    isDetectingWall = enemy.IsWallDetected();
    isPlayerInMinAgroRange = enemy.CheckPlayerInMinAgroRange();
}
```

- *Метод DoChecks*: викликає базовий метод DoChecks і виконує перевірки, чи ворог виявив стіну, край платформи або гравця у зонах мінімальної та максимальної агресії.

```
public override void Enter()
{
    base.Enter();

    enemy.SetVelocityEnemy(stateData.movementSpeed);
}
```

- *Метод Enter*: викликає базовий метод Enter і встановлює швидкість ворога для початку руху.

```
public override void Exit()
{
    base.Exit();
}
```

- *Метод Exit*: викликає базовий метод Exit, логіка завершення стану тут відсутня, але її можна додати за необхідності.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		42

```
public override void LogicUpdate()
{
    base.LogicUpdate();
}
```

- *Метод LogicUpdate:* викликає базовий метод LogicUpdate, в якому можна додати специфічну логіку для оновлення стану.

```
public override void PhysicsUpdate()
{
    base.PhysicsUpdate();
}
```

- *Метод PhysicsUpdate:* викликає базовий метод PhysicsUpdate, де можна додати фізичні перевірки.

Клас E9_MoveState

Клас E9_MoveState спеціалізується для ворога типу EnemyAssassin. Він наслідує MoveState та додає специфічну логіку для цього типу ворога.

Опис методів класу E9_MoveState

```
public class E9_MoveState : MoveState
{
    private EnemyAssassin enemyAssassin;

    public E9_MoveState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
        D_MoveState stateData, EnemyAssassin enemyAssassin) : base(stateMashine, enemy, animBoolName,
        stateData)
    {
        this.enemyAssassin = enemyAssassin;
    }
}
```

- *Конструктор:* успадковує базовий конструктор MoveState і додає параметр enemyAssassin, який зберігається для використання у специфічній логіці.

```
public override void DoChecks()
{
    base.DoChecks();
}
```

- *Метод DoChecks:* викликає базовий метод DoChecks. У разі необхідності можна додати додаткові перевірки для enemyAssassin.

```
public override void Enter()
{
    base.Enter();
}
```

- *Метод Enter:* викликає базовий метод Enter. Може бути розширений специфічною логікою для enemyAssassin.

```
public override void Exit()
{
    base.Exit();
}
```

- *Метод Exit:* викликає базовий метод Exit. Може бути розширений специфічною логікою для enemyAssassin.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		43

```

public override void LogicUpdate() {
    base.LogicUpdate();
    if (isPlayerInMinAgroRange)
    {
        stateMashine.ChangeState(enemyAssassin.playerDetectedState);
    }
    else if (isDetectingWall || !isDetectingLedge)
    {
        enemyAssassin.idleState.SetFlipAfterIdle(true);
        stateMashine.ChangeState(enemyAssassin.idleState);
    }
}

```

- *Метод LogicUpdate:* викликає базовий метод LogicUpdate. Додає логіку для перевірки, чи знаходиться гравець у зоні мінімальної агресії. Якщо так, змінює стан на playerDetectedState. Якщо ворог натрапив на стіну або досяг краю платформи, змінює стан на idleState.

```

public override void PhysicsUpdate()
{
    base.PhysicsUpdate();
}

```

- *Метод PhysicsUpdate:* викликає базовий метод PhysicsUpdate. У разі необхідності можна додати додаткову фізичну логіку.

Використання класів

Ці класи демонструють, як можна організувати складну поведінку ворогів у грі за допомогою машини станів. Використання базових і похідних класів дозволяє створювати модульний, легко розширюваний код, який можна адаптувати для різних типів ворогів і поведінки.

Таке структурування допомагає забезпечити високу читабельність коду, легкість його підтримки і можливість повторного використання. В результаті, можна легко додавати нові стани або змінювати існуючі, не порушуючи загальну архітектуру системи.

Клас Entity є базовим для всіх сутностей у грі і містить загальні властивості та методи, які можуть бути використані в різних типах сутностей. Основна функціональність цього класу включає:

1. Перевірки зіткнень з ґрунтом та стінами:

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		44

○ Методи `IsGroundDetected()` та `IsWallDetected()` використовуються для визначення чи знаходиться сутність на ґрунті та чи є перешкода перед нею [14-15].

2. *Логіка відповіді на отримання урону:*

○ Методи `DamageEffect()` та `HitKnockback()` відповідають за відтворення анімації вдару та реалізацію ефекту від удару, такого як відкидання.

3. *Керування властивостями фізики сутності:*

○ Методи `SetVelocity()` та `ZeroVelocity()` використовуються для керування швидкістю сутності та зупинки її руху.

Клас `Enemy` наслідується від `Entity` і розширює його функціонал для конкретного типу сутності - ворога. Він містить додаткові властивості та методи, специфічні для ворога, такі як:

1. *Машину станів для керування поведінкою ворога:*

○ Поле `stateMashine` та методи `Update()` та `FixedUpdate()` використовуються для визначення та виконання поточного стану ворога.

2. *Логіку виявлення гравця:*

○ Методи `CheckPlayerInMinAgroRange()`, `CheckPlayerInMaxAgroRange()` та інші використовуються для виявлення гравця та визначення стану ворога в залежності від відстані до гравця.

3. *Методи для встановлення швидкості ворога:*

○ Методи `SetVelocityEnemy(float velocity)`, `SetVelocityEnemy(int direction, float velocity)` та `SetVelocityEnemy(float velocity, Vector2 angle, int direction)` встановлюють швидкість руху ворога відповідно до заданих параметрів. Вони використовуються для керування рухом ворога у різних ситуаціях, таких як рух вперед, назад або під кутом.

2.5 Розгляд станів для `State Mashine`

У цій кваліфікаційній роботі реалізація станів для `State Machine` є важливим компонентом, який визначає та керує різноманітними станами

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		45

системи. Використання State Machine дозволяє організувати функціонал системи шляхом розділення його на окремі стани та визначення умов переходу між ними. Це сприяє зрозумілості, легкості розробки, розуміння та налагодження системи.

Наразі проведеться детальний аналіз та реалізація програмної частини станів для State Machine. Розглядаються ключові класи та їх взаємодія з використанням сучасних підходів до програмування, які спрямовані на підвищення ефективності та надійності системи. Описані методи та прийоми демонструють, як шляхом реалізації станів можна оптимізувати роботу програмного забезпечення, забезпечуючи його стабільність та гнучкість у змінних умовах експлуатації. Всього було написано 25 станів для ворогів:

1. IdleState: стан бездіяльності, коли ворог не рухається.
2. MoveState: стан руху, коли ворог переміщується по карті.
3. PlayerDetectedState: стан, коли ворог виявляє гравця.
4. ChargeState: стан атаки, коли ворог кидається на гравця.
5. StunState: стан оглушення, коли ворог не може рухатися.
6. StrongMeleeAttack: стан сильної ближньої атаки.
7. RangeTripleTargetAttackState: стан атаки на трьох цілей з дальньої відстані.
8. RangeTargetAttackState: стан атаки на одну ціль з дальньої відстані.
9. RangeAttackState: загальний стан атаки з дальньої відстані.
10. MeleeAttackState: загальний стан ближньої атаки.
11. LookForPlayerState: стан пошуку гравця.
12. LazerAttackState: стан атаки лазером.
13. JumpAttackState: стан атаки стрибком.
14. JumpAttackDone: стан завершення атаки стрибком.
15. DodgeState: стан ухилення від атаки.
16. DebatState: стан коли ворог отримує негативні ефекти.
17. DeathState: стан смерті.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		46

18. DashState: стан ривка.
19. DangleState: стан бовтання на мотузці.
20. BossShieldState: стан захисного щита боса.
21. BossRollState: стан перекидання боса.
22. BossRollIsOverState: стан завершення перекидання боса.
23. BackTeleportState: стан телепортації за спину гравця.
24. AttackState: загальний стан атаки.
25. AppearenceState: стан появи на карті.

Буде детально розглянуто роботу деяких з цих станів, зокрема: IdleState, MoveState, PlayerDetectedState, AttackState, MeleeAttackState, RangeAttackState.

Детальний розгляд станів

MoveState (Стан руху)

```
public class MoveState : EnemyState
{
    protected D_MoveState stateData;
    protected bool isDetectingWall;
    protected bool isDetectingLedge;
    protected bool isPlayerInMinAgroRange;
    protected bool isPlayerInMaxAgroRange;

    public MoveState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
        D_MoveState stateData) : base(stateMashine, enemy, animBoolName)
    {
        this.stateData = stateData;
    }
}
```

Цей стан відповідає за рух ворога. Він зберігає дані про те, чи виявляє ворог стіну або виступ (isDetectingWall, isDetectingLedge).

```
public override void DoChecks()
{
    base.DoChecks();
    isPlayerInMaxAgroRange = enemy.CheckPlayerInMaxAgroRange();
    isDetectingLedge = enemy.IsGroundDetected();
    isDetectingWall = enemy.IsWallDetected();
    isPlayerInMinAgroRange = enemy.CheckPlayerInMinAgroRange();
}
```

Під час перевірок він використовує методи ворога для перевірки оточення і знаходження гравця.

```
public override void Enter()
{
    base.Enter();
}
```

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		47

```
enemy.SetVelocityEnemy(stateData.movementSpeed); }
```

При вході в стан, встановлюється швидкість ворога на основі даних зі скрипту D_MoveState.

IdleState (Стан бездіяльності)

```
public class IdleState : EnemyState
{
    protected D_IdleState stateData;
    protected bool flipAfterIdle;
    protected float idleTime;
    protected bool isIdleTimeOver;
    protected bool isPlayerInMinAgroRange;
    protected bool isPlayerInMaxAgroRange;
    protected bool performCloseRangeAction;

    public IdleState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName, D_IdleState stateData) : base(stateMashine, enemy, animBoolName)
    {
        this.stateData = stateData;
    }

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

        isPlayerInMinAgroRange = enemy.CheckPlayerInMinAgroRange();
        isPlayerInMaxAgroRange = enemy.CheckPlayerInMaxAgroRange();
        performCloseRangeAction = enemy.CheckPlayerInCloseRangeAction();
    }
}
```

Цей стан відповідає за бездіяльність ворога. Він зберігає таймер бездіяльності (idleTime) і випадковим чином встановлює новий час бездіяльності за допомогою методу SetRandomIdleTime.

```
public override void Enter()
{
    base.Enter();

    enemy.SetVelocityEnemy(0f);

    isIdleTimeOver = false;

    SetRandomIdleTime();
}

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

    if (flipAfterIdle)
    {
        enemy.Flip();
    }
}

public override void LogicUpdate()
```

```

{    base.LogicUpdate();
    if (Time.time > startTime + idleTime)
    {        isIdleTimeOver = true;    }    }
public void SetFlipAfterIdle(bool flip)
{    flipAfterIdle = flip;    }
public void SetRandomIdleTime()
{    idleTime = Random.Range(stateData.minIdleTime,stateData.maxIdleTime);    }

```

Ворог може перевертатися після виходу з цього стану (flipAfterIdle). Під час перевірок стан використовує методи ворога для визначення близькості гравця.

PlayerDetectedState (Стан виявлення гравця)

```

public class PlayerDetectedState : EnemyState
{
    protected D_PlayerDetected stateData;
    protected bool isPlayerInMinAgroRange;
    protected bool isPlayerInMaxAgroRange;
    protected bool performLongRangeAction;
    protected bool performCloseRangeAction;
    protected bool performStrongAttackRangeAction;
    protected bool isDetectingLedge;

    public PlayerDetectedState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
D_PlayerDetected stateData) : base(stateMashine, enemy, animBoolName)
    {
        this.stateData = stateData;    }

```

Цей стан відповідає за поведінку ворога, коли він виявляє гравця. Він зберігає дані про те, чи гравець знаходиться в межах досяжності для різних типів атак.

```

public override void DoChecks()
{
    base.DoChecks();
    isPlayerInMinAgroRange = enemy.CheckPlayerInMinAgroRange();
    isPlayerInMaxAgroRange = enemy.CheckPlayerInMaxAgroRange();
    isDetectingLedge = enemy.IsGroundDetected();
    performCloseRangeAction = enemy.CheckPlayerInCloseRangeAction();    }

public override void Enter()
{
    base.Enter();
    enemy.rb.bodyType = RigidbodyType2D.Dynamic;
    enemy.SetVelocityEnemy(0f);
    performLongRangeAction = false;    }

```

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		49

```

public override void LogicUpdate()
{
    base.LogicUpdate();
    if(Time.time >= startTime + stateData.longRangeActionTime)
    {
        performLongRangeAction = true;
    }
}

```

Після встановленого часу запускається дальня атака (longRangeActionTime). Під час перевірок використовуються методи ворога для визначення оточення та стану атаки.

AttackState (Базовий клас для станів атаки)

```

public class AttackState : EnemyState
{
    protected Transform attackPosition;
    protected bool isAnimationFinished;
    protected bool isPlayerInMinAgroRange;
    protected bool strongAttackCoolwodn;
    protected bool performCloseRangeAction;

    public AttackState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName, Transform attackPosition) : base(stateMashine, enemy, animBoolName)
    {
        this.attackPosition = attackPosition;
    }

    public override void DoChecks()
    {
        base.DoChecks();
        performCloseRangeAction = enemy.CheckPlayerInCloseRangeAction();
        isPlayerInMinAgroRange = enemy.CheckPlayerInMinAgroRange();
    }
}

```

Цей клас є основою для інших класів станів атаки. Він зберігає посилання на точку атаки (attackPosition) і стан анімації атаки (isAnimationFinished). Він перевіряє, чи гравець знаходиться поблизу для ближньої атаки (isPlayerInMinAgroRange).

```

public override void Enter()
{
    base.Enter();
    enemy.atism.attackState = this;
    isAnimationFinished = false;
    enemy.SetVelocityEnemy(0f);
}

```

Віртуальні функції TriggerAttack та FinishAttack реалізуються в дочірніх класах для конкретної поведінки атаки.

```

public virtual void TriggerAttack()
{
}

```

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		50

```
public virtual void FinishAttack()
{
    isAnimationFinished = true;
}
```

MeleeAttackState (Стан ближньої атаки)

```
public class MeleeAttackState : AttackState
{
    protected D_MeleeAttack stateData;
    protected AttackDetails attackDetails;

    public MeleeAttackState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
    Transform attackPosition, D_MeleeAttack stateData ) : base(stateMashine, enemy, animBoolName, attackPosition)
    {
        this.stateData = stateData;
    }

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

        enemy.rb.bodyType = RigidbodyType2D.Dynamic;

        attackDetails.damageAmount = stateData.attackDamage;

        attackDetails.position = enemy.rb.transform.position;
    }
}
```

Цей стан відповідає за ближню атаку ворога. Він використовує дані зі скрипту D_MeleeAttack для визначення радіуса атаки (attackRadius) та сили атаки (attackDamage).

```
public override void TriggerAttack()
{
    base.TriggerAttack(); Collider2D[] detectedObjects = Physics2D.OverlapCircleAll(attackPosition.position,
    stateData.attackRadius, stateData.whatIsPlayer);
    foreach (Collider2D collider in detectedObjects)
    {
        PlayerStats target = collider.GetComponent<PlayerStats>();
        enemy.stats.DoDamage(target);
    }
}
```

Ворог завдає шкоди гравцям у межах радіуса атаки під час виконання анімації атаки (TriggerAttack).

RangeAttackState (Стан дальньої атаки)

```
public class RangeAttackState : AttackState
{
    protected D_RangeAttackState stateData;
    protected GameObject projectile;
    protected Projectile projectileScript;
    protected bool isPlayerInMaxAgroRange;

    public RangeAttackState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
    Transform attackPosition, D_RangeAttackState stateData) : base(stateMashine, enemy, animBoolName,
    attackPosition)
    {
        this.stateData = stateData;
    }

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

        isPlayerInMaxAgroRange = enemy.CheckPlayerInMaxAgroRange();
    }
}
```

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		51

Цей стан відповідає за дальню атаку ворога. Він використовує дані зі скрипту `D_RangeAttackState` для отримання снаряда (`projectile`), швидкості снаряда (`projectileSpeed`), відстані польоту снаряда (`projectileTravelDistance`) та його шкоди (`projectileDamage`).

```
public override void TriggerAttack()
{
    base.TriggerAttack();

    projectile = GameObject.Instantiate(stateData.projectile, attackPosition.position,
    attackPosition.rotation);

    projectileScript = projectile.GetComponent<Projectile>();

    projectileScript.FireProjectile(stateData.projectileSpeed, stateData.projectileTravelDistance,
    stateData.projectileDamage);
}
}
```

Ворог стріляє снарядом у напрямку гравця під час виконання анімації атаки (`TriggerAttack`).

Скрипти з даними станів (Data ScriptableObjects)

- `D_MoveState`: містить дані про стан руху, такі як швидкість руху ворога.
- `D_IdleState`: містить дані про стан бездіяльності, такі як мінімальний та максимальний час бездіяльності.
- `D_PlayerDetected`: містить дані про стан виявлення гравця, зокрема час, після якого ворог здійснює дальню атаку.
- `D_MeleeAttack`: містить дані про стан ближньої атаки, такі як радіус ураження, сила атаки та шар, що визначає, що вважається гравцем.
- `D_RangeAttackState`: містить дані про стан дальньої атаки, такі як об'єкт снаряда, його силу пошкодження, швидкість та дальність польоту.

Переваги використання цих скриптів:

- *Налаштування поведінки ворога*: ви можете змінювати значення цих змінних в інспекторі Unity, щоб налаштувати швидкість руху, силу атаки тощо.
- *Роз'єднання даних від коду*: ці скрипти дозволяють змінювати дані станів без необхідності змінювати код станів.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		52

– *Покращена читабельність коду:* код станів стає більш читабельним, оскільки логіка поведінки чітко відокремлена від коду станів.

2.6 Налаштування аніматора та ворога

Буде розглянуто клас `EnemyAssassin`, що відповідає за управління поведінкою ворожого асасина у відеогрі. Клас відповідає за належну координацію анімацій та дій ворога в грі, забезпечуючи плавний та реалістичний геймплей. У цьому контексті буде розглянуто як поля, методи, так і зв'язок класу з рештою системи управління грою та графічним двигуном Unity.

У наведеному коді описується клас `EnemyAsassin`, який наслідує від базового класу `Enemy`. Цей клас визначає поведінку ворога в грі, використовуючи різні стани для керування його анімаціями та логікою.

1. *Оголошення станів та даних:*

○ Клас `EnemyAsassin` оголошує приватні та публічні змінні для різних станів ворога, таких як `idleState`, `moveState`, `deathState` тощо.

```
public class EnemyAsassin : Enemy
{
    protected Player player;

    public E9_IdleState idleState { get; private set; }
    public E9_DeathState deathState { get; private set; }
    public E9_DodgeState dodgeState { get; private set; }
    public E9_LookForPlayerState lookForPlayerState { get; private set; }
    public E9_MeleeAttackState meleeAttackState { get; private set; }
}
```

○ Також оголошуються змінні для зберігання даних цих станів, такі як `idleStateData`, `moveStateData` тощо.

```
[SerializeField] private D_IdleState idleStateData;
[SerializeField] private D_MoveState moveStateData;
[SerializeField] private D_PlayerDetected playerDetectedData;
[SerializeField] private D_LookForPlayer lookForPlayerData;
[SerializeField] private D_MeleeAttack meleeAttackData;
```

2. *Method Start:*

○ В методі `Start` викликається базовий метод `Start` класу `Enemy`.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		53

- Ініціалізується змінна `player`, яка знаходить об'єкт типу `Player` в сцені.

```
protected override void Start()
{
    base.Start();

    player = Transform.FindAnyObjectByType<Player>();

    moveState = new E9_MoveState(stateMashine, this, "move", moveStateData, this);

    idleState = new E9_IdleState(stateMashine, this, "idle", idleStateData, this);

    playerDetectedState = new E9_PlayerDetectedState(stateMashine, this, "playerDetected",
    playerDetectedData, this);

    meleeAttackState = new E9_MeleeAttackState(stateMashine, this, "meleeAttackOne", attackCheck,
    meleeAttackData, this);

    lookForPlayerState = new E9_LookForPlayerState(stateMashine, this, "lookForPlayer",
    lookForPlayerData, this);

    deathState = new E9_DeathState(stateMashine, this, "death", deathStateData, this);

    stateMashine.Initialize(dangeState);
}
```

- Ініціалізуються всі стани ворога за допомогою відповідних конструкторів. Наприклад, `moveState = new E9_MoveState(stateMashine, this, "move", moveStateData, this);`.

- В кінці методу ініціалізується машина станів, встановлюючи початковий стан (`dangeState`).

```
protected override void FixedUpdate()
{
    base.FixedUpdate();

    CheckDamageAndVisibility();

    if (stats.currentHealth <= 0)
    {
        stateMashine.ChangeState(deathState);
    }
}
```

3. Метод *FixedUpdate*:

- В цьому методі, крім виклику базового методу `FixedUpdate`, перевіряється стан здоров'я ворога. Якщо здоров'я дорівнює нулю або менше, машина станів переходить у стан `deathState`.

```
private void CheckDamageAndVisibility()
{
    if (stats.damaged && !CheckPlayerInMaxAgroRange())
    {
        stateMashine.ChangeState(lookForPlayerState);

        stats.damaged = false;
    }
}
```

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		54

}

4. Метод *CheckDamageAndVisibility*:

○ Метод перевіряє, чи отримав ворог ушкодження (stats.damaged) і чи знаходиться гравець в зоні агресії. Якщо ворог отримав ушкодження і гравець не в зоні агресії, машина станів переходить у стан lookForPlayerState.

```
protected override void OnDrawGizmos()
{
    base.OnDrawGizmos();
    Gizmos.DrawWireSphere(attackCheck.position, attackCheckRadius);
    Gizmos.DrawLine(playerUnderCheck.position, playerUnderCheck.position + (Vector3)(Vector2.down
* checkPlayerUnderDistance));
}
```

5. Метод *OnDrawGizmos*:

○ В цьому методі відображаються візуальні підказки для зони атаки та перевірки наявності гравця під ворогом.

Взаємодія з аніматором

Кожен стан ворога (наприклад, E9_MoveState, E9_IdleState, E9_MeleeAttackState) наслідуює відповідний базовий стан і має методи для керування анімаціями.

Метод Enter: Викликається при вході в стан і активує відповідну анімацію за допомогою аніматора, наприклад:

```
public override void Enter()
{
    base.Enter();
    enemy.anim.SetBool(animBoolName, true);
}
```

Метод Exit: Викликається при виході зі стану і деактивує анімацію, наприклад:

```
public override void Exit()
{
    base.Exit();
    enemy.anim.SetBool(animBoolName, false);
}
```

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		55

Метод LogicUpdate: Виконує логіку стану, наприклад, перевіряє умови для переходу до інших станів і керує анімаціями відповідно.

Приклади використання аніматора у станах

E9_MeleeAttackState

```
public class E9_MeleeAttackState : AttackState
{
    // інші змінні і методи

    public override void TriggerAttack()
    {
        base.TriggerAttack();
        enemy.anim.SetTrigger("meleeAttack");
        // логіка атаки
    }

    public override void FinishAttack()
    {
        base.FinishAttack();
        enemy.anim.SetBool("animBoolName", false);
    }
}
```

У цьому стані метод TriggerAttack викликає анімацію атаки, а метод FinishAttack визначає, коли атака завершена і коли слід перейти до іншого стану.

E9_DeathState

```
public class E9_DeathState : EnemyState
{
    // інші змінні і методи

    public override void Enter()
    {
        base.Enter();
        enemy.anim.SetBool("isDead", true);
    }
}
```

```

public override void Exit()
{
    base.Exit();
    enemy.anim.SetBool("isDead", false);
}
}

```

У цьому стані метод Enter встановлює анімацію смерті, а метод Exit очищає стан.

Переваги цього підходу

Цей підхід дозволяє створити гнучку та модульну структуру для керування різними станами ворога та його анімаціями, що полегшує розробку і тестування ігрової логіки.

Приклад ініціалізації станів

```

public E9_MoveState moveState { get; private set; }

void Start()
{
    moveState = new E9_MoveState(stateMachine, this, "move", moveStateData, this);
    // ініціалізація інших станів
}

public class E9_MoveState : MoveState
{
    private EnemyAsassin enemyAsassin;

    public E9_MoveState(EnemyStateMachine stateMachine, Enemy enemy, string animBoolName,
        D_MoveState stateData, EnemyAsassin enemyAsassin) : base(stateMachine, enemy, animBoolName, stateData)
    {
        this.enemyAsassin = enemyAsassin;
    }
}

public class MoveState : EnemyState
{
    protected D_MoveState stateData;

    public MoveState(EnemyStateMachine stateMachine, Enemy enemy, string animBoolName,
        D_MoveState stateData) : base(stateMachine, enemy, animBoolName)
    {
        this.stateData = stateData;
    }
}

```

Оскільки використано конструктор для ініціалізації станів, нам навіть не потрібно нічого додатково оголошувати. Просто підставляючи назву анімації в конструкторі, ми автоматично встановлюємо true при вході в стан і false при виході з нього. Це забезпечує плавний перехід між анімаціями та спрощує керування логікою ворога.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		57

```

public abstract class EnemyState
{
    protected EnemyStateMachine stateMachine;
    protected Enemy enemy;
    protected string animBoolName;
    protected float startTime;

    public EnemyState(EnemyStateMachine stateMachine, Enemy enemy, string animBoolName)
    {
        this.stateMachine = stateMachine;
        this.enemy = enemy;
        this.animBoolName = animBoolName;
    }
}

```

Цей підхід дозволяє створити гнучку та модульну структуру для керування різними станами ворога та його анімаціями. Використання машин станів (State Machines) разом із чітко визначеними даними станів забезпечує легкість в управлінні складною поведінкою ворога. Кожен стан має свої власні методи для обробки входу (Enter), виходу (Exit), оновлення логіки (LogicUpdate) та фізичних властивостей (PhysicsUpdate), що забезпечує ізоляцію логіки кожного стану та уникнення небажаних перетинів між ними.

Така структура сприяє полегшенню розробки ігрової логіки, оскільки кожен стан легко модифікувати або розширювати, не впливаючи на інші частини коду. Це дозволяє розробникам легко додавати нові стани або змінювати існуючі, швидко адаптуючи поведінку ворога до нових вимог або умов гри.

Крім того, модульний підхід сприяє поліпшенню тестування. Оскільки кожен стан є окремою сутністю, його можна тестувати незалежно від інших станів, що знижує ймовірність виникнення помилок і полегшує діагностику та виправлення проблем. Завдяки чіткій структурі коду і зрозумілим залежностям між станами, можна легко знаходити і виправляти помилки, а також оптимізувати код для кращої продуктивності.

Висновки до другого розділу

У цьому розділі було розглянуто кілька ключових аспектів розробки програмного забезпечення для ігрового проекту. Починаючи з проектування

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		58

на основі UML та розробки архітектури поведінки ворогів, ми перейшли до створення системи збереження та завантаження даних.

Було детально розглянуто реалізацію програмної частини State Machine, що є ключовою складовою для керування поведінкою ворогів у грі. Це дозволило нам легко організувати та керувати станами ворогів, щоб забезпечити їхню різноманітну та реалістичну поведінку.

Окремо було розглянуто стани для State Machine, де детально проаналізували різні стани ворогів, такі як рух, бездіяльність, атака тощо. Це сприяло більш глибокому розумінню процесу взаємодії ворогів з гравцем та середовищем у грі.

Останній аспект, який було розглянуто, це налаштування аніматора та ворога. Це важлива частина процесу розробки, оскільки анімація гри відіграє ключову роль у створенні іммерсивного геймплею.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		59

РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ

3.1 Робота State mashine босів локацій

Найголовнішим ворогом на локації є її бос який зустрічається на фінальній стадії дослідження, тобто є концем рівня саме тому робота їх State Machine має бути найбільш комплексною та гнучкою. Кожен бос має дві стадії [16, 17] перехід до другої відбувається при нанесенні шкоди в 50% від його максимального здоров'я. Кожна стадія включає в себе свої унікальні стани, завдяки чому змінюється патерн поведінки ворога і бій стає більш цікавим та динамічним.

Бос першої локації “Forest Keeper” налічує в собі 9 станів, сім з яких на першій стадії і два додаткових на другій. До того як гравець потрапляє в зону битви з босом він знаходиться в стані Idle очікуючи (Рис 3.1). Коли починається перша стадія бою бос використовує простий патерн атак, а саме переكاتи (roll) і одиничні постріли (rangeAttack), як зображено на рисунках 3.2 – 3.3. Перемекання між станами залежить від таймеру та відстані гравця до ворога.

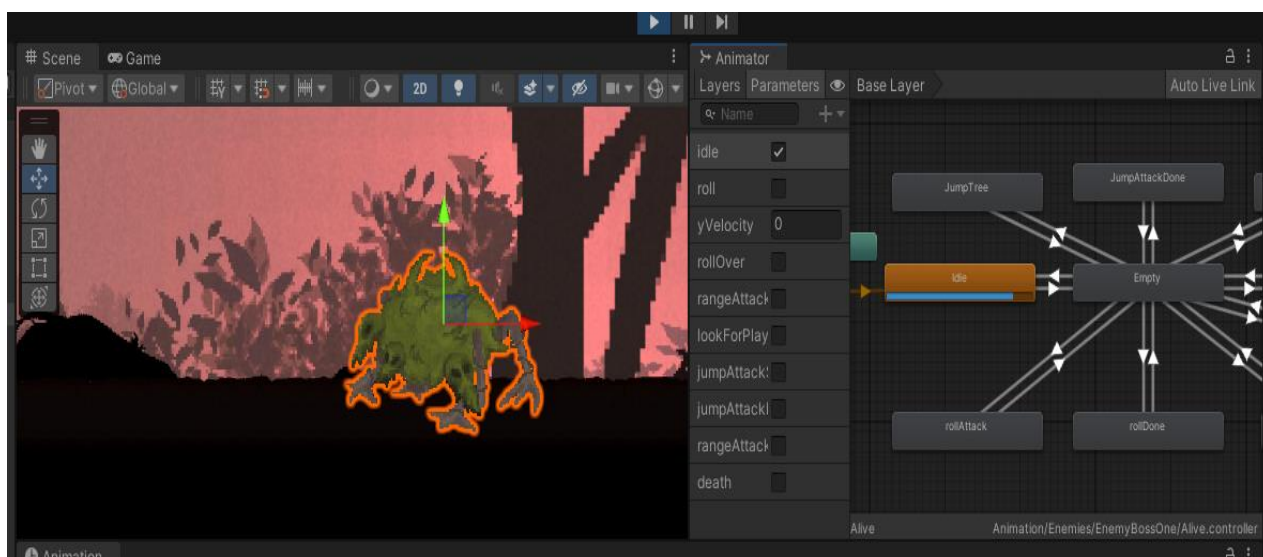


Рисунок 3.1 - Стан Idle

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		60

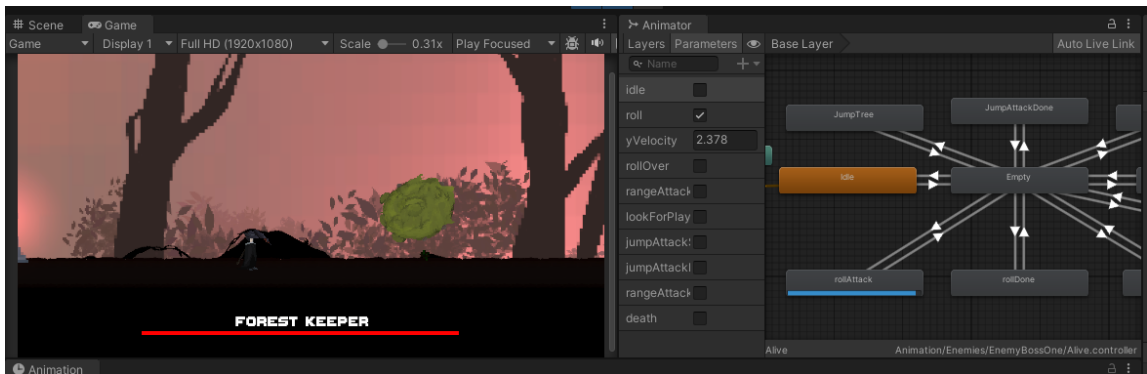


Рисунок 3.2 - Стан roll

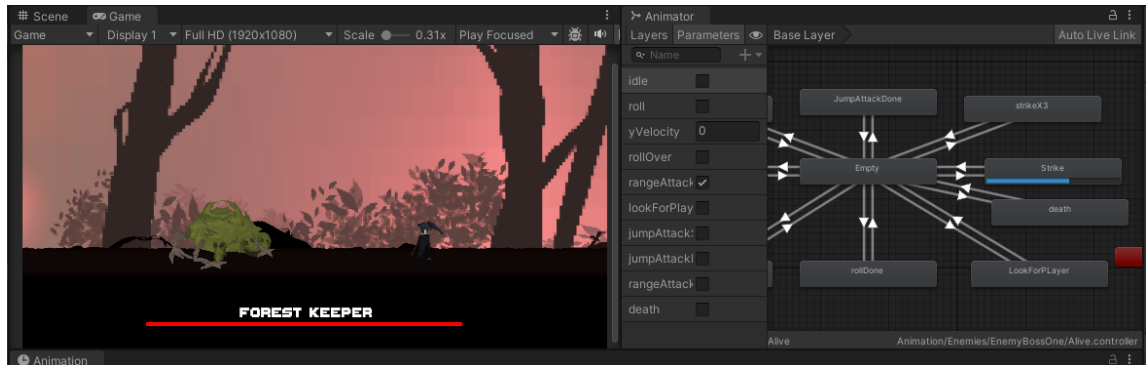


Рисунок 3.3 - Стан rangeAttack

У другій стадії з'являються такі стани як: rangeAttackX, jumpAttackState. Перший стан відповідає за потрібний постріл – поліпшену версію rangeAttack, а другий стан це нова атака – стрибок з повітря (рис. 3.4 – 3.5). Перемекання між станами працює аналогічно до першої стадії. Після перемоги над босом вмикається стан смерті (deathState) (рис. 3.6).

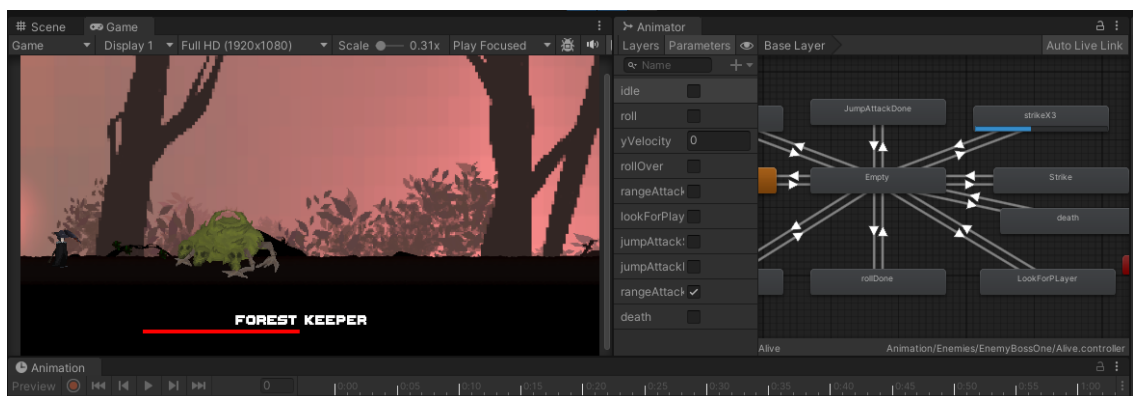


Рисунок 3.4 - Стан rangeAttackX

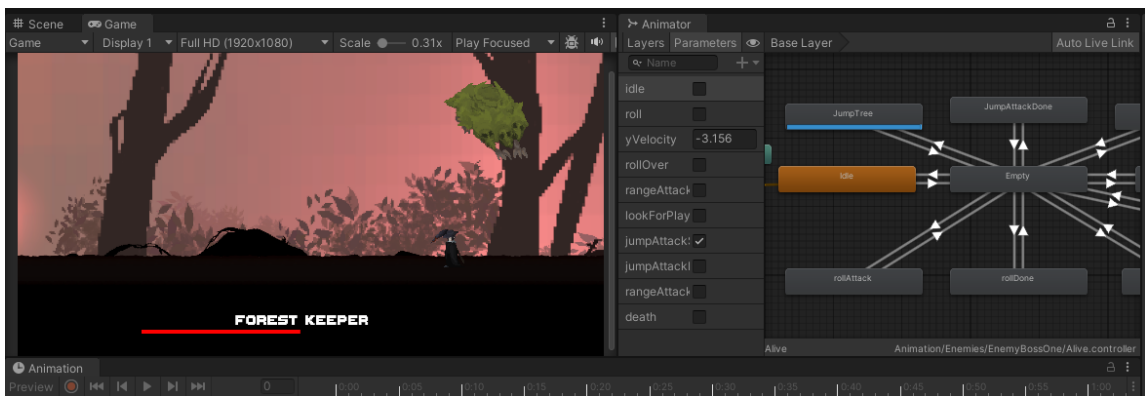


Рисунок 3.5 - Стан jumpAttackState

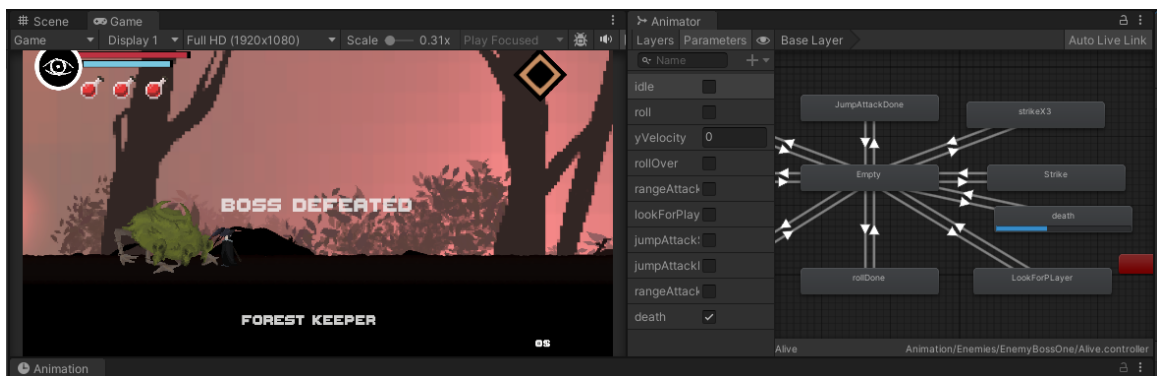


Рисунок 3.6 - Стан deathState

Бос другої локації “Fallen knight” так само, як і перший налічує в собі 9 станів, сім з яких на першій стадії і два додаткових на другій. Перед початком бою ворог перебуває у стані очікування. На першій стадії він підбігає до гравця та використовує атаку ближнього бою (рис. 3.7). На другій стадії додається блокування шкоди від гравця за допомогою щита та телепортація за спину, як показано на рисунках 3.8 та 3.9.

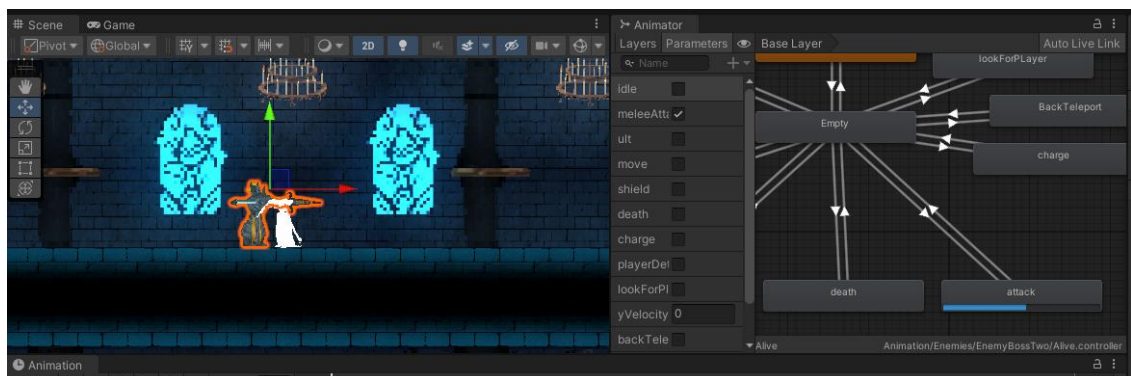


Рисунок 3.7 - Стан meleeAttackState

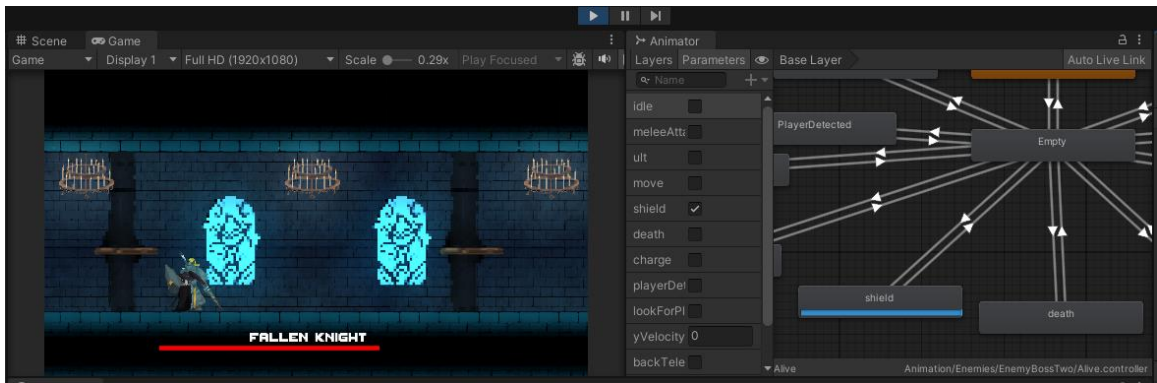


Рисунок 3.8 - Стан shield

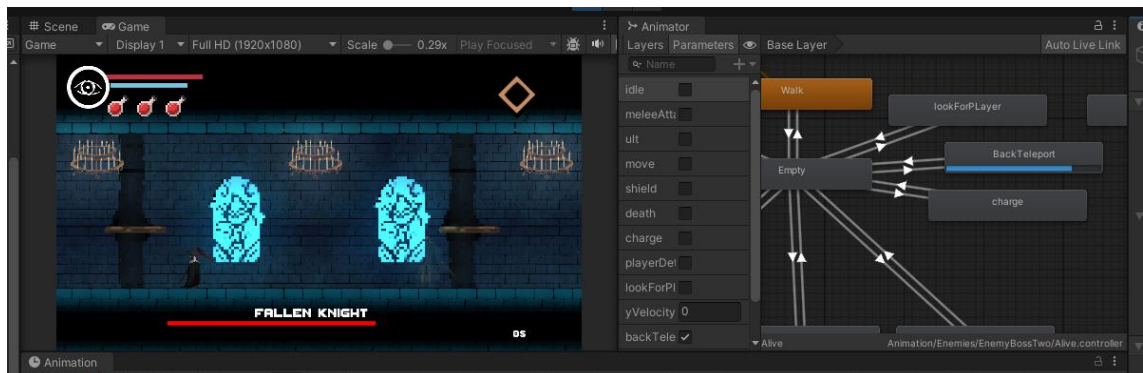


Рисунок 3.9 - Стан backTeleport

3.2 Тестування роботи додатку

Тестування роботи додатку "Dethrone". Для кращого розуміння проблем, які виникають у взаємодії ворогів та гравця в грі, ми проаналізуємо низку випадків тестування, де були виявлені недоліки. Наведені таблиці, bug reports [18-20], містять детальну інформацію про проблеми, включаючи їх опис, середовище, вихідну інформацію, візуальні докази, дії для відтворення, очікувані та фактичні результати, а також серйозність і пріоритет проблеми, і використану техніку тестування. Наша мета - знайти та виправити ці недоліки для покращення якості геймплею. З ними можна ознайомитись на рисунках 3.10 – 3.16 та в таблицях 3.1 – 3.5.

Таблиця 3.1

Проблема взаємодії ворога з грою

ID/Ім'я:	Game world-не правильна взаємодія ворога
пис/резюме:	Підчас гри ігровий інтелект ворога коли доходить до прірви він зупиняється але часитина тіла весить у повітрі.
Середовище:	PC,Windows,unity 2022.3.10f1
Вихідна інформація:	Об'єкт який знаходиться в ієрархії ворога GroundCheck
Візуальний доказ:	Рисунок 3.1
Дія для відтворення:	Запустити гру та очікувати допоки ворог не дійде по прірви
Очікувані результати:	Ворог повинен зупинитись прямо перед виступом
Фактичні результати:	Ворог звисає з виступа
Severity and Priority:	Серйозність: покращення. Пріорітет середній
ТЕХНІКА ТЕСТ ДИЗАЙНУ:	Error Guessing



Рисунок 3.10 - Візуальний доказ до Game world-не правильна взаємодія ворога

Таблиця 3.2

Проблема з передачею шкоди ворога гравцеві

ID/Ім'я:	Enemy- не проходить шкода
Опис/резюме:	Підчас ігрового процесу коли ворог вдаряє об'єкт з тегом player відомості про пошкодження не доходять до самого об'єкта
Середовище:	PC,Windows,unity 2022.3.10f1
Вихідна інформація:	Помилка в консолі
Візуальний доказ:	Рисунок 3.2 та 3.3
Дія для відтворення:	Очікування допоки ворог вдарить об'єкт
Очікувані результати:	Шкода пройде у гравця відніметься здоров'я
Фактичні результати:	Інформація про шкоду не доходить до гравця
Severity and Priority:	Серйозність: критична. Пріорітет: високий
ТЕХНІКА ТЕСТ ДИЗАЙНУ:	Cause/Effect

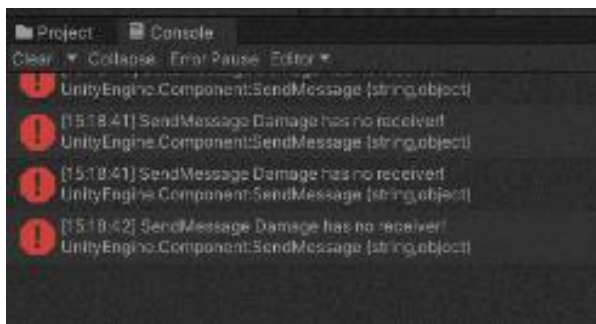


Рисунок 3.11 - Візуальний доказ до Enemy- не проходить шкода

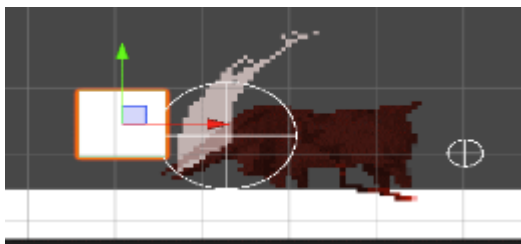


Рисунок 3.12 - Візуальний доказ до Enemy- не проходить шкода

Таблиця 3.3

Проблема зі зміною кадрів анімації для ворога-лучника

ID/Ім'я:	EnemyArcher-не змінили карти анімації
Опис/резюме:	Під час ігрового процесу коли ворог переходить в стан Player Detected
Середовище:	PC, Windows, unity 2022.3.10f1
Вихідна інформація:	Animator в ворога Archer
Візуальний доказ:	Рисунок 3.4 та 3.5
Дія для відтворення:	Ворог має зустріти гравця
Очікувані результати:	Має використовуватись нова модель ворога
Фактичні результати:	Використовується стара модель ворога
Severity and Priority:	Серйозність: критична. Пріоритет: високий
ТЕХНІКА ТЕСТ ДИЗАЙНУ:	Cause/Effect



Рисунок 3.13 - Візуальний доказ до EnemyArcher-не змінили карти анімації



Рисунок 3.14 - Візуальний доказ до EnemyArcher-не змінили карди анімації

Таблиця 3.4

Проблема зі зміною станів для ворога-лучника

ID/Ім'я:	Enemy Archer- не виходить зі стану Player Detected
Опис/резюме:	Підчас того як ворог зустрічає гравця він переходиться в PlayerDecetedState але від більше з нього не виходить
Середовище:	PC,Windows,unity 2022.3.10f1
Вихідна інформація:	Нереалізований перехід між стейтами
Візуальний доказ:	Рисунок 3.6
Дія для відтворення:	Ворог бачить гравця
Очікувані результати:	Має вистрелити з лука
Фактичні результати:	Стоїть
Severity and Priority:	Серйозність: критична. Пріорітет:високий
ТЕХНІКА ТЕСТ ДИЗАЙНУ:	Boundary Value Analysis

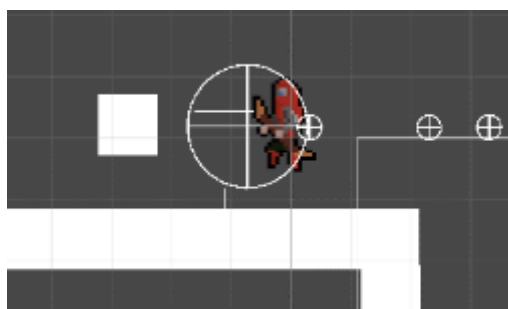


Рисунок 3.15 - Візуальний доказ до Enemy Archer- не виходить зі стану Player Detected

Таблиця 3.5

Проблема з колізією ворогів між собою

ID/Ім'я:	Енему - не можуть пройти скрізь один одного
Опис/резюме:	Підчас ігрово процесу коли вороги перетинаються вони не моєуть пройти поміж один одного
Середовище:	PC,Windows,unity 2022.3.10f1

Вихідна інформація:	Праця зі слоями
Візуальний доказ:	Рисунок 3.7
Дія для відтворення:	Вороги мають зіткнутись
Очікувані результати:	Вороги мають пройти скрізь один одного
Фактичні результати:	Вороги штовхають один одного
Severity and Priority:	Серйозність: критична. Пріоритет: високий
ТЕХНІКА ТЕСТ ДИЗАЙНУ:	Equivalence Partitioning

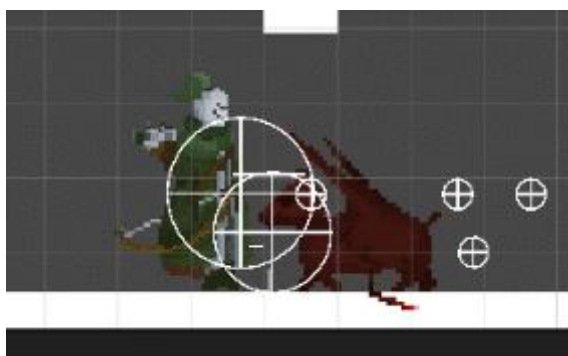


Рисунок 3.16 - Візуальний доказ до Епену - не можуть пройти скрізь один одного

Висновки до третього розділу

У результаті тестування гри "Dethrone" було виявлено декілька серйозних проблем, які потребують негайного виправлення. Перш за все, існують проблеми з взаємодією ворогів з грою, такі як неправильна поведінка ворогів під час наближення до прірви та передача шкоди ворогом гравцеві. Крім того, виявлені проблеми з анімацією ворогів-лучників і зі зміною їх станів під час взаємодії з гравцем. Не менш важливою є проблема колізії між ворогами, яка призводить до неправильної поведінки та взаємодії між ними.

Ці проблеми варто виправити, оскільки вони значно впливають на якість геймплею та загальний досвід гравців. Пріоритетними завданнями є виправлення проблем з передачею шкоди ворогів гравцеві та з взаємодією ворогів з грою, оскільки вони мають критичний характер для коректності гри. Також важливо виправити проблеми з анімацією та станами ворогів, а також колізією між ними, щоб забезпечити більш плавний та реалістичний геймплей.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		67

ВИСНОВКИ

Протягом виконання кваліфікаційної роботи було розроблено ігровий інтелект за допомогою state machine з використанням рушія Unity 2D.

Було чітко сформульовано практичне завдання – розробка поведінки ворогів та механік збереження і завантаження для гри в жанрах Metroidvania і Soulslike. Було здійснено аналіз вимог до ігрового інтелекту, що послужило основою для подальшого проектування. Було проведено аналіз та визначення вимог до програмного забезпечення.

В другому розділі було розглянуто ключові аспекти розробки програмного забезпечення для гри. Від проектування на основі UML та розробки архітектури поведінки ворогів до створення системи збереження та завантаження даних. Також докладно розглянуто реалізацію програмної частини State Machine та налаштування аніматора та ворога.

Проте, під час тестування у третьому розділі було виявлено декілька серйозних проблем, які потребують негайного виправлення. Проблеми з взаємодією ворогів з грою, передачею шкоди гравцеві, анімацією ворогів-лучників, зміною їх станів та колізією між ними впливають на якість геймплею та загальний досвід гравців.

Після виправлення цих проблем та повторного тестування ми очікуємо значне поліпшення у геймплеї гри "Dethrone" та задоволення від її проходження для гравців.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		68

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hollow Knight [Електронний ресурс] – Режим доступу до ресурсу: https://store.steampowered.com/app/367520/Hollow_Knight/.
2. What is an AI model? [Електронний ресурс] – Режим доступу до ресурсу: <https://www.ibm.com/topics/ai-model>.
3. DARK SOULS [Електронний ресурс] – Режим доступу до ресурсу: https://store.steampowered.com/app/374320/DARK_SOULS_III/.
4. Blasphemous [Електронний ресурс] – Режим доступу до ресурсу: <https://store.steampowered.com/app/774361/Blasphemous/>.
5. Harel D. Звичайна Finite-state machine [Електронний ресурс] / David Harel // itermis. – 2019. – Режим доступу до ресурсу: https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview_what_are_state_machines.
6. State machine [Електронний ресурс] // MDN Web Docs Glossary. – 2023. – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Glossary/State_machine?retiredLocale=uk.
7. Basic State Mashine [Електронний ресурс] // Statecharts. – 2021. – Режим доступу до ресурсу: <https://statecharts.dev/what-is-a-state-machine.html>.
8. Pattern State [Електронний ресурс] // Refactoring Guru. – 2020. – Режим доступу до ресурсу: <https://refactoring.guru/uk/design-patterns/state>.
9. PlayerPrefs [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/uk/530/ScriptReference/PlayerPrefs.html>.
10. Mohamed H. Unity Feature 101: Basic Saving Using PlayerPrefs [Електронний ресурс] / Hijazi Mohamed. – 2021. – Режим доступу до ресурсу: <https://bootcamp.uxdesign.cc/unity-feature-101-basic-saving-using-playerprefs-2fc737d1ac7b>.
11. JsonUtility [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: <https://docs.unity3d.com/ScriptReference/JsonUtility.html>.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		69

12. Документація компонент Animator [Електронний ресурс] // Unity Technologies. – 2017. – Режим доступу до ресурсу: <https://docs.unity3d.com/ru/530/Manual/class-Animator.html>.
13. Introduction to Sprite Animations [Електронний ресурс] // Unity Technologies. – 2022. – Режим доступу до ресурсу: <https://learn.unity.com/tutorial/introduction-to-sprite-animations#>.
14. Vector2 [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/ScriptReference/Vector2.html>.
15. Vector3 [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/ScriptReference/Vector3.html>.
16. Davis M. Scripting a Multi-Stage Boss Battle [Електронний ресурс] / Micha Davis. – 2021. – Режим доступу до ресурсу: <https://medium.com/nerd-for-tech/scripting-a-multi-stage-boss-battle-86fb1c3b984d>.
17. How to Create Boss Fight Mechanics in Unity [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: <https://educationecosystem.com/ramondev/RYrEr-how-to-create-boss-fight-mechanics-in-unity>.
18. Bug report [Електронний ресурс] – Режим доступу до ресурсу: <https://qalight.ua/baza-znaniy/bug-report/>.
19. Sviatoslav A. How to Write a Bug Report [Електронний ресурс] / A. Sviatoslav, K. Elena. – 2023. – Режим доступу до ресурсу: <https://rubygarage.org/blog/how-to-write-a-quality-bug-report>.
20. Peham T. What Is A Bug Report? The Essential Guide + Examples [Електронний ресурс] / Thomas Peham – Режим доступу до ресурсу: <https://usersnap.com/blog/what-is-a-bug-report/>.

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		70

ДОДАТКИ

		Іщук О.Ю.			КН.КР.Б – 122 – 24 – ПЗ	Арк.
		Марчук Г. В				
Змн.	Арк.	№ докум.	Підпис	Дата		71

Лістинг Enemy_Dog

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

public class EnemyDog : Enemy
{
    public E1_IdleState idleState { get; private set; }

    public E1_MoveState moveState { get; private set; }

    public E1_PlayerDetectedState playerDetectedState { get; private set; }

    public E1_ChargeState chageState { get; private set; }

    public E1_LookForPlayer lookForPlayerState { get; private set; }

    public E1_MeleeAttack meleeAttackState { get; private set; }

    public E1_DeathState deathState { get; private set; }

    public E1_StunState stunState { get; private set; }
    [SerializeField]
    private D_IdleState idleStateData;
    [SerializeField]
    private D_MoveState moveStateData;
    [SerializeField]
    private D_PlayerDetected playerDetectedData;
    [SerializeField]
    private D_ChargeState chageStateData;
    [SerializeField]
    private D_LookForPlayer lookForPlayerStateData;
    [SerializeField]
    private D_MeleeAttack meleeAttackStateData;
    [SerializeField]
    private D_StunState stunStateData;
    [SerializeField]
    private D_DeathState deathStateData;

    protected override void Start()
    {

        moveState = new E1_MoveState(stateMashine, this, "move", moveStateData, this);

        idleState = new E1_IdleState(stateMashine, this, "idle", idleStateData, this);

        playerDetectedState = new E1_PlayerDetectedState(stateMashine, this, "playerDetected",
        playerDetectedData, this);
```

```

    chageState = new E1_ChargeState(stateMashine, this,"charge",chageStateData, this);

    lookForPlayerState = new E1_LookForPlayer(stateMashine, this, "lookForPlayer",
lookForPlayerStateData, this);

    meleeAttackState = new E1_MeleeAttack(stateMashine, this, "meleeAttack", attackCheck,
meleeAttackStateData, this);

    stunState = new E1_StunState(stateMashine, this, "stun", stunStateData, this);

    deathState = new E1_DeathState(stateMashine, this,"death", deathStateData, this);

    stateMashine.Initialize(moveState);
}
protected override void OnDrawGizmos()
{
    base.OnDrawGizmos();
    Gizmos.DrawWireSphere(attackCheck.position, attackCheckRadius);
}

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

    if (isDead)
    {
        stateMashine.ChangeState(deathState);
    }
else if (isStunned && stateMashine.currentState != stunState)
    {
        stateMashine.ChangeState(stunState);
    }
}

```

Лістинг Enemy

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

public class Enemy : Entity
{
    public EnemyStateMashine stateMashine;

    public D_Enemy enemyData;
    public AnimationToStatemachine atsm { get; private set; }
    public int lastDamageDirection { get; private set; }

    [SerializeField]
    private Transform playerCheck;
    [SerializeField]
    private Transform groundCheckStun;
    private float currentHealth;
    private float currentStunResistance = 0;
    private float lastDamageTime;

    private Vector2 velocityWokrSpace;

    protected bool isStunned;
    protected bool isDead;
    protected override void Start()
    {
        base.Start();
        currentHealth = enemyData.maxHealth;
        currentStunResistance = enemyData.stunResistance;

        atsm = GetComponentInChildren<AnimationToStatemachine>();
        stateMashine = new EnemyStateMashine();
    }

    protected override void Update()
    {

```

```

stateMashine.currentState.LogicUpdate();

anim.SetFloat("yVelocity", rb.velocity.y);

if (Time.time >= lastDamageTime + enemyData.stunRecoveryTime)
{
    ResetStunResistance();
}
}

protected override void FixedUpdate()
{
    base.FixedUpdate();
    stateMashine.currentState.PhysicsUpdate();
}

public virtual void DamageHop(float velocity)
{
    velocityWokrSpace.Set(rb.velocity.x, velocity);
    rb.velocity = velocityWokrSpace;
}

public virtual void ResetStunResistance()
{
    isStunned = false;
    currentStunResistance = enemyData.stunResistance;
}

//public virtual void Damage(AttackDetails attackDetails)
//{
//    lastDamageTime = Time.time;
//    currentHealth -= attackDetails.damageAmount;
//    currentStunResistance -= attackDetails.stunDamageAmount;
//    DamageHop(enemyData.damageHopSpeed);
//    if (attackDetails.position.x > rb.transform.position.x)
//    {
//        lastDamageDirection = -1;
//    }
//    else
//    {
//        lastDamageDirection = 1;
//    }
//}

```

```

public override void Damage()
{
    base.Damage();
    currentHealth = enemyData.maxHealth;

    Instantiate(enemyData.hitParticle,rb.transform.position,
Quaternion.Euler(0f,0f,Random.Range(0f,360f)));
    if(currentHealth <= 0)
    {
        isDead = true;
    }
}
public virtual void SetVelocityEnemy(float velocity)
{
    velocityWokrSpace.Set(facingDirection * velocity, rb.velocity.y);
    rb.velocity = velocityWokrSpace;
}
public virtual void SetVelocityEnemy(float velocity, Vector2 angle, int direction)
{
    angle.Normalize();
    velocityWokrSpace.Set(angle.x * velocity * direction, angle.y * velocity);
    rb.velocity = velocityWokrSpace;
}
public virtual bool CheckGourndAround()
{
    return Physics2D.OverlapCircle(groundCheckStun.position,
enemyData.groundCheckRadius, whatIsGround);
}

public virtual bool CheckPlayerInMinAgroRange()
{
    return Physics2D.Raycast(playerCheck.position, rb.transform.right,
enemyData.minAgroDistance, enemyData.whatIsPlayer);
}
public virtual bool CheckPlayerInMaxAgroRange()
{
    return Physics2D.Raycast(playerCheck.position, rb.transform.right,
enemyData.maxAgroDistance, enemyData.whatIsPlayer);
}
public virtual bool CheckPlayerInCloseRangeAction()
{

```

```

Physics2D.Raycast(playerCheck.position,rb.transform.right,enemyData.closeRangeActionDistance, enemyData.whatIsPlayer);
    }

    public override void Flip()
    {
        base.Flip();
    }

    protected override void OnDrawGizmos()
    {
        Gizmos.DrawLine(wallCheck.position, wallCheck.position + (Vector3)(Vector2.right *
facingDirection * wallCheckDistance));
        Gizmos.DrawLine(groundCheck.position, groundCheck.position + (Vector3)(Vector2.down
* groundCheckDistance));
        Gizmos.DrawWireSphere(playerCheck.position + (Vector3)(Vector2.right *
enemyData.closeRangeActionDistance), 0.2f);
        Gizmos.DrawWireSphere(playerCheck.position + (Vector3)(Vector2.right *
enemyData.minAgroDistance), 0.2f);
        Gizmos.DrawWireSphere(playerCheck.position + (Vector3)(Vector2.right *
enemyData.maxAgroDistance), 0.2f);
    }
}

```

Лістинг Enemy State

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

public class EnemyState
{
    protected EnemyStateMashine stateMashine;
    protected Enemy enemy;

    public float startTime { get; protected set; }

    protected string animBoolName;

    public EnemyState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName)
    {
        this.stateMashine = stateMashine;
        this.enemy = enemy;
        this.animBoolName = animBoolName;
    }

    public virtual void Enter()
    {
        startTime = Time.time;
        enemy.anim.SetBool(animBoolName, true);
        DoChecks();
    }

    public virtual void LogicUpdate()
    {
    }

    public virtual void Exit()
    {
        enemy.anim.SetBool(animBoolName, false);
    }

    public virtual void PhysicsUpdate()
    {
        DoChecks();
    }

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

Лістинг Enemy State Mashine

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

using UnityEngine;
public class EnemyStateMashine
{
    public EnemyState currentState { get; private set; }

    public void Initialize(EnemyState startingState)
    {
        currentState = startingState;
        currentState.Enter();
    }

    public void ChangeState(EnemyState newState)
    {
        currentState.Exit();
        currentState = newState;
        currentState.Enter();
    }
}
```

Лістинг IdleState

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

public class IdleState : EnemyState
{
    protected D_IdleState stateData;

    protected bool flipAfterIdle;

    protected float idleTime;
    protected bool isIdleTimeOver;
    protected bool isPlayerInMinAgroRange;

    public IdleState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
D_IdleState stateData) : base(stateMashine, enemy, animBoolName)
    {
        this.stateData = stateData;
    }

    public override void DoChecks()
    {
        base.DoChecks();
        isPlayerInMinAgroRange = enemy.CheckPlayerInMinAgroRange();
    }

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

        enemy.SetVelocityEnemy(0f);
        isIdleTimeOver = false;
        SetRandomIdleTime();
    }
    public override void Exit()
    {
        base.Exit();

        if (flipAfterIdle)
        {
            enemy.Flip();
        }
    }
    public override void LogicUpdate()
    {
        base.LogicUpdate();
        if (Time.time > startTime + idleTime)
```

```
        {
            isIdleTimeOver = true;
        }
    }
    public override void PhysicsUpdate()
    {
        base.PhysicsUpdate();

    }
    public void SetFlipAfterIdle(bool flip)
    {
        flipAfterIdle = flip;

    }
    public void SetRandomIdleTime()

    {
        idleTime = Random.Range(stateData.minIdleTime,stateData.maxIdleTime);
    }
}
```

Лістинг D_IdleState

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[CreateAssetMenu(fileName = "newIdleStateData", menuName = "Data/State Data/Idle State")]

public class D_IdleState : ScriptableObject
{
    public float minIdleTime = 1f;
    public float maxIdleTime = 2f;
}
```

Лістинг E1_IdleState

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

public class E1_IdleState : IdleState
{
    private EnemyDog enemy1;
    public E1_IdleState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
D_IdleState stateData, EnemyDog enemy1) : base(stateMashine, enemy1, animBoolName,
stateData)
    {
        this.enemy1 = enemy1;
    }

    public override void Enter()
    {
        base.Enter();
    }

    public override void Exit()
    {
        base.Exit();
    }

    public override void LogicUpdate()
    {
        base.LogicUpdate();
        if (isPlayerInMinAgroRange)
        {
            stateMashine.ChangeState(enemy1.playerDetectedState);
        }

        else if (isIdleTimeOver)
        {
            stateMashine.ChangeState(enemy1.moveState);
        }
    }

    public override void PhysicsUpdate()
    {
        base.PhysicsUpdate();
    }
}
```

Лістинг Move State

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

public class MoveState : EnemyState
{
    protected D_MoveState stateData;

    protected bool isDetectingWall;
    protected bool isDetectingLedge;
    protected bool isPlayerInMinAgroRange;

    public MoveState(EnemyStateMashine stateMashine, Enemy enemy, string animBoolName,
D_MoveState stateData) : base(stateMashine, enemy, animBoolName)
    {
        this.stateData = stateData;
    }

    public override void DoChecks()
    {
        base.DoChecks();
        isDetectingLedge = enemy.IsGroundDetected();
        isDetectingWall = enemy.IsWallDetected();

        isPlayerInMinAgroRange = enemy.CheckPlayerInMinAgroRange();
    }

    public override void Enter()
    {
        base.Enter();
        enemy.SetVelocityEnemy(stateData.movementSpeed);
    }
    public override void Exit()
    {
        base.Exit();
    }

    public override void LogicUpdate()
    {
        base.LogicUpdate();
    }

    public override void PhysicsUpdate()
    {
        base.PhysicsUpdate();}
}
```