МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук, проф.
_____Ю. П. Кондратенко
«____»_____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ЗАСТОСУНОК ДЛЯ ДОСЛІДЖЕННЯ ПОВЕДІНКИ NPC В ІГРОВОМУ СЕРЕДОВИЩІ UNITY

Спеціальність 122 «Комп'ютерні науки»

122 – КРБ – 402.22010216

Виконав студент 4-го курсу, групи 402
_____В. В. Кулідобрєв
«19» червня 2024 р.

Керівник: д-р фіз.-мат. наук., професор
_____Е. А. Лисенков
«19» червня 2024 р.

Миколаїв – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет ім. Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

Рівень вищої освіти **бакалавр**
Спеціальність **12 «Комп'ютерні науки»**
(шифр і назва)
Галузь знань **12 «Інформаційні технології»**
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____ Ю. П. Кондратенко
«____» _____2024 р.

З А В Д А Н Н Я
на виконання кваліфікаційної роботи

Видано студенту групи 402 факультету комп'ютерних наук Кулідобрєву Владиславу Володимировичу.

1. Тема кваліфікаційної роботи «Застосунок для дослідження поведінки NPC в ігровому середовищі Unity».

Керівник роботи Лисенков Едуард Анатолійович, д-р фіз.-мат. наук., професор.

Затв. наказом Ректора ЧНУ ім. Петра Могили від «28» грудня 2023 р. № 271

2. Строк представлення кваліфікаційної роботи студентом «19» червня 2024 р.

3. Вхідні (початкові) дані до роботи: варіанти алгоритмів для аналізу поведінки неігрових персонажів.

Очікуваний результат: система для дослідження оптимального алгоритму поведінки NPC.

4. Перелік питань, що підлягають розробці (зміст пояснювальної записки):

– аналіз теоретичних та практичних аспектів програмування поведінки NPC;

– огляд існуючих підходів до реалізації поведінки NPC в ігрових середовищах;

– порівняльний аналіз переваг та недоліків кожних з виокристаних методів;

– розробка застосунку на платформі Unity, що дозволить реалізувати обидва методи програмування поведінки NPC.

5. Перелік графічного матеріалу: 23 рисунки, 4 таблиці, презентація.

6. Завдання до спеціальної частини: «Розрахунок рівня освітлення в робочому місці»

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис
Спеціальна частина з охорони праці	Алексєєва А. О., доент кафедри екоголії	

Керівник роботи ____д-р фіз.-мат. наук., професор Лисенков Е. А._____
(наук. ступінь, вчене звання, прізвище та ініціали)

_____
(підпис)

Завдання прийнято до виконання ___Кулідобрєв В. В._____
(прізвище та ініціали)

_____
(підпис)

Дата видачі завдання «_14_» _____січня____ 2024 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Застосунок для дослідження поведінки NPC в ігровому середовищі Unity

№	Найменування роботи	Початок	Закінчення	Примітки
1	Подання заяви на затвердження теми та керівників КРБ	10.11.2023	15.11.2023	Виконано
2	Отримання завдання на виконання КРБ	10.01.2024	15.01.2024	Виконано
3	Складання календарного плану роботи на весь період виконання КРБ	16.01.2024	30.01.2024	Виконано
4	Отримання завдання на переддипломну практику	15.04.2024	29.04.2024	Виконано
5	Проходження переддипломної практики, збір та аналіз матеріалів до КРБ	29.04.2024	11.05.2024	Виконано
6	Розробка звіту з переддипломної практики	12.05.2024	15.05.2024	Виконано
7	Виконання КРБ: аналіз поведінки неігрових персонажів, огляд існуючих методів реалізації поведінки, розробка ПЗ	13.05.2024	22.06.2024	Виконано
8	Перший попередній захист КРБ на засіданні комісії кафедри	27.05.2024	27.05.2024	Виконано
9	Доробка та остаточне оформлення КРБ	28.05.2024	09.06.2024	Виконано
10	Другий попередній захист КРБ на засіданні комісії кафедри	10.06.2024	10.06.2024	Виконано
11	Подання КРБ рецензенту	13.06.2024	13.06.2024	Виконано
11	Подання КРБ, її електронної копії та інших документів (відгуку, рецензії) до захисту	17.06.2024	21.06.2024	
12	Захист БКР перед екзаменаційною комісією (ЕК)	24.06.2024	28.06.2024	

Розробив студент _____Кулідобрєв В. В._____ _____
(прізвище та ініціали) (*підпис*)

Керівник роботи д-р фіз.-мат. наук., професор Лисенков Е. А _____
(наук. ступінь, вчене звання, прізвище та ініціали) *(підпис)*

«_29_» _____01_____ 2024 р.

АНОТАЦІЯ

кваліфікаційної роботи студента групи 402 ЧНУ ім. Петра Могили

Кулідобрєва Владислава Володимировича

Тема: «Застосунок для дослідження поведінки NPC в ігровому середовищі Unity»

Актуальність даної теми обумовлена швидким розвитком індустрії відеоігор та зростаючими вимогами гравців до реалістичності та складності ігрового процесу. Сучасні ігри все більше орієнтуються на створення інтерактивних та динамічних світів, де NPC відіграють ключову роль у забезпеченні захоплюючого ігрового досвіду. Використання передових методів, таких як дерева поведінки та нечітка логіка, дозволяє створювати більш адаптивних і реалістичних персонажів, що можуть приймати обґрунтовані рішення в різних ситуаціях. Це сприяє підвищенню якості ігрового процесу та задоволеності користувачів.

Об'єкт роботи – процес розробки застосунку для дослідження стратегій прийняття рішень.

Предмет роботи – методи стратегій прийняття рішень неігрових персонажів у комп'ютерній грі на платформі Unity, зокрема використання дерев поведінки та нечіткої логіки.

Метою кваліфікаційної роботи є розробка застосунку для дослідження різних методів прийняття рішень в поведінці NPC. Ця робота спрямована на розширення теоретичних знань та практичних навичок у галузі розробки штучного інтелекту для ігор. Вона може стати основою для подальших досліджень та розробок у цій сфері.

Пояснювальна записка складається зі вступу, трьох розділів, висновків та додатків.

У першому розділі аналізується поведінка NPC в ігрових середовищах, описуються особливості їхньої реалізації та аналізуються можливості використання штучного інтелекту в ігровій розробці.

У другому розділі досліджено методи реалізації поведінки неігрових персонажів, зокрема дерева поведінки та нечітку логіку. Проведено порівняльний аналіз переваг та недоліків вибраних стратегій прийняття рішень NPC.

У третьому розділі описано проєктування та реалізацію застосунку для дослідження поведінки NPC. Розглянуто проєктування ігрового середовища, реалізацію NPC з різними стратегіями прийняття рішень та було здійснено порівняння отриманих результатів стратегій прийняття рішень.

У результаті роботи розроблено застосунок, який дозволяє вивчати та порівнювати різні стратегії прийняття рішень NPC в ігровому середовищі. Використання цього застосунку може допомогти розробникам ігор у виборі оптимальних стратегій для реалізації поведінки NPC залежно від контексту та вимог гри.

Кваліфікаційна робота містить 66 сторінок, 23 рисунки, 4 таблиці, 26 використаних джерел та 3 додатки.

Ключові слова: неігрові персонажі, стратегії прийняття рішень, штучний інтелект, дерева поведінки, нечітка логіка.

ABSTRACT

qualification work of a student of group 402 at Petro Mohyla National University

Kulidobriev Vladyslav

Topic: «An application for studying the behavior of NPCs in the Unity game environment»

The relevance of this topic is due to the rapid development of the video game industry and the growing demands of players for realism and complexity of the gameplay. Modern games are increasingly focused on creating interactive and dynamic worlds where NPCs play a key role in providing an immersive gaming experience. The use of advanced techniques such as behavioural trees and fuzzy logic allows you to create more adaptive and realistic characters that can make informed decisions in different situations. This contributes to the quality of the gameplay and user satisfaction.

Object of work – the process of developing an application for researching decision-making strategies.

Subject of work – methods of decision-making strategies of non-game characters in a computer game on the Unity platform, in particular, the use of behavioural trees and fuzzy logic.

The purpose of the qualification work is to develop an application for researching various methods of decision-making in NPC behaviour. This work is aimed at expanding theoretical knowledge and practical skills in the field of artificial intelligence development for games. It can become the basis for further research and development in this area.

The explanatory note consists of an introduction, three chapters, conclusions, and appendices.

The first section analyses the behaviour of NPCs in gaming environments, describes the peculiarities of their implementation, and analyses the possibilities of using artificial intelligence in game development.

The second section explores methods of implementing the behaviour of non-game characters, in particular, behavioural trees and fuzzy logic. A comparative analysis of the advantages and disadvantages of the selected NPC decision-making strategies is carried out.

The third section describes the design and implementation of an application for studying NPC behaviour. The design of the game environment, the implementation of NPCs with different decision-making strategies, and a comparison of the results of the decision-making strategies were considered.

As a result of the work, an application has been developed that allows you to study and compare different decision-making strategies of NPCs in the game environment. The use of this application can help game developers in choosing the best strategies for implementing NPC behaviour depending on the context and requirements of the game.

The qualification work contains 66 pages, 23 figures, 4 tables, 26 references and 3 appendices.

Keywords: non-player character, decision-making strategies, artificial intelligence, behavior tree, fuzzy logic.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ШІ	– штучний інтелект
BT	– Behavior tree
FL	– Fuzzy logic
FSM	– Finite state machine
NPC	– Non-Player Character

ВСТУП

Ігрова індустрія – одна з швидкозростаючих та прибуткових. З кожним роком кількість користувачів тільки зростає, тому є попит на створення і випуск нових ігор, які б відповідали потребам суспільства. В рамках цієї роботи було зосереджено на одному з таких важливих аспектів комп'ютерних ігор – поведінці неігрових персонажів (NPC). Сучасні ігри все більше фокусуються на створенні реалістичних та адаптивних NPC, здатних взаємодіяти з гравцями та оточенням на високому рівні. У багатьох сучасних іграх NPC відіграють ключову роль у створенні живого та динамічного ігрового середовища. Їхні дії та реакції можуть визначати хід гри та впливати на враження гравців.

Об’єкт кваліфікаційної роботи: процес розробки застосунку для дослідження стратегій прийняття рішень неігрових персонажів.

Предмет кваліфікаційної роботи: методи розробки та впровадження стратегій прийняття рішень неігрових персонажів у комп'ютерній грі на платформі Unity, зокрема використання дерев поведінки та нечіткої логіки.

Метою цієї роботи є розробка застосунку для дослідження різних методів поведінки NPC в ігровому середовищі на основі двох ключових методів: дерев поведінки та нечіткої логіки. Ця робота спрямована на розширення теоретичних знань та практичних навичок у галузі розробки штучного інтелекту для ігор. Вона може стати основою для подальших досліджень та розробок у цій сфері.

Для досягнення цієї мети передбачено наступні завдання:

– дослідити теоретичні та практичні аспекти програмування поведінки NPC в ігровому середовищі;

– розробити застосунок на платформі Unity, що дозволить реалізувати обидва методи програмування поведінки NPC;

– визначити переваги та недоліки кожного підходу прийняття рішень;

– провести порівняльний аналіз ефективності кожного методу на практиці.

1 АНАЛІЗ ПОВЕДІНКИ НЕІГРОВИХ ПЕРСОНАЖІВ В КОНТЕКСТІ ІГРОВИХ СЕРЕДОВИЩ

1.1 Опис поведінки Non-Player Characters в ігрових середовищах

Non-Player Characters (NPC) в іграх – це персонаж, який не контролюється гравцем-людиною. Замість цього ним зазвичай керує комп'ютер через заздалегідь визначену або гнучку поведінку. NPC можна розділити на три групи залежно від їхньої поведінки по відношенню до гравців: ворожі, дружні та нейтральні. Вони можуть виконувати різні ролі, наприклад, давати квести, торгувати з гравцем, надавати інформацію або бути ворогами, яких гравець повинен перемогти [3].

Поведінка NPC може бути простою або складною, залежно від потреб гри. Прості NPC можуть просто ходити по заздалегідь визначеному маршруту або повторювати одні й ті ж фрази. Складні NPC можуть мати власні цілі, пам'ятати події, що відбувалися з ними раніше, і більш динамічно реагувати на дії гравця [5].

Для реалізації поведінки NPC використовуються різні стратегії прийняття рішень. Деякими з найпоширеніших стратегій можуть бути [15]:

– машини скінченних станів – це простий і ефективний спосіб моделювання поведінки NPC. FSM складається з набору станів, між якими може переходити NPC. Перехід між станами визначається поточним станом NPC, його вхідними даними та правилами переходу;

– дерева поведінки є більш гнучким способом моделювання поведінки NPC. Дерево поведінки складається з вузлів, які представляють дії або умови. NPC рухається по дереву поведінки, виконуючи дії та перевіряючи умови;

– нечітка логіка – це система логічного висновку, яка дозволяє NPC приймати рішення на основі нечітких або невизначених даних. Нечітка логіка може використовуватися для моделювання складних емоцій та рішень NPC.

Актуальність даної теми полягає в постійному розвиток ігрової галузі та зростаючий попит на реалістичність, іммерсію та інтерактивність у ігрових

середовищах. Оскільки поведінка неігрових персонажів (NPC) створює враження реального світу в грі, їх поведінка є важливою для досягнення цих цілей. У цьому контексті розвиток ШІ стає стратегічно важливим, оскільки він дозволяє створювати більш складних, реалістичних і непередбачуваних NPC, що робить геймплей більш захоплюючим і цікавим для гравців [8].

Дослідження та реалізація різних стратегій прийняття рішень для NPC, таких як машини скінченних станів, дерева поведінки та нечітка логіка, має безпосередній практичний вплив на ігрову індустрію. Наприклад, FSM підвищує стабільність поведінки NPC, створюючи їх з визначеним набором дій і реакцій на певні події. Дерева поведінки дають NPC більшу гнучкість і дозволяють використовувати складніші алгоритми, що робить їх більш адаптивними до різних ситуацій у грі. Нечітка логіка підвищує автентичність NPC, дозволяючи їм приймати рішення на основі своїх справжніх емоцій і ситуації [8].

Вивчення поведінки NPC також може допомогти у створенні більш ефективних методів штучного інтелекту, які можна використовувати в інших областях, таких як робототехніка або автоматизація процесів.

1.2 Аналіз жанрів комп'ютерних ігор з використанням NPC

Відеогра використовує аудіовізуальні інструменти для створення віртуального середовища, в якому гравець може грати. Найпопулярнішим способом розваги та відпочинку є відеоігри. Це відхід від реального світу, хоча б на короткий проміжок часу, і потраплення в привабливий «віртуальний Всесвіт» [26]. Відеоігри вже перетворилися на окремий вид мистецтва, яке має велику кількість шанувальників, незважаючи на те, що на початку вони вважалися дрібничками.

Усі люди грали в ігри. Спочатку вони були спортивними, потім словесними, а потім настільними. Комп’ютерні ігри з’явилися разом з розвитком електроніки.

Фізик Уільям Хігінбот створив першу відеогру в 1958 році.

Він хотів зробити наукову лабораторію Брукхейвена інтерактивною. В лабораторії гра «Теніс для двох» стала проривом. У той час багато гостей хотіли пограти в гру, яка тоді виглядала як тонкі сині лінії на невеликому екрані осцилографу. Але на той час це було щось незвичайне, що викликало інтерес до нової розробки.

Комп’ютерна гра — це комп’ютерна програма, яка використовує ігровий двигун і написана на мові програмування [2].

На сьогодні комп’ютерні ігри класифікуються за жанрами або категоріями залежно від того, як гравці взаємодіють у грі: кількість гравців, методи взаємодії гравців і тип платформи. Відеоігри є групою ігор, які мають спільний ігровий процес. Як гравець взаємодіє в ігровому світі, визначає жанр відеогри. Ігри на персональних комп’ютерах, ігрові консолі, браузерні ігри, мобільні ігри та ігрові автомати є прикладами різних типів платформ [12].

Графічні зображення включають від першої особи, від третьої особи, віртуальну реальність, доповнену реальність і тривимірні графіки.

У наш час не існує чіткого та однозначного розподілу комп'ютерних ігор за жанрами. А тим паче жанрів, де використовуються NPC. У певній грі можуть бути поєднані характерні риси кількох різних жанрів. Тому жанри ігор також поділяються на піджанри, щоб отримати більш точне уявлення про тип гри [20].

Класифікація ігор за жанрами та піджанрами, де є велика ймовірність зіткнутись з механіками NPC:

– action: ігри включають фізичні випробування, які потрібно подолати гравцеві. Ці ігри залишаються найпопулярнішими серед гравців завдяки швидкому темпу гри;

– платформер: це спосіб, за допомогою якого персонаж взаємодіє з платформами, бігаючи, стрибаючи та падаючи протягом усього ігрового процесу. Наприклад, Donkey Kong і Super Mario Bros;

– шутер: це використання вогнепальної зброї проти опонента. Наприклад, Counter-Strike, Fallout та Call of Duty;

– файтінг: це гра-поєдинок, де потрібно битися, зазвичай рукою. В таких іграх є можливість вибрати персонажа, який може володіти певним типом бою. Наприклад, у бойових іграх Mortal Combat і Super Smash Bros;

– бійки: персонаж б’ється з супротивником. Наприклад, Double Dragon і Teenage Mutant Ninja Turtles 2;

– стелс-ігри: гравці атакують раптово, ховаючись від супротивника. Наприклад, Hitman 2: Blood Money, Batman: Arkham Asylum і Assassins Creed;

– action-travel: це квести, у яких необхідно подолати перешкоди за допомогою отриманого предмета. Наприклад, The Last of Us, Little Nightmares 2 і Resident Evil Village;

– квест (броділки): персонажи взаємодіють з іншими персонажами, щоб вирішувати головоломки за допомогою підказок. Наприклад, Machinarium і Minecraft;

– рольові ігри: це ігри, в яких переважно використовується фантазійне середовище;

– симуляції: це імітація реального (або вигаданого) світу для імітації реальних подій. Наприклад, Sims 1-4, Youtubers Life та Spore;

– стратегії на основі традиційних настільних ігор.

1.3 Можливості та перспективи використання ШІ в ігровій розробці

Вплив ігрової промисловості на суспільство та культуру стає все більш помітним, оскільки це є однією з найбільш швидко розвиваються сфер у сучасному світі. Ігри, які стають все більш популярним видом розваг і розвитку, можуть мати значний вплив на психічне, соціальне та фінансове життя людей. Розробники комп’ютерних ігор використовували нові інструменти та технології, щоб зробити ігри більш складними, реалістичними та цікавими. Штучний інтелект є однією з технологій, яку активно розробляють.

Штучний інтелект – це область дослідження, яка спрямована на розробку програм і систем, що можуть самостійно навчатися, приймати рішення та виконувати завдання, що традиційно виконувалися виключно людьми.

По-перше, у сучасних комп'ютерних іграх, використання штучного інтелекту може виявитися корисним для поліпшення геймплею та створення більш реалістичних ігрових персонажів. Він може бути застосований для розробки більш складних і захоплюючих ігрових світів, що забезпечують глибший досвід гри. Тому подальший розвиток цієї технології може сприяти створенню ще більш персоналізованих та реалістичних ігор з більшим різноманіттям варіантів та виборів для гравців [4].

Системи адаптивного геймплею, які аналізують дії гравця та впливають на зміни геймплею в режимі реального часу, є прикладом використання штучного інтелекту в іграх. Наприклад, алгоритм може створити ігрову стратегію, яка включає альтернативні варіанти дій, якщо гравець часто вибирає одні й ті ж дії в певних ситуаціях. Для того, щоб зрозуміти, як це працює, розглянемо приклад гри, яка належить до жанру Action-RPG. У цій грі можуть бути різні характеристики героя, такі як його здоров’я, міцність, швидкість руху та сила атаки. Наприклад, система адаптивного геймплею може збільшити складність гри, додати більше ворогів або змінити поведінку ворогів, щоб зробити гру більш складною. Більше того, якщо гравець не використовує певні характеристики героя, такі як швидкість руху, система може налаштувати гру таким чином, щоб зробити її більш динамічною та швидкою, залучаючи гравця до використання цієї конкретної характеристики [1].

Нейромережі можуть допомогти визначити поведінку гравців і створювати ігрових персонажів, які можуть взаємодіяти з гравцями на більш реалістичному рівні. Крім того, системи адаптивного геймплею можуть допомагати гравцям різного досвіду та навичок. Наприклад, система може дозволити гравцю зосередитися на основних елементах гри, зменшуючи складність гри, якщо гравець

має низький досвід, а якщо гравець має високий досвід, система може збільшити складність гри, додавши нові завдання.

По-друге, штучний інтелект може допомогти розробникам оптимізувати ігри, полегшувати процес розробки (наприклад, генеруючи елементи коду), зменшити кількість випадкових складнощів, покращити ігрові сервіси та платформи, вирішувати проблеми з балансуванням гри, запобігати читерству та дозволити гравцям насолоджуватися емоційно насиченим ігровим досвідом, наближеним до реального життя. Ігрові боти, які можуть бути створені за допомогою штучного інтелекту, можуть допомогти гравцям вирішити складні ситуації та підвищити їхню успішність у грі [4].

Ігри також можуть покращити когнітивні та психологічні функції людини, а також розвивати такі навички, як реакція, координація рухів і прийняття рішень. Ігри можуть використовувати штучний інтелект для терапії та навчання [21]:

– практика соціальних навичок: нейромережі можуть використовуватися в деяких іграх для навчання соціальних навичок, таких як співпереживання, емпатія та спілкування;

– забезпечення інклюзивності: ШІ може зробити ігри доступними та інтерактивними для людей з різними обмеженнями. Наприклад, можна створити ігри для людей з фізичними обмеженнями за допомогою голосового управління або створити ігри за допомогою брейнкомп’ютерних інтерфейсів для людей зі складними обмеженнями руху;

– покращений підхід до навчання: штучний інтелект може допомогти в навчанні мови, імітуючи реальні ситуації та спілкуючи з віртуальними персонажами;

– психологічна допомога: деякі ігрові програми можуть бути розроблені, щоб допомогти людям розслабитися та медитувати;

– терапія: ігри можуть бути корисними для людей з психологічними проблемами. Вони, наприклад, можуть допомогти в лікуванні тривоги, депресії та посттравматичного стресового розладу.

Тим не менш, є деякі недоліки використання штучного інтелекту. Навчання нейромереж може бути складним і вимагати великої кількості ресурсів, а штучний інтелект може приймати рішення, які не зовсім точні або не враховують моральні та етичні принципи. Збільшення залежності від цієї технології може призвести до меншого ручного контролю над грою та змінами в реальному часі. Гравці, які не мають найновішого обладнання або доступу до швидкого Інтернету, можуть відчувати, що використання штучного інтелекту погіршує їхні ігри. Крім того, існує ймовірність створення штучного інтелекту, який або занадто складний для гравців, або занадто простий і нудний [21].

1.4 Постановка завдання дослідження

Мета розробки полягає у створенні програмного забезпечення, яке дозволить порівнювати стратегії прийняття рішень NPC (Non-Player Character) у відеогрі, реалізовані за допомогою дерев поведінки та нечіткої логіки. Програма має бути розроблена на платформі Unity.

Створення ігрового середовища:

- розробка ігрового середовища у двомірному просторі на платформі Unity;
- відображення на мапі двох NPC з візуальними об'єктами;
- генерація ігрової мапи з елементами лікування та зброї.

Розробка двох різних типів NPC з різними стратегіями прийняття рішень:

- NPC з поведінкою, що базується на дереві рішень;
- NPC з поведінкою, що базується на нечіткій логіці.

Реалізація геймплейної логіки:

- забезпечення безперервної гри, де раунди розпочинаються автоматично після завершення попереднього;
- підрахунок кількості вбивств кожного NPC;
- генерація нової ігрової мапи та розташування на ній елементів (аптечки та кристали з поліпшенням шкоди) після завершення кожного раунду;

– підрахунок кількості зібраних аптечок та кристалів із поліпшенням шкоди кожним NPC;

Реалізація дослідження:

– створення можливості вибору варіанту присвоєння поведінки двом NPC (дерево рішень проти дерева рішень, дерево рішень проти нечіткої логіки та нечітка логіка проти нечіткої логіки);

– створення механізму для збору статистичних даних про стратегії поведінки NPC;

– візуалізація та збереження даних для подальшого аналізу та порівняння ефективності стратегій.

Функціональні вимоги:

– програма має працювати на платформі Unity;

– всі NPC мають мати базові характеристики: здоров'я, швидкість, дальність атаки;

– NPC мають взаємодіяти з елементами на мапі (аптечки та кристали з поліпшенням шкоди);

– ведення статистики та збір даних про результати гри.

За допомогою цієї програми можна вивчати, який тип поведінки краще впорається з надходячими завданнями та умовами середовища гри. Програма може бути корисною для дослідників у галузі штучного інтелекту та ігрової розробки, а також для розробників ігор, які хочуть вдосконалити інтелектуальний рівень своїх NPC. Вона надає можливість експериментувати з різними підходами до управління NPC та визначити оптимальні стратегії для певного типу гри або середовища.

Висновки до розділу 1

У першму розділі БКР проведено аналіз предметної області та сформульовано постановку задачі для дослідження. Було детально розглянуто поведінку NPC в ігрових середовищах, що є важливим аспектом сучасних

комп'ютерних ігор. Зроблено огляд комп'ютерних ігор як об'єкту дослідження, з акцентом на їхні характеристики та вимоги до штучного інтелекту.

Також було проаналізовано можливості та перспективи використання штучного інтелекту в розробці ігор, що дозволило виявити основні напрямки розвитку цієї технології в індустрії.

На основі проведеного аналізу сформульовано завдання дослідження, яке включає розробку застосунку для порівняння стратегій прийняття рішень NPC за допомогою дерев поведінки та нечіткої логіки. Було визначено мету і завдання роботи, що є основою для подальших етапів дослідження та розробки застосунку.

2 МЕТОДИ РЕАЛІЗАЦІЇ ПОВЕДІКИ НЕІГРОИХ ПЕРСОНАЖІВ

2.1 Дерева поведінки у відеоіграх

Дерево поведінки – формалізований метод побудови поведінки NPC. Всі стани персонажа організовані у вигляді деревоподібної ієрархічної структури, що є його особливістю. Дерево поведінки включає в себе всі потенційні стани, у яких можуть перебувати NPC [22, 10]. Коли у грі відбувається якась подія, ШІ проводить аналіз станів NPC, обираючи найактуальніший для поточної ситуації. Дерево поведінки ідеально підходить для систематизації станів NPC в іграх, у яких є багато механік і елементів для геймплея.

Замість того, щоб визначити кінцевий набір станів персонажа, створюється дерево вузлів з гілками, які демонструють різні типи поведінки. Дерева поведінки спрощують пріоритети дій: оскільки робота дочірніх вузлів визначає роботу гілки, а отже, і дерева, поєднання складених «і» та «або» прямо дає набір пріоритетних функцій, які автоматично обробляють переривання, послідовності та альтернативні варіанти. Крім того, ця структура легко читається та візуалізується (див. рис. 2.1). Достатньо слідкувати за гілками, які є пріоритетними, і перевіряти, чи успішна вона, щоб визначити, чи можна продовжувати цю гілку чи слід переходити до наступної [16].

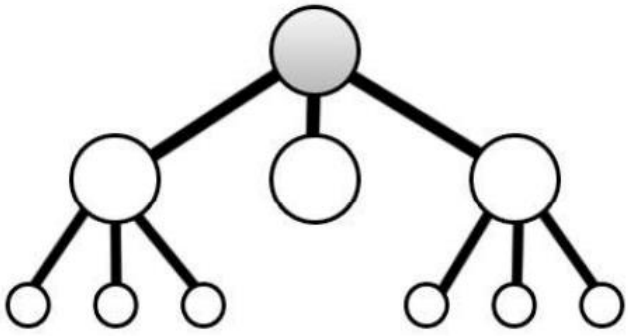

Рисунок 2.1 – Базова структура дерева

Але найбільшою перевагою дерев поведінки є те, що вони дуже масштабовані та модульні. Видалення або додавання гілки фактично додає або видаляє функцію для персонажа. Так само можна просто скопіювати та вставити частину поведінки з піддеревом. Деякі піддерева можуть навіть служити автономними бітами ігрової логіки, які можна повторно імпортувати та використовувати в дереві поведінки іншого персонажа, щоб реалізувати фрагменти системи, такі як слідування за ціллю та патрулювання між групами точок [6].

Крім того, така модульність означає, що можна будувати дерево по частинах і мати стратегію поступового проектування. Дерева поведінки набагато більш гнучкі, ніж кінцеві автомати, де стани повинні бути максимально віддаленими один від одного. Незважаючи на те, що кожен вузол автономний і функціонує самостійно, існує певний стан спільних даних, до якого можна отримати доступ із різних точок дерева, щоб отримати глобальні змінні, які можуть бути залежними від контексту.

За класичним визначенням, дерево поведінки – це орієнтоване дерево, що включає набір вузлів та ребер. Корінь дерева поведінки – це вершина без батьків. З іншого боку, вузли без нащадків є листям цього дерева [14].

Існує також складені вузли. Складені вузли називаються так, оскільки вони мають одного або більше нащадків. Їх стан повністю базується на результатах обчислення дочірніх вузлів, і поки вони обчислюються, вузол перебуває у стані "виконання". Існує кілька різновидів складених вузлів, які здебільшого визначаються шляхом обчислення їхніх дочірніх вузлів [19, 7].

1. Вузли послідовності – вся послідовність дочірніх вузлів повинна завершитися успішно, щоб бути оцінена як успіх. Вся послідовність повідомить про невдачу, якщо будь-який з дочірніх елементів на будь-якому кроці послідовності повертає false. Важливо відзначити, що послідовності зазвичай виконуються зліва направо. Вузол послідовності зображено рамкою з міткою "→", як показано на рис. 2.2. Псевдокод вузла послідовності відображено на рис. 2.3.

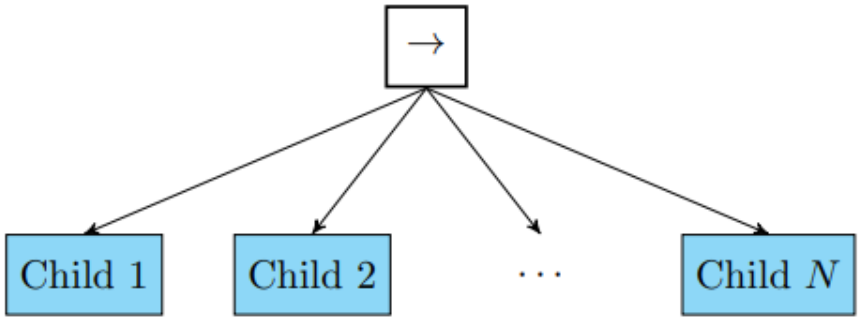

Рисунок 2.2 – Графічне представлення вузла послідовності з N дочірніми елементами.

```
Sequence() function return status
   for each child node[i]
      childStatus <-Tick(node[i])
      if childStatus equals RUNNING
         return RUNNING
      elseif childStatus equals FAILURE
         return FAILURE
      end
   end
   return SUCCESS
end
```

Рисунок 2.3 – Псевдокод вузла послідовності

2. Вузли селектори – на відміну від послідовності, селектори набагато більш лояльні до своїх дочірніх вузлів. Якщо будь-який з дочірніх вузлів у послідовності селектора повертає true, селектор скаже: «ех, досить!» і поверне true негайно, не обчислюючи більше жодного з дочірніх вузлів. Єдиний спосіб, за допомогою якого вузол-селектор може повернути помилку, якщо всі його дочірні елементи обчислені, але жоден з них не поверне успіху. Вузол селектор зображено рамкою з міткою "?", як показано на рис. 2.4. Псевдокод вузла селектора відображена на рис. 2.5.

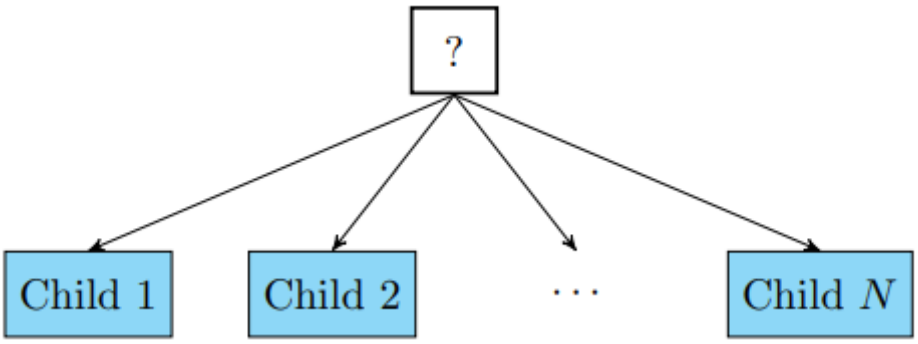

Рисунок 2.4 – Графічне представлення вузла-селектора з N дочірніми вузлами

```
Selector() function return status
   for each child node[i]
      childStatus <-Tick(node[i])
      if childStatus equals RUNNING
         return RUNNING
      elseif childStatus equals SUCCESS
         return SUCCESS
      end
   end
   return FAILURE
end
```

Рисунок 2.5 – Псевдокод вузла селектора

Окрім вузлів керування, також існують вузли виконання (дії та стану). Вузли дії включають відтворення анімації, зміну стану персонажа або будь-яку дію, що змінює стан гри. З іншого боку, вузол стану зазвичай використовується для перевірки деяких значень. Тести на близькість, тестування стану персонажа та перевірка лінії видимості є прикладами вузлів умов. Умова повертає успіх, якщо умова виконана, інакше повертає невдачу.

Вузол завжди повертає один з наступних станів [7]:

– успіх – умова, яку перевірив вузол, виконана;
– відмова – умова, яку перевірив вузол, не була і не буде виконана;
– виконується – дійсність умови, яку перевіряє вузол, не було визначено. Можна сприймати це як: «Будь ласка, зачекайте».

2.2 Нечітка логіка в ігровому середовищі

Найпростіший спосіб визначити, що таке нечітка логіка – порівняти її з бінарною логікою. Навіть у випадках, коли оцінюються декілька значень, всі значення мають рівно два результати, тобто є бінарними. Нечіткі значення, з іншого боку, мають набагато ширший діапазон можливостей, де кожне значення представлено як число з плаваючою комою замість цілого числа. В таких випадках, значення більше не сприймається як 0 або 1, а починається сприйматись як від 0 до 1 [9].

Поширеним прикладом для опису нечіткої логіки є температура. Нечітка логіка дозволяє нам приймати рішення на основі неконкретних даних. Не знаючи точної температури, можна вийти на вулицю в теплий сонячний день і переконатися, що там тепло. І навпаки, якби опинитися на Алясці взимку, можна зрозуміти, що там холодно, знову ж таки, не знаючи точної температури. Терміни «холодний», «прохолодний», «теплий» і «гарячий» є розмитими. Існує велика кількість невизначеності щодо того, в який момент стан переходить від теплого до гарячого. За допомогою набору правил нечітка логіка дозволяє моделювати ці ідеї як множини та визначати, наскільки вони правильні або неправильні [6].

Люди, як прийнято говорити, мають певну сіру зону під час прийняття рішень. Таким чином, не все є чорно-білим. Те ж саме стосується і факторів, які покладаються на нечітку логіку. Скажімо, відчули голод через кілька годин без їжі. В який момент були настільки голодні, щоб піти на перекус? Можна позначити час одразу після їжі як 0, а 1 – як точку, де наблизилися до голоду. На рисунку 2.6 зображено цей момент [9].

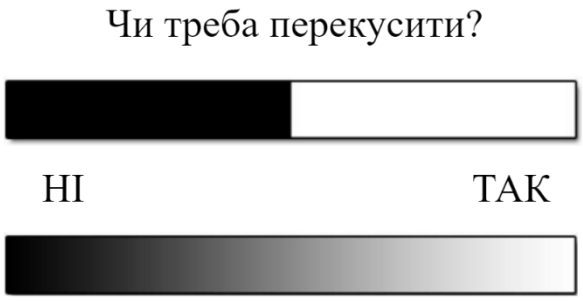

Рисунок 2.6 – Шкала рівня голоду

Багато факторів впливають на процес прийняття рішень. Це дозволяє контролерам нечіткої логіки враховувати стільки даних, скільки потрібно. Продовжимо розглядати приклад «чи треба перекусити?». Було розглянуто лише одне значення для прийняття цього рішення, а саме час, що минув з моменту останнього прийому їжі. Однак є інші речі, які можуть вплинути на це рішення, такі як скільки енергії витрачається і наскільки ліниві в цей конкретний момент. Як би там не було, можна побачити, як кілька вхідних значень можуть впливати на вихід, який можна охарактеризувати як «ймовірність з'їсти ще одну порцію їжі».

Зрозумівши прості ідеї, що лежать в основі нечіткої логіки, легко почати думати про безліч корисних використань. Нечітка логіка чудово справляється з отриманням даних, їх оцінкою, подібною до дії людини, а потім перетворенням цих даних у інформацію, яку система може використовувати .

Контролери нечіткої логіки можуть використовуватися в багатьох різних контекстах. Деякі з них більш очевидні, ніж інші, і хоча вони не порівнюються з нашим використанням в ігровому ігровому ШІ, вони слугують для ілюстрації певних позицій [17]:

– системи опалення, вентиляції та кондиціонування повітря: приклад температури, коли йдеться про нечітку логіку, є хорошим теоретичним прикладом пояснення нечіткої логіки, а також дуже поширеним реальним прикладом контролерів нечіткої логіки, які працюють;

– машини: сучасні автомобілі мають дуже складні комп’ютеризовані системи, включаючи систему кондиціонування повітря, автоматизовані системи гальмування та систему подачі палива. Насправді комп’ютери в автомобілях створили набагато ефективніші системи, ніж звичайні двійкові системи;

– смартфон: операційні системи сучасних смартфонів оптимізують яскравість екрану, враховуючи освітлення навколишнього середовища, колір даних, що відображаються, і рівень заряду акумулятора;

– пральні машини: більшість сучасних пральних машин за останні два десятиліття використовують нечітку логіку. Розмір завантаження, брудність води, температура та інші фактори враховуються від циклу до циклу, щоб оптимізувати використання води, енергоспоживання та час.

Нечітка логіка була протестована у відеоіграх, які намагаються використовувати простий дизайн разом з інтелектуальними елементами, як і в багатьох інших академічних методах штучного інтелекту. Нечітка логіка відповідає цим вимогам, оскільки її мова досить проста для аналізу, що призводить до швидкого та простого проектування [11].

У 1996 році Ларрі О'Брайен представив нечітку логіку в журналі Game Developer Magazine як офіційну концепцію розробки ігор. З тих пір багато джерел згадують нечітку логіку як одну з корисних технік для розробки ігрового штучного інтелекту.

Використовуючи визначені нечіткі змінні та їхні набори, можна створити нечіткі правила. Ці правила зазвичай мають форму if-then і використовують граматику булевої логіки. Правила визначають значення вихідних змінних, враховуючи поточні значення вхідних змінних. В нечіткій логіці такі оператори, як AND, OR і NOT, також мають значення. Після створення правил працює механізм нечіткого виведення, який робить висновки на основі правил, що керують системою, і значень змінних, які зараз є. Таким чином, вихідні змінні отримують нечітке значення [18].

Нечітка логіка може бути корисною для ігрового штучного інтелекту в багатьох аспектах. Зокрема, її можна використовувати для прийняття рішень NPC, таких як вибір предметів або зброї, керування рухом юнітів, як у системах керування, оцінки загроз ШІ-супротивником і класифікації, наприклад, ранжування гравців і NPC за рівнем здоров’я або потужності за допомогою нечітких змінних.

2.3 Аналіз вибраних стратегій прийняття рішень NPC

Аналіз стратегій прийняття рішень NPC є важливим аспектом у багатьох областях комп'ютерних наук, включаючи ігрову розробку, робототехніку та штучний інтелект. Для наглядності порівняння стратегій прийняття рішень побудована порівняльна таблиця переваг та недоліків кожного з них (див. табл. 2.1).

Таблиця 2.1 –Порівняння переваг та недоліків обраних стратегій прийняття рішень

Алгоритм	**Переваги**	**Недоліки**
Дерева поведінки	Дерева поведінки ефективні, оскільки їх легко зрозуміти, впровадити та підтримувати. Вони пропонують простий і логічний спосіб моделювання поведінки ШІ, що скорочує час і зусилля, необхідні для розробки гри [16].	Дерева поведінки можуть стати досить складними, коли в дереві багато вузлів. Це може ускладнити розуміння потоку поведінки ШІ.

Продовження таблиці 2.1

Алгоритм	**Переваги**	**Недоліки**
Дерева поведінки	Дерева поведінки є гнучкими, оскільки дозволяють створювати складні моделі поведінки, об'єднуючи прості моделі поведінки. Їх можна адаптувати до різних ігрових середовищ, таких як головоломки, екшн та симулятори.	Дерева поведінки обмежені в діапазоні поведінки, яку вони можуть представляти. Хоча вони ефективні для простої або помірної поведінки, вони можуть бути не настільки ефективними для великомасштабної поведінки або поведінки, яка є нелінійною.
	Дерева поведінки можна модифікувати, оскільки вони дозволяють змінювати поведінку ШІ під час виконання. Це означає, що ШІ-агент може адаптуватися до мінливих умов гри під час її проходження.	Дерева поведінки може бути важко оптимізувати через велику кількість вузлів у дереві. Оптимізація вимагає багато часу і зусиль, що може бути дорого коштувати.
	Дерева поведінки можна налагоджувати, оскільки вони надають розробникам чітке візуальне уявлення про те, як поводиться ШІ-агент. Це дозволяє розробникам легко виявляти проблеми в поведінці ШІ та швидко їх виправляти.	Тестування поведінки дерева поведінки вимагає багато часу і зусиль, оскільки існує багато можливих шляхів через дерево.

Закінчення таблиці 2.1

Алгоритм	Переваги	Недоліки
Нечітка логіка	Нечітка логіка дозволяє здійснювати надійне управління в невизначених або змінних умовах, що робить її придатною для реальних застосувань, де точне математичне моделювання ускладнене [17].	Проектування та налаштування контролерів нечіткої логіки може бути складним і трудомістким процесом, що вимагає досвіду і великих обсягів тестування.
	Нечітка логіка може обробляти неточні та нечіткі вхідні дані, що робить її придатною для завдань, які вимагають людських міркувань.	Системи нечіткої логіки можуть бути складними для інтерпретації та розуміння, що ускладнює прогнозування їхньої поведінки у всіх ситуаціях.
	Контролери нечіткої логіки можуть адаптуватися до змін у навколишньому середовищі або динаміки системи без необхідності перепрограмування.	Впровадження контролерів нечіткої логіки може вимагати додаткових обчислювальних ресурсів у порівнянні з традиційними методами управління.

Висновки до розділу 2

Проаналізувавши наведені технології та методи для реалізації поставленої задачі, зроблено наступні висновки.

1. Розглянуто дерева поведінки як один із основних методів реалізації логіки прийняття рішень NPC у відеоіграх. Визначено, що дерева поведінки є потужним та гнучким інструментом, який дозволяє створювати складні та адаптивні алгоритми поведінки;

2. Вивчено та проаналізовано нечітку логіку, яка застосовується для моделювання складних та невизначених середовищ у відеоіграх. Було виявлено, що нечітка логіка дозволяє NPC приймати рішення в умовах невизначеності та з обмеженою інформацією.

3. Проведено порівняльний аналіз стратегій прийняття рішень NPC, які базуються на деревах поведінки та нечіткій логіці. Визначено переваги та недоліки кожного підходу, а також сфери їх найефективнішого застосування. Зокрема, було виявлено, що дерева поведінки краще підходять для сценаріїв з чітко визначеними правилами, тоді як нечітка логіка ефективніша в умовах невизначеності та варіативності.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ ДЛЯ ДОСЛІДЖЕННЯ ПОВЕДІНКИ NPC

3.1 Проектування ігрового середовища

3.1.1 Налаштування ігрових об’єктів

Так як в застосунку реалізована механіка підбирання об’єктів, які будуть давати бонуси, та механіка стрільби, слід створити відповідні об’єкти для впровадження їх у гру. Для досягнення реалістичності та інтерактивності ігрового середовища було створено та налаштовано три основні типи об'єктів:

- модель кулі;
- модель кристалу для поліпшення наносимої шкоди;
- модель аптечки, який відновлює здоров’я NPC.

Ці моделі були імпортовані в Unity і перетворені на Prefab об'єкти. На рисунках 3.1-3.3 показані створені Prefab з 3D моделей.

Рисунок 3.1 – Prefab 3D моделі кулі

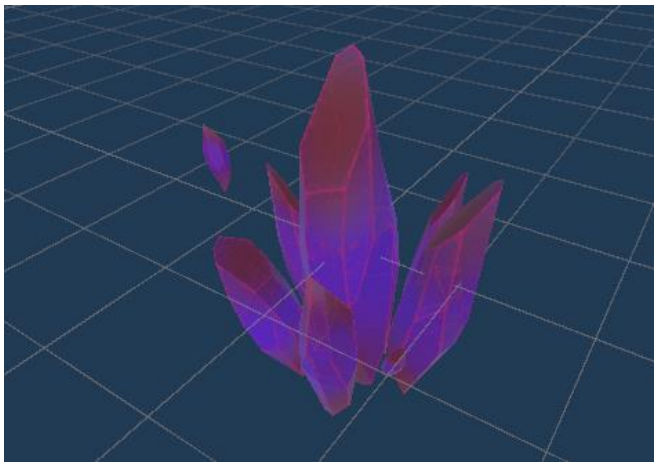

Рисунок 3.2 – Prefab 3D моделі кристалу

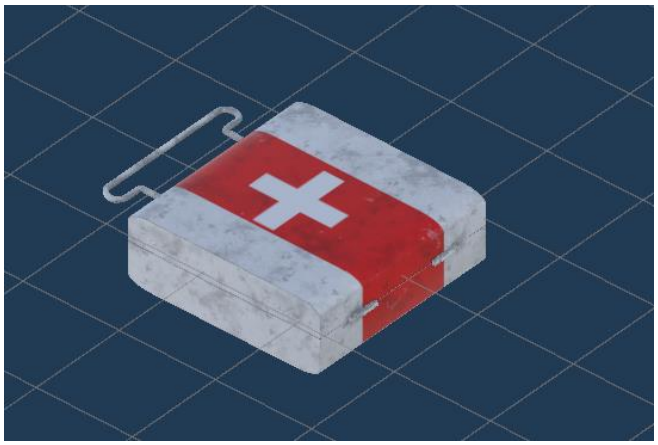

Рисунок 3.3 – Prefab 3D моделі аптечки

Prefab в Unity дозволяє створити шаблон об'єкта, який можна багаторазово використовувати у грі. Використання Prefab спрощує управління об'єктами та їх налаштування, оскільки зміни в Prefab автоматично відображаються на всіх його екземплярах у сцені.

Кожен Prefab був налаштований за допомогою наступних компонентів:

– Mesh Renderer;
– Collider;
– Rigidbody.

Mesh Renderer відповідає за візуалізацію 3D моделі в ігровому середовищі. Він надає можливість налаштовувати матеріали, кольори та текстури, які будуть застосовані до поверхні моделі. Це дозволяє створювати більш реалістичні та естетично привабливі об'єкти [25].

Collider визначає фізичну оболонку об'єкта, яка використовується для обчислення зіткнень з іншими об'єктами у грі. Типи Collider можуть варіюватися, і для кожної моделі було обрано відповідний тип [23].

Для моделі кулі – Capsule Collider, що забезпечує капсульну оболонку.

Для моделі кристалу – Sphere Collider, що забезпечує сферичну оболонку.

Для моделі аптечки – Box Collider, що забезпечує прямокутну оболонку.

Rigidbody додає об'єкту фізичні властивості, такі як маса, гравітація та сили, що діють на нього. Це дозволяє об'єктам реагувати на фізичні закони в ігровому світі, забезпечуючи реалістичну динаміку руху та зіткнень. Rigidbody налаштований так, щоб об'єкти могли правильно взаємодіяти з NPC та іншими елементами гри [24].

Головним налаштуванням усіх об’єктів є налаштування Collider як тригеру, тобто об'єкт буде викликати події при входженні в нього, але не має фізичного впливу.

Створення та налаштування цих компонентів для кожного з об'єктів кулі, кристалу та аптечки забезпечує їх коректну роботу в ігровому середовищі, дозволяючи NPC взаємодіяти з ними відповідно до логіки гри.

3.1.2 Розробка 3D мапи

Для реалізації проєкту була створена 3D мапа, яка дозволяє забезпечити комплексний ігровий простір. Мапа має форму квадрата, що забезпечує симетрію та рівні умови для кожного NPC. Візуалізація ігрового простору була організована таким чином, щоб гравці мали змогу чітко бачити переміщення NPC та їх взаємодію з об'єктами мапи.

Для забезпечення кращої видимості та зручності спостереження за поведінкою NPC, камера у грі була налаштована на статичну позицію зверху. Це дозволяє чітко бачити всі дії NPC, їх переміщення та взаємодію з елементами мапи.

Статична камера зверху також сприяє кращому аналізу та дослідженню алгоритмів поведінки NPC.

На початку кожного раунду на мапі розташовуються два NPC, кожен з яких реалізує свій алгоритм поведінки. Кожен NPC розташований на своїй половині мапи, що забезпечує початкову рівновіддаленість від ресурсів та один від одного. Це дозволяє досліджувати їх поведінку в умовах рівних стартових позицій.

На мапі розташовані такі елементи:

– стіни;
– аптечки;
– кристали з поліпшенням шкоди.

Вміст елементів арени показано на рисунку 3.4.

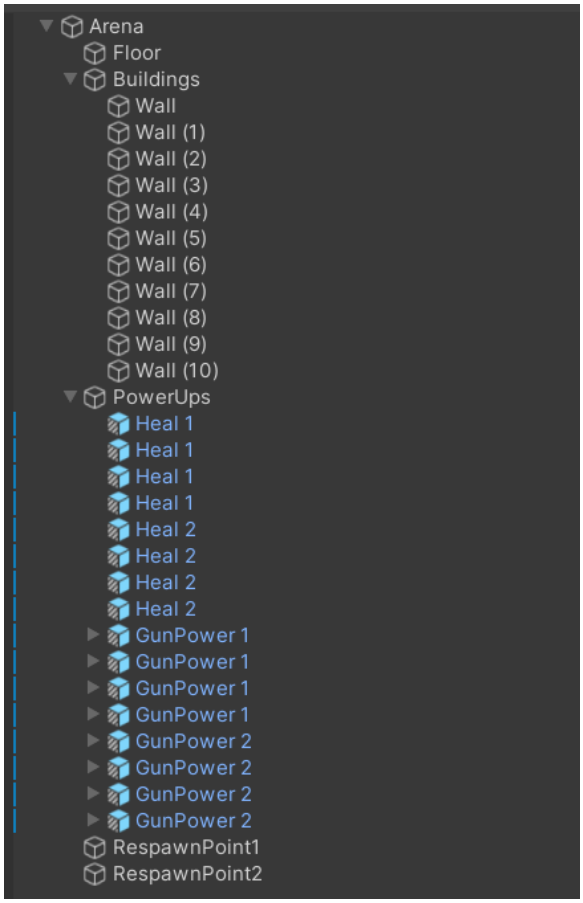

Рисунок 3.4 – Вміст мапи арени

На мапі розміщені стіни, які створюють перешкоди для NPC. Стіни змушують NPC шукати оптимальні шляхи для досягнення своїх цілей, що додає складності та реалістичності їх поведінці.

На кожній половині мапи розташовані чотири аптечки. Аптечки забезпечують відновлення здоров'я NPC, що підібрав їх. Однак при запуску кожного раунду генерується лише три з чотирьох аптечок, що додає елемент випадковості та варіативності в гру.

Так само, як і з аптечками, на кожній половині мапи розташовані чотири кристали, які збільшують шкоду NPC. З цих чотирьох кристалів генеруються рандомно лише три, що знову ж таки додає непередбачуваності. Це забезпечує унікальність кожного раунду та виключає повторення однакових сценаріїв, що є важливим для дослідження різних стратегій поведінки NPC.

Шаблон мапи показано на рисунку 3.5.

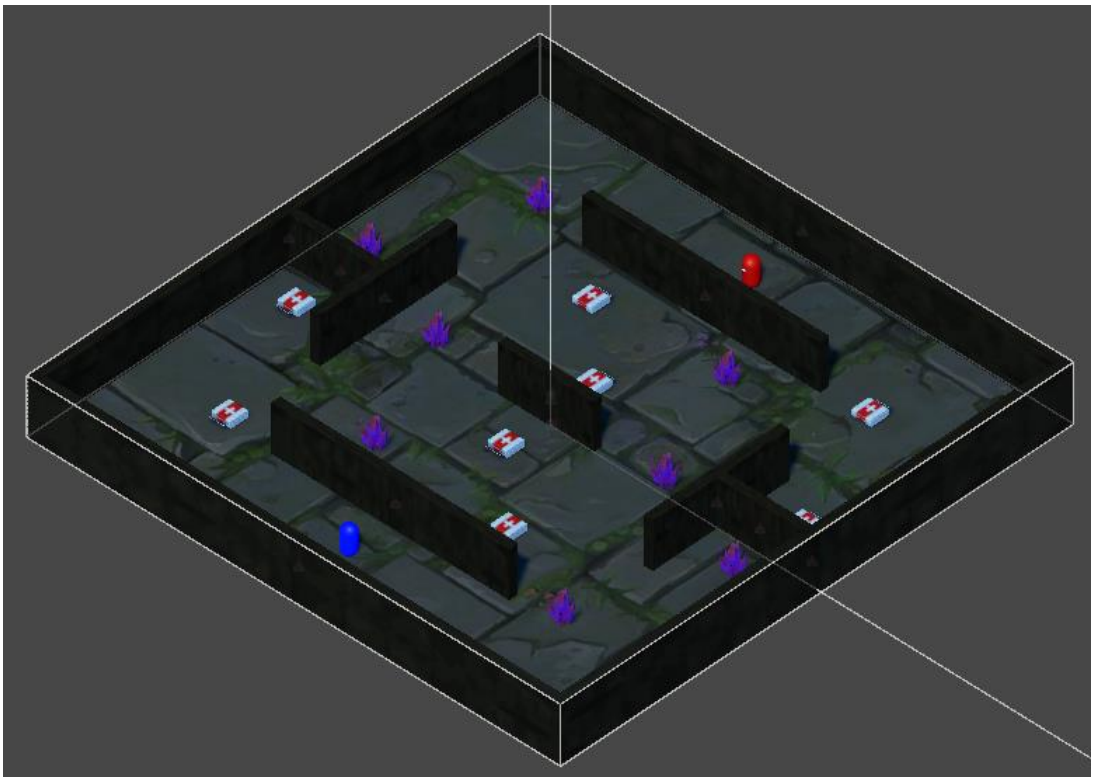

Рисунок 3.5 – Зовнішній вигляд шаблону мапи арени

3.1.3 Навігація NPC мапою

Для реалізації ефективної навігації NPC мапою була використана система Navigation Mesh (NavMesh). NavMesh дозволяє NPC обчислювати оптимальні маршрути для переміщення, враховуючи наявні перешкоди (стіни) та динамічно змінюване середовище (розташування аптечок та кристалів).

Для роботи з цією механікою були використані наступні компоненти [13]:

– NavMesh – це основна структура даних, яка використовується для опису поверхонь ігрового рівня. Ця структура використовується персонажами для обчислення шляху до певної точки;

– NavMeshAgent – це компонент, призначений для ігрових об'єктів, які згодом повинні переміщатися по рівню. Ці об'єкти називаються агентами.

Процес створення NavMesh [13]:

– підготовка мапи: перед налаштуванням NavMesh, підготовлюється мапа з усіма перешкодами, такими як стіни та інші об'єкти;

– запечіння (Bake) NavMesh: використовуючи компонент Navigation, мапа була «спечена» (baked), що означає створення навігаційної сітки на основі заданих параметрів агента. Це процес, при якому Unity автоматично генерує навігаційну сітку, враховуючи всі перешкоди та налаштування;

– налаштування областей (NavMesh Areas): для додаткової оптимізації деякі області мапи були позначені як різні типи поверхонь, що дозволяє NPC приймати кращі рішення під час навігації.

Після налаштування NavMesh агента та створення навігаційної сітки, NPC можуть використовувати її для ефективного переміщення по мапі. Це включає:

– обхід перешкод: NPC можуть обчислювати маршрути, що дозволяють обходити стіни та інші об'єкти;

– досягнення цілей: NPC можуть знаходити найкоротші шляхи до аптечок та кристалів, забезпечуючи ефективне використання ресурсів;

– реалістична взаємодія: використання NavMesh забезпечує плавну та реалістичну поведінку NPC під час їх переміщення по ігровому середовищу.

3.2 Реалізація NPC з різними стратегіями прийняття рішень

3.2.1 Ієрархія моделі NPC

Для більш структурованого коду, було прийнято рішення використати принцип програмування «наслідування» для реалізації повідінки NPC. Основою ієрархії є базовий клас NPC, від якого успадковуються два похідні класи: NPCBehaviorTree та NPCFuzzyLogic, які реалізують різні стратегії прийняття рішень (див. рис. 3.6).

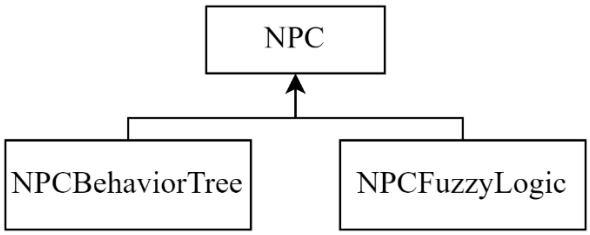

Рисунок 3.6 – Ієрархія класів реалізації поведінки NPC

Базовий клас NPC є фундаментом для всіх NPC у грі. Він містить базову логіку та механіку, яку успадковують і розширюють інші класи. Основні функції базового класу включають:

– встановлення та зберігання базових характеристик NPC, таких як здоров'я, швидкість руху, дальність атаки;

– взаємодія з елементами на мапі (аптечка та кристали);

– обробка базових подій, таких як отримання урону та смерть NPC.

Цей клас забезпечує стандартну поведінку для всіх NPC, дозволяючи успадковувати і розширювати його функціонал у спеціалізованих класах.

Клас NPCBehaviorTree успадковується від базового класу NPC і реалізує поведінку NPC на основі алгоритму дерева поведінки. Основні аспекти цього класу включають:

– реалізація дерева поведінки, що дозволяє NPC приймати рішення на основі певних умов та дій;

– використання вузлів дерева для визначення послідовності дій NPC (наприклад, пошук аптечки, атакуючі дії);

– гнучкість у додаванні нових типів поведінки шляхом розширення дерева.

Цей клас дозволяє моделювати складну поведінку NPC, використовуючи структурований підхід дерева рішень.

Клас NPCFuzzyLogic також успадковується від базового класу NPC і реалізує поведінку на основі алгоритму нечіткої логіки. Основні аспекти цього класу включають:

– використання нечітких множин і правил для прийняття рішень NPC;

– оцінка різних можливих дій NPC на основі ступеня належності до нечітких множин.

Цей клас забезпечує адаптивну та гнучку поведінку NPC, дозволяючи враховувати більше факторів при прийнятті рішень.

Створення ієрархії моделі NPC з використанням базового класу та двох похідних класів, NPCBehaviorTree та NPCFuzzyLogic, забезпечує структурований і розширюваний підхід до моделювання різних стратегій поведінки NPC у грі. Це дозволяє ефективно додавати нові механіки у поведінці неігрових персонажів та додавати нові поведінки іншими алгоритмами. Таким чином, код є легким у розширенні.

3.2.2 Базовий клас NPC

Скрипт класу NPC у грі Unity забезпечує базову логіку та функціональність для неігрового персонажа, який буде використовувати вже кожен окремий NPC в своїй поведіні. Він включає в себе константи, змінні стану та методи, які використовуються для керування NPC, включаючи їхні атаки, переміщення та взаємодію з іншими об'єктами в грі. Схема моделі класу NPC показано на рисунку 3.7.

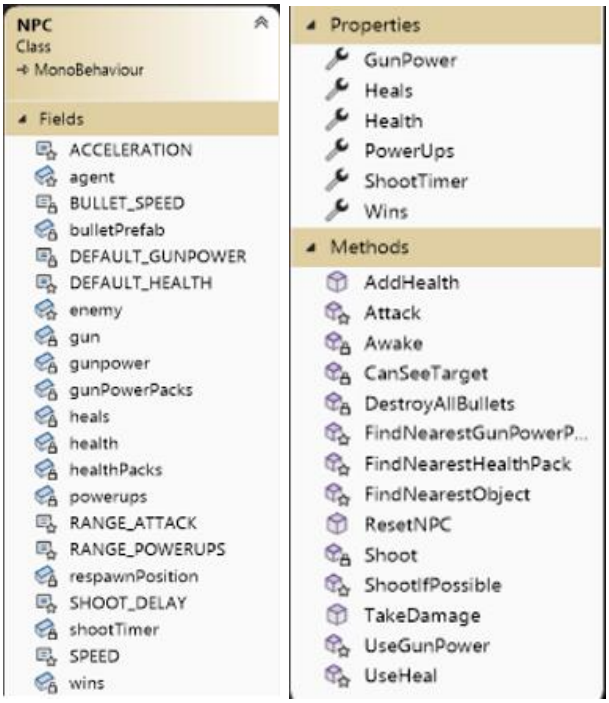

Рисунок 3.7 – Модель базового класу NPC

У класі використовуються такі константи для ініціалізації базових значень налаштування NPC:

– SPEED: швидкість руху NPC (12f). Використовується для налаштування компонента NavMeshAgent;

– SHOOT_DELAY: затримка між пострілами (0.75f). Використовується для визначення частоти стрільби;

– DEFAULT_HEALTH: стандартне значення здоров'я NPC (100). Використовується при ініціалізації та скиданні стану NPC;

– RANGE_ATTACK: дальність атаки NPC (10). Використовується для встановлення дистанції атаки;

– RANGE_POWERUPS: дальність, на якій NPC може збирати аптечки та кристали (2). Використовується при встановленні цілі для підбирання пакунків;

– BULLET_SPEED: швидкість кулі (30.0f). Використовується для встановлення швидкості руху кулі після пострілу;

– DEFAULT_GUNPOWER: стандартна сила зброї (5). Використовується при ініціалізації та скиданні стану NPC.

Окрім констант з базовими значенням налаштування, в класі також містяться змінні стану, які змінюються динамічно протягом виконання програми. Основними з них є:

– health: здоров'я NPC, яке змінюються при нанесенні шкоди або при підбиранні аптечки. Початково встановлюється на DEFAULT_HEALTH;

– gunpower: шкода NPC, яке змінюються при підбиранні поліпшення шкоди. Початково встановлюється на DEFAULT_GUNPOWER;

– wins: кількість перемог NPC. Встановлюється на 0 при ініціалізації;

– heals: кількість підібраних лікувальних пакунків. Встановлюється на 0 при ініціалізації;

– powerups: кількість підібраних пакунків з поліпшенням шкоди. Встановлюється на 0 при ініціалізації.

І також в класі містяться методи, в яких вже виконується сама логіка виконання певних дій NPC. В базовому класі реалізовані ті методи, які будуть використовуватись в будь-якому випадку, не залежно від того, який алгоритм поведінки виконує той чи інший NPC.

1. Метод Awake(). Цей метод викликається при першій ініціалізації об'єкта NPC. Він встановлює початкові значення для здоров'я, сили зброї, кількості перемог, кількості пакунків лікування та пакунків з додатковою шкодою. На

початку методу шукаються всі згенеровані об'єкти з тегом «Heal» та «GunPower» і зберігаються відповідно в масивах healthPacks та gunPowerPacks. Також ініціалізується компонент NavMeshAgent, що відповідає за навігацію NPC, з заданими швидкістю та прискоренням.

2. Метод ResetNPC(). Цей метод скидає стан NPC до початкового. Він вимикає компонент NavMeshAgent, переміщує NPC на позицію відродження (вказану у змінній respawnPosition) та знову вмикає компонент. Потім скидаються значення здоров'я та сили зброї до початкових.

3. Метод AddHealth(int heal). Цей метод додає здоров'я NPC на певне значення при підбиранні аптечки, але не більше максимального значення DEFAULT_HEALTH.

4. Метод TakeDamage(int damage). Цей метод зменшує здоров'я NPC на певне значення при отримані шкоди. Якщо після зменшення здоров'я воно стає менше або дорівнює 0, то NPC та його ворог скидаються до початкового стану, а ворог отримує одну перемогу.

5. Метод CanSeeTarget(). Цей метод перевіряє, чи NPC бачить ворога. Використовується промінь (Raycast) з позиції одного NPC у напрямку зору, щоб визначити, чи зустрічається якийсь об'єкт на шляху до NPC. Якщо так, то NPC бачить ворога.

6. Метод Shoot(). Цей метод виконує постріл зі зброї NPC. Створюється куля, яка летить у напрямку ворога з встановленою швидкістю. Куля отримує початкову силу від зброї NPC.

7. Метод FindNearestHealthPack(). Цей метод знаходить найближчий лікувальний пакунок для NPC. Перебирає всі активні об'єкти з тегом «Heal» та обчислює відстань між NPC та кожним пакунком. Повертає пакунок з найменшою відстанню.

8. Метод FindNearestGunPowerPack(). Аналогічно до попереднього методу, перебирає всі активні об'єкти з тегом «GunPower» та знаходить найближчий.

9. Метод FindNearestObject(GameObject object1, GameObject object2). Цей метод знаходить найближчий об'єкт серед двох заданих. Використовується для вибору найближчого пакунка (лікування або зброї) для NPC.

10. Метод UseHeal(). Цей метод встановлює ціль для NPC – найближчий пакунок лікування, який знаходиться в методі FindNearestHealthPack(). NPC починає рухатися до цієї цілі з встановленою дистанцією зупинки.

11. Метод UseGunPower(). Аналогіно до попердeнього методу, встановлює ціль для NPC – найближчий пакунок з додатковою шкодою, який знаходиться в методі FindNearestGunPowerPack().

12. Метод Attack(). Цей метод виконує атаку на ворога. NPC встановлює відстань зупинки до ворога та намагається стріляти. Він також намагається рухатися в напрямку ворога.

Ці методи разом забезпечують основну функціональність NPC в грі, включаючи переміщення, атаку, взаємодію з об'єктами на мапі та управління станом NPC.

3.2.3 Реалізація класу поведінки NPC алгоритмом дерева рішень

Скрипт класу NPCBehaviorTree у грі Unity представляє собою реалізацію поведінки NPC за допомогою алгоритму дерева рішень. Схема моделі класу NPCBehaviorTree показано на рисунку 3.8.

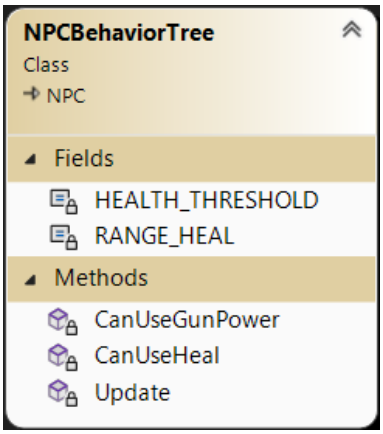

Рисунок 3.8 – Модель класу NPC з поведінкою дерева рішень

У класі використовуються такі константи для ініціалізації значень необхідних для цього алгоритму поведінки:

– HEALTH_THRESHOLD: поріг низького рівня здоров'я. Ця константа визначає мінімальний рівень здоров'я NPC, при досягненні якого NPC буде вважатися в «низькому стані» і спробує скористатися можливістю лікування;

– RANGE_HEAL: відстань, на якій NPC може використати можливість лікування. Встановлює максимальну відстань, на якій NPC зможе підійти до пакунка з лікуванням.

Також в класі містяться певні методи для реалізації механіки поведінки.

1. Метод Update(). Цей метод викликається кожен кадр та керує основною логікою NPC за допомогою дерева рішень. Перевіряє, чи може NPC використати можливість лікування (CanUseHeal()). Якщо так, то NPC лікується. Якщо не може лікуватися, перевіряє, чи може NPC використати можливість покращення зброї (CanUseGunPower()). Якщо може, то він рухається до пакунка з покращенням зброї. Якщо NPC не може ні лікуватися, ні використовувати покращення зброї, то він атакує ворога.

2. Метод CanUseHeal(). Цей метод перевіряє, чи може NPC скористатися можливістю лікування. Перевіряє наявність аптечки, поточний рівень здоров'я NPC та відстань до ворога. Повертає true, якщо можливість лікування доступна, та false в іншому випадку. Код реалізації цього методу наведено нижче.

```
bool CanUseHeal()
{
   return FindNearestHealthPack() != null &&
      Health <= HEALTH_THRESHOLD &&
      Vector3.Distance(gameObject.transform.position, enemy.transform.position) >= RANGE_HEAL;
}
```

3. Метод CanUseGunPower(). Цей метод перевіряє, чи може NPC скористатися можливістю покращення зброї. Перевіряє наявність пакунка з покращенням зброї та чи цей пакунок ближчий за ворога. Повертає true, якщо можливість покращення зброї доступна, та false в іншому випадку.

Код реалізації цього методу наведено нижче.

```
bool CanUseGunPower()
{
   return FindNearestGunPowerPack() != null &&
      (FindNearestObject(enemy, FindNearestGunPowerPack()).tag == "GunPower");
}
```

Цей клас демонструє просту реалізацію алгоритму дерева рішень для управління поведінкою NPC в грі. Він визначає, які дії повинні бути виконані в залежності від поточного стану NPC та його оточення. Блок-схема реалізації цього алгоритму показано на рисунку 3.9.

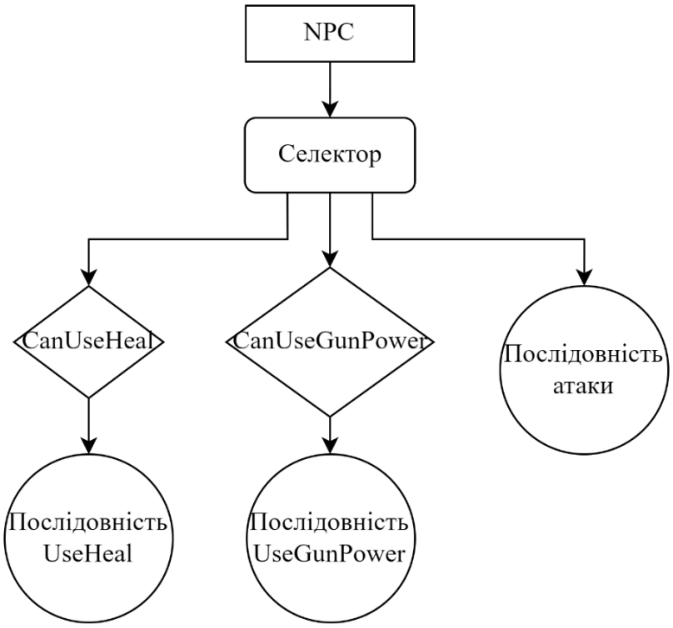

Рисунок 3.9 – Блок-схема поведінки за деревом рішень

На самому початку, використовуючи селектор, перевіряється перший умовний вузол «CanUseHeal». Якщо він повертає true, він переходить до послідовності «UseHeal», якщо повертає false, селектор переходить до наступного умовного вузла «CanUseGunPower». І так само, як і раніше – у випадку true вузол переходить до послідовності UseGunPower, коли false – селектор переходить до послідовності атаки. У реалізації дерева рішень спочатку визначається структура дерева та вказується умови, за яких мають прийматися подальші рішення. Потім на кожному кроці гри дерево переглядається та приймаються рішення на основі

поточного стану гри та результатів аналізу умов. Результати дерева рішень можуть бути повернуті як конкретні дії, здійснені персонажем у грі, наприклад атака. Завдяки цьому дерева рішень дозволяють автоматизувати управління персонажами в грі, що підвищує її достовірність і реалістичність.

3.2.4 Реалізація класу поведінки NPC алгоритмом нечіткої логіки

Скрипт класу NPCFuzzyLogic у грі Unity представляє собою реалізацію поведінки NPC за допомогою алгоритму нечіткої логіки. Схема моделі класу NPCFuzzyLogic показано на рисунку 3.9.

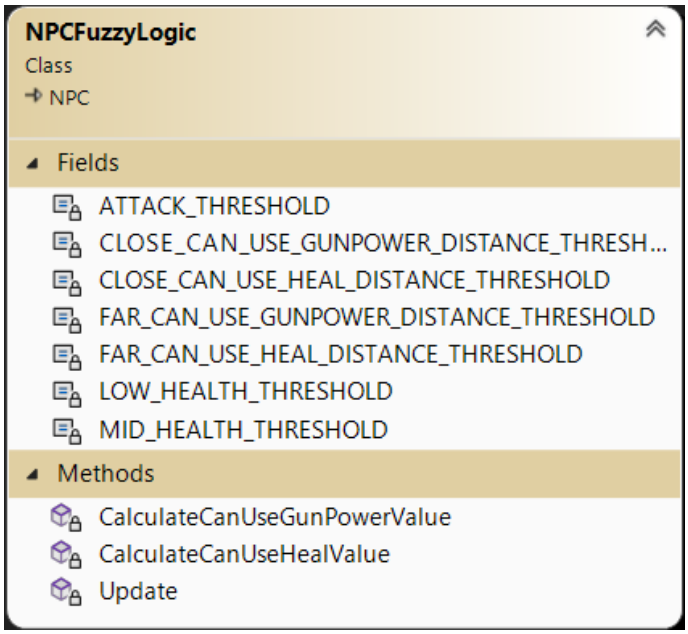

Рисунок 3.9 – Модель класу NPC з поведінкою нечіткої логіки

У класі використовуються такі константи для ініціалізації значень необхідних для цього алгоритму поведінки:

– LOW_HEALTH_THRESHOLD: поріг низького рівня здоров'я (25). Використовується для визначення, коли NPC має низьке здоров'я;

– MID_HEALTH_THRESHOLD: поріг середнього рівня здоров'я (50). Використовується для визначення інтервалу середнього рівня здоров'я NPC;

– CLOSE_CAN_USE_HEAL_DISTANCE_THRESHOLD: відстань (5), на якій NPC може розглянути можливість лікування (ближній поріг);

– FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD: відстань (10), на якій NPC може розглянути можливість лікування (далекий поріг);

– CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD: відстань (15), на якій NPC може розглянути можливість поліпшення шкоди (ближній поріг);

– FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD: відстань (20), на якій NPC може розглянути можливість поліпшення шкоди (далекий поріг);

– ATTACK_THRESHOLD: поріг для визначення можливості атаки. Використовується для порівняння значень можливостей дій.

Також в класі містяться певні методи для реалізації механіки поведінки.

1. Метод Update(). Цей метод викликається кожен кадр та керує основною логікою NPC. Обчислює значення можливості використання поліпшення шкоди (CalculateCanUseGunPowerValue()) та можливості лікування (CalculateCanUseHealValue()). Якщо значення можливості використання поліпшення шкоди більше або дорівнює значенню можливості лікування і більше порогу атаки, NPC використовує поліпшення шкоди. Якщо значення можливості лікування більше або дорівнює значенню можливості використання поліпшення шкоди і більше порогу атаки, NPC лікується. Якщо жодна з можливостей не перевищує порогу атаки, NPC атакує ворога.

2. Метод CalculateCanUseHealValue(). Цей метод обчислює значення можливості лікування для NPC. Якщо пакунок з лікуванням не знайдено, повертає 0.0. Обчислює значення низького здоров'я (lowHealthValue) на основі поточного рівня здоров'я NPC:

– якщо здоров'я менше або дорівнює LOW_HEALTH_THRESHOLD, значення низького здоров'я дорівнює 1.0;

– якщо здоров'я більше LOW_HEALTH_THRESHOLD та менше або дорівнює MID_HEALTH_THRESHOLD, значення низького здоров'я обчислюється лінійно від 1.0 до 0.0.

Також обчислює значення близької відстані до ворога (closeDistanceValue):

– якщо відстань до ворога більша або дорівнює FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD, значення дорівнює 1.0;

– якщо відстань до ворога між CLOSE_CAN_USE_HEAL_DISTANCE_THRESHOLD та FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD, значення обчислюється лінійно від 0.0 до 1.0.

Метод обчислює та повертає мінімальне значення з lowHealthValue та closeDistanceValue. Код реалізації цього методу наведено нижче.

```
private float CalculateCanUseHealValue()
{
   if (FindNearestHealthPack() == null)
      return 0.0f;

   float lowHealthValue = 0.0f;
   if (Health <= LOW_HEALTH_THRESHOLD)
   {
      lowHealthValue = 1.0f;
   }
   else if (Health > LOW_HEALTH_THRESHOLD && Health <= MID_HEALTH_THRESHOLD)
   {
      lowHealthValue = (MID_HEALTH_THRESHOLD - Health) / (MID_HEALTH_THRESHOLD -
LOW_HEALTH_THRESHOLD);
   }

   float distanceToEnemy = Vector3.Distance(gameObject.transform.position, enemy.transform.position);

   float closeDistanceValue = 0.0f;
   if (distanceToEnemy >= FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD)
   {
      closeDistanceValue = 1.0f;
   }
   else if (distanceToEnemy < FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD && distanceToEnemy >=
CLOSE_CAN_USE_HEAL_DISTANCE_THRESHOLD)
   {
      closeDistanceValue = 1 - (FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD - distanceToEnemy) /
(FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD - CLOSE_CAN_USE_HEAL_DISTANCE_THRESHOLD);
   }

   float canUseHealValue = Mathf.Min(lowHealthValue, closeDistanceValue);

   return canUseHealValue;
}
```

3. Метод CalculateCanUseGunPowerValue(). Цей метод обчислює значення можливості використання покращення зброї для NPC. Якщо пакунок з покращенням зброї не знайдено, повертає 0.0. Обчислює значення близької відстані до ворога (closeEnemyValue):

– якщо відстань до ворога більша або дорівнює FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD, значення дорівнює 1.0.

– якщо відстань до ворога між CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD та FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD, значення обчислюється лінійно від 0.0 до 1.0.

Також обчислює значення близької відстані до пакунка з покращенням зброї (closeGunPowerValue):

– якщо відстань до пакунка менша або дорівнює CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD, значення дорівнює 1.0.

– якщо відстань до пакунка між CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD та FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD, значення обчислюється лінійно від 1.0 до 0.0.

Метод обчислює та повертає мінімальне значення з closeEnemyValue та closeGunPowerValue. Код реалізації цього методу наведено нижче.

```
private float CalculateCanUseGunPowerValue()
{
   if (FindNearestGunPowerPack() == null)
      return 0.0f;

   float distanceToEnemy = Vector3.Distance(gameObject.transform.position, enemy.transform.position);

   float closeEnemyValue = 0.0f;
   if (distanceToEnemy >= FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD)
   {
      closeEnemyValue = 1.0f;
   }
   else if (distanceToEnemy < FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD && distanceToEnemy >=
CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD)
   {
      closeEnemyValue = 1 - (FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD - distanceToEnemy) /
(FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD -
CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD);
   }

   float distanceToGunPower = Vector3.Distance(gameObject.transform.position,
FindNearestGunPowerPack().transform.position);

   float closeGunPowerValue = 0.0f;
   if (distanceToGunPower <= CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD)
   {
      closeGunPowerValue = 1.0f;
   }
   else if (distanceToGunPower > CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD &&
distanceToGunPower <= FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD)
   {
      closeGunPowerValue = (FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD - distanceToGunPower) /
(FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD -
CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD);
   }

   float canUseGunPowerValue = Mathf.Min(closeEnemyValue, closeGunPowerValue);

   return canUseGunPowerValue;
}
```

Цей клас демонструє реалізацію нечіткої логіки для управління поведінкою NPC в грі. Нечітка логіка дозволяє NPC приймати рішення на основі нечітких умов, що робить поведінку NPC більш гнучкою та адаптивною до різних ситуацій в ігровому середовищі.

3.3 Реалізація геймплейної логіки

3.3.1 Налаштування вибору варіантів поведінки

Застосунок передбачає вибір варіантів поведінки NPC. Було створено меню, яке дозволяє користувачу вибрати один із трьох варіантів:

– дерево рішень проти дерева рішень (BTvsBT);

– дерево рішень проти нечіткої логіки (BTvsFL);

– нечітка логіка проти нечіткої логіки (FLvsFL).

Відповідно до вибраного варіанту запускається відповідна сцена з NPC, які мають певну поведінку. На рисунку 3.10 показано меню вибору моделі поведінки.

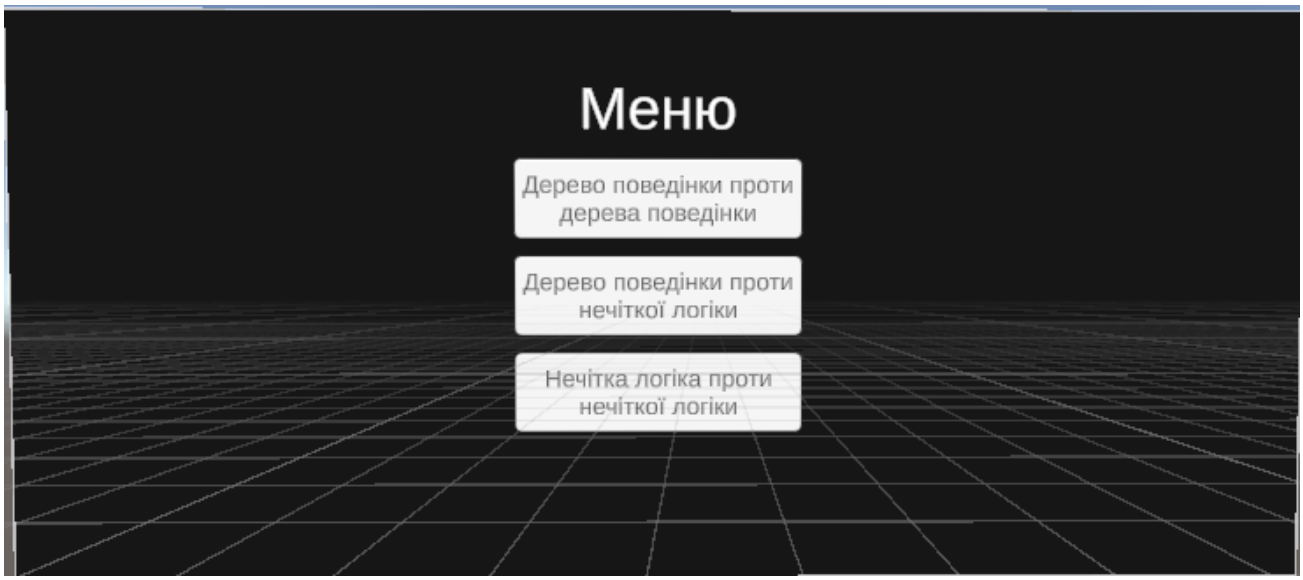

Рисунок 3.10 – Меню вибору моделі поведінки

Кожна кнопка має свій обробник подій, при натисканні на яку, буде відкриватись відповідна сцена за допомогою SceneManager.LoadScene(). Є три обробника подій:

– OnClickBTvsBT(): завантажує сцену «BTvsBT», де обидва NPC використовують дерева рішень;

– OnClickBTvsFL(): завантажує сцену «BTvsFL», де один NPC використовує дерево рішень, а інший – нечітку логіку;

– OnClickFLvsFL(): завантажує сцену «FLvsFL», де обидва NPC використовують нечітку логіку.

3.3.2 Впровадження допоміжних механік

В застосунку передбачена рандомна генерація аптечок та кристалів, що поліпшують шкоду, на кожній половині мапи. Ці об'єкти розміщуються по 3 на кожній половині мапи, де знаходяться NPC: у одного NPC три аптечки та три кристали, і у іншого NPC також три аптечки та три кристали. Такий підхід реалізовано з кількох причин:

– симетричне розміщення об'єктів: розміщення однакової кількості аптечок та кристалів на кожній половині гарантує, що обидва NPC знаходяться в однакових умовах NPC. Це важливо для об'єктивного дослідження стратегій їхньої поведінки.

– запобігання від повторювання сценаріїв: рандомне розміщення об’єктів зменшує можливість повторювання сценаріїв, що, в свою чергу, дає змогу досліджувати унікальки результати раундів;

– пошук ресурсів: NPC змушені шукати аптечки та кристали, що вимагає від них активного пересування по карті. Це дає можливість більш чітко побачити різницю між поведінкою, заснованою на дереві рішень, і поведінкою, що ґрунтується на нечіткій логіці;

– вплив ресурсів на результат: спостереження за тим, як збір різних типів ресурсів (аптечок та кристалів) впливає на ефективність NPC, дозволяє краще зрозуміти, яка стратегія є більш оптимальною.

Якщо NPC зіткається з об’єктом аптечки або кристалу, викликається тригер OnTriggerEnter(Collider other), який обробляє це зіткнення. Якщо кристал зіткнувся з NPC, він збільшує шкоду NPC, збільшує лічильник використаних кристалів і видаляється з мапи.

Фрагмент коду цієї механіки показано нижче.

```
private void OnTriggerEnter(Collider other)
{
   if (other.gameObject.tag == "NPC")
   {
      other.gameObject.GetComponent<NPC>().GunPower += GUNPOWER;
      other.gameObject.GetComponent<NPC>().PowerUps += 1;
      gameObject.SetActive(false);
   }
}
```

Аналогічним чином працює і аптечка, якщо аптечка зіткнулася з NPC, вона відновлює здоров'я NPC, збільшує лічильник використаних аптечок і видаляється з мапи. Фрагмент коду цієї механіки показано нижче.

```
private void OnTriggerEnter(Collider other)
{
   if (other.gameObject.tag == "NPC")
   {
      other.gameObject.GetComponent<NPC>().AddHealth(HEAL);
      other.gameObject.GetComponent<NPC>().Heals += 1;
      gameObject.SetActive(false);
   }
}
```

Також, коли NPC приймає рішення атакувати ворога, він випускає кулі у ворога, як було зазначено в попередньому підрозділі. В цьому випадку генерується об’єкт кулі, яка летить у ворога. Якщо NPC зіткається з об’єктом кулі, так само викликається тригер OnTriggerEnter(Collider other), який обробляє це зіткнення. Якщо куля зіткнулася з об'єктом, що має тег «Environment», тобто оточуюче середовище, вона знищується. Якщо куля зіткнулася з ціллю, вона знищується і завдає цілі шкоди. Фрагмент коду цієї механіки показано нижче.

```
void OnTriggerEnter(Collider other)
{
   if (other.gameObject.tag == "Environment")
   {
      Destroy(gameObject);
   }
   else if (other.gameObject == Target)
   {
      Destroy(gameObject);
      Target.GetComponent<NPC>().TakeDamage(Damage);
   }

}
```

Рандомна генерація аптечок і кристалів з поліпшенням шкоди на мапі стимулює обох NPC до активної поведінки, а також дозволяє їм об’єктивно оцінити різні стратегії прийняття рішень. Це не тільки збільшує динаміку ігрового процесу,

але й сприяє отриманню важливих даних для дослідження оптимальності алгоритмів поведінки NPC.

3.3.3 Підрахунок статистичних даних

Для реалізації підрахунку статистичних даних в застосунку на Unity відстежуються різні параметри, що описують стан кожного NPC, а також загальну статистику гри. Статистичні дані включають в себе наступні аспекти:

- рівень здоров'я кожного NPC;
- кількість можливої нанесеної шкоди;
- кількість підібраних аптечок і кристалів;
- лічильник раундів;
- кількість перемог кожного NPC.

Кожен NPC має початковий рівень здоров'я та рівень шкоди. Кожного разу, коли NPC отримує шкоду, його рівень здоров'я зменшується. Якщо NPC підбирає аптечку, його рівень здоров'я збільшується. Підбір кристалів збільшує рівень шкоди, що завдається NPC противнику.

Гра складається з кількох раундів, у кожному з яких NPC змагаються один з одним. Кожен раз, коли один з NPC перемагає у раунді, його перемога записується до загального рахунку перемог, відбувається перезапуск раунду та оновлення даних про рівень здоров'я та школи. Таким чином, можна відстежувати, який з NPC має більшу кількість перемог за весь процес дослідження.

Окрім відображення рівня здоров'я та шкоди, також ведеться облік кількості підібраних аптечок та кристалів кожним NPC. Інформація про кількість підбирання аптечок та кристілв не скидується з початком кожним раундом. Це дозволяє оцінити, наскільки активно кожен NPC використовує доступні ресурси для покращення своїх характеристик.

На рисунку 3.11 показано шаблон статистичної частини екрану.

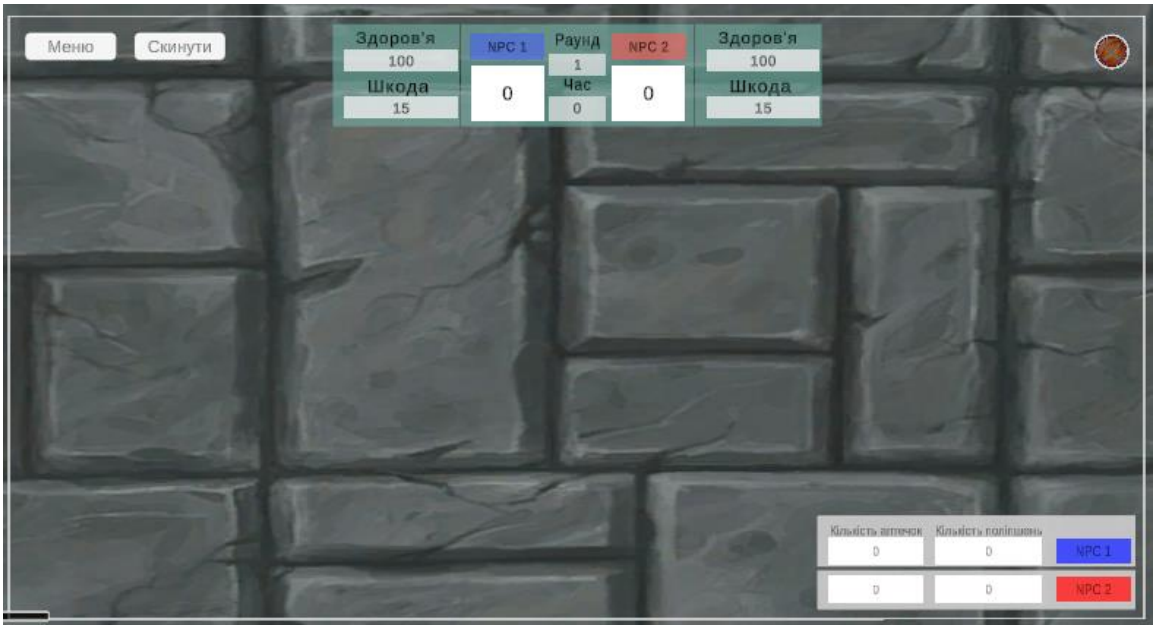

Рисунок 3.11 – Статистична частина екрану

3.4 Порівняння ефективностей стратегій прийняття рішень

Для оцінки ефективності стратегій прийняття рішень NPC в грі на Unity було проведено три експерименти. Кожен експеримент тривав 10 хвилин, протягом яких фіксувалися кількість перемог, зібраних аптечок та покращень шкоди для кожного NPC. Результати експериментів наведені нижче.

Експеримент 1 – дерево рішень проти дерева рішень. Для наглядності порівняння статистичних даних була побувана таблиця 3.1.

Таблиця 3.1 – Статистичні дані режиму «Дерево рішень проти дерева рішень»

Параметр	**NPC 1 (Дерево)**	**NPC 2 (Дерево)**
Кількість перемог	21	19
Кількість підібраних аптечок	52	59
Кількість підібраних покращень шкоди	91	83

Обидва NPC, що використовували алгоритм дерева поведінки, показали більш-менш однакові результати. Кількість перемог, підібраних аптечок та покращень шкоди знаходяться на схожому рівні, що свідчить про рівну ефективність цієї стратегії для обох NPC. Результат експерименту можна побачити на рисунку 3.12.

Рисунок 3.12 – Статистичні дані за 10 хвилин роботи режиму «Дерево рішень проти дерева рішень»

Експеримент 2 – нечітка логіка проти нечіткої логіки.

Розглянемо детальніше реалізацію алгоритму нечіткої логіки.

Як вже було сказано раніше, в коді порівнюються значення приналежності [0-1] до наборів даних: «CalculateCanUseGunPowerValue» та «CalculateCanUseHealValue». Для переходу до дії набір також повинен мати значення більше, ніж параметр «ATTACK_THRESHOLD», який у даному випадку дорівнює 0.2. Метод розрахунку значення приналежності до набору, що відповідає за використання аптечки, враховує такі фактори, як поточний стан здоров'я NPC і відстань до ворога. Обидва фактори мають два порогові значення, в дужках наведено їхні значення, наведені в проекті:

– Стан здоров'я: LOW_HEALTH_THRESHOLD (25), MID_HEALTH_THRESHOLD (50);

– Відстань до ворога: CLOSE_CAN_USE_HEAL_DISTANCE_THRESHOLD (5), FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD (10).

На рисунку 3.13 показана візуалізація значення приналежності до множини, що відповідає за використання аптечки.

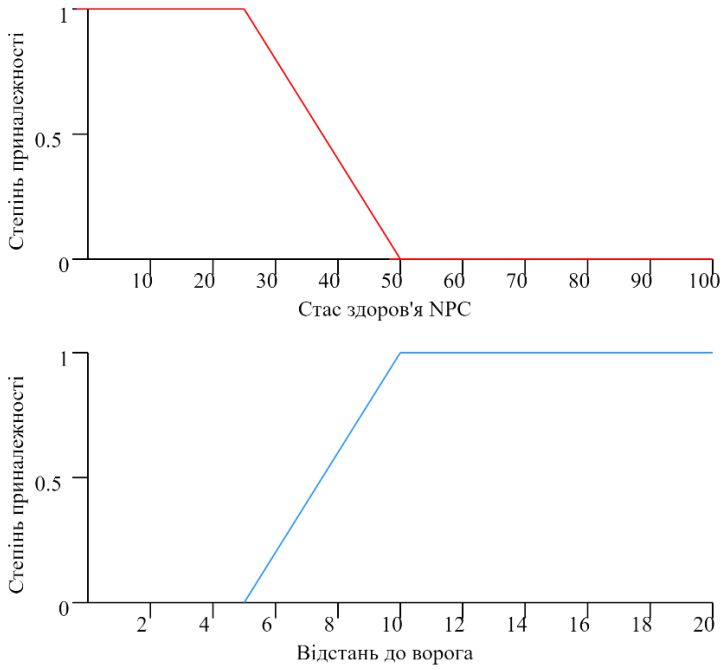

Рисунок 3.13 – Візуалізація значення приналежності до множини, що відповідає за використання аптечки

Наприклад, якщо значення здоров'я NPC нижче значення LOW_HEALTH_THRESHOLD, функція приналежності поверне значення 1. Якщо значення здоров'я дорівнює 35, то повернене значення буде вже 0.6, а якщо значення здоров'я NPC більше, ніж MID_HEALTH_THRESHOLD, то повернеться 0. Ситуація протилежна, коли ми враховуємо відстань до ворога – якщо ця відстань дорівнює 4, то значення приналежності буде дорівнювати 0, коли відстань збільшиться до 9, значення зміниться на 0.8, а коли відстань знову збільшиться і

буде більшою за FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD, значення приналежності буде дорівнювати 1. Завдяки таким правилам, NPC швидше відмовиться від використання аптечки і зосередиться на атаці, якщо ворог знаходиться практично під боком.

У випадку методу, який обчислює значення приналежності до множини, що відповідає за використання пліпшення шкоди, враховується відстань до ворога і відстань до найближчого пліпшення шкоди. При визначенні порогів для цих відстаней використовувалися такі параметри:

– CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD (15);
– FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD (20).

На рисунку 3.14 показана візуалізація значення приналежності до множини, що відповідає за використання поліпшення шкоди.

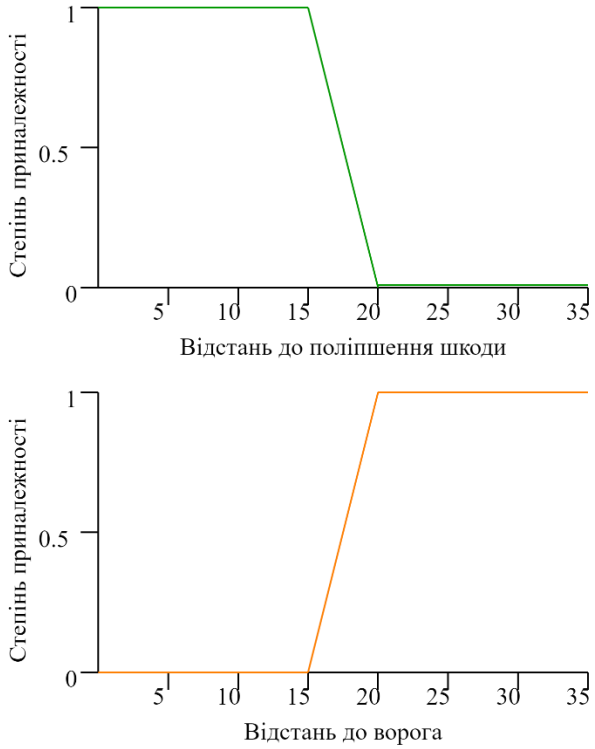

Рисунок 3.14 – Візуалізація значення приналежності до множини, що відповідає за використання поліпшення шкоди

Як і в попередньому випадку – якщо відстань до підкріплення нижче вказаного значення CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD,

то буде повернуто значення 1, тоді як коли ця відстань дорівнює 18, то значення приналежності до множини буде дорівнювати 0.4, коли ця відстань збільшиться до 24, то значення приналежності буде дорівнювати 0. При обчисленні ступеня приналежності по відстані до противника ситуація знову змінюється на зворотну. Значення нижче CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD будуть дорівнювати 0, приклад відстані 16 поверне 0.2, а решта значень, більших за FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD поверне 1.

Коли враховується декілька факторів (наприклад, стан здоров'я та відстань до ворога), їхні значення агрегуються за правилами нечіткої логіки, і їхній мінімум повертається як значення функцій «CalculateCanUseHealValue» та «CalculateCanUseGunPowerValue». У реалізації, показаній вище, значення, які визначають використання аптечки або використання поліпшення шкоди, розраховуються на основі різних факторів. Потім на основі цих значень приймається рішення, яке рішення буде використане в той чи інший момент часу.

Для наглядності порівняння отрмианих статистичних даних була побувана таблиця 3.2.

Таблиця 3.2 – Статистичні дані режиму «Нечітка логіки проти нечіткої логіки»

Параметр	**NPC 1 (Логіка)**	**NPC 2 (Логіка)**
Кількість перемог	20	21
Кількість підібраних аптечок	50	38
Кількість підібраних покращень шкоди	90	103

Обидва NPC, що використовували алгоритм нечіткої логіки, також показали схожі результати. Різниця в кількості перемог, підібраних аптечок та покращень шкоди не є значною, що свідчить про рівну ефективність цієї стратегії для обох NPC. Результат експерименту можна побачити на рисунку 3.15.

Рисунок 3.15 – Статистичні дані за 10 хвилин роботи режиму «Нечітка логіки проти нечіткої логіки»

Експеримент 3 – дерево рішень проти нечіткої логіки. Для наглядності порівняння статистичних даних була побувана таблиця 3.3.

Таблиця 3.3 – Статистичні дані режиму «Нечітка логіки проти нечіткої логіки»

Параметр	**NPC 1 (Дерево)**	**NPC 2 (Логіка)**
Кількість перемог	33	10
Кількість підібраних аптечок	76	40
Кількість підібраних покращень шкоди	105	112

Коли один NPC використовував дерево поведінки, а інший - нечітку логіку, NPC з деревом поведінки показав значно кращі результати. NPC з деревом поведінки виграв більше разів і підібрав більше аптечок, хоча NPC з нечіткою логікою зібрав більше покращень шкоди. Дерево рішень частіше статистично приймає рішення підібрати аптечку, що дозволяє NPC підтримувати високий рівень здоров'я і, відповідно, вигравати більше боїв. Така поведінка може бути

обумовлена чіткістю правил та пріоритетів в дереві рішень, які дозволяють NPC ефективніше реагувати на небезпеку та потребу в лікуванні. Результат експерименту можна побачити на рисунку 3.16.

Рисунок 3.16 – Статистичні дані за 10 хвилин роботи режиму «Дерево рішень проти нечіткої логіки»

Алгоритм дерева рішень виявився оптимальнішим у даних умовах завдяки своїй здатності приймати швидкі та чіткі рішення на основі визначених умов. Нечітка логіка може бути оптимальнішою у складніших сценаріях, де необхідно враховувати багато факторів одночасно і де потрібна більш гнучка адаптація до змінних умов. Наприклад, у випадках, коли на полі бою присутні більше NPC або інші перешкоди, нечітка логіка може краще адаптуватися і приймати більш виважені рішення.

Висновки до розділу 3

В результаті проведеної роботи був розроблений застосунок для дослідження поведінки NPC в ігровому середовищі Unity. Основною метою даного застосунку було дослідження ефективності різних стратегій прийняття рішень.

Ігрове середовище було спроектоване таким чином, щоб максимально точно відображати умови, в яких NPC можуть приймати рішення щодо атак на ворогів, підбору аптечок для відновлення здоров'я або підбору поліпшень для збільшення завданої шкоди. Для цього була побудована мапа, на якій два NPC взаємодіяли відповідно до обраних стратегій.

У процесі виконання програми збиралися та аналізувалися такі дані:

- рівень здоров'я кожного NPC;
- кількість нанесеної шкоди;
- кількість підібраних аптечок і кристалів кожним NPC;
- кількість перемог кожного NPC.

Аналіз результатів дослідження проводився для трьох варіантів протистоянь:

- дерево поведінки проти дерева поведінки;
- дерево поведінки проти нечіткої логіки;
- нечітка логіка проти нечіткої логіки.

Випробування показали, що в ситуаціях, коли обидва NPC використовували однакову поведінкову стратегію (дерево поведінки або нечітка логіка), їхні показники були майже однаковими. Однак, коли один NPC реалізував стратегію дерева поведінки, а інший – стратегію нечіткої логіки, було виявлено наступні закономірності:

- NPC з деревом поведінки виграв більше разів і підібрав більше аптечок;
- NPC з нечіткою логікою зібрав більше поліпшень для збільшення завданої шкоди.

Це пояснюється тим, що алгоритм дерева поведінки частіше приймає рішення підібрати аптечку, що дозволяє NPC підтримувати високий рівень здоров'я

і, відповідно, вигравати більше боїв. Така поведінка обумовлена чіткістю правил та пріоритетів в дереві рішень, які дозволяють NPC ефективніше реагувати на небезпеку та потребу в лікуванні.

Алгоритм дерева рішень виявився оптимальнішим у даних умовах завдяки своїй здатності приймати швидкі та чіткі рішення на основі визначених умов. Нечітка логіка, у свою чергу, може бути оптимальнішою у складніших сценаріях, де необхідно враховувати багато факторів одночасно і де потрібна більш гнучка адаптація до змінних умов. Наприклад, у випадках, коли на полі бою присутні більше NPC або інші перешкоди, нечітка логіка може краще адаптуватися і приймати більш виважені рішення.

ВИСНОВКИ

У даній роботі було розглянуто та досліджено сучасні підходи до розробки поведінки NPC в ігрових середовищах. Аналіз предметної області дозволив виявити ключові аспекти поведінки NPC та їх вплив на ігровий процес.

Досліджено можливості використання штучного інтелекту, зокрема дерев поведінки та нечіткої логіки, для створення більш реалістичних та адаптивних NPC. Порівняння методів дерев поведінки та нечіткої логіки в плані їх використання для прийняття рішень NPC показало, що кожен з них має свої сильні сторони та недоліки. Дерева поведінки забезпечують структурованість та простоту в реалізації, тоді як нечітка логіка дозволяє враховувати більш широкий спектр вхідних даних та робить поведінку NPC більш гнучкою.

У результаті проведеної роботи був розроблений застосунок для дослідження поведінки NPC в ігровому середовищі Unity. Основною метою цього застосунку було дослідження ефективності різних стратегій прийняття рішень.

Аналіз результатів показав, що коли обидва NPC використовували однакову стратегію, їхні показники були майже однаковими. Однак, коли NPC використовували різні алгоритми поведінки, NPC з деревом поведінки вигравав більше разів. Це пояснюється тим, що цей алгоритм частіше приймає рішення підібрати аптечку, дозволяючи NPC підтримувати високий рівень здоров'я і вигравати більше боїв. Дерево рішень виявилося оптимальнішим у цих умовах завдяки здатності приймати швидкі та чіткі рішення на основі визначених умов. Нечітка логіка, у свою чергу, може бути кращою в складніших сценаріях, де необхідно враховувати ще більше факторів одночасно.

Результати цієї роботи можуть мати широке застосування в розробці ігор. Їх можна використовувати для покращення ігрового досвіду та створення більш складних і цікавих ігор. Ця робота спрямована на розширення теоретичних знань та практичних навичок у галузі розробки штучного інтелекту для ігор. Вона може стати основою для подальших досліджень та розробок у цій сфері.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Георгіус Н. Я., Тогелиус Ю. Artificial Intelligence and Games. Oxford : Springer, 2018. 337 с.

2. Гордієнко А. В. Комп'ютерні ігри на їх позитивні психологічні ефекти. Київ, 2017. 35 с.

3. Мілінгтон І. Створення штучного інтелекту в іграх. США : Taylor & Francis, 2006. 856 с.

4. Паласиос Х. Програмування штучного інтелекту в іграх. Київ, 2017. 272 с.

5. Ashworth J. The Game Master's Book of Non-Player Characters. USA : Media Lab Books, 2021. 272 с.

6. Barrera R. Unity AI Game Programming. Birmingham : Mumbaj, 2015. 232 с.

7. Colledanchise M. Behavior Trees in Robotics. Stockholm : Doctor, 2017. 75 с.

8. Corva Veo. Non-Player Character. USA, 2021. 376 с.

9. Gegov A. Complexity Management in Fuzzy Systems: A Rule Base Compression Approach. Luxemburg : Springer, 2007. 364 с.

10. Heineman G.T., Pollice G., Selkow S. Algorithms in a Nutshell. Boston : O'Reilly Media, 2016. 390 с.

11. Hooda D.S, Raich V. Fuzzy Logic Models and Fuzzy Control. Oxford, 2017. 409 с.

12. Klirwater D. What Defines Video Game Genre? Thinking about Genre Study. Канада : The Journal of the Canadian Game Studies Association, 2011. 49 с.

13. Kodeco. Pathfinding With NavMesh: Getting Started. URL: https://www.kodeco.com/16977649-pathfinding-with-navmesh-getting-started (дата звернення 25.05.2024).

14. Marcotte R., Hamilton H. J. Behavior Trees for Modelling Artificial Intelligence in Games. Regina, 2017. 114 с.

15. McEwen D. Non-Player Character. Melbourne, 2023. 98 с.

16. Ogren P. Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. Minneapolis : AIAA Guidance, 2012. 17 с.

17. Pirovano M. Fuzzy Tactics: A scripting game that leverages fuzzy logic as an engaging game mechanic: дис. канд. тех. наук. Мілано, 2014. 13 с.

18. Rabin S. Game AI Pro: Collected Wisdom of Game AI Professionals. USA : A K Peters/CRC Press, 2023. 626 с.

19. Roche D. Randomness in Computing, Game Trees Unit. USA : United States Naval Academy, 2013. 263 с.

20. The Many Different Types of Video Games & Their Subgenres. iD Tech. URL: https://www.idtech.com/blog/different-types-of-video-game-genres (дата звернення 04.05.2024).

21. The present and future of Artificial Intelligence in Game Development. URL: https://logicsimplified.com/newgames/the-present-and-future-of-artificial-intelligence-in-game-development/ (дата звернення 07.05.2024).

22. Unity Learn. Behaviour Trees. URL: https://learn.unity.com/project/behaviour-trees (дата звернення 09.05.2024).

23. Unity Manual. Colliders. URL: https://docs.unity3d.com/ru/2020.1/Manual/CollidersOverview.html (дата звернення 18.05.2024).

24. Unity Manual. Configure Rigidbody Colliders. URL: https://docs.unity3d.com/Manual/rigidbody-configure-colliders.html (дата звернення 16.05.2024).

25. Unity Manual. Mesh Renderer component. URL: https://docs.unity3d.com/Manual/class-MeshRenderer.html (дата звернення 17.05.2024).

26. Why We Like Video Games (Maybe We're Control Freaks). AARP. URL: https://www.aarp.org/home-family/personal-technology/info-2021/video-games-pastimes.html (дата звернення 01.05.2024).

ДОДАТОК А

Лістинг коду базового класу NPC

```
using UnityEngine;
using UnityEngine.AI;

public class NPC : MonoBehaviour
{
    protected const float SPEED = 12f;
    protected const float ACCELERATION = 55f;
    protected const float SHOOT_DELAY = 0.75f;
    protected const int DEFAULT_HEALTH = 100;
    protected const int RANGE_ATTACK = 10;
    protected const int RANGE_POWERUPS = 2;
    private const float BULLET_SPEED = 30.0f;
    private const int DEFAULT_GUNPOWER = 5;

    private int health;
    private int gunpower;
    private int wins;
    private int heals;
    private int powerups;
    private float shootTimer = 0.0f;

    protected NavMeshAgent agent;

    private GameObject[] healthPacks;
    private GameObject[] gunPowerPacks;

    [SerializeField] protected GameObject enemy;
    [SerializeField] private GameObject bulletPrefab;
    [SerializeField] private Transform respawnPosition;
    [SerializeField] private Transform gun;

    public int Health
    {
        get => health;
        set => health = value;
    }

    public int GunPower
    {
        get => gunpower;
        set => gunpower = value;
    }

    public int Wins
    {
        get => wins;
        set => wins = value;
    }

    public int Heals
    {
        get => heals;
        set => heals = value;
    }

    public int PowerUps
```

```
{
    get => powerups;
    set => powerups = value;
}


public float ShootTimer
{
    get => shootTimer;
    set => shootTimer = value;
}

void Awake()
{
    Health = DEFAULT_HEALTH;
    GunPower = DEFAULT_GUNPOWER;
    Wins = 0;
    Heals = 0;
    PowerUps = 0;
    healthPacks = GameObject.FindGameObjectsWithTag("Heal");
    gunPowerPacks = GameObject.FindGameObjectsWithTag("GunPower");

    agent = GetComponent<NavMeshAgent>();
    agent.speed = SPEED;
    agent.acceleration = ACCELERATION;
}

public void ResetNPC()
{
    agent.enabled = false;
    agent.gameObject.transform.position = respawnPosition.transform.position;
    agent.gameObject.transform.rotation = respawnPosition.transform.rotation;
    agent.enabled = true;

    Health = DEFAULT_HEALTH;
    GunPower = DEFAULT_GUNPOWER;
}

public void AddHealth(int heal)
{
    Health += heal;

    if (Health > DEFAULT_HEALTH)
        Health = DEFAULT_HEALTH;
}

public void TakeDamage(int damage)
{
    Health -= damage;

    if (Health <= 0)
    {
        ResetNPC();
        enemy.GetComponent<NPC>().ResetNPC();
        enemy.GetComponent<NPC>().Wins+=1;
    }
}

protected void ShootIfPossible()
{
    if (CanSeeTarget())
```

```
    {
        if (Time.time > ShootTimer)
        {
            Shoot();
            ShootTimer = Time.time + SHOOT_DELAY;
        }
    }
}

private bool CanSeeTarget()
{
    Vector3 direction = enemy.transform.position - transform.position;
    RaycastHit hit;

    if (Physics.Raycast(transform.position, direction, out hit))
    {
        if (hit.collider.gameObject.transform.position == enemy.transform.position)
        {
            return true;
        }
    }
    return false;
}

private void Shoot()
{
    Vector3 directionToEnemy = (enemy.transform.position - transform.position).normalized;
    Quaternion bulletRotation = Quaternion.LookRotation(directionToEnemy) * Quaternion.Euler(0, 90, 0);

    GameObject bullet = Instantiate(bulletPrefab, gun.transform.position, bulletRotation);
    bullet.GetComponent<Bullet>().Init(GunPower, enemy);
    bullet.GetComponent<Rigidbody>().velocity = directionToEnemy * BULLET_SPEED;
}

protected GameObject FindNearestHealthPack()
{
    float distance = Mathf.Infinity;
    GameObject nearestHealthPack = null;

    foreach (GameObject heal in healthPacks)
    {
        if (heal.activeSelf)
        {
            float currDistance = Vector3.Distance(transform.position, heal.transform.position);
            if (currDistance < distance)
            {
                distance = currDistance;
                nearestHealthPack = heal;
            }
        }
    }

    return nearestHealthPack;
}

protected GameObject FindNearestGunPowerPack()
{
    float distance = Mathf.Infinity;
    GameObject nearestGunPowerPack = null;

    foreach (GameObject gunPower in gunPowerPacks)
```

```
        {
            if (gunPower.activeSelf)
            {
                float currDistance = Vector3.Distance(transform.position, gunPower.transform.position);
                if (currDistance < distance)
                {
                    distance = currDistance;
                    nearestGunPowerPack = gunPower;
                }
            }
        }

        return nearestGunPowerPack;
    }

    protected GameObject FindNearestObject(GameObject object1, GameObject object2)
    {
        if (object1 == null)
            return object2;
        else if (object2 == null)
            return object1;


        if (Vector3.Distance(transform.position, object1.transform.position) <=
            Vector3.Distance(transform.position, object2.transform.position))
            return object1;
        else
            return object2;
    }

    private void DestroyAllBullets()
    {
        GameObject[] bullets = GameObject.FindGameObjectsWithTag("Bullet");

        for(int i = bullets.Length-1; i >= 0; i--)
        {
            Destroy(bullets[i]);
        }
    }

    protected void UseHeal()
    {
        agent.stoppingDistance = RANGE_POWERUPS;
        agent.SetDestination(FindNearestHealthPack().transform.position);
    }

    protected void UseGunPower()
    {
        agent.stoppingDistance = RANGE_POWERUPS;
        agent.SetDestination(FindNearestGunPowerPack().transform.position);
    }

    protected void Attack()
    {
        agent.stoppingDistance = RANGE_ATTACK;
        transform.rotation = Quaternion.LookRotation(enemy.transform.position - transform.position, Vector3.up);
        ShootIfPossible();
        agent.SetDestination(enemy.transform.position);
    }
}
```

ДОДАТОК Б

Лістинг коду NPC алгоритмом дерева рішень

```
using UnityEngine;

public class NPCBehaviorTree : NPC
{
    private const int HEALTH_THRESHOLD = 50;
    private const int RANGE_HEAL = 5;

    void Update()
    {
        if (CanUseHeal())
        {
            UseHeal();
        }
        else if (CanUseGunPower())
        {
            UseGunPower();
        }
        else
        {
            Attack();
        }
    }
    bool CanUseHeal()
    {
        return FindNearestHealthPack() != null && Health <= HEALTH_THRESHOLD &&
Vector3.Distance(gameObject.transform.position, enemy.transform.position) >= RANGE_HEAL;
    }

    bool CanUseGunPower()
    {
        return FindNearestGunPowerPack() != null && (FindNearestObject(enemy, FindNearestGunPowerPack()).tag ==
"GunPower");
    }
}
```

ДОДАТОК В

Лістинг коду NPC алгоритмом нечіткої логіки

```
using UnityEngine;

public class NPCFuzzyLogic : NPC
{
   private const int LOW_HEALTH_THRESHOLD = 25;
   private const int MID_HEALTH_THRESHOLD = 50;
   private const int CLOSE_CAN_USE_HEAL_DISTANCE_THRESHOLD = 5;
   private const int FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD = 10;
   private const int CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD = 15;
   private const int FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD = 20;
   private const float ATTACK_THRESHOLD = 0.2f;

   void Update()
   {
      if (CalculateCanUseGunPowerValue() >= CalculateCanUseHealValue() && CalculateCanUseGunPowerValue() >
ATTACK_THRESHOLD)
      {
         UseGunPower();
      }
      else if (CalculateCanUseHealValue() >= CalculateCanUseGunPowerValue() && CalculateCanUseHealValue() >
ATTACK_THRESHOLD)
      {
         UseHeal();
      }
      else
      {
         Attack();
      }
   }


   private float CalculateCanUseHealValue()
   {
      if (FindNearestHealthPack() == null)
         return 0.0f;

      float lowHealthValue = 0.0f;
      if (Health <= LOW_HEALTH_THRESHOLD)
      {
         lowHealthValue = 1.0f;
      }
      else if (Health > LOW_HEALTH_THRESHOLD && Health <= MID_HEALTH_THRESHOLD)
      {
         lowHealthValue = (MID_HEALTH_THRESHOLD - Health) / (MID_HEALTH_THRESHOLD -
LOW_HEALTH_THRESHOLD);
      }

      float distanceToEnemy = Vector3.Distance(gameObject.transform.position, enemy.transform.position);

      float closeDistanceValue = 0.0f;
      if (distanceToEnemy >= FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD)
      {
         closeDistanceValue = 1.0f;
      }
```

```
        else if (distanceToEnemy < FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD && distanceToEnemy >=
CLOSE_CAN_USE_HEAL_DISTANCE_THRESHOLD)
        {
            closeDistanceValue = 1 - (FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD - distanceToEnemy) /
(FAR_CAN_USE_HEAL_DISTANCE_THRESHOLD - CLOSE_CAN_USE_HEAL_DISTANCE_THRESHOLD);
        }

        float canUseHealValue = Mathf.Min(lowHealthValue, closeDistanceValue);

        return canUseHealValue;
    }

    private float CalculateCanUseGunPowerValue()
    {
        if (FindNearestGunPowerPack() == null)
            return 0.0f;

        float distanceToEnemy = Vector3.Distance(gameObject.transform.position, enemy.transform.position);

        float closeEnemyValue = 0.0f;
        if (distanceToEnemy >= FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD)
        {
            closeEnemyValue = 1.0f;
        }
        else if (distanceToEnemy < FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD && distanceToEnemy >=
CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD)
        {
            closeEnemyValue = 1 - (FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD - distanceToEnemy) /
(FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD -
CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD);
        }

        float distanceToGunPower = Vector3.Distance(gameObject.transform.position,
FindNearestGunPowerPack().transform.position);

        float closeGunPowerValue = 0.0f;
        if (distanceToGunPower <= CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD)
        {
            closeGunPowerValue = 1.0f;
        }
        else if (distanceToGunPower > CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD &&
distanceToGunPower <= FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD)
        {
            closeGunPowerValue = (FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD - distanceToGunPower) /
(FAR_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD -
CLOSE_CAN_USE_GUNPOWER_DISTANCE_THRESHOLD);
        }

        float canUseGunPowerValue = Mathf.Min(closeEnemyValue, closeGunPowerValue);

        return canUseGunPowerValue;
    }
}
```