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
_____Юрій КОНДРАТЕНКО
«____»_____ 20___ р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВР

НЕЙРОМЕРЕЖЕВІ ТЕХНОЛОГІЇ ДЛЯ ПОКРАЩЕННЯ ІНТЕРАКТИВНИХ ПРОЦЕСІВ В ІГРАХ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____Єлизавета СТРЕЗЄВА
«___»_____ 2025 р.

Керівник канд. техн. наук, доцент _____ Євген СІДЕНКО
«___»_____ 2025 р.

Миколаїв – 2025

Чорноморський національний університет імені Петра Могили

(повне найменування закладу вищої освіти)

Факультет	Комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Юрій КОНДРАТЕНКО

«____» _____ 20___ р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Стрезєва Єлизавета Миколаївна

(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи «Нейромережеві технології для покращення інтерактивних процесів в іграх».

Керівник роботи Сіденко Євген Вікторович, доцент кафедри інтелектуальних інформаційних систем, канд. техн. наук, доцент.

Затв. наказом Ректора ЧНУ ім. Петра Могили від «18» жовтня 2024 р. № 321.

2. Строк представлення кваліфікаційної роботи студентом «20» червня 2025 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: проведення навчання нейронної мережі для неігрових персонажів, для виконання заданих завдань.

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

– дослідити існуючі симулятори дикої природи та проаналізувати підходи до побудови поведінки NPC;

– ознайомитися з можливостями Unity ML-Agents Toolkit;

– реалізувати навчання NPC;

– провести тестування, зібрати дані про ефективність навчання ігрових агентів.

5. Перелік графічних матеріалів: презентація.

Керівник роботи _____ Євген СІДЕНКО

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Єлизавета СТРЕЗЄВА

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «18» грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної роботи

Тема: Нейромережеві технології для покращення інтерактивних процесів в іграх.

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	13.10.2024	25.10.2024	Виконано
2	Аналіз предметної області та постановка задачі	02.12.2024	07.12.2024	Виконано
3	Огляд літературних джерел за темою кваліфікаційної роботи, зокрема огляд публікацій та аналогічних систем, щодо використання нейронних мереж в відеоіграх	10.12.2024	25.12.2024	Виконано
4	Огляд існуючих архітектур нейронних мереж для вирішення поставленої задачі	21.04.2025	21.04.2025	Виконано
5	Реалізація обраних технологій з аналізом отриманих результатів	22.04.2025	02.05.2025	Виконано
6	Перший попередній захист КР на засіданні комісії кафедри	03.05.2025	04.05.2025	Виконано
7	Корегування роботи за результатами попереднього захисту	06.05.2025	07.06.2025	Виконано
8	Другий попередній захист КР на засіданні комісії кафедри	16.05.2025	16.05.2025	Виконано
9	Доробка та остаточне оформлення КР	17.05.2025	07.06.2025	Виконано
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	30.05.2025	30.05.2025	Виконано

Керівник роботи _____ Євген СІДЕНКО

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Єлизавета СТРЕЗЄВА

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану
«29» січня 2025 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувачки групи 401 ЧНУ ім. Петра Могили

Стрезєвої Єлизавети Миколаївни

на тему: **«НЕЙРОМЕРЕЖЕВІ ТЕХНОЛОГІЇ ДЛЯ ПОКРАЩЕННЯ ІНТЕРАКТИВНИХ ПРОЦЕСІВ В ІГРАХ»**

Кваліфікаційну роботу присвячено розробці симулятора дикої природи з використанням штучного інтелекту та нейромережевих технологій. Актуальність роботи зумовлена необхідністю створення реалістичної ігрової екосистеми, де NPC (неігрові персонажі) демонструють адаптивну поведінку на основі навчання штучного інтелекту.

Об’єктом кваліфікаційної роботи є процес моделювання поведінки NPC у симуляторі дикої природи.

Предметом кваліфікаційної роботи є нейронні мережі для створення реалістичної поведінки ігрових агентів.

Метою роботи є покращення інтерактивних процесів в іграх за рахунок розробки симулятора дикої природи з NPC, поведінка яких моделюється за допомогою нейронних мереж, що дозволяє створити інтерактивний та динамічний ігровий світ.

Для досягнення поставленої мети визначено такі завдання:

– дослідити існуючі симулятори дикої природи та проаналізувати підходи до побудови поведінки NPC;

– ознайомитися з можливостями Unity ML-Agents Toolkit;

– реалізувати навчання NPC;

– провести тестування, зібрати дані про ефективність навчання ігрових агентів.

Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків та переліку джерел. У вступі обґрунтовано актуальність роботи, визначено об’єкт, предмет, мету та завдання дослідження.

У першому розділі виконано аналіз предметної області, вивчено сучасні підходи до моделювання поведінки NPC та можливості застосування нейронних мереж у симуляціях. Розглянуто основні технології, що використовуються у галузі.

Другий розділ присвячено вибору та аналізу технологічних компонентів, зокрема ігрового рушія, мов програмування, бібліотек машинного навчання та системи тренування ML-агентів. Виконано порівняльний аналіз інструментів та обґрунтовано вибір Unity у поєднанні з ML-Agents.

У третьому розділі описано процес розробки інформаційної системи симулятора дикої природи, включно з проєктуванням структури, налаштуванням середовища, створенням моделей поведінки NPC та їх інтеграцією в ігровий процес. Розроблено архітектуру симулятора та визначено методи обробки даних про стан NPC та ігрового середовища.

Четвертий розділ зосереджено на програмній реалізації та тестуванні. Наведено деталі реалізації компонентів системи, описано алгоритми поведінки NPC, налаштування системи тренування ML-агентів та проведено тестування симулятора. Визначено критерії оцінки якості роботи моделі та проаналізовано результати тестування.

У висновках підсумовано результати виконаної роботи, сформульовано основні досягнення та визначено можливі напрями подальшого розвитку проєкту.

Кваліфікаційна робота викладена на 87 сторінках, містить 4 розділи, 58 ілюстрацій, 9 таблиць, 3 додатки та 27 джерел у переліку посилань.

Ключові слова: *штучний інтелект, нейронні мережі, симулятор дикої природи, Unity, машинне навчання, ML-Agents, поведінка NPC.*

ABSTRACT

qualification work of

the student of 401 group of Petro Mohyla Black Sea National University

Strezieva Yelyzaveta Mykolaivna

Theme: **‘Neural network technologies for improving interactive processes in games’**

The qualification work is devoted to the development of a wildlife simulator using artificial intelligence and neural network technologies. The relevance of the work is determined by the need to create a realistic gaming ecosystem where NPCs (non-player characters) demonstrate adaptive behavior based on artificial intelligence learning.

The object of the thesis is the process of modeling NPC behavior in a wildlife simulator.

The subject of the thesis is neural networks for creating realistic behavior of game agents.

The goal of the work is to improve interactive processes in games by developing a wildlife simulator with NPCs whose behavior is modeled using neural networks, allowing for the creation of an interactive and dynamic game world.

To achieve this goal, the following tasks have been defined:

– research existing wildlife simulators and analyze approaches to building NPC behavior;

– familiarize yourself with the capabilities of the Unity ML-Agents Toolkit;

– implement NPC training;

– conduct testing, collect data on the effectiveness of game agent training.

The thesis consists of an introduction, four chapters, conclusions, and a list of sources. The introduction justifies the relevance of the work and defines the object, subject, goal, and tasks of the research.

The first chapter analyzes the subject area, studies modern approaches to modeling NPC behavior, and the possibilities of using neural networks in simulations. The main technologies used in the industry are considered.

The second chapter is devoted to the selection and analysis of technological components, in particular the game engine, programming languages, machine learning libraries, and the ML-agent training system. A comparative analysis of tools is performed, and the choice of Unity in combination with ML-Agents is justified.

The third chapter describes the process of developing an information system for a wildlife simulator, including designing the structure, configuring the environment, creating NPC behavior models, and integrating them into the gameplay. The simulator architecture is developed, and methods for processing data about the state of NPCs and the game environment are defined.

The fourth chapter focuses on software implementation and testing. Details of the implementation of system components are provided, NPC behavior algorithms are described, the ML agent training system is configured, and the simulator is tested. Criteria for evaluating the quality of the model are defined and the test results are analyzed.

The conclusions summarize the results of the work, formulate the main achievements, and identify possible directions for further development of the project.

The bachelor's thesis is presented on 87 pages, contains 4 sections, 58 illustrations, 9 tables, 3 appendices, and 27 sources in the list of references.

Keywords: *artificial intelligence, neural networks, wildlife simulator, Unity, machine learning, ML-Agents, NPC behavior.*

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ШІ – Штучний Інтелект

AR/VR – Augmented/Virtual Reality

FBX – Filmbox

FSM – Finite State Machine

GLTF – GL Transmission Format

MCTS – Monte Carlo Tree Search

ML – Machine Learning

ML-Agents – Machine Learning Agents

NPC – Non-Player Character

ONNX – Open Neural Network Exchange

PPO – Proximal Policy Optimization

RL – Reinforcement Learning

SAC – Soft Actor-Critic

UI – User Interface

UXML – Unity Extensible Markup Language

USS – Unity Style Sheets

ВСТУП

У сучасному світі комп'ютерні ігри є не лише засобом розваги, а й потужним інструментом для навчання, тренувань і досліджень. Важливим аспектом сучасних ігор є взаємодія гравця з неігровими персонажами (NPC), що може суттєво вплинути на якість ігрового досвіду. Проте традиційні підходи до створення NPC мають обмеження, адже вони часто працюють за заздалегідь прописаними сценаріями, що робить їхню поведінку передбачуваною та неадаптивною.

Актуальність даної роботи зумовлена необхідністю вдосконалення інтелектуальних можливостей NPC за допомогою сучасних технологій машинного навчання. Використання нейронних мереж та платформи Unity з бібліотекою ML-Agents дозволяє навчати NPC самостійно ухвалювати рішення та адаптуватися до дій гравця в режимі реального часу. Це відкриває можливості для створення реалістичних симуляцій дикої природи, де NPC поводяться як справжні тварини, реагуючи на зміни середовища та взаємодію з гравцем.

Розробка такої системи не лише підвищує рівень інтерактивності ігор, але й сприяє дослідженню адаптивної поведінки агентів, що має значення для інших галузей, зокрема робототехніки та дослідження штучного інтелекту.

1 АНАЛІЗ СУЧАСНОГО СТАНУ ПОСТАВЛЕНОЇ ЗАДАЧІ

1.1 Опис предметної області

Відеоігри (ігри, розроблені для настільних систем та портативних пристроїв, таких як ноутбуки, планшети та смартфони) стали значущою індустрією для XXI століття завдяки тому, що в часі вони постійно еволюціонували і зуміли нав'язати в наш час домінуючі медіа культури.

У дослідженнях, присвячених еволюції відеоігор, зазначається, що їх можна розділити на кілька поколінь, залежно від технічних і технологічних змін в індустрії. [7]

Перше покоління охоплює період з 1971 по 1977 роки та пов’язане із появою популярних аркадних ігор і перших ігрових консолей. Друге покоління починається з випуском Video Computer System компанією Atari у 1977 році. Цей період отримав назву «Золота епоха аркадних ігор», яка тривала до 1983 року, коли індустрія пережила кризу.

У наступний період 1989–1996 років відбулися суттєві удосконалення в області графіки та пам'яті, що дозволило створювати ігри з кращою візуальною якістю та складнішим геймплеєм.

П’яте покоління (1994–1999) відзначилося появою мобільних платформ та портативних пристроїв для ігор. У шостому періоді (1999–2004) компанія Sega покинула ринок, поступившись місцем Microsoft. Сьоме покоління, яке триває з 2004 року до сьогодні, характеризується зростанням популярності онлайн-ігор та ігор для мобільних пристроїв.

Відеоігри від самого початку тісно пов'язані з розвитком новітніх технологій.

Програмний або первинний штучний інтелект у відеоіграх визначається як набір програмних методів, що використовуються в ігровому рушії для імітації інтелекту в діях керованих комп'ютером персонажів, що відображає одну з основних характеристик інтелекту, яка полягає в «сприйнятті релевантних зв'язків у конкретній ситуації». [2]

Ігровий рушій, який керує ШІ у відеоіграх, окрім традиційних методів ШІ, також включає алгоритми з теорії управління, робототехніки, комп'ютерної графіки та інформатики в цілому.

Починаючи з найперших ігор, гравці взаємодіяли з автономними сутностями у відеоіграх, яких зазвичай називають неігровими персонажами (NPC). NPC населяють ігровий світ, щоб забезпечити захоплюючий ігровий досвід, і тому вони повинні бути максимально реалістичними у своєму зовнішньому вигляді, рухах, діалогах і рішеннях. Залежно від типу гри та її цілей, ШІ може мати різні ролі та функції в управлінні цими різними об'єктами у відеогрі.

Один з найпоширеніших прийомів, відомий як алгоритм кінцевого автомата (FSM), був впроваджений у дизайн відеоігор на початку 1990-х років (рис. 1.1). У FSM дизайнер узагальнює всі можливі ситуації, з якими може зіткнутися штучний інтелект, а потім програмує конкретну реакцію для кожної ситуації. По суті, ШІ FSM негайно реагує на дії гравця своєю запрограмованою поведінкою. [7]

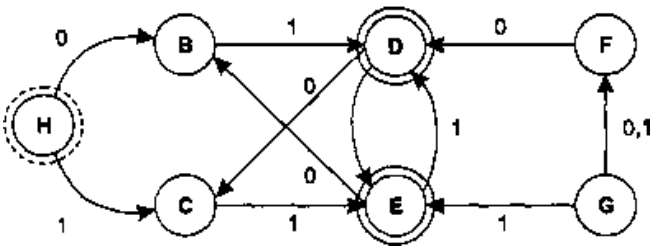

Рисунок 1.1 – Схема роботи алгоритму кінцевого автомата

У іграх-шутерах штучний інтелект може атакувати гравця, але відступати, коли здоров'я аватара стає небезпечно низьким, що призводить до потенційного «кінця гри». У грі на основі FSM конкретний персонаж має можливість виконувати чотири основні дії у відповідь на потенційні сценарії: шукати допомоги, ухилятися, блукати і діяти. Багато відомих ігор, таких як Battlefield, Call of Duty і Tomb Raider, містять успішні приклади розробки ШІ на основі FSM.

Більш просунутий метод, який розробники використовують для покращення індивідуалізованого ігрового досвіду, - це алгоритм Монте-Карло.

Алгоритм MCTS був створений, щоб уникнути аспекту повторення, присутнього в алгоритмі FSM (рис. 1.2). Алгоритм MCTS спочатку оцінює всі можливі ходи, доступні NPC в даний момент. Потім, для кожного з цих можливих ходів, він аналізує всі дії, якими гравець міг би відповісти. Після цього він повертається, щоб оцінити оцінку NPC на основі дій гравця.

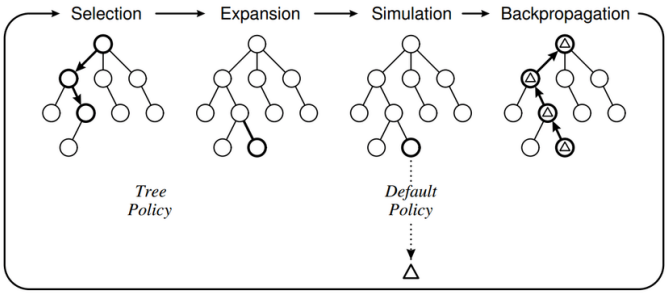

Рисунок 1.2 – Схема роботи алгоритму MCTS

Цей алгоритм штучного інтелекту був використаний компанією IBM для створення Deep Blue, першого шахового суперкомп'ютера.

Подібний алгоритм використовується в багатьох стратегічних іграх. Однак, оскільки в них набагато більше можливих ходів, ніж у шахах, врахувати їх усі неможливо. У таких іграх алгоритм MCTS випадковим чином вибирає деякі з можливих ходів. В результаті дії NPC стають набагато менш передбачуваними для гравців.

Одним з жанрів ігор, в якому широко використовуються NPC це симулятори. Першою грою-симулятором традиційно вважається The Sumerian Game, яка вийшла в 1964 році. Ця гра була дуже раннім прикладом управлінської симуляції, яка ставила перед гравцями завдання побудувати месопотамське місто та керувати ним.

Сьогодні симулятори та імітаційні ігри є більш популярними та реалістичними, ніж будь-коли. Їх продовжують розробляти для різних платформ, включаючи комп'ютери, відеоігрові консолі та мобільні пристрої.

Хоча симулятори та імітаційні ігри дозволяють відволіктися від повсякденної

рутини, вони все частіше використовуються як інструменти для студентів та професіоналів, щоб відточувати свої навички. Біржові симулятори або симулятори менеджменту часто зустрічаються в бізнес-школах, симулятори водіння використовуються гонщиками, а симулятори польотів - для навчання пілотів.

В іграх-симуляторах тварин гравці беруть на себе роль якоїсь дикої істоти. Деякі з них мають захопливі та розважальні сюжети, цікавих персонажів і складну ігрову механіку, що імітує виживання тварин у лісі чи іншому природному середовищі.

1.2 Огляд відеоігор-аналогів

Для складання специфікації вимог розроблюваного застосунку треба розглянути та проаналізувати існуючі альтернативи та виділити їх переваги, недоліки. З усього різноманіття можливих варіантів було обрано Wolf Quest, The Sims та Microsoft Flight Simulator.

WolfQuest – це освітній симулятор, який дозволяє гравцям втілитися у вовка в Єллоустонському національному парку. Гра акцентує увагу на екології та поведінці вовків, пропонуючи реалістичне моделювання життя в дикій природі. [1]

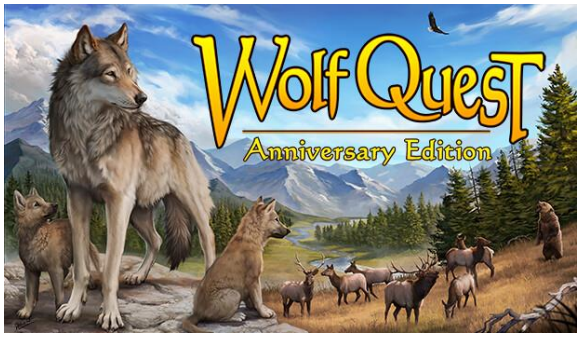

Рисунок 1.3 – Обкладинка гри Wolf Quest у Steam

У WolfQuest, особливо в версії Anniversary Edition, NPC вовки демонструють складну поведінку, що включає полювання, спарювання, захист території та взаємодію з іншими тваринами. Хоча конкретні технічні деталі реалізації ШІ не були широко оприлюднені, відомо, що гра використовує поведінкові дерева

(behavior trees) для моделювання дій NPC (рис. 1.4).

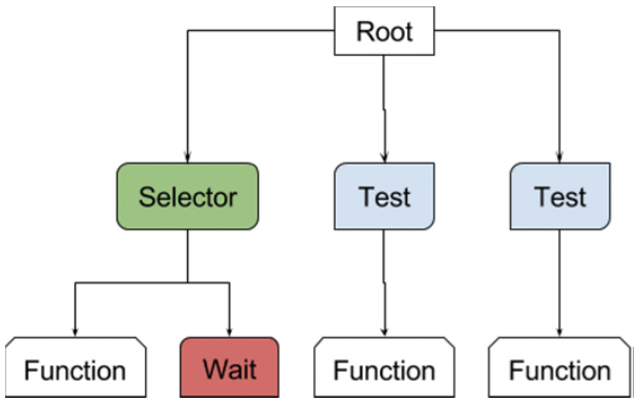

Рисунок 1.4 – Схема роботи алгоритму поведінкового дерева

Таблиця 1.1 – Характеристика Wolf Quest.

Розробник	EduWeb
Рік випуску	2007; 2019 – WolfQuest Anniversary Edition
Жанр	3D симулятор дикої природи
Платформи	Windows, macOS, Android, IOS
Технології	Unity Engine
Переваги	1) гра має значну освітню складову, яка допомагає гравцям дізнаватись нову інфомацію про життя вовків у дикій природі; 2) гравці можуть досліджувати різні ландшафти Єллоустонського національного парку і взаємодіяти з іншими тваринами в реальному часі; 3) у грі наявний кооперативний режим, що додає грі соціального аспекту; 4) гра пропонує детальну модель полювання, розмноження та інших аспектів життя вовків.
Недоліки	1) обмеженні графічні можливості; 2) ігровий процес обмежений кількома основними місіями, що може швидко набриднути при тривалому проходженні.

The Sims – це серія симуляторів життя, де гравці створюють та керують віртуальними персонажами, званими "Сімами", у їхньому повсякденному житті. Гра дозволяє будувати будинки, розвивати кар'єру, заводити стосунки та багато іншого.

Рисунок 1.5 – Обкладинка гри The Sims 4 у Steam

У серії The Sims, зокрема в The Sims 3, NPC керуються системою корисності (utility system), яка дозволяє їм приймати рішення на основі поточних потреб та бажань (рис. 1.6). Ця система враховує різні фактори, такі як голод, втома, соціальні потреби тощо, і визначає, яку дію персонаж виконає наступною.

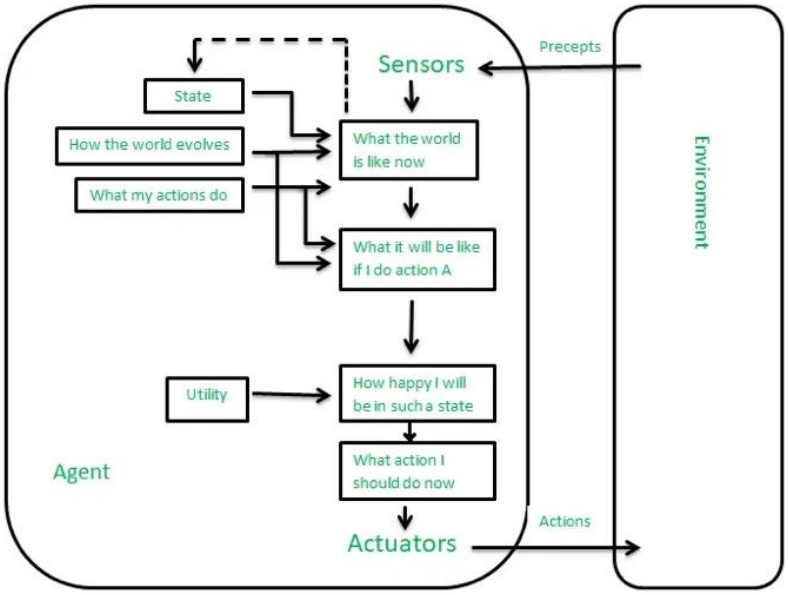

Рисунок 1.6 – Схема роботи алгоритму системи корисності

Крім того, в The Sims 3 було впроваджено модифіковану версію розподілу Больцмана, який дозволяє моделювати поведінку зі стохастичним (ймовірнісним) вибором дій, а не лише найоптимальнішої (1.1). Це додає реалістичності в поведінку – персонажі не завжди поводяться раціонально.

$$P(a_i) = \frac{e^{\frac{U(\alpha_i)}{T}}}{\sum_j e^{\frac{U(\alpha_j)}{T}}}, \quad (1.1)$$

де $P(a_i)$ – ймовірність вибору дії a_i;

$U(\alpha_i)$ – корисність (utility) або оцінка цієї дії;

T – температура, що контролює «випадковість» вибору.

Таблиця 1.2 – Характеристика гри The Sims.

Розробник	Maxis
Рік випуску	2000
Жанр	Симулятор життя
Платформи	Microsoft, macOS, PlayStation, Xbox, Nintendo
Технології	Власні рушії Maxis
Переваги	1) the Sims дозволяє гравцям створювати власні історії, будувати будинки та взаємодіяти з іншими персонажами, що дає величезну свободу в ігровому процесі; 2) гра має простий і інтуїтивно зрозумілий інтерфейс, що робить її доступною для широкої аудиторії; 3) реалістична симуляція емоцій та взаємодій між персонажами створює значну глибину для гри, що дозволяє зануритися в процес.
Недоліки	1) багато важливих функцій і контенту доступні тільки через додаткові пакети розширення, що може бути незадовільним для деяких гравців. 2) відсутність серйозних завдань.

Microsoft Flight Simulator – це високореалістичний авіаційний симулятор, який використовує дані з Bing Maps та хмарні обчислення Microsoft Azure для

створення детального зображення Землі. Гра пропонує реалістичну фізику польоту та погодні умови в реальному часі.

Рисунок 1.7 – Обкладинка гри у Steam

У Microsoft Flight Simulator використовується комбінація хмарних обчислень, машинного навчання та штучного інтелекту для створення реалістичного світу. Компанія Blackshark.ai розробила систему, яка аналізує супутникові знімки та фотограмметричні дані, щоб автоматично генерувати 3D-моделі будівель, дерев та інших об'єктів.

Для NPC, таких як диспетчери повітряного руху та пасажири, використовується технологія Azure Cognitive Services для створення природного синтезу мови та поведінки (рис. 1.8).

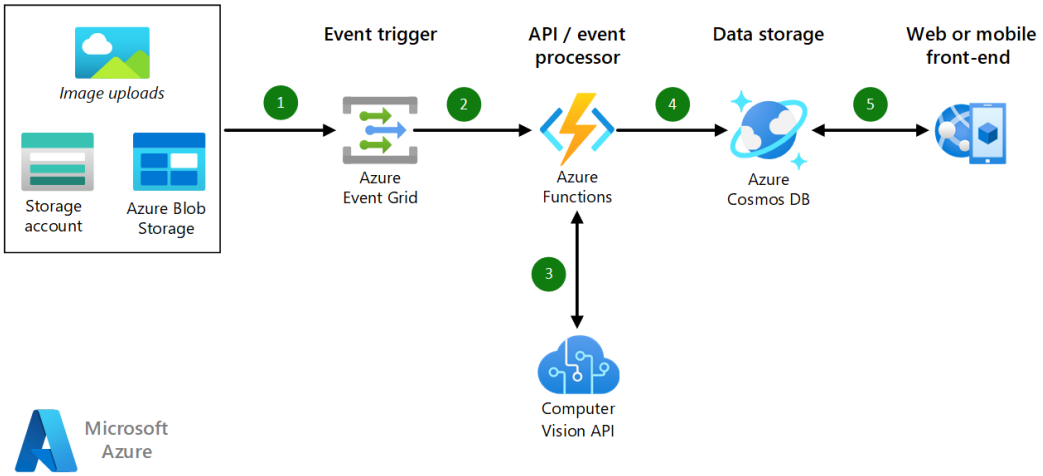

Рисунок 1.8 – Схема роботи Azure Cognitive Services

Таблиця 1.3 – Характеристика гри Microsoft Flight Simulator.

Розробник	Asobo Studio
Рік випуску	2020
Жанр	Авіаційний симулятор
Платформи	Windows, Xbox Series X/S
Технології	Власний рушій Asobo, Microsoft Azure AI, Bing Maps
Переваги	1) один з найбільш реалістичних авіаційних симуляторів на ринку з відмінною графікою та точною фізикою польоту; 2) використання реальних даних; 3) метеорологічні умови в реальному часі; 4) гравці можуть летіти разом з іншими у багатокористувацькому режимі
Недоліки	1) високі системні вимоги; 2) висока вартість; 3) відсутність сюжету.

1.3 Постановка задачі

Запропонована відеогра є 3D-симулятором дикої природи, у якому гравець має змогу спостерігати та взаємодіяти з віртуальним середовищем, населеним різними видами диких тварин. На відміну від класичних симуляторів, де поведінка тварин зазвичай задається через наперед прописані сценарії, у цій грі використано підхід із застосуванням машинного навчання — зокрема, бібліотеки Unity ML-Agents — для створення автономних агентів із можливістю адаптації до середовища. Кожен NPC персонаж є автономним агентом, який ухвалює рішення на основі власного досвіду, внутрішніх потреб та поточних умов середовища.

Актуальність цієї роботи полягає у потребі створення інтерактивних ігор нового покоління, в яких віртуальні істоти (тварини) поводяться не як

запрограмовані об’єкти, а як адаптивні агенти з елементами самонавчання. Такий підхід дозволяє формувати більш природну симуляцію дикої природи та підвищує якість геймерського досвіду.

Об’єктом кваліфікаційної роботи є процес моделювання поведінки NPC у симуляторі дикої природи.

Предметом кваліфікаційної роботи є нейронні мережі для створення реалістичної поведінки ігрових агентів.

Метою роботи є покращення інтерактивних процесів в іграх за рахунок розробки симулятора дикої природи з NPC, поведінка яких моделюється за допомогою нейронних мереж, що дозволяє створити інтерактивний та динамічний ігровий світ.

Для досягнення поставленої мети визначено такі завдання:

– дослідити існуючі симулятори дикої природи та проаналізувати підходи до побудови поведінки NPC;

– ознайомитися з можливостями Unity ML-Agents Toolkit;

– реалізувати навчання NPC;

– провести тестування, зібрати дані про ефективність навчання ігрових агентів.

Висновки до розділу 1

У першому розділі було здійснено аналіз предметної області, а саме було проведено роботу з пошуку та ознайомлення з галуззю розробки відеоігор, сучасний стан індустрії, а також зв’язок з використанням штучного інтелекту для покращення користувацького досвіду у відеоіграх через навчання неігрових персонажів.

Було здійснено огляд, а також порівняльний аналіз та дослідження технологій нейронних мереж для аналогів відеоігор у жанрі симулятор, таких як Wolf Quest, The Sims та Microsoft Flight Simulator. Встановлено, що на даний момент незважаючи на те що ринок відеоігор активно розвивається, ігри жанру

симулятор не є жанром з активним розвитком і популярністю, через нестачу різноманітності та великого вибору саме для симуляторів дикої природи. Були визначені мета, актуальність та завдання роботи.

2 ВИБІР ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ

Правильний вибір технологічного стеку є ключовим етапом у процесі створення сучасної 3D-відеогри, оскільки він безпосередньо впливає на якість візуалізації, продуктивність, можливості реалізації штучного інтелекту та масштабованість проєкту. Під час цього етапу було проведено детальний аналіз наявних технологій з урахуванням особливостей розробки симулятора дикої природи, специфіки взаємодії з NPC-персонажами та потреб у використанні нейромережевих моделей для їхнього навчання.

У цьому розділі представлено основні технологічні компоненти, обрані для реалізації гри: рушій для розробки 3D-середовища, мову програмування, бібліотеки машинного навчання, систему для тренування ML-агентів, а також інструменти для створення користувацького інтерфейсу та налагодження поведінки штучного інтелекту.

2.1 Вибір рушія для розробки 3D гри симулятора

Вибір рушія для створення будь-якої відеогри є надзвичайно важливим етапом. Він є одним із ключових етапів, що безпосередньо впливає на всі аспекти проєкту — від технічної реалізації до візуальної якості, продуктивності та можливостей масштабування. Правильно підібраний рушій забезпечує розробників необхідними інструментами для ефективної побудови ігрових механік, управління графікою, створення штучного інтелекту та реалізації інтерактивних сценаріїв. Саме тому аналіз особливостей сучасних рушіїв є критично важливим для обґрунтованого вибору платформи, яка найкраще відповідатиме цілям і завданням проєкту.

В сфері розробки відеоігор двома найвідомішими і найпопулярнішими рушіями для розробки є Unity та Unreal Engine.

Платформа **Unity** використовується для створення двовимірних, тривимірних відеоігор, відеоігор віртуальної та доповненої реальності та інших

симуляцій. Рушій спочатку був запущений у 2005 році для створення відеоігор, а згодом був проданий в інші галузі, включаючи кіно та виробництво автомобілів. У 2022 році Unity додала до набору інструментів Unity Software послуги хостингу серверів та підбору ігор, а в березні 2023 року компанія оголосила про запуск ринку для активів, створених за допомогою ШІ, включаючи персонажів, оточення, анімацію та звукові ефекти.

Рисунок 2.1 – Логотип рушія Unity

Серед найбільш успішних та популярних відеоігор, які були створенні за допомогою рушія Unity виділяють Pokemon Go, The Forest та Cuphead. З перелічених прикладів ігор Pokemon Go є мобільною грою доповненої реальності, для якої Unity забезпечив кросплатформену підтримку (iOS і Android), швидкий рендеринг 3D-графіки та інтеграцію з камерами для AR-функціоналу (рис. 2.2).

Рисунок 2.2 – Приклад використання доповненої реальності у грі

В той час The Forest є грою на виживання від першої особи, особливістю якої є великий відкритий світ, просунута симуляція виживання та складна поведінка

неігрових персонажів (рис. 2.3). У грі було реалізовано складну систему штучного інтелекту для керування NPC, яка базується на поєднанні класичних алгоритмів поведінки з можливостями рушія Unity. Метою цієї системи є створення переконливої моделі поведінки ворожих істот (наприклад, канібалів), яка б забезпечувала динамічний, адаптивний і психологічно напружений ігровий досвід.

Рисунок 2.3 – Сцена гри

Для гри Cuphead, що має ретельно опрацьовану 2D-анімацію та потребує високої ефективності при відтворенні складних візуальних ефектів, Unity забезпечив такі суттєві переваги як здатність ефективно працювати з 2D-анімацією, можливості для інтеграції анімацій і фізики, а також забезпечення мультиплатформності, що сприяло успішному запуску гри на різних пристроях (рис. 2.4).

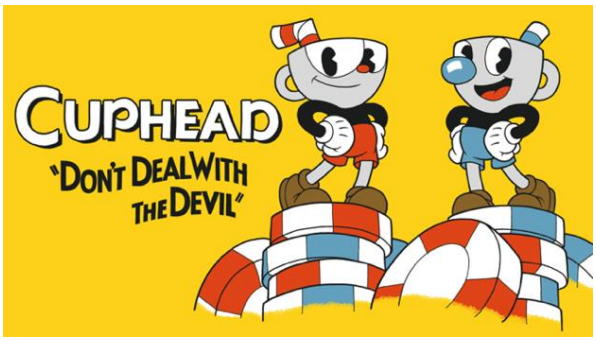

Рисунок 2.4 – Обкладинка гри у Steam

Таблиця 2.1 – Переваги та недоліки Unity.

Переваги	1) має простий та зручний інтерфейс, що підходить для новачків та розробників середнього рівня; 2) підтримує понад 25 платформ, включаючи мобільні пристрої, ПК, консолі, веб, AR/VR; 3) дозволяє створювати навчальні середовища для NPC на основі машинного навчання.
Недоліки	1) не підтримує фотореалістичну графіку на високому рівні; 2) величезні сцени та великі проєкти можуть потребувати додаткової оптимізації.

Unreal Engine підтримує високоякісну 3D-графіку завдяки використанню потужних рендеринг-програм та системи фізики. Він був розроблений компанією Epic Games і вперше представлений у 1998 році. Unreal Engine відзначається високим рівнем продуктивності, який забезпечує детальну візуалізацію та інтерфейси для створення складних ігор.

Одною з найвідоміших ігор, розроблених за допомогою Unreal Engine є багатокористувацька мультиплеєрна гра Fortnite. Fortnite побудована на технології Unreal Engine 4, яка надає високу якість графіки, підтримку для великих відкритих карт та інтерактивних об'єктів, таких як будівлі, земля та предмети (рис. 2.5).

Рисунок 2.5 – Логотип гри

Ще одною відомою грою, яка була розроблена за допомогою Unreal Engine є рольова бойова гра з відкритим світом – Hogwarts Legacy (рис. 2.6). Unreal Engine здатний підтримувати великі відкриті світи, що є важливим для таких ігор, де

гравці можуть досліджувати величезні локації. Двигун підтримує високий рівень оптимізації, що дозволяє працювати з великими обсягами даних та складними сценаріями.

Рисунок 2.6 – Обкладинка гри у Steam

Ще одна гра Unreal Tournament була розроблена на базі рушія Unreal Engine, який забезпечував високу графічну якість і дозволяв створювати великі, деталізовані арени для боїв (рис. 2.7). Гра була створена з орієнтацією на багатокористувацький режим, де гравці могли змагатися у різних типах матчів.

Рисунок 2.7 – Обкладинка гри

Таблиця 2.2 – Переваги та недоліки Unreal Engine.

Переваги	1) забезпечує неймовірно детальну і реалістичну графіку, що дозволяє розробникам створювати ігри з вражаючими візуальними ефектами; 2) має вбудовані інструменти для створення мультиплеєрних ігор з високою ефективністю синхронізації та низькою затримкою; 3) має унікальну технологію Blueprints Visual Scripting – візуальний інтерфейс для створення ігрової логіки без необхідності програмувати вручну.
Недоліки	1) для розробки великих проектів потрібні потужні комп'ютери; 2) ігри, розроблені на Unreal Engine, часто мають вищі вимоги до графічних карт і процесорів; 3) обмежена документація для певних інструментів.

Таблиця 2.3 – Характеристика рушіїв Unity та Unreal Engine.

Характеристика	Unity	Unreal Engine
Основна мова програмування	C#	C++
Платформи	Понад 25 платформ	Понад 20 платформ
Типи відеоігор	2D/3D, інді та комерційні проєкти	3D, фотореалістичні ігри, AAA проєкти
Графіка	Підтримує 2D та 3D графіку	Високоякісна 3D графіка, фотореалістичні ефекти
Інструменти AI	ML-Agents	Behavior Trees
Ресурси	Unity Asset Store	Unreal Marketplace
Масштабованість	Підходить для невеликих та середніх проєктів	Підходить для великих, складних проєктів (AAA)

Обидва інструменти мають потужний функціонал, широкий спектр можливостей і активну спільноту користувачів. Unreal Engine демонструє виняткову якість графіки, містить розвинуту систему візуального програмування

Blueprints, а також є популярним серед розробників високобюджетних ігор. Водночас Unity вирізняється більшою гнучкістю у створенні кросплатформних додатків, простішим інтерфейсом, широкими можливостями для інтеграції з системами машинного навчання (зокрема ML-Agents) та меншими вимогами до апаратного забезпечення.

З огляду на цілі проєкту було вирішено обрати Unity як основний рушій. Його підтримка ML-Agents, доступність для початківців, добра документація та активна спільнота забезпечують зручні умови для навчання NPC за допомогою машинного навчання та подальшої інтеграції в ігрове середовище.

2.2 Вибір мови програмування

Успішна реалізація відеоігрового проєкту значною мірою залежить від правильного вибору мови програмування, оскільки вона визначає структуру, продуктивність, можливість інтеграції з рушієм та зовнішніми бібліотеками. Враховуючи специфіку проєкту — було проаналізовано три мови програмування: C++, Python та C#. Нижче представлено порівняльну характеристику цих мов з погляду доцільності їх використання в даному контексті.

Таблиця 2.4 – Порівняльний аналіз мов програмування.

Характеристика	**C++**	**Python**	**C#**
Продуктивність	Висока	Низька	Висока
Простота використання	Низька, складний синтаксис	Висока, інтуїтивний синтаксис	Середня, об’єктно-орієнтований підхід
Сфера застосування	AAA-ігри, системне програмування	AI, прототипування, обробка даних	Розробка ігор у Unity, десктоп-додатки
Інтеграція з ML	Потребує ручної інтеграції	Велика кількість бібліотек	Можлива через ML-Agents у Unity

Кінець таблиці 2.4

Підтримка рушіїв	Unreal Engine	Лише через сторонні інструменти	Unity
Складність навчання NPC	Висока складність	Проста реалізація та тренування моделей	Збалансована, має зручний інтерфейс

Попри високу продуктивність C++ та його широке застосування у великих комерційних проєктах, ця мова вимагає значних ресурсів на налаштування, розгортання та відлагодження коду. Крім того, складність інтеграції із сучасними бібліотеками штучного інтелекту, що написані переважно на Python, ускладнює її використання в межах проєкту зі складною поведінкою NPC.

Python, зі свого боку, ідеально підходить для реалізації машинного навчання та прототипування алгоритмів, однак його низька продуктивність і відсутність повноцінної підтримки у рушіях для 3D-ігор робить його малопридатним як основну мову для відеогри з повноцінною графікою в реальному часі.

C# поєднує у собі переваги об'єктно-орієнтованої архітектури, високу ефективність в екосистемі Unity, простоту реалізації логіки гравця і NPC, а також можливість зручного підключення ML-модулів через Unity ML-Agents. Велика спільнота, якісна документація та нативна інтеграція із рушієм Unity дозволяють значно пришвидшити розробку, оптимізувати витрати часу та уникнути багатьох технічних труднощів.

2.3 Вибір бібліотеки машинного навчання

При розробці гри-симулятора з навчанням NPC тварин ключову роль відіграє правильний вибір бібліотеки машинного навчання. Було проведено аналіз низки сучасних ML-бібліотек. Основні критерії відбору включають підтримку навчання з підкріпленням, інтеграцію з рушієм Unity, документацію, активність спільноти та можливості візуалізації процесу навчання.

Unity ML-Agents ToolKit. ML-агенти дозволяють іграм та симуляціям

слугувати середовищем для навчання інтелектуальних агентів в Unity. Навчання можна проводити за допомогою reinforcement learning, imitation learning, neuroevolution або будь-яких інших методів. Навчені агенти можуть бути використані для багатьох випадків використання, включаючи управління поведінкою NPC (у різних налаштуваннях, таких як мультиагентні та змагальні), автоматизоване тестування збірок гри та оцінювання різних рішень ігрового дизайну перед випуском.

Unity ML-Agents Toolkit поєднує можливості двох мов програмування: Python, який використовується для навчання моделей, та C#, що забезпечує інтеграцію з середовищем Unity. Архітектура інструментарію побудована на основі Python API, що за своєю структурою нагадує OpenAI Gym, і дозволяє використовувати сучасні алгоритми навчання з підкріпленням, реалізовані на базі PyTorch або TensorFlow.

Серед ключових особливостей ML-Agents — підтримка алгоритмів PPO, SAC та імітативного навчання (Behavioral Cloning), а також вбудовані засоби для візуалізації поведінки агентів у реальному часі. Інструмент активно підтримується розробниками та постійно оновлюється разом із новими версіями Unity.

Однією з вагомих переваг є можливість експорту натренованих моделей у форматі ONNX, що дозволяє запускати inference без необхідності активного Python-середовища під час виконання гри.

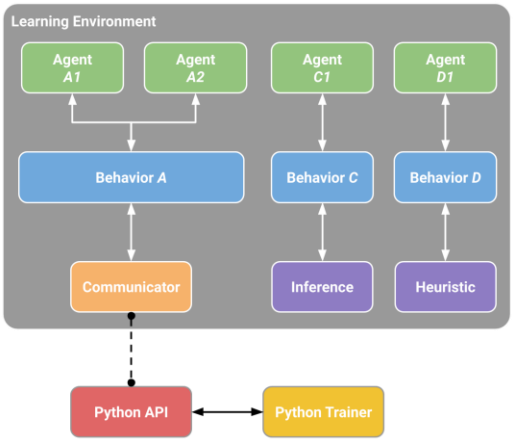

Рисунок 2.8 – Схема роботи ML-Agents

TensorFlow підтримує як низькорівневі обчислення на основі графів, так і високорівневі API (наприклад, Keras) для швидкої побудови нейронних мереж. Основною мовою використання є Python, але також існують обгортки для C++, JavaScript, Java тощо, що дозволяє застосовувати моделі в різних середовищах, включно з мобільними додатками та вебплатформами.

Він активно підтримується в ML-Agents Toolkit, що дозволяє тренувати агентів за допомогою глибокого навчання з підкріпленням. TensorFlow також підтримує експорт моделей у формат SavedModel чи ONNX, що дає змогу ефективно імплементувати навчену поведінку у самій грі.

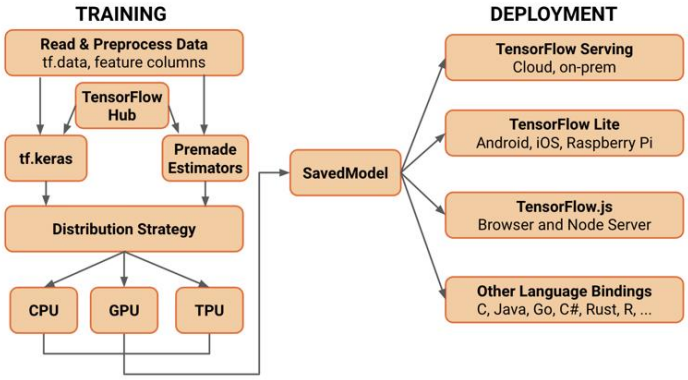

Рисунок 2.9 – Схема роботи TensorFlow

PyTorch – це повнофункціональний фреймворк для побудови моделей глибокого навчання, який є різновидом машинного навчання, що широко використовується в таких додатках, як розпізнавання зображень та обробка мови. Написаний на Python, він відносно простий у вивченні та використанні для більшості розробників машинного навчання. PyTorch вирізняється чудовою підтримкою графічних процесорів та використанням автодиференціації у зворотному режимі, що дозволяє змінювати графіки обчислень ефективніше.

Фреймворк поєднує в собі ефективні та гнучкі внутрішні бібліотеки Torch з GPU-прискоренням та інтуїтивно зрозумілий інтерфейс Python, який фокусується на швидкому створенні прототипів, читабельному коді та підтримці якомога

ширшого спектру моделей глибокого навчання. Pytorch дозволяє розробникам використовувати знайомий підхід імперативного програмування, але при цьому виводити дані у вигляді графіків.

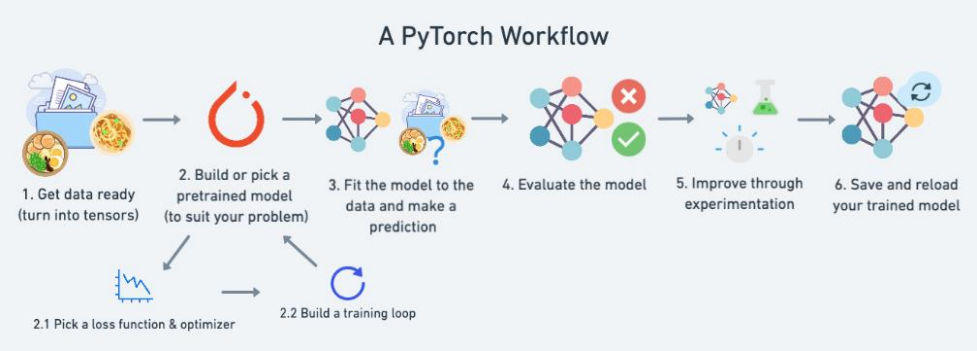

Рисунок 2.10 – Схема роботи PyTorch

Таблиця 2.5 – Порівняльний аналіз бібліотек машинного навчання.

Характеристика	**Unity ML-Agents**	**TensorFlow**	**PyTorch**
Мова	Python та C#	Python, C++	Python, C++
Основна спеціалізація	Навчання агентів в Unity	Глибоке навчання	Дослідження, прототипування
Переваги	Глибока інтеграція з Unity, підтримка RL, візуалізація	Потужність, широкі можливості, TensorBoard	Зручний синтаксис, динамічні графи
Недоліки	Обмежена гнучкість	Складність у підключенні до ігор без ML-Agents	Менш оптимізований для продакшену
Сумісність з Unity	Повна	Часткова	Часткова

Після детального дослідження та аналізу різних наявних бібліотек для машинного навчання було визначено, що оптимальним вибором є ML-Agents Toolkit. Ця бібліотека спеціально розроблена для використання з Unity та дозволяє будувати складні сценарії поведінки, використовуючи глибоке підкріплене навчання. Крім того, вона підтримує інтеграцію з такими потужними

фреймворками як PyTorch та TensorFlow, що дозволяє використовувати гнучкі моделі в межах знайомої ігрової екосистеми.

Хоча TensorFlow і PyTorch мають вищу гнучкість, їх самостійне використання для ігрових агентів вимагає складнішої інтеграції. ML-Agents надає всі необхідні інструменти для ефективного навчання NPC у форматі, зручному для розробників Unity, і при цьому зберігає можливість глибокої кастомізації моделей за потреби.

2.4 Вибір інструментів для створення користувацького інтерфейсу

Для реалізації UI у Unity використовується система Unity UI Toolkit, яка поєднує компоненти візуального редактора та декларативну розмітку на базі UXML і USS.

UI Toolkit розроблений для оптимальної продуктивності та адаптивності на різних платформах і натхненний стандартними веб-технологіями. Якщо у вас є досвід розробки веб-сторінок або додатків, основні концепції UI Toolkit можуть бути вам знайомі. За допомогою UI Toolkit ви можете розділити структуру, стиль і поведінку інтерфейсу, подібно до того, як це робиться з HTML, CSS і JavaScript (рис. 2.11).

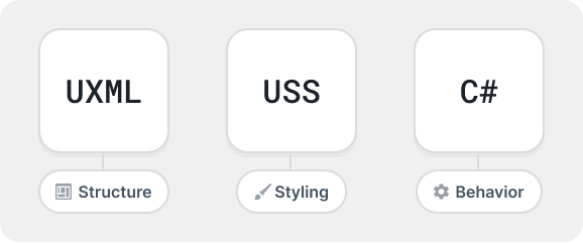

Рисунок 2.11 – Функції UI Toolkit

Файли Unity Extensible Markup Language (UXML) - це текстові файли, які визначають структуру інтерфейсу користувача. UXML разом з USS полегшує менш технічним користувачам визначення макету та стилю інтерфейсу.

Файли USS - це текстові файли, натхненні каскадними таблицями стилів

(CSS) з HTML. Синтаксис USS подібний до синтаксису CSS, але USS включає перевизначення та налаштування для кращої роботи з Unity.

Компоненти інтерфейсу визначаються та управляються як об'єкти або вузли у візуальному дереві (рис. 2.12). Візуальне дерево - це об'єктний граф, що складається з легких вузлів, який містить всі елементи у вікні або панелі. Воно визначає кожен інтерфейс, створений за допомогою UI Toolkit.

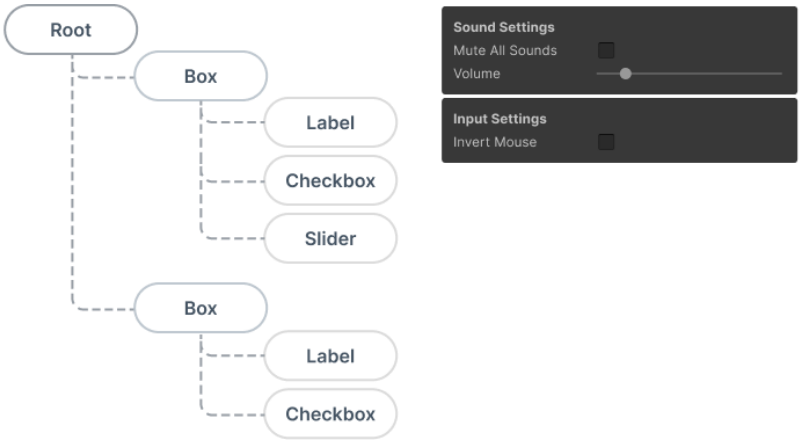

Рисунок 2.12 – Схема візуального дерева

Висновки до розділу 2

У цьому розділі було здійснено комплексний аналіз ключових технологічних компонентів, необхідних для розробки симулятора дикої природи з елементами штучного інтелекту. Зокрема, було обґрунтовано вибір рушія Unity, що забезпечує широкі можливості для створення інтерактивного 3D-середовища та має гнучку інтеграцію з ML-Agents для реалізації навчання NPC. Також визначено оптимальні мови програмування: Python — для побудови і навчання моделей ШІ, та C# — для взаємодії з рушієм Unity. Проведено порівняльний аналіз бібліотек машинного навчання (TensorFlow, PyTorch) та систем тренування агентів, де ML-Agents виявилась найдоцільнішою через глибоку інтеграцію з Unity і підтримку сучасних алгоритмів навчання.

Окрему увагу приділено вибору інструментів для реалізації користувацького

інтерфейсу та візуалізації поведінки агентів у грі. Зіставлення можливостей UI Toolkit дало змогу сформувати ефективний підхід до побудови інтерфейсу, що відповідає вимогам сучасної геймдев-індустрії. Обраний стек технологій забезпечує гнучкість, масштабованість та відповідність технічним вимогам до розробки інноваційної гри з навчанням NPC.

3 РОЗРОБКА 3D ВІДЕОГРИ СИМУЛЯТОРА ДИКОЇ ПРИРОДИ

Розробка 3D відеогри симулятора дикої природи є ключовим етапом проєкту, що визначає якість графіки, інтерактивність і реалістичність поведінки NPC. Цей процес охоплює створення та налаштування віртуального середовища, розробку механік взаємодії з ігровим світом та забезпечення стабільної роботи гри.

У цьому розділі описано основні компоненти системи: ігрове середовище, що включає тривимірні моделі та текстури ландшафту, модуль управління NPC із вбудованими алгоритмами штучного інтелекту, а також графічний інтерфейс користувача..

3.1 Пошук та вибір ігрових асетів та інших необхідних елементів гри

Для забезпечення якісної графіки та реалістичної симуляції дикої природи в розроблюваній грі було виконано пошук та відбір відповідних ігрових асетів і допоміжних елементів. Процес вибору асетів включав аналіз доступних ресурсів на платформі Unity Asset Store, де представлені моделі, текстури, анімації та звукові ефекти.

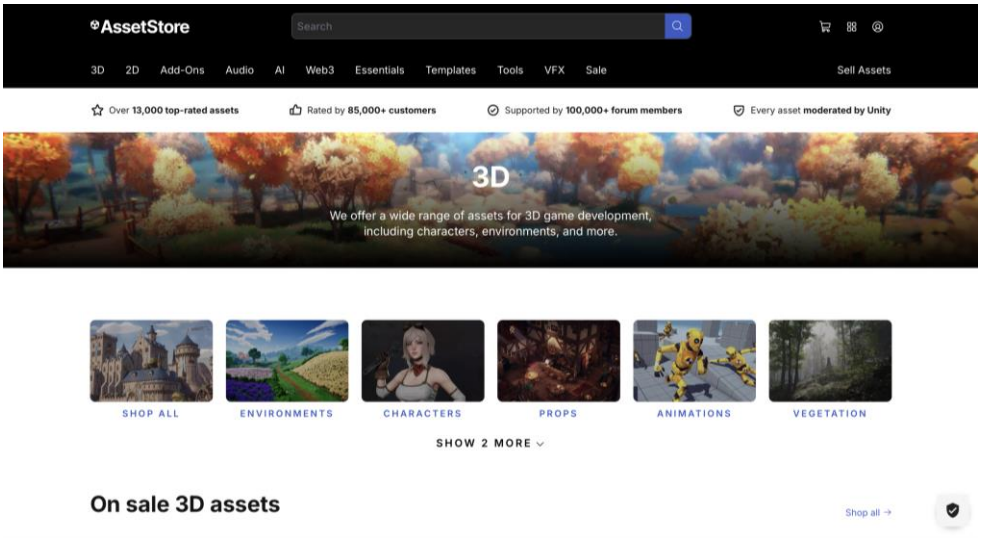

Рисунок 3.1 – Інтерфейс Unity Asset Store

Під час вибору асетів для гри враховувалися кілька ключових факторів. По-

перше, важливою була якість графіки та оптимізація: були обрані моделі з високою роздільною здатністю текстур, але водночас стежили за тим, щоб кількість полігонів залишалася оптимальною для забезпечення продуктивності. Це дозволяло зберегти якість зображення без надмірного навантаження на систему.

Також асети підбиралися з урахуванням тематики проєкту — ландшафти, рослинність і тварини відображали природне середовище симулятора дикої природи. Важливо було забезпечити їх сумісність із рушієм Unity, що дозволило легко налаштовувати матеріали, анімації та інші властивості. Окрему увагу було приділено ліцензійним умовам: використано лише ті моделі, які мали дозволи на комерційне використання, що забезпечує правову чистоту проєкту.

Найкращим вибором 3D-моделей для створення лісу став набір асетів Lowpoly Environment від Polytope Studio через свою візуальну складову та роздільну здатність.

Lowpoly моделі це це 3D-об'єкти, що мають низьку кількість полігонів (трикутників або чотирикутників), які використовуються для побудови форми об'єкта. Вони створюються з мінімальною деталізацією, що дозволяє знизити навантаження на графічний процесор та оперативну пам'ять, забезпечуючи високу продуктивність навіть на слабких пристроях.

Low Poly моделі можуть поєднуватися з сучасними технологіями освітлення та текстурування, що дозволяє отримувати візуально привабливий результат без значного навантаження на систему.

Прикладом гри, в якій використовуються саме такі 3D-об’єкти є Poly Bridge (рис. 3.2). Використання Low Poly стилю допомагає зосередитися на геймплеї без перевантаження деталями.

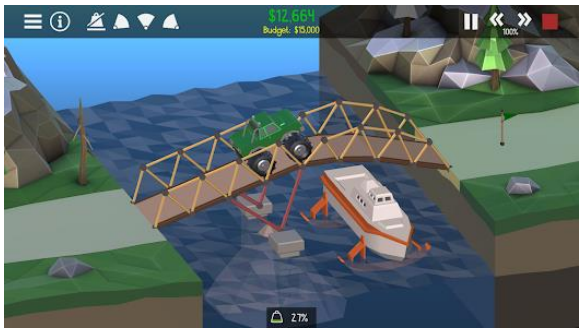

Рисунок 3.2 – Вигляд гри Poly Bridge

Low Poly моделі мають протилежність — High Poly (високополігональні) моделі, які містять тисячі або навіть мільйони полігонів. Вони використовуються для фотореалістичних текстур і складних анімацій. Наприклад такою грою є The Last Of Us 2 (рис. 3.3).

Рисунок 3.3 – Вигляд гри The Last Of Us 2

Набір асетів складається з 25 3D-об’єктів, таких як дерева, кущі, квіти та інші рослини.

Рисунок 3.4 – Приклад вигляду об’єкту яблуні

Для створення реалістичного ігрового середовища у проєкті було використано інструмент Terrain, який є стандартним засобом у Unity для проєктування та налаштування ландшафту. Цей інструмент забезпечує широкий спектр можливостей для моделювання різноманітних природних ландшафтів, від гірських масивів і пагорбів до рівнин, лісів та водойм(рис. 3.5).

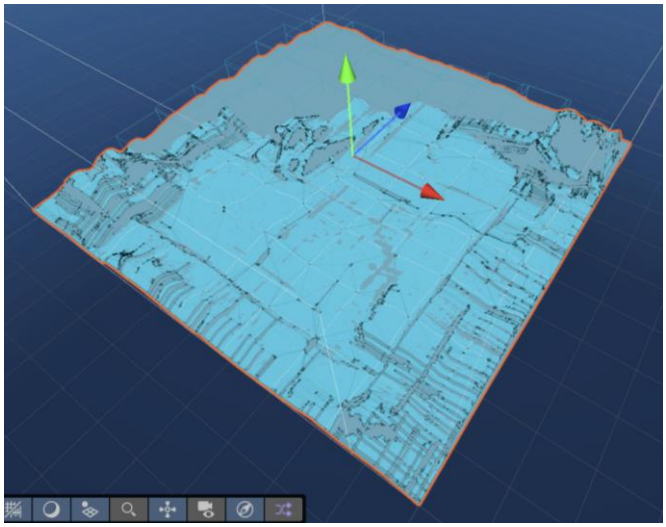

Рисунок 3.5 – Вигляд створеного ландшафту

Основний процес створення ландшафту за допомогою Terrain починається зі створення базової поверхні, яку можна деформувати за допомогою кистей різних форм і розмірів. Використовуючи ці кисті, можна легко додавати гори, долини та інші елементи рельєфу. Налаштування параметрів кисті, таких як інтенсивність і радіус, дозволяє досягти плавних переходів або створити різкі контури скель і ярів.

Для додаткової реалістичності поверхня ландшафту текстурується за допомогою спеціальних матеріалів. Terrain підтримує багатошарові текстури, що дозволяє комбінувати різні типи поверхонь, такі як трава, пісок, каміння та сніг. Це забезпечує природний вигляд середовища та допомагає підкреслити різноманітність біомів у симуляторі дикої природи.

Особливістю Terrain є можливість додавання рослинності та дерев. Завдяки вбудованому інструменту розсіювання об'єктів можна швидко створювати густі

ліси, луки та чагарники(рис. 3.6). При цьому рослинність автоматично оптимізується для підвищення продуктивності гри — дерева та трава з'являються і зникають залежно від відстані до камери.

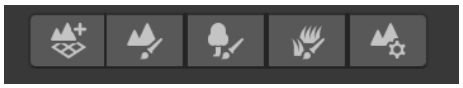

Рисунок 3.6 – Інструменти розсіювання об'єктів

Використовуючи цей інструмент було розміщено дерева та інші рослини по всьому ландшафту (рис. 3.7).

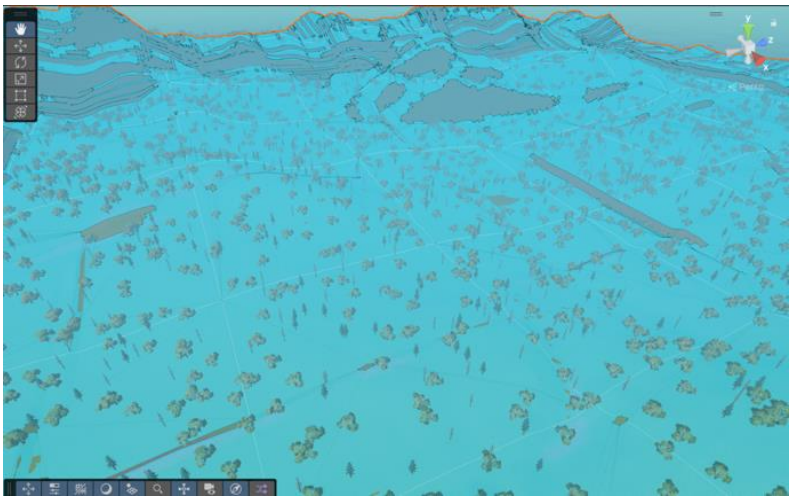

Рисунок 3.7 – Вигляд ландшафту після розміщення об’єктів

Для створення реалістичної атмосфери в симуляторі дикої природи було вирішено використовувати 3D-моделі тварин, зокрема оленів та вовків(рис. 3.8). Ці тварини є типовими представниками лісової екосистеми та мають яскраво виражену поведінку хижак-жертва. Це дозволяє продемонструвати можливості штучного інтелекту та симуляції поведінки у грі.

Рисунок 3.8 – Модель оленя

Під час вибору моделей увага приділялася кільком важливим критеріям. Моделі мали бути створені у стилі low-poly — це означає невелику кількість полігонів, що знижує навантаження на систему і дозволяє підтримувати високу продуктивність навіть за наявності багатьох NPC на екрані. Водночас такі моделі зберігають достатню деталізацію, що робить їх привабливими для гравця.

Другою важливою характеристикою була анімація. Обрані моделі оленів і вовків мали набір готових анімацій: для оленів та вовків — патрулювання території, спокійна хода та біг.

Також враховувалася сумісність із рушієм Unity. Моделі у форматах FBX та GLTF було імпортовано в проєкт, де їх легко можна було налаштувати та адаптувати. Це включало коригування матеріалів і текстур для досягнення природного вигляду шерсті та інших деталей.

Ще одним важливим аспектом були ліцензійні умови. Усі обрані моделі мають дозволи на використання в комерційних проєктах, що унеможливлює юридичні проблеми у майбутньому.

3.2 Імпорт асетів у відеогру

Процес імпорту 3D-об'єктів у ігровий застосунок на базі Unity складається з кількох основних етапів. Перед імпортом моделей необхідно переконатися, що вони відповідають вимогам проєкту. Для цього використовуються моделі у форматах FBX, GLTF або OBJ, які підтримуються Unity.

Імпорт асетів у Unity здійснюється через вкладку "Assets" або простим перетягуванням файлів у проєкт (рис. 3.9). Після цього Unity автоматично створює попередній перегляд моделей та текстур. Якщо асети містять анімації, Unity автоматично розпізнає їх та додає до вкладки "Animation".

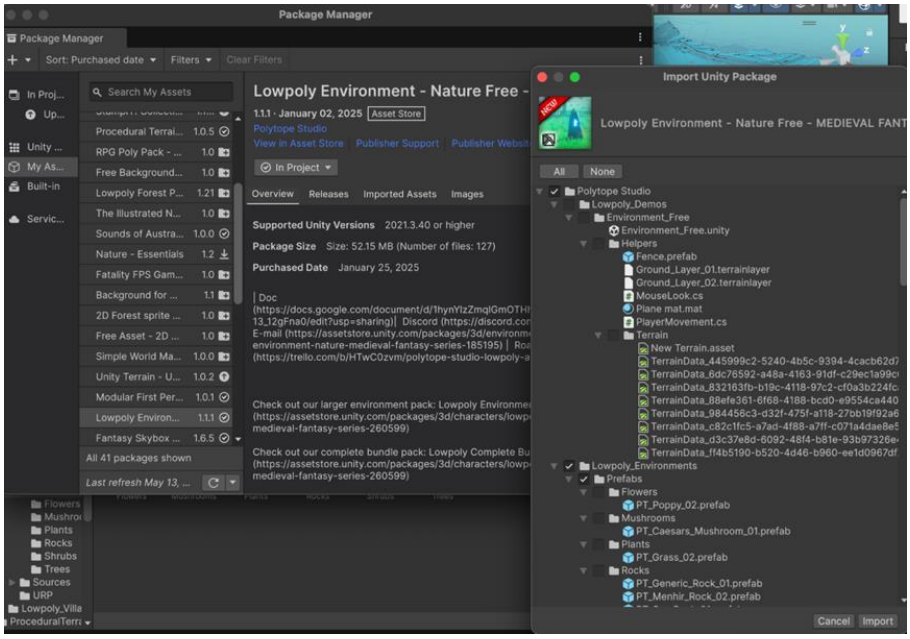

Рисунок 3.9 – Процес імпорту набору асетів у проєкт

Для зручного використання імпортованих моделей у сценах Unity створюються префаби (рис. 3.10). Префаб — це шаблон об'єкта, який можна багаторазово використовувати. Це означає, що зміни у префабі автоматично відображатимуться на всіх його копіях у проєкті.

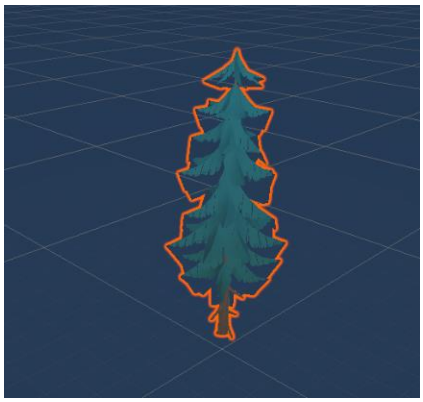

Рисунок 3.10 – Префаб дерева

3.3 Створення основних сцен гри

Основні сцени є фундаментальними елементами будь-якої гри, оскільки саме вони формують її загальний ігровий досвід, визначають навігацію між рівнями, систему збереження прогресу, а також основний геймплей.

Було створено 3 основні сцени, а саме сцену головного меню, сцену налаштувань та сцену самої гри(рис. 3.11). Кожна з цих сцен має свої унікальні елементи та функціонал, що забезпечують зручність та привабливість гри для користувачів.

Рисунок 3.11 – Вигляд сцени головного меню

3.4 Створення скриптів

Створення скриптів у Unity є ключовим етапом розробки симулятора дикої природи, адже саме скрипти визначають поведінку об'єктів, взаємодію між ними та загальну функціональність гри.

Меню головного екрану. Для коректної роботи меню головного екрану було створено скрипти для обробки основних подій таких як старт гри, завантаження гравця зі збережених координат попередньої гри, налаштувань та виходу з гри.

MainMenu.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;
```

```
public class MainMenu : MonoBehaviour
{
    public GameObject player;
    public void PlayGame()
    {
        SceneManager.LoadScene("MainScene");
    }
public void OpenSettings()
{
    SceneManager.LoadScene("settings");
}
   public void LoadGame()
{
    if (PlayerPrefs.HasKey("SavedGame"))
    {
        Debug.Log("Game Loaded");
        PlayerPrefs.SetInt("LoadPosition", 1);
        SceneManager.LoadScene("MainScene");
    }
    else
    {
        Debug.Log("No saved game found");
    }
}
private IEnumerator FindPlayerAndLoadPosition()
{
    GameObject player = null;
    while (player == null)
    {
        player = GameObject.FindWithTag("Player");
        yield return null;
    }

    if (PlayerPrefs.HasKey("PlayerX"))
    {
        float x = PlayerPrefs.GetFloat("PlayerX");
        float y = PlayerPrefs.GetFloat("PlayerY");
        float z = PlayerPrefs.GetFloat("PlayerZ");

        player.transform.position = new Vector3(x, y, z);
        Debug.Log("Player Position Loaded: " + player.transform.position);
    }
}
    public void ExitGame()
    {
        Debug.Log("Game Closed");
        Application.Quit();
    }
}
```

Наступною важливою сценою, яка нерозривно пов’язана з сценою головного меню є сцена налаштувань гри(рис. 3.12). В ній користувач може налаштувати гучність музики та зберегти всі зміни.

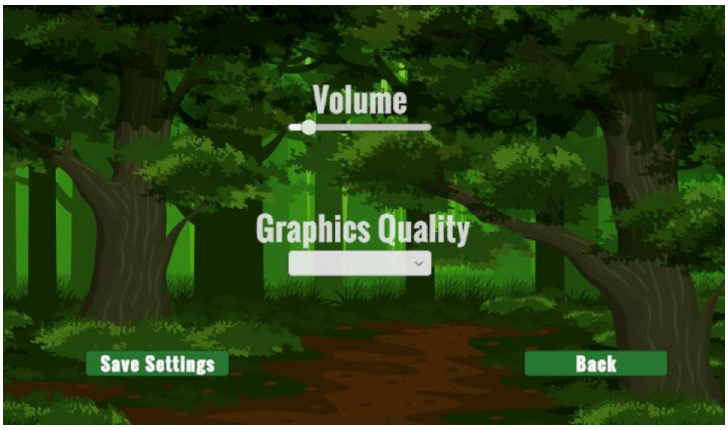

Рисунок 3.12 – Вигляд сцени налаштувань гри

Settings.cs

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

       public class SettingsScript : MonoBehaviour
       {
        public Dropdown qualityDropdown;
           public Slider volumeSlider;
           public Button saveButton;
           public Button backButton;

           void Start()
           {
               if (qualityDropdown == null) qualityDropdown =
GameObject.Find("Dropdown_Quality")?.GetComponent<Dropdown>();
               if (volumeSlider == null) volumeSlider =
GameObject.Find("Slider_Volume")?.GetComponent<Slider>();
               if (saveButton == null) saveButton =
GameObject.Find("Button_SaveSettings")?.GetComponent<Button>();
               if (backButton == null) backButton =
GameObject.Find("Button_Back")?.GetComponent<Button>();

               if (qualityDropdown == null || volumeSlider == null || saveButton == null
|| backButton == null)
               {
                   Debug.LogError("Не всі елементи UI знайдені! Переконайся, що вони є в
сцені.");
                   return;
               }

               qualityDropdown.ClearOptions();
               List<string> options = new List<string> { "Low", "Medium", "High", "Ultra"
};
               qualityDropdown.AddOptions(options);
         int savedQuality = PlayerPrefs.GetInt("GraphicsQuality", 1);
               qualityDropdown.value = savedQuality;
               qualityDropdown.RefreshShownValue();
               qualityDropdown.onValueChanged.AddListener(delegate {
ChangeQuality(qualityDropdown.value); });
```

```
            volumeSlider.value = PlayerPrefs.GetFloat("VolumeLevel", 10.0f);
            volumeSlider.onValueChanged.AddListener(ChangeVolume);

            saveButton.onClick.AddListener(SaveSettings);
            backButton.onClick.AddListener(GoBack);
        }

        void ChangeQuality(int index)
        {
            Debug.Log("Обрана якість: " + index);
            PlayerPrefs.SetInt("GraphicsQuality", index);
        }

       void ChangeVolume(float value)
    {
        Debug.Log("Гучність: " + value);
        AudioListener.volume = value;
        PlayerPrefs.SetFloat("VolumeLevel", value);
    }

        void SaveSettings()
        {
            PlayerPrefs.Save();
            Debug.Log("Налаштування збережено!");
        }

        void GoBack()
        {
            Debug.Log("Повернення в головне меню...");
            SceneManager.LoadScene("menu");
        }
    }
```

Крім налаштувань UI елементів у сцені головного меню, також необхідно додати UI елементи до сцени гри, а саме панель, яку гравець зможе викликати безпосередньо під час гри. В цій панелі необхідні такий функціонал як збереження гри на поточному прогресі гравця та вихід у головне меню (рис. 3.13).

Рисунок 3.13 – Вигляд панелі під час гри

У скрипті UIManager.cs наявні необхідні методи для зупинки гри під час перебування у панелі меню такі як `TogglePauseMenu()` за допомогою якого змінюється масштаб часу а також заблоковується курсор у грі. Після цього метод `LockCursor()` встановлює режим курсору – для сцени гри блокує, а для панелі меню відновлює для вільного переміщення.

UIManager.cs

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class UIManager : MonoBehaviour
{
    public Image heartImage;
    public GameObject heartContainer; // Контейнер для сердець
    public GameObject pauseMenu;
    public Button saveButton, exitButton;

    public GameObject inventoryContainer; // Контейнер для інвентаря
    public Image[] inventorySlots = new Image[3];  // 3 слоти для інвентаря

    private CanvasGroup pauseCanvasGroup;
    private int maxHealth = 4;
    private Image[] hearts;
    private bool isPaused = false;
    private PlayerStats playerStats;

    void Start()
    {
        playerStats = GameObject.FindWithTag("Player").GetComponent<PlayerStats>();

        hearts = new Image[maxHealth];
        AddHearts();

        pauseCanvasGroup = pauseMenu.GetComponent<CanvasGroup>();
        if (pauseCanvasGroup == null)
            pauseCanvasGroup = pauseMenu.AddComponent<CanvasGroup>();

        SetPauseMenuActive(false);
        exitButton.onClick.AddListener(ExitToMainMenu);

        LockCursor(true);
    }

    void Update()
    {
        if                (Input.GetKeyDown(KeyCode.LeftCommand)                ||
Input.GetKeyDown(KeyCode.RightCommand))
        {
            TogglePauseMenu();
        }

        if (playerStats != null)
```

```
            {
                UpdateHealth(playerStats.currentHealth);
            }
        }

        void AddHearts()
        {
            for (int i = 0; i < maxHealth; i++)
            {
                Image heart = Instantiate(heartImage, heartContainer.transform);
                heart.rectTransform.anchoredPosition = new Vector2(i * 50, 0);
                heart.rectTransform.localScale = Vector3.one * 0.6f;
                hearts[i] = heart;
            }
        }

        public void AddItemToInventory(Sprite fruitSprite)
        {
            for (int i = 0; i < inventorySlots.Length; i++)
            {
                if (inventorySlots[i].sprite == null)  // Якщо слот порожній
                {
                    inventorySlots[i].sprite = fruitSprite; // Додаємо фрукт до слота
                    inventorySlots[i].enabled = true; // Включаємо слот
                    return; // Виходимо після додавання
                }
            }
        }

        void TogglePauseMenu()
        {
            isPaused = !isPaused;
            SetPauseMenuActive(isPaused);
            Time.timeScale = isPaused ? 0 : 1;
            LockCursor(!isPaused);
        }

        void SetPauseMenuActive(bool active)
        {
            pauseCanvasGroup.alpha = active ? 1 : 0;
            pauseCanvasGroup.interactable = active;
            pauseCanvasGroup.blocksRaycasts = active;
        }

        public void ExitToMainMenu()
        {
            Time.timeScale = 1;
            LockCursor(false);
            SceneManager.LoadScene("menu");
        }

        void LockCursor(bool lockCursor)
        {
            Cursor.lockState = lockCursor ? CursorLockMode.Locked : CursorLockMode.None;
            Cursor.visible = !lockCursor;
        }

        public void UpdateHealth(int currentHealth)
```

```
        {
                for (int i = 0; i < hearts.Length; i++)
                {
                    hearts[i].gameObject.SetActive(i < currentHealth);
                }
            }
        }
```

Зважаючи на функцію збереження прогресу гравця через кнопку save Game та завантаження гри на моменті збереження через Load Game, створено скрипт PlayerPositionLoader.cs. За допомогою методу `LoadPlayerPosition()` знаходиться гравець у сцені та завантажуються координати. Корутин використовується замість звичайного методу, щоб забезпечити надійне завантаження навіть за умови, що гравець завантажується із затримкою.

PlayerPositionLoader.cs

```
        using UnityEngine;
        using System.Collections;
        public class PlayerPositionLoader : MonoBehaviour
        {
            private void Start()
            {
                if (PlayerPrefs.HasKey("LoadPosition") && PlayerPrefs.GetInt("LoadPosition")
== 1)
                {
                    StartCoroutine(LoadPlayerPosition());
                    PlayerPrefs.SetInt("LoadPosition", 0);
                }
            }

            private IEnumerator LoadPlayerPosition()

          {
                GameObject player = null;

                while (player == null)
                {
                    player = GameObject.FindWithTag("Player");
                    yield return null;
                }

                float x = PlayerPrefs.GetFloat("PlayerX");
                float y = PlayerPrefs.GetFloat("PlayerY");
                float z = PlayerPrefs.GetFloat("PlayerZ");

                player.transform.position = new Vector3(x, y, z);
```

```
        Debug.Log("Player position loaded: " + player.transform.position);
         }
    }
```

Висновки до розділу 3

У цьому розділі було детально розглянуто процес розробки 3D відеогри, що охоплював вибір та імпорт ігрових асетів, створення основних сцен та розробку скриптів для керування ігровим процесом. Було обґрунтовано вибір Low Poly моделей для оптимізації продуктивності та забезпечення візуальної привабливості гри.

Особлива увага приділялася налаштуванню ігрового середовища, яке включає тривимірні моделі рослинності та тварин, що відображають природний світ симулятора.

Розробка скриптів забезпечила функціональність гри, таку як збереження та завантаження прогресу гравця, а також налаштування параметрів через графічний інтерфейс користувача. Це дозволяє створити інтерактивний ігровий процес та забезпечити стабільну роботу гри на різних платформах.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ НЕЙРОННИХ МЕРЕЖ ТА ТЕСТУВАННЯ

4.1 Встановлення та налаштування Unity ML Toolkit

Встановлення та налаштування Unity ML-Agents Toolkit є важливим етапом для реалізації інтелектуальної поведінки неігрових персонажів у симуляторі дикої природи. Цей інструмент дозволяє поєднати можливості машинного навчання з ігровим середовищем, забезпечуючи гнучку систему навчання агентів. Для ефективної роботи необхідно встановити всі залежності, налаштувати середовище тренування та інтегрувати ML-агентів у проєкт на Unity. Це створює основу для подальшого навчання моделей та тестування адаптивної взаємодії віртуальних тварин із середовищем.

Для встановлення Unity ML-Agents Toolkit необхідно виконати кілька послідовних кроків. Насамперед потрібно мати встановлену актуальну версію Unity Hub та Unity Editor. Далі встановлюється Python за рекомендованою версію в документації (рис. 4.1).

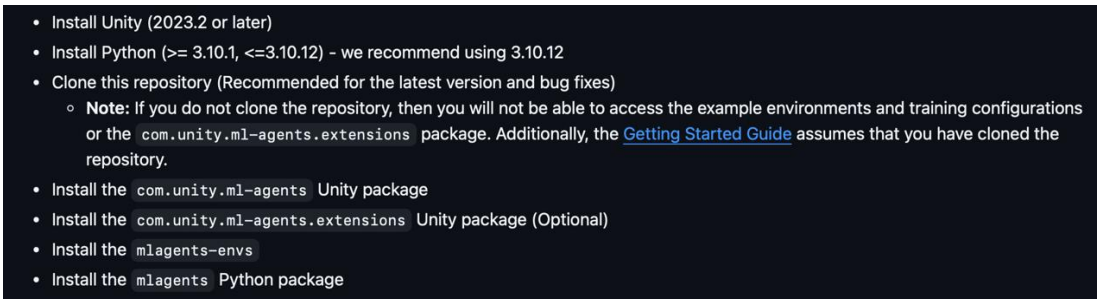

Рисунок 4.1 – Рекомендований спосіб встановлення бібліотеки з офіційної документації Unity

Наступним кроком створюється окреме віртуальне середовище, у якому за допомогою менеджера пакетів pip встановлюються всі необхідні бібліотеки, а саме pytorch (рис. 4.2).

```
(venv) mac@MacBook-Pro-mac project % pip install torch==2.7.0

Collecting torch==2.7.0
  Downloading torch-2.7.0-cp313-none-macosx_11_0_arm64.whl.metadata (29 kB)
Collecting filelock (from torch==2.7.0)
  Downloading filelock-3.18.0-py3-none-any.whl.metadata (2.9 kB)
Collecting typing-extensions>=4.10.0 (from torch==2.7.0)
  Downloading typing_extensions-4.13.2-py3-none-any.whl.metadata (3.0 kB)
Collecting setuptools (from torch==2.7.0)
  Downloading setuptools-80.8.0-py3-none-any.whl.metadata (6.6 kB)
Collecting sympy>=1.13.3 (from torch==2.7.0)
  Downloading sympy-1.14.0-py3-none-any.whl.metadata (12 kB)
Collecting networkx (from torch==2.7.0)
  Downloading networkx-3.4.2-py3-none-any.whl.metadata (6.3 kB)
Collecting jinja2 (from torch==2.7.0)
  Downloading jinja2-3.1.6-py3-none-any.whl.metadata (2.9 kB)
Collecting fsspec (from torch==2.7.0)
  Downloading fsspec-2025.5.0-py3-none-any.whl.metadata (11 kB)
Collecting mpmath<1.4,>=1.1.0 (from sympy>=1.13.3->torch==2.7.0)
  Downloading mpmath-1.3.0-py3-none-any.whl.metadata (8.6 kB)
Collecting MarkupSafe>=2.0 (from jinja2->torch==2.7.0)
  Downloading MarkupSafe-3.0.2-cp313-cp313-macosx_11_0_arm64.whl.metadata (4.0 kB)
Downloading torch-2.7.0-cp313-none-macosx_11_0_arm64.whl (68.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 68.6/68.6 MB 2.2 MB/s eta 0:00:00
Downloading sympy-1.14.0-py3-none-any.whl (6.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.3/6.3 MB 2.1 MB/s eta 0:00:00
Downloading mpmath-1.3.0-py3-none-any.whl (536 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 536.2/536.2 kB 2.9 MB/s eta 0:00:00
Downloading typing_extensions-4.13.2-py3-none-any.whl (45 kB)
Downloading filelock-3.18.0-py3-none-any.whl (16 kB)
Downloading fsspec-2025.5.0-py3-none-any.whl (196 kB)
Downloading jinja2-3.1.6-py3-none-any.whl (134 kB)
Downloading MarkupSafe-3.0.2-cp313-cp313-macosx_11_0_arm64.whl (12 kB)
Downloading networkx-3.4.2-py3-none-any.whl (1.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 1.9 MB/s eta 0:00:00
Downloading setuptools-80.8.0-py3-none-any.whl (1.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.2/1.2 MB 694.6 kB/s eta 0:00:00
Installing collected packages: mpmath, typing-extensions, sympy, setuptools, networkx, MarkupSafe, fsspec, filelock, jinja2, torch
Successfully installed MarkupSafe-3.0.2 filelock-3.18.0 fsspec-2025.5.0 jinja2-3.1.6 mpmath-1.3.0 networkx-3.4.2 setuptools-80.8.0 sympy
-1.14.0 torch-2.7.0 typing-extensions-4.13.2
```

Рисунок 4.2 – Встановлення pytorch

Після цього в Unity додається пакет ML-Agents, підключається необхідна сцена, налаштовується Academy, середовище та поведінка агентів. Завершальним етапом є перевірка з’єднання між Unity та Python середовищем, що дозволяє запускати навчання (рис. 4.3).

```
(myvenv) mac@MacBook-Pro-mac python % mlagents-learn --help
usage: mlagents-learn [-h] [--env ENV_PATH] [--resume] [--deterministic] [--force] [--run-id RUN_ID] [--initialize-from RUN_ID] [--seed SEED]
                      [--inference] [--base-port BASE_PORT] [--num-envs NUM_ENVS] [--num-areas NUM_AREAS] [--debug] [--env-args ...]
                      [--max-lifetime-restarts MAX_LIFETIME_RESTARTS] [--restarts-rate-limit-n RESTARTS_RATE_LIMIT_N]
                      [--restarts-rate-limit-period-s RESTARTS_RATE_LIMIT_PERIOD_S] [--torch] [--tensorflow] [--results-dir RESULTS_DIR]
                      [--width WIDTH] [--height HEIGHT] [--quality-level QUALITY_LEVEL] [--time-scale TIME_SCALE]
                      [--target-frame-rate TARGET_FRAME_RATE] [--capture-frame-rate CAPTURE_FRAME_RATE] [--no-graphics] [--torch-device DEVICE]
                      [trainer_config_path]

positional arguments:
  trainer_config_path

optional arguments:
  -h, --help            show this help message and exit
  --env ENV_PATH        Path to the Unity executable to train (default: None)
  --resume              Whether to resume training from a checkpoint. Specify a --run-id to use this option. If set, the training code loads an
                        already trained model to initialize the neural network before resuming training. This option is only valid when the models
                        exist, and have the same behavior names as the current agents in your scene. (default: False)
  --deterministic       Whether to select actions deterministically in policy. `dist.mean` for continuous action space, and `dist.argmax` for
                        deterministic action space (default: False)
  --force               Whether to force-overwrite this run-id's existing summary and model data. (Without this flag, attempting to train a model
                        with a run-id that has been used before will throw an error. (default: False)
  --run-id RUN_ID       The identifier for the training run. This identifier is used to name the subdirectories in which the trained model and
                        summary statistics are saved as well as the saved model itself. If you use TensorBoard to view the training statistics,
                        always set a unique run-id for each training run. (The statistics for all runs with the same id are combined as if they were
                        produced by a the same session.) (default: ppo)
  --initialize-from RUN_ID
                        Specify a previously saved run ID from which to initialize the model from. This can be used, for instance, to fine-tune an
                        existing model on a new environment. Note that the previously saved models must have the same behavior parameters as your
                        current environment. (default: None)
  --seed SEED           A number to use as a seed for the random number generator used by the training code (default: -1)
  --inference           Whether to run in Python inference mode (i.e. no training). Use with --resume to load a model trained with an existing run
                        ID. (default: False)
  --base-port BASE_PORT
                        The starting port for environment communication. Each concurrent Unity environment instance will get assigned a port
                        sequentially, starting from the base-port. Each instance will use the port (base_port + worker_id), where the worker_id is
                        sequential IDs given to each instance from 0 to (num_envs - 1). Note that when training using the Editor rather than an
                        executable, the base port will be ignored. (default: 5005)
  --num-envs NUM_ENVS   The number of concurrent Unity environment instances to collect experiences from when training (default: 1)
  --num-areas NUM_AREAS
                        The number of parallel training areas in each Unity environment instance. (default: 1)
  --debug               Whether to enable debug-level logging for some parts of the code (default: False)
  --env-args ...        Arguments passed to the Unity executable. Be aware that the standalone build will also process these as Unity Command Line
                        Arguments. You should choose different argument names if you want to create environment-specific arguments. All arguments
                        after this flag will be passed to the executable. (default: None)
  --max-lifetime-restarts MAX_LIFETIME_RESTARTS
                        The max number of times a single Unity executable can crash over its lifetime before ml-agents exits. Can be set to -1 if no
                        limit is desired. (default: 10)
  --restarts-rate-limit-n RESTARTS_RATE_LIMIT_N
                        The maximum number of times a single Unity executable can crash over a period of time (period set in restarts-rate-limit-
                        period-s). Can be set to -1 to not use rate limiting with restarts. (default: 1)
  --restarts-rate-limit-period-s RESTARTS_RATE_LIMIT_PERIOD_S
                        The period of time --restarts-rate-limit-n applies to. (default: 60)
  --torch               (Removed) Use the PyTorch framework. (default: False)
```

Рисунок 4.3 – Підключення та під’єднання Unity до бібліотеки MLAgents

Для відстеження роботи та прогресу навчання агентів, необхідне підключення Tensorboard. Результати тренування будуть відображені у браузері через localhost (рис. 4.4).

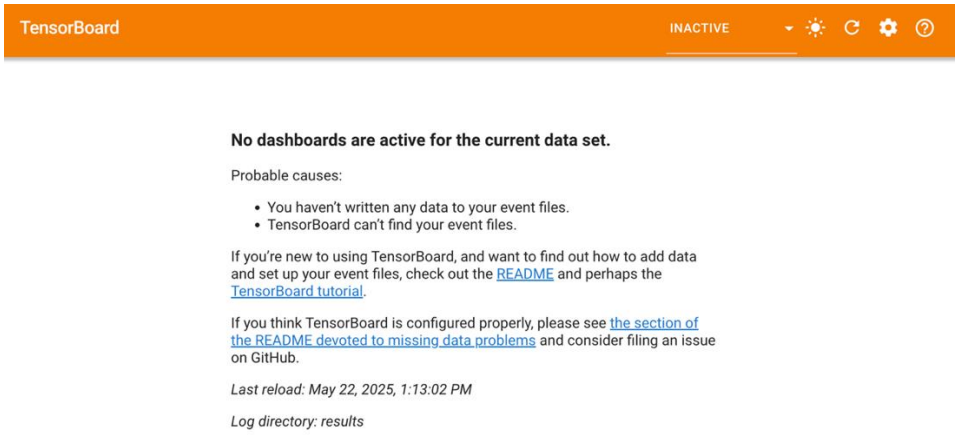

Рисунок 4.4 – Вигляд логів тренування до початку навчання

Останній етапом є встановлення MLAgents саме в Unity використовуючи Package Manager (рис.4.5).

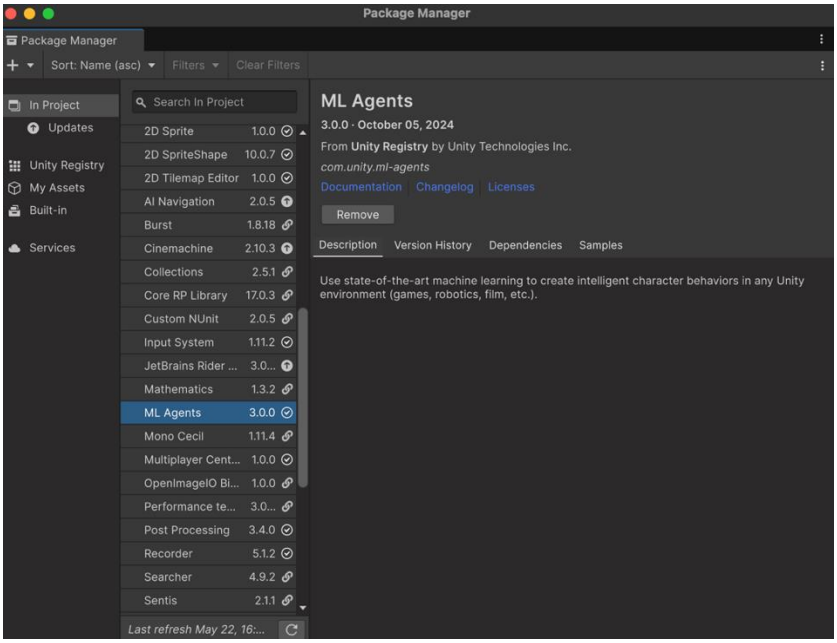

Рисунок 4.5 – Інстальований пакет ML Agents

4.2 Вибір типу тренування та об’єкту

Було обрано оленів як основний об’єкт для навчання за допомогою ML-Agents Toolkit. Такий вибір зумовлений природною поведінкою цих тварин у дикій природі, яку можна змоделювати за допомогою методів машинного навчання. Зокрема, увагу зосереджено на двох аспектах: зграйній поведінці (тобто вмінні оленів триматися разом) та уникненні хижаків (вовків), які виступають загрозою.

Під час навчання за допомогою біблотеки ML-Agents використовується Reinforcement Learning – це тип процесу машинного навчання, який фокусується на прийнятті рішень автономними агентами. Автономний агент - це будь-яка система, яка може приймати рішення і діяти у відповідь на навколишнє середовище незалежно від прямих інструкцій людини-користувача (рис 4.6).

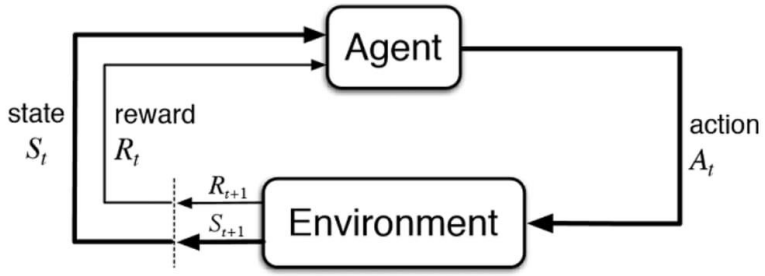

Рисунок 4.6 – Принцип роботи Reinforcement Learning

Основна задача для оленів — триматися поблизу один одного, формуючи зграйку, але при цьому не скупчуватися надто щільно. Вони повинні вчасно виявляти загрозу від вовків і тікати від них, якщо ті наближаються. Водночас, гравець у симуляторі не викликає у них жодної реакції, навіть коли знаходиться поруч.

Для ефективного навчання агентів-оленів система збирає спостереження про поточний стан навколишнього середовища. Кожен олень отримує інформацію про свою позицію, положення найближчих трьох оленів і одного-двох найближчих вовків. Крім того, олень "бачить" напрямок на гравця, але ця інформація не

використовується для реакції у винагороді. Також враховується власна швидкість руху оленя. Ці дані передаються нейронній мережі для прийняття рішень.

Для формування бажаної поведінки оленів у системі прописані певні винагороди та штрафи. Наприклад, олень отримує позитивну оцінку за перебування в зоні зграйності — приблизно на відстані 3-8 метрів від інших оленів. Навпаки, занадто близьке наближення до вовка (менше 2 метрів) карається штрафом. Олені також заохочуються рухатися в напрямку інших оленів, якщо вони занадто віддалені, і караються, якщо тривалий час залишаються наодинці. Відносно гравця — реакції немає, тому на це не накладаються ні винагороди, ні штрафи.

Таблиця 4.1 – Система штрафів та нагород для агентів.

Назва дії	**Нагорода**	**Штраф**
Перебування в зоні зграйності (3-8 м)	+ 0,5	
Близьке наближення до вовка		-1,0
Рух в напрямку інших оленів, якщо далеко	+0,2	
Перебування наодинці довше 5 секунд		-0,2

В результаті навчання олені навчаються самостійно ухвалювати оптимальні рішення: триматися ближче до зграйки, вчасно втікати від вовків, але ігнорувати гравця. Це забезпечує реалістичну адаптивну поведінку, що відповідає поставленим цілям.

Для навчання оленів використовується вектор спостережень розміром близько 30 (залежно від кількості сусідніх оленів і вовків). Дії агентів представлені як два безперервні параметри, що відповідають руху по осях X та Z. Використовується безперервний тип дій (Continuous).

4.3 Тренування агентів

Першим кроком тренування після налаштування та встановлення необхідних бібліотек є створення файлу deer_config.yaml. Це конфігураційний файл, який описує, як тренується агент DeerAgent (олень) за допомогою алгоритму підкріплення PPO (Proximal Policy Optimization) (рис. 4.7).

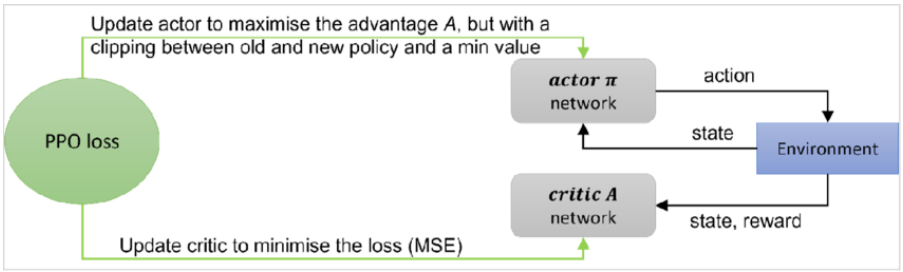

Рисунок 4.7 – Принцип роботи алгоритму підкріплення

deer_config.yaml

```
behaviors:
  DeerAgent:
    trainer_type: ppo
    max_steps: 500000
    time_horizon: 128
    summary_freq: 10000
    hyperparameters:
      batch_size: 1024
      buffer_size: 10240
      learning_rate: 2.5e-4
      beta: 1.0e-3
      epsilon: 0.1
      lambd: 0.95
      num_epoch: 3
  network_settings:
      normalize: true
     hidden_units: 256
      num_layers: 3
    reward_signals:
   extrinsic:
```

```
            gamma: 0.995
            strength: 1.0
```

У файлі задаються ключові параметри тренування агента DeerAgent з алгоритмом PPO, зокрема максимальна кількість кроків (`max_steps`), розмір батча (`batch_size`), швидкість навчання (`learning_rate`) та архітектура нейронної мережі (`hidden_units, num_layers`). Включена нормалізація вхідних даних для підвищення стабільності навчання. Налаштовано параметри для контролю оновлень моделі (`epsilon`) і дисконтування винагороди (`gamma`), що впливає на прийняття рішень агента. Також передбачене регулярне логування метрик для моніторингу прогресу за допомогою TensorBoard.

Наступним кроком є 2 важливих скрипти, а саме UniversalAnimalAI.cs і DeerAgent. **UniversalAnimalAI** – цей скрипт відповідає за базову поведінку будь-якої тварини у грі (додаток А). Він керує переміщенням тварини за допомогою NavMeshAgent, дозволяючи їй випадково блукати територією. Тварина періодично вибирає нову цільову точку в межах заданого радіусу та прямує до неї з різною швидкістю: ходьбою або бігом, залежно від випадку (із заданою ймовірністю). Анімації автоматично перемикаються відповідно до швидкості руху — наприклад, якщо швидкість висока, програється анімація бігу. Для універсальності передбачено змінні для різних типів тварин, що дозволяє використовувати один скрипт для кількох моделей. Територія, якою блукають тварини, створюється за допомогою системи NavMesh у Unity. Вона визначає, де саме можуть пересуватись об'єкти з компонентом NavMeshAgent (рис. 4.8).

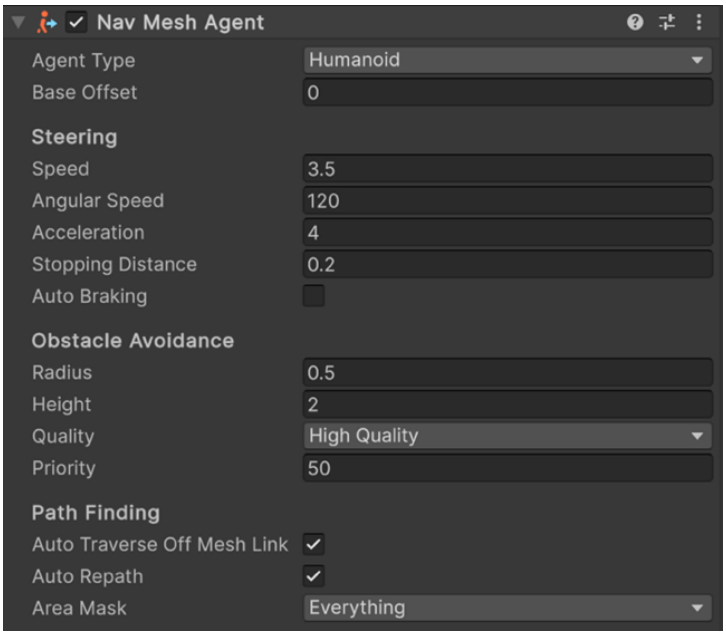

Рисунок 4.8 – Налаштування NavMeshAgent для об’єкту оленя

Таким чином, територія для руху — це поєднання створеного рельєфу, поверхонь сцени, що доступні для переміщення, та згенерованої навігаційної сітки, по якій тварини можуть безпечно пересуватись.

DeerAgent – цей скрипт реалізує агента оленя на основі ML-Agents (додаток Б). Він навчається ухвалювати рішення в динамічному середовищі, спостерігаючи за позиціями інших оленів та вовків. У методі `CollectObservations` агент збирає дані про навколишнє середовище — відстані та напрямки до інших тварин — і передає їх у нейромережу. Метод `OnActionReceived` приймає дії, згенеровані мережею, і визначає, куди агент повинен рухатись: у напрямку зграйки чи навпаки від хижака. Використовується логіка ухилення, руху до центру групи та переміщення на основі поєднання напрямків. Поведінка агента залежить від відстані до вовків — якщо хижак поруч, олень тікає. Також реалізована можливість ручного керування для тестування. Скрипт автоматично оновлює списки інших оленів та хижаків у полі зору агента (рис. 4.9).

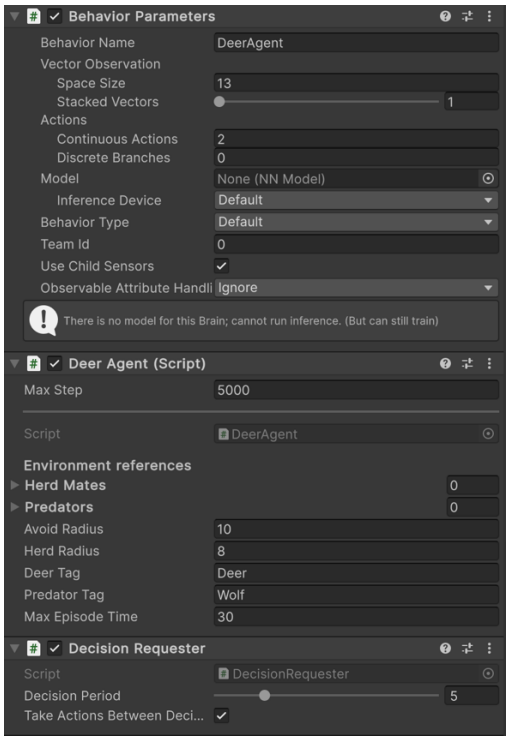

Рисунок 4.9 – Налаштування поведінки оленя

Останнім етапом після всіх налаштувань та підключення скриптів до об’єкту є тренування. Для цього використовується команда `mlagents-learn config/deer_config.yaml --run-id=deer-herd-01 –force` та запускається гра в редакторі Unity (рис. 4.10).

Рисунок 4.10 – Запуск навчання

Перше тренування зупинилось на помилці `AttributeError: 'StrictVersion'`

`object has no attribute 'version'.` Вона часто виникає при несумісності версій Python, mlagents та Unity. Після детального аналізу версій, було перевстановлено mlagents під сумісну версію з Unity.

Наступні 6 тренувань були проведені більш успішно, після кожного проведений аналіз проблеми, яка могла виникнути та виправлено. У додатку В показано результат першого тренування, яке не було успішним через відсутність правильного налаштування системи нагород та штрафів.

На основі графічних даних та системних логів, що відображають процес навчання моделі, ідентифіковано ключові аспекти її поточної ефективності та метрики для подальшого моніторингу. Графіки "Policy/Extrinsic Reward" та "Environment/Cumulative Reward" є фундаментальними для оцінки досягнення цілей, оскільки поточні спостереження демонструють стабільну нульову зовнішню винагороду (рис. 4.11 та рис. 4.12). Це підтверджується логами, які вказують на незавершеність епізодів та відсутність середньої винагороди, що свідчить про значні обмеження у взаємодії агента із середовищем або у визначенні функції винагороди.

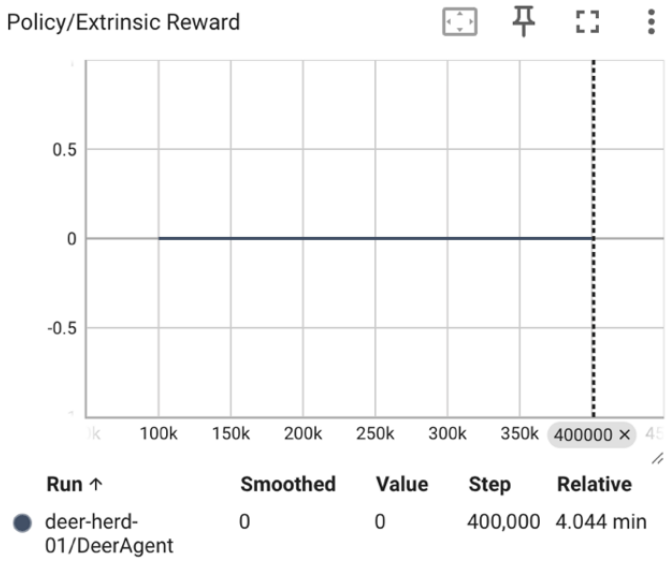

Рисунок 4.11 – Графік Policy/Extrinsic Reward

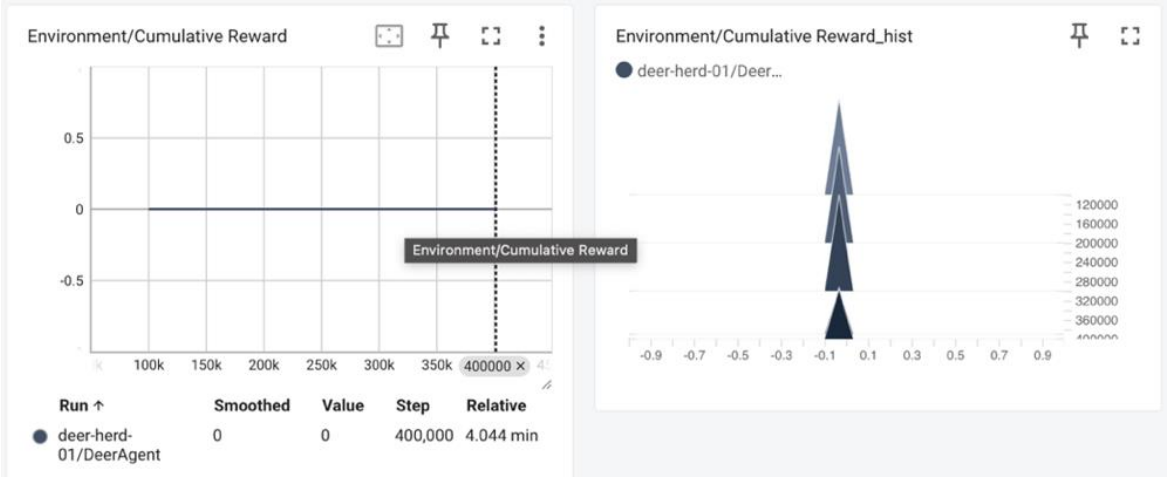

Рисунок 4.12 – Графік та гістограма Environment/Cumulative Reward

Для подальшого порівняння та оцінки оптимізації моделі, зростання значень на графіку "Policy/Extrinsic Reward" та зміщення розподілу на "Environment/Cumulative Reward" у бік позитивних накопичених винагород (із бажаною вужчою дисперсією) будуть основними індикаторами успіху. Паралельно, графік "Losses/Value Loss" є важливим для моніторингу внутрішньої збіжності моделі щодо оцінки значень, тоді як "Policy/Learning Rate", "Policy/Epsilon"та "Policy/Beta" надають інформацію про контроль над гіперпараметрами та стратегіями дослідження/експлуатації, що є невід'ємною частиною ефективного тренувального процесу.

Аналіз наступних графіків "Environment/Cumulative Reward" та "Policy/Extrinsic Reward" для п'яти тренувань демонструє значне покращення продуктивності агента (рис. 4.13).

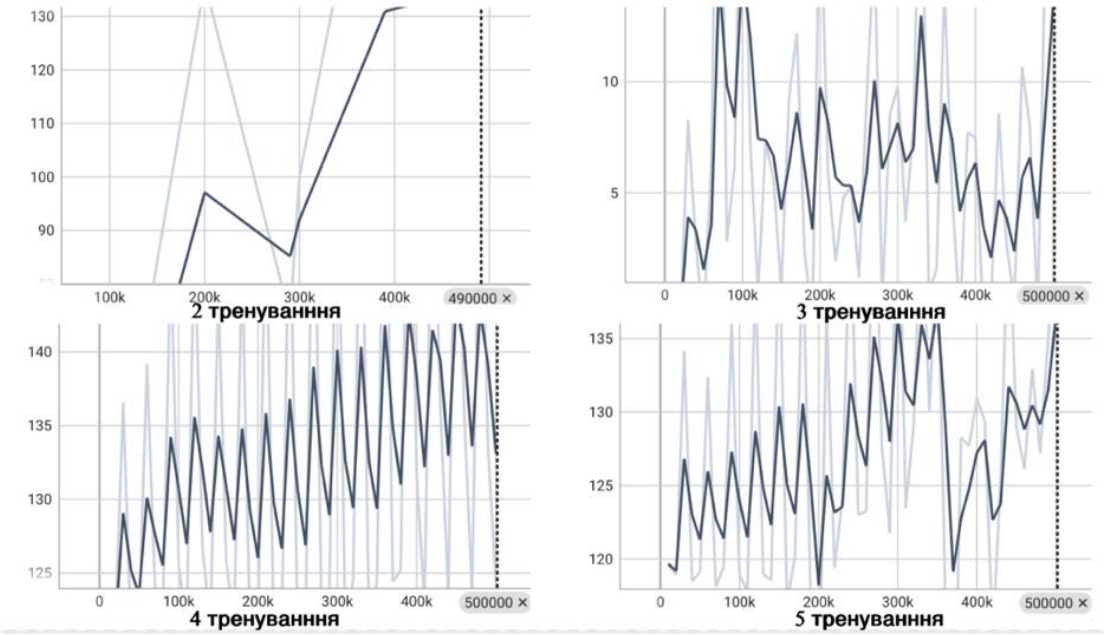

Рисунок 4.13 – Результати тренувань на графіках Environment/Cumulative Reward та Policy/Extrinsic Reward

Друге тренування показало помітне зростання накопиченої винагороди після 300 000 кроків, вказуючи на успішне навчання. У третьому тренуванні графік "Policy/Extrinsic Reward" відобразив коливання миттєвої зовнішньої винагороди в діапазоні від 0 до приблизно 13, що свідчить про періодичне отримання винагороди. Четверте та п'яте тренування на графіку "Environment/Cumulative Reward" продемонстрували стійке зростання накопиченої винагороди з вираженими циклічними патернами, що підкреслює послідовне покращення та епізодичний характер отримання винагород. Для покращення результатів, було оновлено скрипт DeerAgent.cs для стабілізації нагород та штрафів.

Аналіз лог-файлів дозволяє прогнозувати поведінку графіків "Environment/Cumulative Reward" та "Policy/Extrinsic Reward". Для другого тренування, де лог фіксує середню винагороду 0.000 та відсутність завершених епізодів, очікується, що обидва графіки залишатимуться поблизу нуля, підтверджуючи відсутність отриманих зовнішніх винагород (рис. 4.14).

Рисунок 4.14 – Процес 2-го тренування

На противагу цьому, лог шостого тренування демонструє значні позитивні значення середньої винагороди (від 119 до 186), що дозволяє прогнозувати високі, хоча й коливні, позитивні значення на графіку "Policy/Extrinsic Reward" та виражену висхідну тенденцію накопиченої винагороди на графіку "Environment/Cumulative Reward" (рис. 4.15). Ці результати шостого тренування є особливо показовими та заслуговують на детальніший розгляд.

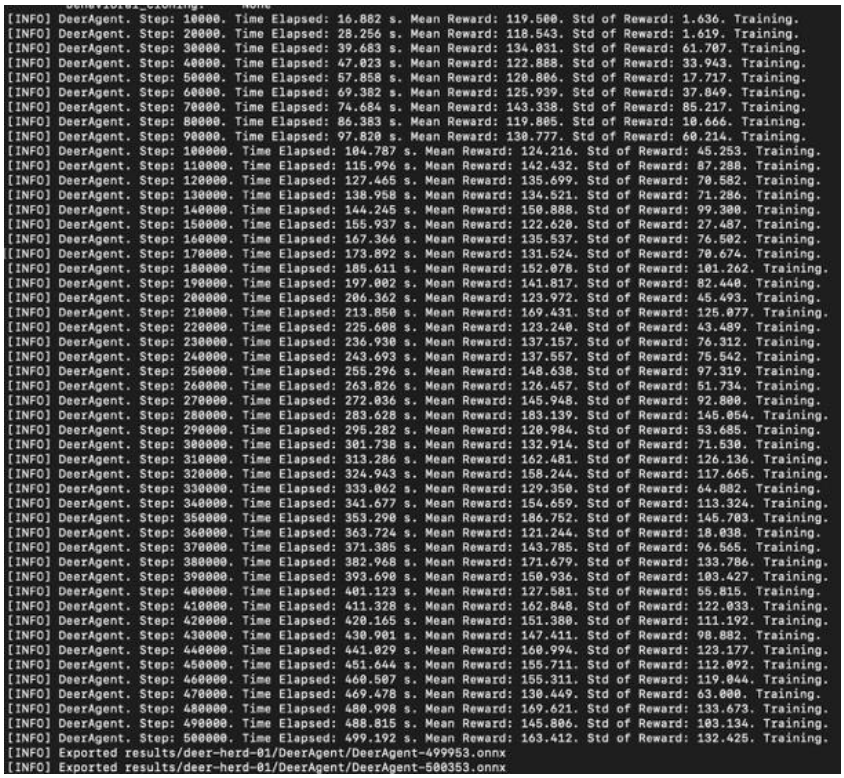

Рисунок 4.15 – Процес 6го тренування

Аналіз результатів шостого тренування демонструє значний прогрес моделі за всіма ключовими метриками. Графік "Policy/Extrinsic Value Estimate" показує стійке зростання оцінки зовнішньої цінності, досягаючи значень близько 95 до 500 000 кроків, що свідчить про ефективне навчання агента передбачати майбутні винагороди. Одночасно, "Policy/Learning Rate" демонструє плавне зменшення швидкості навчання до дуже низьких значень, забезпечуючи стабільну конвергенцію моделі (рис. 4.16).

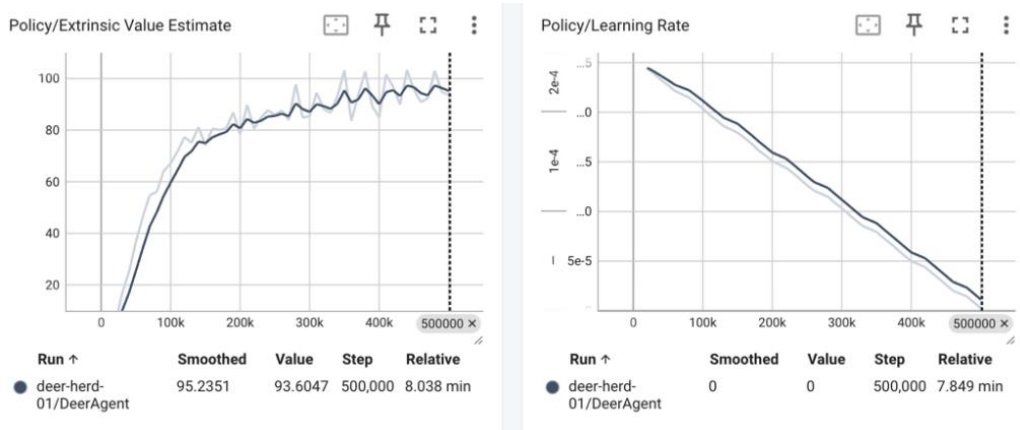

Рисунок 4.16 – Графіки Policy/Extrinsic Value Estimate та Policy/Learning Rate

Графік "Policy/Epsilon" залишається стабільним на рівні 0.1, що вказує на постійний рівень дослідження протягом усього тренування. На відміну від попередніх тренувань, "Policy/Extrinsic Reward" показує високі та коливні позитивні значення зовнішньої винагороди, досягаючи піків понад 150, що підтверджує успішне виконання агентом цільових завдань (рис. 4.17).

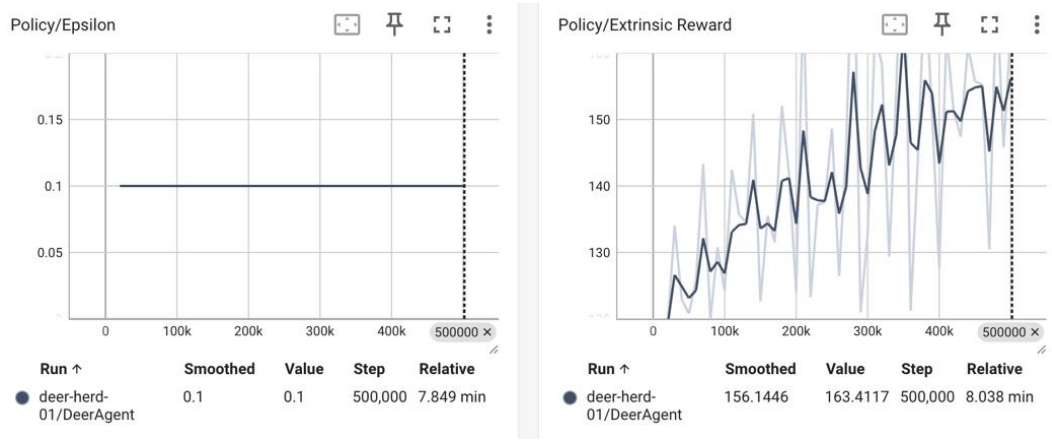

Рисунок 4.17 – Графіки Policy/Epsilon та Policy/Extrinsic Reward

"Policy/Beta" продовжує плавно зменшуватися до нуля, що є типовою стратегією для переходу від дослідження до експлуатації. Графік "Policy/Entropy" демонструє початкове зростання, а потім поступове зниження, що вказує на те, що політика спочатку збільшувала різноманітність дій, а потім ставала більш детермінованою в міру навчання (рис. 4.18).

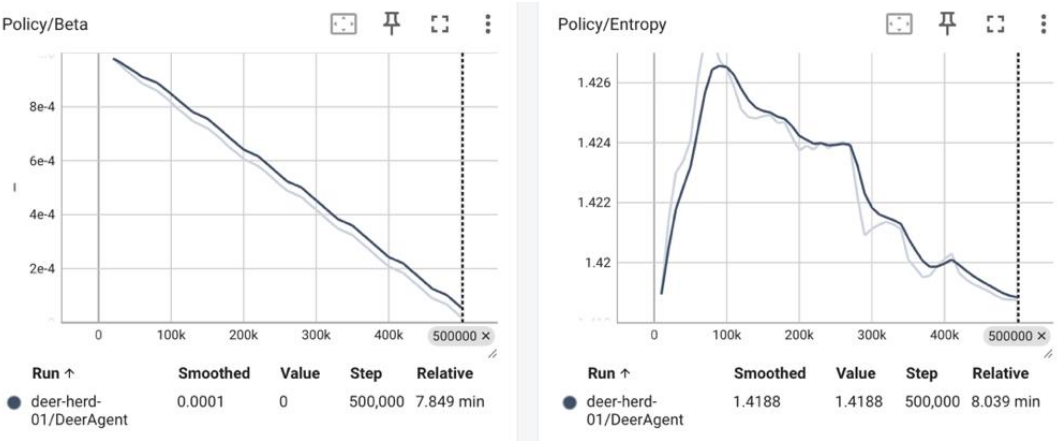

Рисунок 4.18 – Графіки Policy/Beta та Policy/Entropy

"Environment/Cumulative Reward" відображає значне та стабільне зростання накопиченої винагороди, що свідчить про загальне покращення продуктивності агента протягом тренування.

Відповідно, "Environment/Cumulative Reward_hist" показує, що розподіл накопиченої винагороди значно змістився у бік позитивних значень (близько 150), підтверджуючи послідовне отримання високих винагород (рис. 4.19).

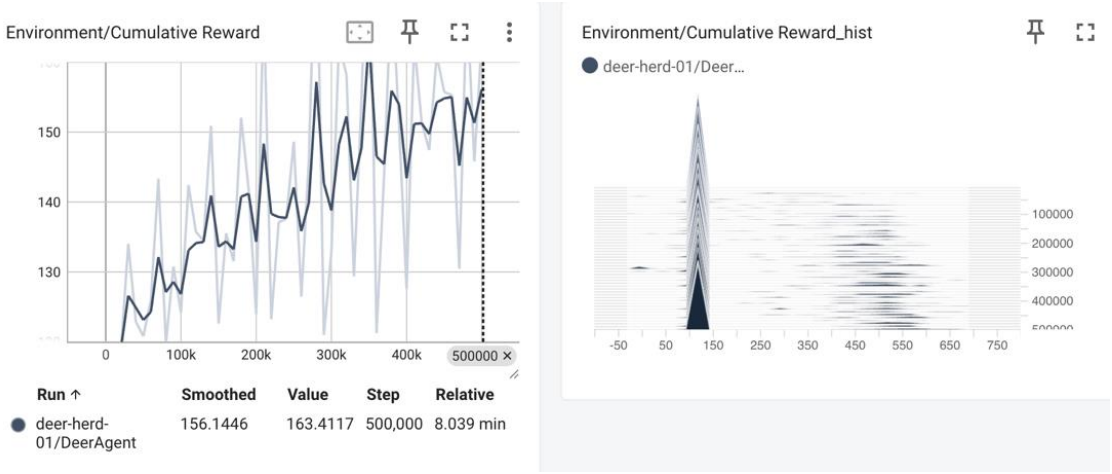

Рисунок 4.19 – Графік та гістограма Environment/Cumulative Reward

Нарешті, "Losses/Policy Loss" демонструє коливання в низькому діапазоні, що вказує на постійну оптимізацію політики. Графік "Losses/Value Loss" також показує загальну тенденцію до зростання після початкового падіння, що може бути пов'язано з більш складною оцінкою цінності в умовах отримання високих винагород.

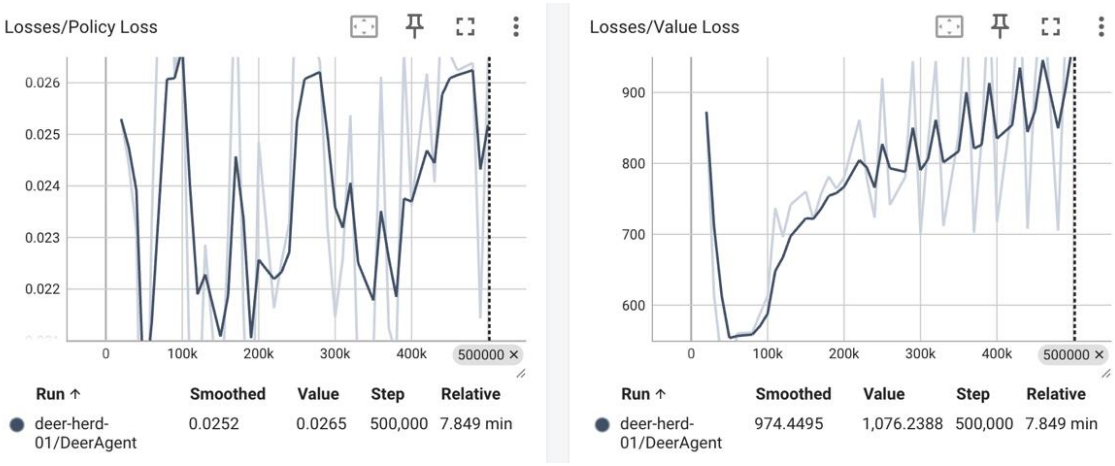

Рисунок 4.20 – Графіки Losses/Value Loss та Losses/Policy Loss

4.4 Тестування та результати навчання агентів

Після завершення процесу навчання нейронної моделі оленів було проведено тестування, щоб оцінити її ефективність у реалістичних умовах середовища. Основна мета — перевірити, наскільки добре агенти навчилися згуртовуватися в стадо та уникати зіткнень із хижаками, демонструючи поведінку, наближену до природної. Під час тестування особлива увага приділялася тому, як олені реагують на наближення вовків і чи зберігають вони оптимальну дистанцію одне від одного. Результати дозволяють зробити висновки про стабільність навченої поведінки, а також виявити можливі недоліки в моделі чи конфігурації середовища.

Поведінка – згуртування у стадо. Після навчання можна спостерігати, що олені дійсно прагнуть триматися ближче одне до одного, формуючи невеликі групки. Проте замість великого стада частіше утворюються пари або трійки, що свідчить про обмежену ефективність зграйного інстинкту в нинішній конфігурації (рис. 4.21 – 4.23).

Рисунок 4.21– Приклад поведінки згуртування

Рисунок 4.22 – Приклад поведінки згуртування

Рисунок 4.23 – Приклад поведінки згуртування

Ймовірною причиною такої поведінки є великий простір локації, що ускладнює виявлення інших особин на відстані. Через це олені не завжди мають

змогу знайти найближчих членів стада та залишаються в менших групах або парах, орієнтуючись лише на локальні взаємодії.

Поведінка – реакції на хижаків. Після навчання модель демонструє стійку реакцію на появу хижаків — олень швидко змінює напрямок руху та намагається втекти на безпечну відстань, переключаючись з ходьби на біг. У більшості випадків спостерігається адекватна реакція уникнення, особливо коли хижак знаходиться в межах заданого радіусу сприйняття, що свідчить про ефективне засвоєння агентом стратегії уникнення загрози (рис. 4.24 та 4.25).

Рисунок 4.24 – Приклад поведінки уникнення хижаків

Рисунок 4.25 – Приклад поведінки уникнення хижаків

4.5 Перспективи вдосконалення

У подальшій роботі можливе вдосконалення як самої нейронної моделі, так і середовища, в якому вона функціонує. Насамперед доцільно ускладнити поведінку

хижаків — наприклад, додати переслідування здобичі або командну взаємодію між ними. Це дозволить створити реалістичніші умови та перевірити здатність оленів до більш складної координації. Також ьожна розширити набір спостережень агента — додати врахування рельєфу, перешкод або стану втоми. Одним із перспективних напрямів є впровадження комунікації між агентами, що може сприяти більш ефективному згуртуванню стада. Ще один підхід — зменшення розмірів території або введення динамічних зон загрози, які змушуватимуть оленів активніше реагувати. Крім того, доцільно провести більше ітерацій. Усі ці зміни допоможуть зробити поведінку агентів ще природнішою та більш адаптивною.

Висновки до розділу 4

Було розглянуто повний цикл навчання агентів на основі нейронної мережі за допомогою Unity ML-Agents Toolkit. Проведено налаштування середовища та інструментів, необхідних для взаємодії Unity з інструментами машинного навчання. Далі було обрано тип тренування та розроблено поведінку агента-оленя, який мав навчитися уникати хижаків і збиратися в групи.

Проведене тренування дозволило досягти базової форми бажаної поведінки: агенти реагували на загрозу й проявляли схильність до згуртованості. Під час тестування було виявлено як позитивні результати, так і обмеження — зокрема, тенденцію утворювати не повноцінне стадо, а невеликі групи. Це частково зумовлено великою територією середовища, що ускладнює взаємодію між агентами.

Окрему увагу приділено аналізу результатів шести навчальних запусків за допомогою TensorBoard, що дозволило простежити динаміку показників винагороди, втрат та стабільності поведінки моделі протягом тренувань.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було розроблено 3D-відеогру симулятор дикої природи з використанням нейромережевих технологій для моделювання адаптивної поведінки неігрових персонажів.

Основною метою дослідження було створення реалістичного ігрового середовища, в якому NPC ухвалюють рішення на основі досвіду, внутрішніх потреб та змін у навколишньому середовищі.

Поставлені завдання виконані, а саме:

- досліджено предметну область та існуючі симулятори дикої природи, зокрема Wolf Quest, The Sims і Microsoft Flight Simulator; виконано аналіз їхніх технологічних підходів до поведінки NPC;
- проведено огляд і вибір оптимального інструментарію для реалізації проєкту;
- реалізовано навчання NPC за допомогою Unity ML-Agents Toolkit із використанням алгоритмів машинного навчання на основі підкріплення;
- створено ігрове середовище з NPC-агентами, що демонструють поведінку хижаків і жертв;
- проведено тестування симулятора та зібрано дані про ефективність навчання агентів, зокрема на основі критеріїв успішної адаптації NPC до середовища.

У ході реалізації проєкту підтверджено актуальність створення симуляторів із реалістичною, адаптивною поведінкою NPC. Завдяки вибору сучасного технологічного стеку та використанню навчання з підкріпленням досягнуто високої інтерактивності в ігровому процесі. Розроблений прототип продемонстрував можливості інтеграції нейромережевих підходів до побудови динамічної ігрової екосистеми.

Результати дослідження можуть бути використані як основа для подальшої розробки подібних ігор.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. About WolfQuest 2.7. WolfQuest Support. URL: https://support.wolfquest.org/help/about-wolfquest-2-dot-7(дата звернення: 03.05.2025).

2. Diaz C. G., Perry P. and Fiebrink R., "Interactive Machine Learning for More Expressive Game Interactions," *2019 IEEE Conference on Games (CoG)*, London, UK, 2019, pp. 1-2, doi: 10.1109/CIG.2019.8848007.

3. Filipovic A. The Role of Artificial Intelligence in Video Game Development. Central and Eastern European Online Library. URL: https://www.ceeol.com/search/viewpdf?id=1251564 (дата звернення: 03.05.2025).

4. GĂITĂNARU A. VIDEO GAMES IN THE XXIst CENTURY. Central and Eastern European Online Library. URL: https://www.ceeol.com/search/viewpdf?id=537523 (дата звернення: 03.05.2025).

5. Graham D. An Introduction to Utility Theory. In: Millington I., Funge J. (eds) Game AI Pro: Collected Wisdom of Game AI Professionals. CRC Press, 2013. URL: https://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf (дата звернення: 20.05.2025).

6. History of WolfQuest. WolfQuest. URL: https://www.wolfquest.org/about/history (дата звернення: 03.05.2025).

7. How The Sims Accidentally Invented the Cosy Game Genre. University of Portsmouth. URL: https://www.port.ac.uk/news-events-and-blogs/blogs/popular-culture/how-the-sims-accidentally-invented-the-cosy-game-genre (дата звернення: 02.05.2025).

8. Koleva M. Simulation Games and the Educational Process: Conceptual and Didactic Aspects. Central and Eastern European Online Library. URL: https://www.ceeol.com/search/viewpdf?id=537523 (дата звернення:

02.05.2025).

9. Microsoft Flight Simulator (2020). PCGamingWiki. URL: https://www.pcgamingwiki.com/wiki/Microsoft_Flight_Simulator_%282020%29 (дата звернення: 03.05.2025).

10. Microsoft Flight Simulator: The Future of Game Development. Microsoft Developer. URL: https://developer.microsoft.com/en-us/games/articles/2021/07/microsoft-flight-simulator-the-future-of-game-development/ (дата звернення: 03.05.2025).

11. Mladenović I. Simulation Games as a Method of Historical Learning: Between Immersion and Manipulation. Central and Eastern European Online Library. URL: https://www.ceeol.com/search/viewpdf?id=1251564 (дата звернення: 02.05.2025).

12. Newcastle University. Artificial Intelligence 1: Finite State Machines. Newcastle University. URL: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/previousinformation/artificialintelligence1finitestatemachines/2016%20Tutorial%208%20-%20Finite%20State%20Machines.pdf (дата звернення: 16.05.2025).

13. Schulman J., Wolski F., Dhariwal P., Radford A., Klimov O. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347. URL: https://arxiv.org/abs/1707.06347 (дата звернення: 16.05.2025).

14. Simpson C. Behavior Trees for AI: How They Work. Game Developer. URL: https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work (дата звернення: 16.05.2025).

15. TensorFlow – Official Website. Google Brain. URL: https://www.tensorflow.org (дата звернення: 04.05.2025).

16. The Sims. Encyclopædia Britannica. URL: https://www.britannica.com/topic/The-Sims (дата звернення: 02.05.2025).

17. Unity – Real-Time Development Platform. Unity Technologies. URL: https://unity.com (дата звернення: 04.05.2025).

18. Unity Technologies. Get started with UI Toolkit. Unity Documentation. URL: https://docs.unity3d.com/6000.1/Documentation/Manual/UIE-simple-ui-toolkit-workflow.html (дата звернення: 12.05.2025).

19. Unity Technologies. Introduction to UI Toolkit. Unity Documentation. URL: https://docs.unity3d.com/Manual/ui-systems/introduction-ui-toolkit.html (дата звернення: 04.05.2025).

20. Unity Technologies. ML-Agents Toolkit Documentation. Unity Technologies. URL: https://unity-technologies.github.io/ml-agents/ML-Agents-Toolkit-Documentation/ (дата звернення: 12.05.2025).

21. Unity Technologies. ML-Agents Toolkit Documentation (v3.0). Unity Documentation. URL: https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/manual/index.html (дата звернення: 04.05.2025).

22. Unity Technologies. ML-Agents Toolkit Documentation (v3.0). Unity Documentation. URL: https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/manual/index.html (дата звернення: 12.05.2025).

23. Unity Technologies. ML-Agents Toolkit Overview. Unity Technologies. URL: https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/ (дата звернення: 12.05.2025).

24. Unity Technologies. UI Toolkit - Unity Manual. Unity Documentation. URL: https://docs.unity3d.com/6000.1/Documentation/Manual/UIElements.html (дата звернення: 12.05.2025).

25. Unreal Engine – Real-Time 3D Creation Tool. Epic Games. URL: https://www.unrealengine.com/en-US (дата звернення: 04.05.2025).

26. Wang K., Li P. and Liu Y., "Research of Drilling Simulation System Based on Unity 3D," *2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA)*, Shenyang, China, 2022, pp. 288-292.

27. Zhang X. Learn AI Game Playing Algorithm Part II — Monte Carlo Tree Search. Medium. URL: https://xyzml.medium.com/learn-ai-game-playing-algorithm-part-ii-monte-carlo-tree-search-2113896d6072 (дата звернення: 03.05.2025).

ДОДАТОК А

Скрипт UniversalAnimalAI

```
using UnityEngine;
using UnityEngine.AI;

public class UniversalAnimalAI : MonoBehaviour
{
   [SerializeField]
    private NavMeshAgent agent;
    public NavMeshAgent Agent => agent;
    private Animator animator;
    private Vector3 destination;

    [Header("Animal Settings")]
    public float wanderRadius = 250f;
    public float minWalkTime = 5f; // Мінімальний час ходьби перед зупинкою
    public float maxWalkTime = 12f; // Максимальний час ходьби перед зупинкою
    public float runSpeedThreshold = 3.5f; // Мінімальна швидкість для бігу
    public float walkSpeed = 1.5f; // Швидкість ходьби
    public float runSpeed = 4f; // Швидкість бігу
    public float runChance = 0.2f; // 20% шанс бігу

    [Header("Animal Type Settings")]
    public bool isDog = false; // Якщо true, тварина є собакою
    public string walkAnimName = "Deer_001_walk"; // Назва анімації для ходьби
    public string runAnimName = "Deer_001_run"; // Назва анімації для бігу
    public string idleAnimName = "Deer_001_idle"; // Назва анімації для бездіяльності

    public string dogWalkAnimName = "Dog_001_walk"; // Назва анімації для собаки (ходьба)
    public string dogRunAnimName = "Dog_001_run"; // Назва анімації для собаки (біг)
    public string dogIdleAnimName = "Dog_001_idle"; // Назва анімації для собаки
(бездіяльність)

    void Start()
    {
        agent = GetComponent<NavMeshAgent>();
        animator = GetComponent<Animator>();

        if (animator == null)
        {
            Debug.LogError("Animator not found on " + gameObject.name);
        }

        if (agent == null)
        {
            Debug.LogError("NavMeshAgent not found on " + gameObject.name);

}
```

```
        agent.speed = walkSpeed; // Початково тварина йде

 StartCoroutine(Wander());
 }

    private System.Collections.IEnumerator Wander()
    {
        while (true)
        {
            SetRandomDestination();
            yield return new WaitForSeconds(Random.Range(minWalkTime, maxWalkTime)); //
Час ходьби перед зміною напрямку
        }
    }
    public void MoveTo(Vector3 targetPos, bool run = false)
{
    agent.SetDestination(targetPos);
    agent.speed = run ? runSpeed : walkSpeed;
}


    void Update()
    {
        if (animator == null) return; // Перевірка, щоб уникнути помилок

        float speed = agent.velocity.magnitude;
        Debug.Log("Current Speed: " + speed);

        if (speed >= runSpeedThreshold)
        {
            PlayAnimation(isDog ? dogRunAnimName : runAnimName);
        }
        else if (speed > 0.1f)
        {
            PlayAnimation(isDog ? dogWalkAnimName : walkAnimName);
        }
        else
        {
            PlayAnimation(isDog ? dogIdleAnimName : idleAnimName);
        }
    }

    void PlayAnimation(string animName)
    {
        if (animator.HasState(0, Animator.StringToHash(animName)))
        {
            animator.Play(animName);
        }

  else
        {
```

```
            Debug.LogError($"Animation '{animName}' not found in Animator on
{gameObject.name}");

}
   }

    void SetRandomDestination()
    {
        Vector3 randomDirection = Random.insideUnitSphere * wanderRadius;
        randomDirection += transform.position;
        NavMeshHit hit;

        if (NavMesh.SamplePosition(randomDirection, out hit, wanderRadius,
NavMesh.AllAreas))
        {
            destination = hit.position;
            agent.SetDestination(destination);

            // 20% шанс бігу
            if (Random.value < runChance)
            {
                agent.speed = runSpeed;
            }
            else
            {
                agent.speed = walkSpeed;
            }
        }
    }
}
```

ДОДАТОК Б

Скрипт DeerAgent

```
using UnityEngine;
using UnityEngine.AI;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;
using System.Collections.Generic;

public class DeerAgent : Agent
{
    private UniversalAnimalAI animalAI;

    [Header("Environment references")]
    public Transform[] herdMates;
    public Transform[] predators;

    public float avoidRadius = 10f;
    public float herdRadius = 8f;

    public string deerTag = "Deer";
    public string predatorTag = "Wolf";

    private float timeAlone = 0f;
    private float aloneThreshold = 5f;

    private float episodeTimer = 0f;
    public float maxEpisodeTime = 30f;

    public override void Initialize()
    {
        animalAI = GetComponent<UniversalAnimalAI>();
        UpdateHerdMates();
        UpdatePredators();
    }

    public override void OnEpisodeBegin()
    {
        Vector3 center = Vector3.zero;
        float range = 20f;

        Vector3 randomPoint = center + new Vector3(
            Random.Range(-range, range),
            0,
            Random.Range(-range, range)
        );


   if (NavMesh.SamplePosition(randomPoint, out NavMeshHit hit, 5f, NavMesh.AllAreas))
```

```
            {

transform.localPosition = hit.position;

      animalAI.Agent.Warp(hit.position);
      }
      else
      {
         Debug.LogWarning("Не вдалося знайти позицію на NavMesh в OnEpisodeBegin");
      }

      timeAlone = 0f;
      episodeTimer = 0f;

      UpdateHerdMates();
      UpdatePredators();
   }

   public override void CollectObservations(VectorSensor sensor)
   {
      sensor.AddObservation(transform.localPosition);
      sensor.AddObservation(animalAI.Agent.velocity);

      float closestPredDist = 1000f;
      Vector3 closestPredDir = Vector3.zero;

      foreach (var pred in predators)
      {
         float dist = Vector3.Distance(transform.localPosition, pred.localPosition);
         if (dist < closestPredDist)
         {
            closestPredDist = dist;
            closestPredDir = (transform.localPosition - pred.localPosition).normalized;
         }
      }

      sensor.AddObservation(closestPredDist / avoidRadius);
      sensor.AddObservation(closestPredDir);

      Vector3 herdCenter = Vector3.zero;
      int count = 0;
      foreach (var mate in herdMates)
      {
         if (mate == null || mate == transform) continue;
         float dist = Vector3.Distance(transform.localPosition, mate.localPosition);

if (dist < herdRadius)
         {
            herdCenter += mate.localPosition;

            count++;
```

```
            }

}
        herdCenter = (count > 0) ? herdCenter / count : transform.localPosition;
        Vector3 toHerdCenter = (herdCenter - transform.localPosition).normalized;
        sensor.AddObservation(toHerdCenter);
    }

    public override void OnActionReceived(ActionBuffers actions)
    {
        episodeTimer += Time.deltaTime;
        if (episodeTimer > maxEpisodeTime)
        {
            AddReward(1.0f);  // Винагорода за виживання
            EndEpisode();
            return;
        }

        float moveX = Mathf.Clamp(actions.ContinuousActions[0], -1f, 1f);
        float moveZ = Mathf.Clamp(actions.ContinuousActions[1], -1f, 1f);
        Vector3 moveDir = new Vector3(moveX, 0, moveZ);

        float closestPredDist = 1000f;
        Vector3 closestPredPos = Vector3.zero;
        foreach (var pred in predators)
        {
            float dist = Vector3.Distance(transform.localPosition, pred.localPosition);
            if (dist < closestPredDist)
            {
                closestPredDist = dist;
                closestPredPos = pred.localPosition;
            }
        }

        if (closestPredDist < 5f)
        {
            AddReward(-0.1f);
            if (closestPredDist < 1.5f)
            {
                AddReward(-1.0f);
                EndEpisode();
                return;
            }
        }

        Vector3 herdCenter = Vector3.zero;
        int count = 0;
        foreach (var mate in herdMates)

  {
            if (mate == null || mate == transform) continue;
```

```
  float dist = Vector3.Distance(transform.localPosition, mate.localPosition);
    if (dist < herdRadius)

{
                herdCenter += mate.localPosition;
                count++;
            }
        }

        herdCenter = (count > 0) ? herdCenter / count : transform.localPosition;
        float distToHerdCenter = Vector3.Distance(transform.localPosition, herdCenter);

        float distReward = Mathf.Clamp01(1 - (distToHerdCenter / herdRadius));
        AddReward(distReward * 0.5f);

        if (animalAI.Agent.velocity.magnitude < 0.1f)
        {
            AddReward(-0.01f);
        }

        if (count > 0)
        {
            Vector3 toHerd = (herdCenter - transform.localPosition).normalized;
            float dot = Vector3.Dot(toHerd, moveDir.normalized);
            AddReward(dot * 0.2f);
        }

        if (count == 0)
        {
            timeAlone += Time.deltaTime;
            if (timeAlone > aloneThreshold)
            {
                AddReward(-0.5f);
                EndEpisode();
                return;
            }
        }
        else
        {
            timeAlone = 0f;
        }
        // Нова винагорода за кількість сусідів
        float groupReward = CalculateGroupReward();

AddReward(groupReward);
        Vector3 targetPos;
 if (closestPredDist < avoidRadius)
        {

 Vector3 awayFromPred = (transform.localPosition - closestPredPos).normalized;
```

```
targetPos = transform.localPosition + awayFromPred * 5f;
        }
        else
        {
            Vector3 toHerd = (herdCenter - transform.localPosition).normalized;
            Vector3 combinedDir = (toHerd + moveDir.normalized).normalized;
            targetPos = transform.localPosition + combinedDir * 3f;
        }

        animalAI.MoveTo(targetPos, run: closestPredDist < avoidRadius);
    }

    private float CalculateGroupReward()
    {
        int neighborsCount = 0;

        foreach (var mate in herdMates)
        {
            if (mate == null || mate == transform) continue;
            float dist = Vector3.Distance(transform.localPosition, mate.localPosition);
            if (dist < herdRadius)
            {
                neighborsCount++;
            }
        }

        float reward = 0f;

        if (neighborsCount >= 4 && neighborsCount <= 6)
        {
            reward = 0.1f;  // позитивна винагорода за оптимальну групу
        }
        else if (neighborsCount > 6)
        {
            reward = -0.05f * (neighborsCount - 6);  // штраф за зайвих
        }
        else
        {
            reward = -0.01f;  // невеликий штраф за занадто мало сусідів
        }

        return reward;

}

    public void UpdateHerdMates()

 {
        List<Transform> deerList = new List<Transform>();
 GameObject[] deers = GameObject.FindGameObjectsWithTag(deerTag);
        foreach (var d in deers)
```

```
{
            if (d.transform != transform)
                deerList.Add(d.transform);
        }
        herdMates = deerList.ToArray();
    }

    public void UpdatePredators()
    {
        List<Transform> predatorList = new List<Transform>();
        GameObject[] preds = GameObject.FindGameObjectsWithTag(predatorTag);
        foreach (var p in preds)
        {
            predatorList.Add(p.transform);
        }
        predators = predatorList.ToArray();
    }
}
```

ДОДАТОК В

Графіки результатів першого тренування

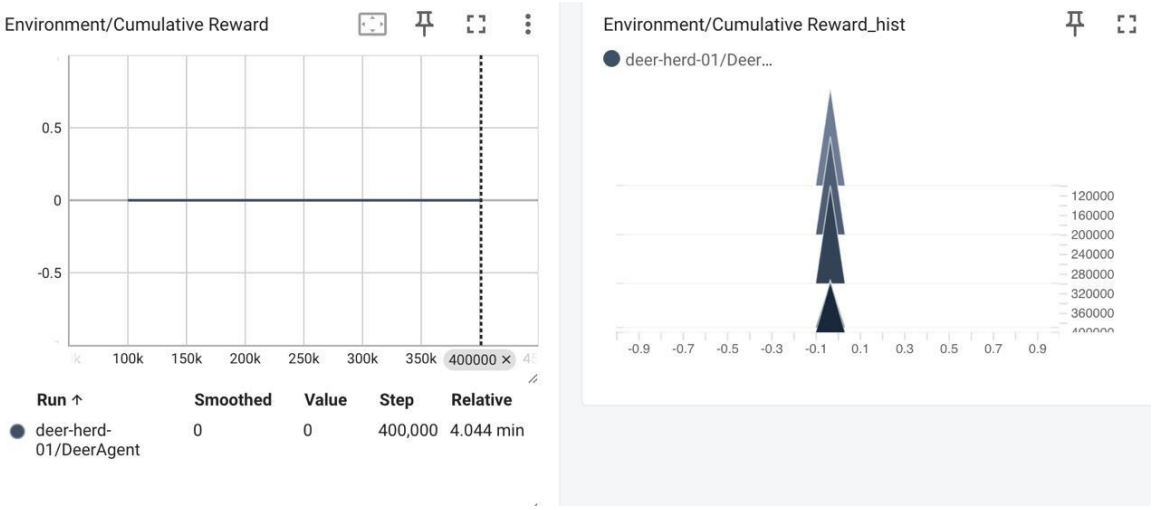

Рисунок В.1 – Графік та гістограма Environment/Cumulative Reward

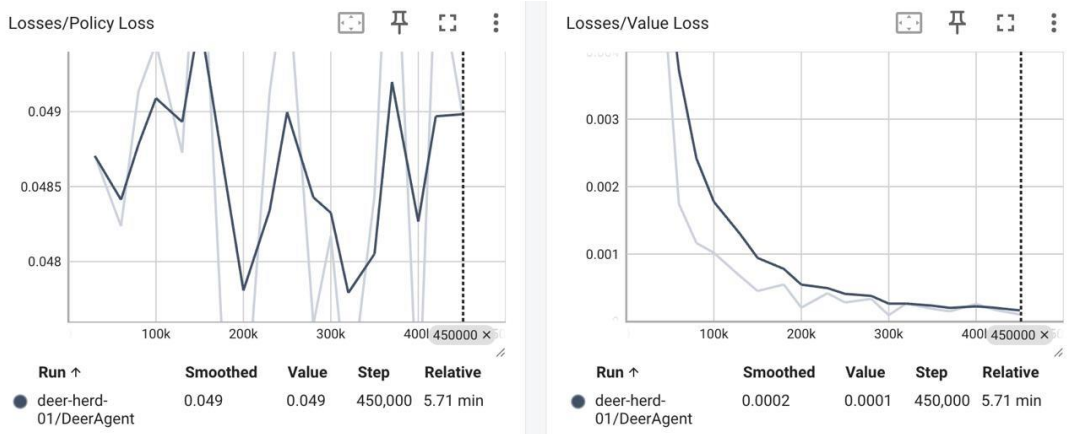

Рисунок В.2 – Графіки Losses/Value Loss та Losses/Policy Loss

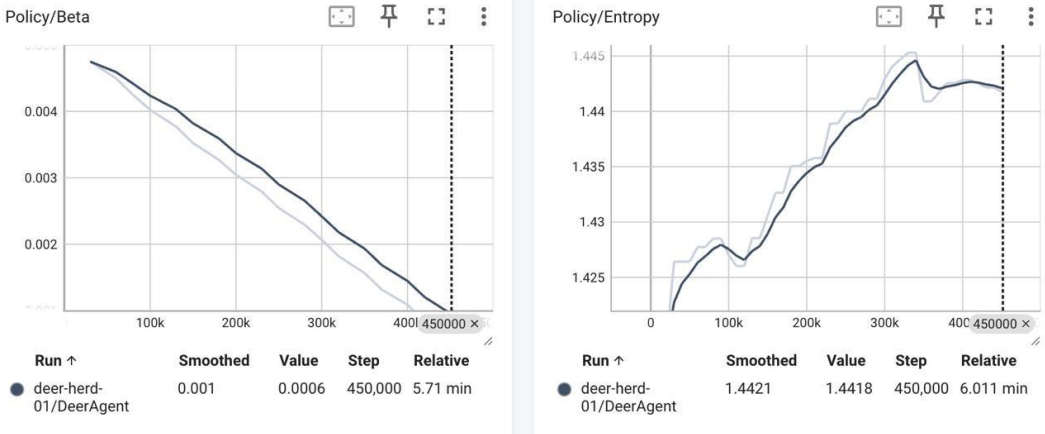

Рисунок В.3 – Графіки Policy/Beta та Policy/Entropy

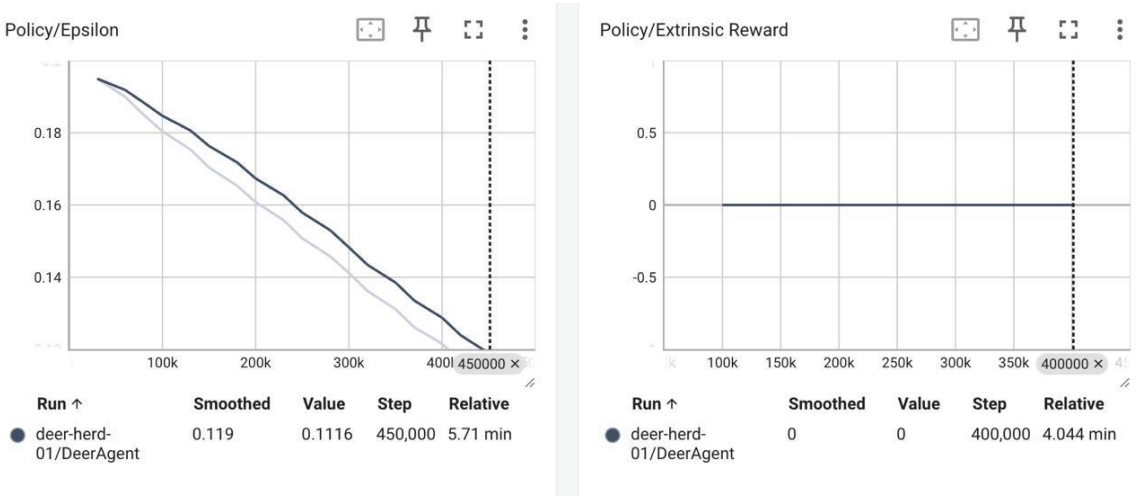

Рисунок В.4 – Графіки Policy/Epsilon та Policy/Extrinsic Reward

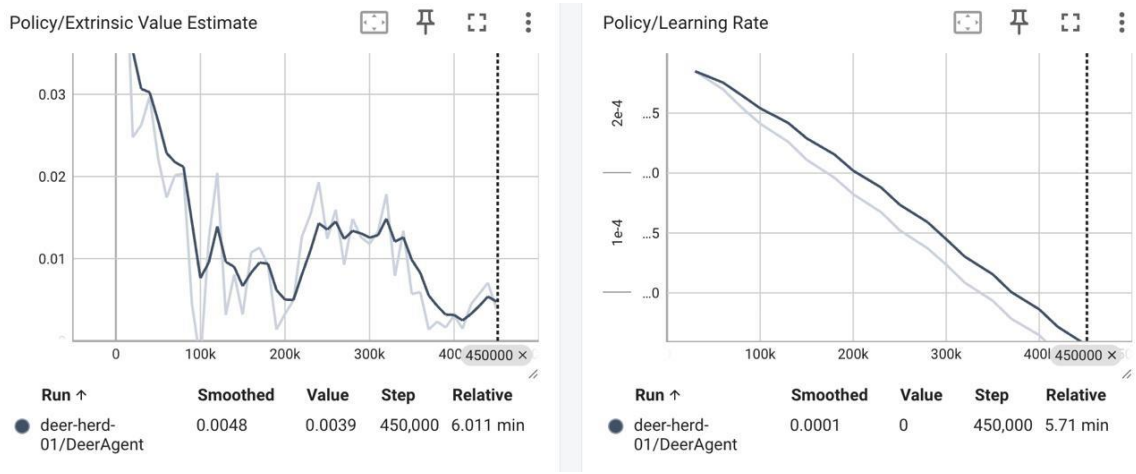

Рисунок В.5 – Графіки Policy/Extrinsic Value Estimate та Policy/Learning Rate