МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

В.о. завідувача кафедри інтелектуальних інформаційних систем

_____Євген СІДЕНКО

«____»_____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА

ІНТЕЛЕКТУАЛЬНА СИСТЕМА СУПРОВОДУ ОБ'ЄКТА В 3D-СЕРЕДОВИЩІ НА БАЗІ UNITY ТА НЕЙРОННИХ МЕРЕЖ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Інтелектуальні інформаційні системи»

Здобувач _____ Василь ГИЛЯКА

«___»_____ 2025 р.

Керівник канд. фіз.мат. наук, доцент _____Інесса КУЛАКОВСЬКА

«___»_____ 2025 р.

Миколаїв – 2025

Чорноморський національний університет імені Петра Могили

(повне найменування закладу вищої освіти)

Факультет	Факультет комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Другий (магістерський)
Освітній ступень	Магістр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Інтелектуальні інформаційні системи

ЗАТВЕРДЖУЮ

В. о. завідувача кафедри
інтелектуальних інформаційних систем
_____ Євген СІДЕНКО
«____» _____ 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

_____ Гиляки Василя Олександровича_____
(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: « Інтелектуальна система супроводу об'єкта в 3D-середовищі на базі Unity та нейронних мереж ».
Керівник роботи: Кулаковська Інесса Василівна, кандидат фіз.-мат. наук, доцент.
Затверджена наказом ЧНУ ім. Петра Могили від «7» липня 2025 р. № 184.
2. Строк представлення кваліфікаційної роботи «___» _____ 2025 р.
3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: тривимірне симульоване середовище, створене в рушії Unity, що містить агента супроводу, рухомий об’єкт-ціль та статичні і динамічні перешкоди; просторові координати агента та цілі, вектори швидкості й напрямку руху, дані сенсорних систем агента (Ray Perception Sensors); параметри фізичної симуляції середовища; конфігураційні параметри алгоритмів навчання з підкріпленням; дані, згенеровані в процесі взаємодії агента з середовищем під час тренування; журнали винагород,

дій та станів для аналізу ефективності навчання. Інтелектуальна система супроводу об’єкта в тривимірному середовищі на базі Unity та нейронних мереж, яка забезпечує автономне відстеження рухомої цілі, адаптивну поведінку агента, стійкість до змін середовища та ефективне уникнення перешкод, а також демонструє здатність до самонавчання з використанням алгоритмів глибинного навчання з підкріпленням.

4. Перелік питань, що підлягають розробці: аналіз сучасного стану та теоретичних засад задачі супроводу об’єктів у тривимірних середовищах, огляд класичних та нейромережевих підходів до відстеження і формування поведінки агентів; дослідження можливостей використання алгоритмів глибинного навчання з підкріпленням для задач супроводу у 3D-просторі та обґрунтування вибору середовища Unity, фреймворку ML-Agents і алгоритму Proximal Policy Optimization; проєктування архітектури інтелектуальної системи супроводу, визначення структури агента, сенсорної системи, простору станів і дій та функції винагороди; розробка тривимірного симульованого середовища та реалізація агента супроводу з використанням нейронної мережі; проведення навчання агента, налаштування гіперпараметрів моделі та аналіз процесу збіжності; тестування та оцінка ефективності розробленої системи супроводу в середовищах різної складності.

5. Перелік графічних матеріалів: презентація, рисунки, таблиці.

Керівник роботи _____ Інесса КУЛАКОВСЬКА

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

Здобувач _____ Василь ГИЛЯКА

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

Дата видачі завдання «07» липня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

кваліфікаційної роботи

Тема: Інтелектуальна система супроводу об'єкта в 3D-середовищі на базі Unity та нейронних мереж

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	29.06.2025	30.06.2025	Виконано
2	Аналіз предметної області та постановка задачі	1.09.2025	10.09.2025	Виконано
3	Огляд наукових публікацій та існуючих підходів до побудови інтелектуальних систем супроводу об’єктів у тривимірних середовищах.	11.09.2025	21.09.2025	Виконано
4	Аналіз алгоритмів навчання з підкріпленням і інструментальних засобів та обґрунтування вибору технологій реалізації системи супроводу.	22.09.2025	25.10.2025	Виконано
5	Реалізація обраних технологій із аналізом отриманих результатів	26.10.2025	24.11.2025	Виконано
6	Перший попередній захист КР на засіданні комісії кафедри	25.11.2025	25.11.2025	Виконано
7	Корегування роботи за результатами попереднього захисту	26.11.2025	08.12.2025	Виконано
8	Другий попередній захист КР на засіданні комісії кафедри	09.12.2025	09.12.2025	Виконано
9	Доробка та остаточне оформлення КР	10.12.2025	13.02.2025	Виконано
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	14.12.2025	15.12.2025	Виконано

Керівник роботи _____ Інесса КУЛАКОВСЬКА

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

Здобувач _____ Василь ГИЛЯКА

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

Дата складання календарного плану
«7» липня 2025 р

АНОТАЦІЯ

до кваліфікаційної роботи здобувача групи 601 ЧНУ ім. П. Могили

Гиляки Василя Олександровича

на тему: «**ІНТЕЛЕКТУАЛЬНА СИСТЕМА СУПРОВОДУ ОБ'ЄКТА В 3D-СЕРЕДОВИЩІ НА БАЗІ UNITY ТА НЕЙРОННИХ МЕРЕЖ**»

Кваліфікаційна робота присвячена розробці та програмній реалізації інтелектуальної системи супроводу об'єкта у тривимірному середовищі на базі рушія Unity з використанням нейронних мереж і методів навчання з підкріпленням. Запропонована система забезпечує автономне відстеження рухомої цілі, адаптивну поведінку агента та ефективну роботу в умовах динамічної зміни просторової конфігурації середовища. Застосування підходів глибинного навчання дозволяє вирішити актуальну проблему створення гнучких і стійких систем супроводу без необхідності ручного проєктування складних алгоритмів керування.

Об'єкт дослідження – процес супроводу рухомого об'єкта у тривимірному віртуальному середовищі.

Предмет дослідження – нейромережеві методи формування поведінки інтелектуального агента та програмні засоби їх реалізації в середовищі Unity.

Мета дослідження – розробка та дослідження інтелектуальної системи супроводу об'єкта в 3D-середовищі з використанням Unity та алгоритмів глибинного навчання з підкріпленням з метою підвищення точності, адаптивності та стійкості поведінки агента.

Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків та додатків. У першому розділі проаналізовано сучасний стан задачі супроводу об'єктів у тривимірних середовищах, розглянуто класичні та нейромережеві підходи до відстеження цілей, а також виконано огляд існуючих наукових досліджень і програмних рішень. У другому розділі розглянуто архітектуру інтелектуальної системи супроводу, описано структуру агента, сенсорні підсистеми, простір станів і дій, а також обґрунтовано вибір алгоритму Proximal Policy Optimization та фреймворку Unity ML-Agents. У третьому розділі описано створення тривимірного симульованого середовища, реалізацію агента супроводу, процес навчання нейронної моделі, підбір гіперпараметрів та аналіз поведінки агента під час тестування. У четвертому розділі наведено результати експериментальних досліджень, оцінено ефективність роботи системи в

середовищах різної складності, а також описано створені лабораторні роботи, які можуть бути використані у навчальному процесі.

У результаті виконання кваліфікаційної роботи розроблено інтелектуальну систему супроводу об’єкта в тривимірному середовищі, яка використовує дані 3D-сцени та нейронну політику для забезпечення точного і стабільного відстеження цілі в реальному часі.

Кваліфікаційна робота містить 96 сторінок, ілюстрації, таблицю, додатки та посилання на використані джерела.

Ключові слова: *супровід об’єктів, тривимірне середовище, Unity, інтелектуальні агенти, нейронні мережі, навчання з підкріпленням, глибинне навчання, ML-Agents, Proximal Policy Optimization, автономна поведінка.*

ABSTRACT

to the qualification work by the student of the group 601 of Petro Mohyla Black Sea National University

Gylyaka Vasyl

on the subject: «**INTELLIGENT OBJECT TRACKING SYSTEM IN A 3D ENVIRONMENT BASED ON UNITY AND NEURAL NETWORKS**»

The qualification thesis is devoted to the development and software implementation of an intelligent object tracking system in a three-dimensional environment based on the Unity engine using neural networks and reinforcement learning methods. The proposed system provides autonomous tracking of a moving target, adaptive agent behavior, and efficient operation under conditions of dynamically changing spatial configurations of the environment. The application of deep learning approaches makes it possible to address the relevant problem of creating flexible and robust tracking systems without the need for manual design of complex control algorithms.

The object of the research is the process of tracking a moving object in a three-dimensional virtual environment.

The subject of the research is neural network–based methods for shaping the behavior of an intelligent agent and software tools for their implementation within the Unity environment.

The purpose of the research is to develop and study an intelligent object tracking system in a 3D environment using Unity and deep reinforcement learning algorithms in order to improve the accuracy, adaptability, and stability of the agent's behavior.

The qualification thesis consists of an introduction, four chapters, conclusions, and appendices. The first chapter analyzes the current state of object tracking tasks in three-dimensional environments, examines classical and neural network–based approaches to target tracking, and reviews existing scientific research and software solutions. The second chapter considers the architecture of the intelligent tracking system, describes the agent structure, sensor subsystems, state and action spaces, and justifies the selection of the Proximal Policy Optimization algorithm and the Unity ML-Agents framework. The third chapter describes the development of a three-dimensional simulated environment, the implementation of the tracking agent, the neural model training process, hyperparameter tuning, and the analysis of agent

behavior during testing. The fourth chapter presents the results of experimental studies, evaluates the system's performance in environments of varying complexity, and describes the developed laboratory works that can be used in the educational process.

As a result of the qualification thesis, an intelligent object tracking system in a three-dimensional environment was developed, which uses 3D scene data and a neural policy to ensure accurate and stable real-time target tracking.

The qualification thesis contains 96 pages, illustrations, a table, appendices, and references to the sources used.

Keywords: *object tracking, three-dimensional environment, Unity, intelligent agents, neural networks, reinforcement learning, deep learning, ML-Agents, Proximal Policy Optimization, autonomous behaviour*

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ІС – інтелектуальна система

ІШІ – штучний інтелект

ПЗ – програмне забезпечення

3D – тривимірний простір

RL – навчання з підкріпленням

DRL – глибинне навчання з підкріпленням

НС – нейронна мережа

АГ – автономний агент

AI – Artificial Intelligence

DRL – Deep Reinforcement Learning

ML – Machine Learning

PPO – Proximal Policy Optimization

CNN – Convolutional Neural Network

NN – Neural Network

API – Application Programming Interface

GPU – Graphics Processing Unit

CPU – Central Processing Unit

IDE – Integrated Development Environment

FPS – Frames Per Second

SDK – Software Development Kit

C# – C Sharp programming language

Python – Python programming language

Unity – Unity Game Engine

ML-Agents – Unity Machine Learning Agents Toolkit

Ray Sensor – Ray Perception Sensor

NavMesh – Navigation Mesh

VR – Virtual Reality

AR – Augmented Reality

ВСТУП

Актуальність. В умовах стрімкого розвитку цифрових технологій та інтелектуальних систем зростає роль тривимірних віртуальних середовищ, які широко використовуються для моделювання складних процесів, навчання автономних агентів і створення інтерактивних програмних рішень. Одним із ключових напрямів у цій галузі є задача супроводу та відстеження об’єктів у 3D-просторі, що має важливе значення для робототехніки, автономної навігації, ігрової індустрії, систем безпеки, а також тренажерів віртуальної та доповненої реальності. Ефективність таких систем безпосередньо залежить від здатності агента точно визначати положення цілі, прогнозувати її подальший рух і своєчасно адаптувати власну поведінку до змін середовища.

Традиційні підходи до супроводу об’єктів у тривимірних середовищах здебільшого ґрунтуються на детермінованих алгоритмах керування, правилах поведінки або заздалегідь заданих математичних моделях руху. Хоча такі методи є відносно простими в реалізації, вони мають суттєві обмеження щодо гнучкості та масштабованості, особливо в умовах динамічних середовищ із непередбачуваною поведінкою цілі або наявністю перешкод. У складних сценаріях подібні алгоритми потребують ручного налаштування та не здатні ефективно адаптуватися до нових умов.

Перспективним напрямом розв’язання цієї проблеми є використання нейронних мереж і методів навчання з підкріпленням [7, 10, 22], які дозволяють агенту самостійно формувати стратегію поведінки на основі взаємодії з середовищем. Алгоритми глибинного навчання з підкріпленням забезпечують можливість навчання у симульованих середовищах, накопичення досвіду та вироблення оптимальних рішень без необхідності явного програмування кожної дії. Це створює передумови для побудови адаптивних і стійких систем супроводу, здатних працювати в умовах змінної геометрії сцени та складної динаміки руху об’єктів.

Особливе місце серед інструментів для реалізації подібних систем займає ігровий рушій Unity, який поєднує засоби тривимірного моделювання, фізичної симуляції та інтеграції алгоритмів штучного інтелекту. Використання фреймворку Unity ML-Agents [1, 3, 17] надає можливість застосовувати сучасні алгоритми навчання з підкріпленням, зокрема Proximal Policy Optimization, для створення інтелектуальних агентів, здатних навчатися в процесі взаємодії з 3D-середовищем і демонструвати автономну поведінку.

Проблема полягає у відсутності універсальних і доступних програмних рішень для інтелектуального супроводу об’єктів у тривимірних середовищах, які поєднували б високу точність відстеження, адаптивність до змін середовища та простоту інтеграції у навчальні й прикладні проєкти. Більшість існуючих реалізацій орієнтовані на вузькоспеціалізовані задачі або потребують значних витрат часу на ручне налаштування поведінки агентів. У зв’язку з цим актуальною є розробка інтелектуальної системи супроводу, здатної навчатися у симульованому 3D-середовищі та ефективно працювати в умовах різної складності.

Метою дослідження є розробка, дослідження та аналіз інтелектуальної системи супроводу об’єкта у тривимірному середовищі з використанням рушія Unity та алгоритмів глибинного навчання з підкріпленням, спрямованих на підвищення точності, адаптивності та стійкості поведінки автономного агента.

Об’єктом дослідження є процес супроводу рухомого об’єкта у тривимірному віртуальному середовищі.

Предметом дослідження є нейромережеві методи формування поведінки інтелектуального агента та програмні засоби їх реалізації у середовищі Unity.

Практичне значення отриманих результатів полягає в можливості використання розробленої інтелектуальної системи супроводу об’єкта як демонстраційної та навчально-прикладної платформи для дослідження алгоритмів навчання з підкріпленням у тривимірних середовищах. Реалізована система може застосовуватися у навчальному процесі під час вивчення дисциплін, пов’язаних із

штучним інтелектом, машинним навчанням, комп’ютерним моделюванням та розробкою ігор, зокрема для демонстрації принципів формування поведінки автономних агентів у середовищі Unity.

Отримані результати також можуть бути використані як основа для створення прототипів систем супроводу та навігації в задачах робототехніки, автономних віртуальних агентів, тренажерів віртуальної та доповненої реальності, а також у ігрових проєктах, де необхідне адаптивне відстеження об’єктів у реальному часі. Архітектура системи та підхід до навчання агента дозволяють легко модифікувати середовище, параметри симуляції та функцію винагороди, що забезпечує можливість подальшого розширення функціональності та адаптації системи до нових сценаріїв використання.

Структура кваліфікаційної роботи. Відповідно до мети, завдань і предмета дослідження кваліфікаційна робота складається із вступу, чотирьох розділів, висновку, списку використаних джерел та 6 додатків. Загальний обсяг кваліфікаційної роботи – 96 сторінок, кількість використаних джерел – 22.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Сучасний стан технологій супроводу об’єктів у 3D-середовищах

Супровід об’єктів у тривимірних віртуальних середовищах є однією з ключових задач, які активно досліджуються в галузях робототехніки, комп’ютерного моделювання, систем безпеки та інтерактивних застосунків. За останні роки розвиток технологій штучного інтелекту, алгоритмів комп’ютерного зору та фізичних симуляцій істотно розширив можливості таких систем, дозволяючи створювати моделі, здатні відстежувати цілі у реальному часі, аналізувати траєкторії та прогнозувати подальший рух.

Проблема супроводу в загальному вигляді полягає у визначенні положення об’єкта, оцінці його швидкості та напрямку руху, а також побудові поведінки агента, що дозволяє безперервно тримати ціль у зоні видимості або в межах заданої дистанції (див. рис. 1.1). У реальних задачах 3D-супровід використовується в таких сферах, як:

- навігація автономних роботів та дронів, що повинні слідкувати за переміщенням об’єктів або уникати зіткнень;

- системи відеоспостереження, де камери відстежують підозрілу активність або переміщення людей;

- комп’ютерні ігри, в яких неігрові персонажі (NPC) переслідують гравця або супроводжують його;

- VR/AR-тренажери, де потрібне точне реагування на рухи користувача;
індустрія анімації, де автоматизоване керування персонажами полегшує роботу аніматорів.

У традиційних системах супровід здійснювався переважно за допомогою класичних алгоритмів, таких як пошук шляху A*, моделі переслідування «steering behaviors», пропорційно-інтегральні контролери, алгоритми прогнозування руху

(наприклад, Kalman Filter [7, 14]). Проте зі зростанням складності віртуальних середовищ ці підходи часто стають недостатніми, оскільки вони погано пристосовані до ситуацій із динамічними перешкодами, складною геометрією рівнів та непередбачуваною поведінкою цілі.

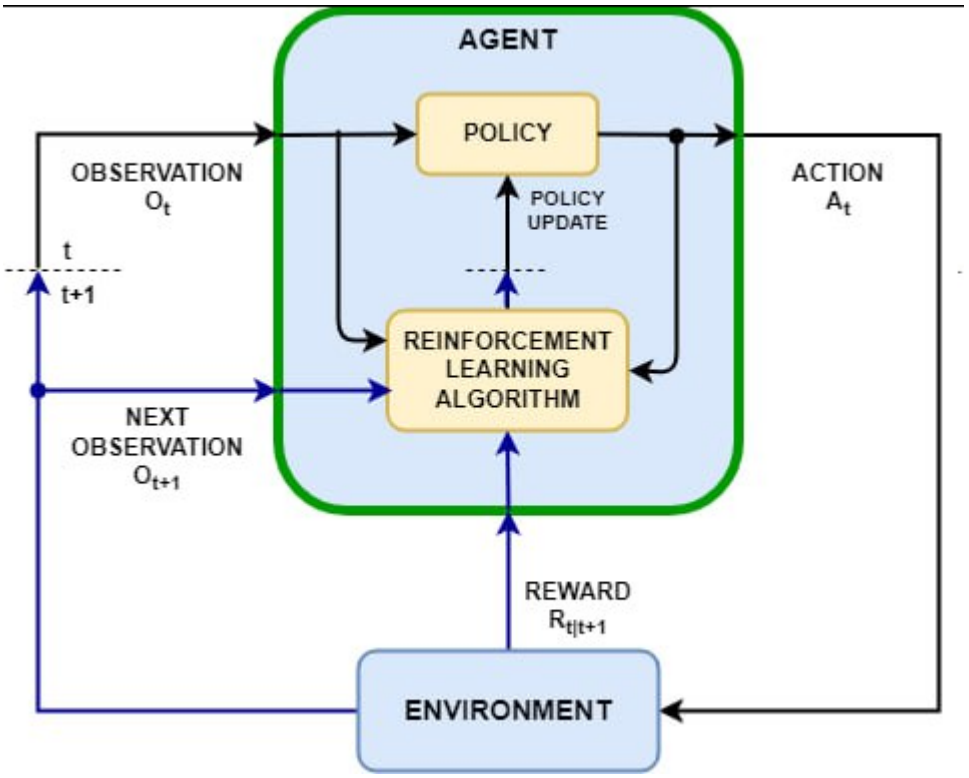

Рисунок 1.1 – Архітектура інтелектуальної системи супроводу

Сучасні тенденції розвитку 3D-систем показують перехід до використання нейронних мереж, які здатні:

- адаптуватися до нових сценаріїв без ручного перепроектування поведінки;
- навчатися на симуляціях, що значно спрощує підготовку даних;
- реалізовувати складні поведінкові моделі, недоступні для класичних методів;
- приймати рішення на основі багатовимірних сенсорних даних.

Разом із тим, інтеграція таких методів у систему супроводу вимагає побудови спеціальної архітектури, що включає модулі сприйняття, аналізу, прийняття рішень та контролю руху агента. Особливої уваги заслуговують рушії

на кшталт **Unity**, які поєднують можливості 3D-моделювання з інструментами машинного навчання через ML-Agents [1, 3, 17], дозволяючи створювати повноцінні інтелектуальні системи у симульованому середовищі (див. рис. 1.2).

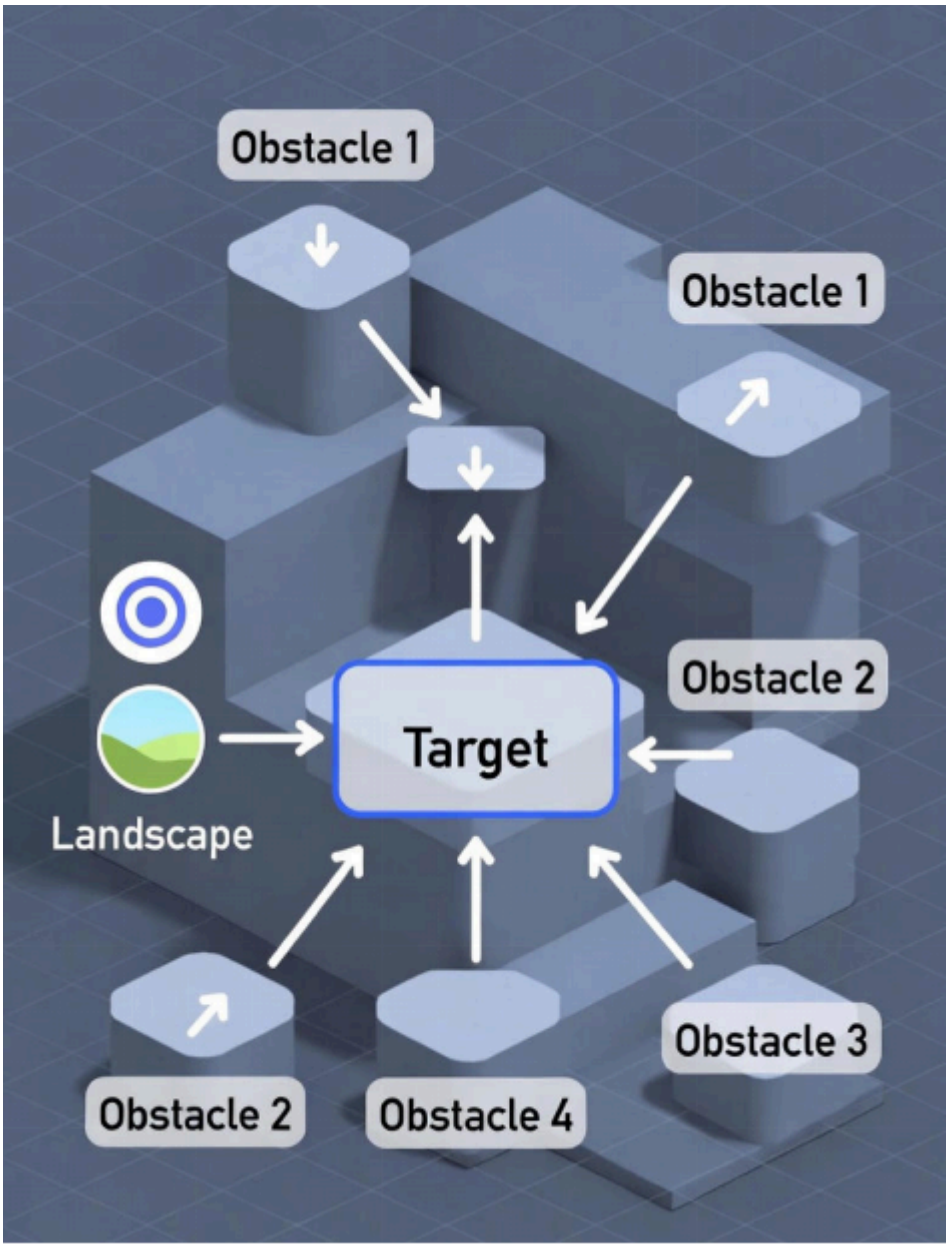

Рисунок 1.2 – Структура 3D-середовища та ключових об’єктів

1.2 Методи відстеження та супроводу об’єктів: класичні та нейромережеві підходи

Задача супроводу об’єкта у тривимірному середовищі є складною багатокомпонентною проблемою, яка поєднує елементи навігації, прогнозування руху, обробки сенсорних даних [5, 9, 10] і прийняття рішень у реальному часі. Протягом останніх років її розв’язання здійснювалося за допомогою різноманітних методів, що можна умовно розділити на дві категорії: класичні алгоритми керування та сучасні нейромережеві підходи. Кожен із них має власні переваги та недоліки, які визначають сферу їх ефективного застосування.

Класичні методи супроводу об’єктів в основному ґрунтуються на математичних моделях руху і передбачають використання детермінованих алгоритмів (див. рис. 1.3). Найбільш поширеними серед них є алгоритми пошуку шляху, такі як A* або Dijkstra, які дозволяють агенту знаходити оптимальну траєкторію у статичних або частково змінних середовищах. У рамках ігрових рушіїв широко застосовуються також навігаційні сіті (NavMesh), що дають змогу ефективно переміщувати персонажів між точками з урахуванням наявних перешкод. Проте подібні підходи мають суттєві обмеження, оскільки вони не здатні динамічно адаптуватися до непередбачуваної поведінки цілі та не враховують високий ступінь мінливості середовища.

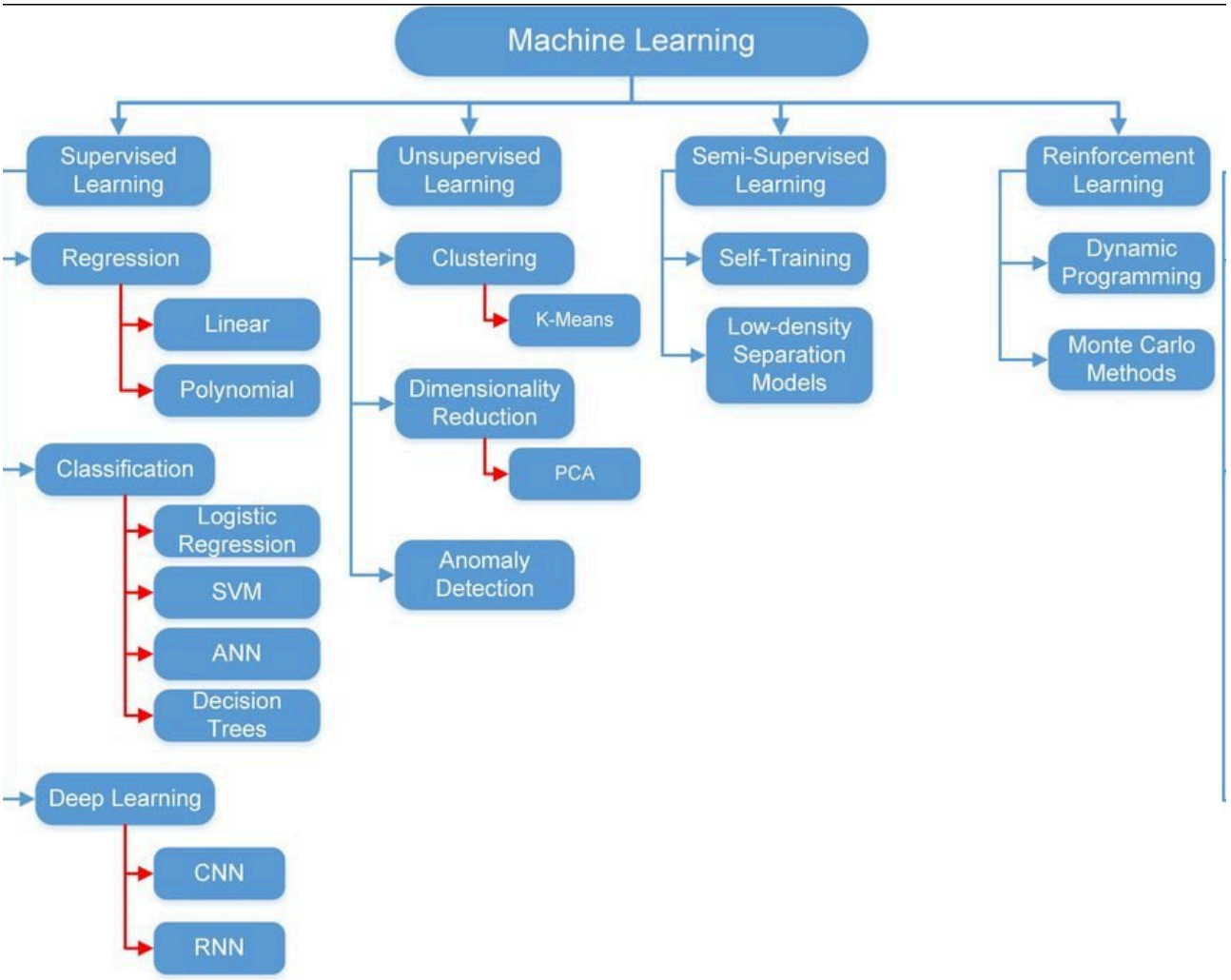

Рисунок 1.3 – Класифікація методів супроводу: класичні та нейромережеві підходи

Розповсюдженим класичним підходом є модель керованої поведінки (steering behaviors), концепція якої була запропонована Крейґом Рейнольдсом [7]. Вона описує рухи агента у вигляді суми сил, що спрямовують його до цілі, допомагають уникати зіткнень або слідувати за певною траєкторією. Хоча такі методи є ефективними у простих сценаріях, у складних тривимірних середовищах з великою кількістю перешкод та динамічних об'єктів вони не забезпечують достатньої точності та гнучкості.

У технічних системах, зокрема в робототехніці, широко застосовуються регулятори типу PID, що контролюють рух у відповідь на відхилення від заданої

позиції. Для прогнозування майбутнього розташування об’єкта використовуються фільтри Калмана, які дозволяють згладжувати шум вимірювань та оцінювати швидкість і напрямок руху. Незважаючи на їхню точність, такі методи також мають недоліки, оскільки потребують детального математичного моделювання середовища і практично не здатні адаптуватися до змін у поведінці цілі.

На противагу класичним рішенням, сучасні методи супроводу все частіше базуються на штучних нейронних мережах, які дають змогу агенту самостійно вивчати структуру середовища, виявляти закономірності у поведінці цілі та адаптуватися до непередбачуваних ситуацій. Особливого поширення набули алгоритми навчання з підкріпленням (Reinforcement Learning), зокрема Deep Reinforcement Learning [5–7, 10], які поєднують можливості глибинних нейронних мереж і теорії оптимального керування. Такі алгоритми дозволяють агенту отримувати досвід шляхом взаємодії з середовищем, формуючи стратегію поведінки на основі позитивних і негативних винагород.

Одним із найбільш ефективних підходів є Proximal Policy Optimization (PPO), який використовується в Unity ML-Agents [2, 6]. Він забезпечує стабільність навчання, високу стійкість до шуму та здатність працювати у середовищах із багатовимірними станами. Нейронна мережа PPO може враховувати позицію агента, напрям погляду, швидкість руху цілі та інші сенсорні дані, формуючи оптимальну дію у кожному кроці симуляції.

У деяких системах поєднуються переваги класичних та нейромережевих методів. Наприклад, агент може використовувати нейронну мережу для визначення цільового напрямку руху, але при цьому застосовувати NavMesh для корекції траєкторії та уникнення складних перешкод. Подібні гібридні моделі забезпечують високу точність супроводу та стійкість поведінки агента у складних сценаріях.

Таким чином, еволюція систем супроводу об’єктів у 3D-середовищах засвідчує поступовий перехід від жорстко заданих алгоритмів до інтелектуальних

методів, здатних самонавчатися та адаптуватися до динаміки середовища. Саме нейромережеві підходи сьогодні вважаються найбільш перспективними для побудови ефективних та гнучких систем супроводу.

1.3 Огляд платформ та інструментів для розробки інтелектуальних 3D-систем

Створення інтелектуальних систем супроводу у тривимірних середовищах потребує використання комплексних технологічних рішень, що охоплюють моделювання 3D-простору, фізичні симуляції, алгоритми машинного навчання та засоби аналізу. На сучасному етапі розвитку індустрії найбільш придатним інструментом для таких задач є рушій Unity, який поєднує в собі гнучкі засоби моделювання та потужну підтримку штучного інтелекту через ML-Agents [1, 17].

Unity забезпечує зручні можливості для побудови тривимірних сцен, що включають об’єкти різної складності, системи освітлення, динамічні перешкоди та фізичні взаємодії. Вбудований фізичний двигун дозволяє моделювати реалістичні рухи, зіткнення, прискорення та інші параметри, що мають вирішальне значення для коректного функціонування системи супроводу. Розробка поведінки агента здійснюється на мові C#, яка забезпечує повний контроль над логікою взаємодії компонентів та дозволяє інтегрувати машинне навчання у робочий цикл симуляції.

Особливої уваги заслуговує фреймворк Unity ML-Agents, який дає змогу реалізувати навчання агентів за допомогою алгоритмів глибинного навчання з підкріпленням. ML-Agents поєднує можливості Python та Unity, що дозволяє використовувати сучасні фреймворки нейронних мереж — TensorFlow або PyTorch — для створення моделей, що визначають поведінку агента. Система сенсорів ML-Agents може збирати дані про стан оточення: відстані до перешкод, координати цілі, напрям руху та іншу інформацію, яка формує вхідні дані для мережі.

TensorFlow тривалий час був основним фреймворком, який використовувався у ML-Agents, завдяки своїй стабільності та поширеності у промислових системах. Його архітектура статичних графів була зручною для великих моделей, але менш гнучкою для експериментальних досліджень. Згодом у ML-Agents було інтегровано PyTorch, який відзначається більшою динамічністю та простотою дебагінгу, що сприяє швидкому прототипуванню нових архітектур.

Крім того, у процесі навчання агентів широко використовуються інструменти для обробки числових даних, такі як NumPy, які значно спрощують роботу з матрицями та векторами. Для візуалізації динаміки навчання застосовуються TensorBoard і Matplotlib, які дозволяють спостерігати зміни винагороди, стабільність політики, швидкість збіжності та інші ключові показники.

Таким чином, сучасні платформи та інструменти створюють комплексну екосистему для розробки інтелектуальних систем супроводу у 3D-середовищах. Поєднання Unity, ML-Agents, PyTorch або TensorFlow забезпечує повний цикл розробки — від побудови середовища до тренування та тестування агентів — що робить цей підхід одним із найефективніших у сфері моделювання інтелектуальних віртуальних систем [4, 8].

1.4 Постановка задачі дослідження

Сучасні інтелектуальні системи, що функціонують у тривимірних середовищах, пред’являють високі вимоги до точності, адаптивності та стійкості агентів, які здійснюють супровід або відстеження об’єктів. У таких системах важливо не лише визначити координати цілі, а й забезпечити правильну реакцію агента на зміни її поведінки, появу перешкод або зміну конфігурації середовища. Враховуючи складність подібних завдань, виникає необхідність застосування

технологій машинного навчання, які дозволяють агента навчати шляхом взаємодії з оточенням та накопичення досвіду.

Постановка задачі у межах цієї роботи ґрунтується на аналізі особливостей супроводу об’єктів у 3D-середовищах та визначенні ключових факторів, що впливають на ефективність такої системи (див. рис. 1.4). Основна проблема полягає у тому, що класичні алгоритми не здатні забезпечити задовільну гнучкість у сценаріях, де поведінка цілі є динамічною або непередбачуваною. Відповідно, необхідно створити систему, в якій агент зможе адаптувати свої дії у процесі навчання, враховуючи широкий спектр можливих ситуацій.

У рамках дослідження передбачається вирішення таких завдань:

- провести аналіз сучасних підходів до супроводу об’єктів у тривимірних середовищах та визначити недоліки класичних методів з позиції адаптивності та гнучкості;

- дослідити можливості сучасних фреймворків машинного навчання, зокрема Unity ML-Agents, TensorFlow та PyTorch, щодо реалізації поведінки агентів у динамічних середовищах;

- розробити структуру інтелектуальної системи супроводу, яка включає 3D-середовище, агента, модуль сенсорики, систему винагороди та модуль машинного навчання;

- реалізувати агентну модель у середовищі Unity з використанням алгоритмів глибинного навчання з підкріпленням;

провести тестування розробленої системи в умовах різної складності середовища, включаючи динамічні перешкоди та змінні траєкторії руху цілі;

- оцінити ефективність поведінки агента, визначити його здатність адаптуватися та правильно реагувати на непередбачувані ситуації;

- розробити рекомендації щодо покращення моделі та можливостей подальшої модернізації системи.

Таким чином, постановка задачі дослідження включає комплексний підхід до створення інтелектуальної системи супроводу об’єкта у 3D-середовищі. Результати виконаної роботи повинні продемонструвати, що застосування методів навчання з підкріпленням забезпечує високий рівень адаптивності агента, здатність працювати у складних динамічних умовах та ефективно вирішувати задачу супроводу без ручного проєктування складних алгоритмів поведінки.

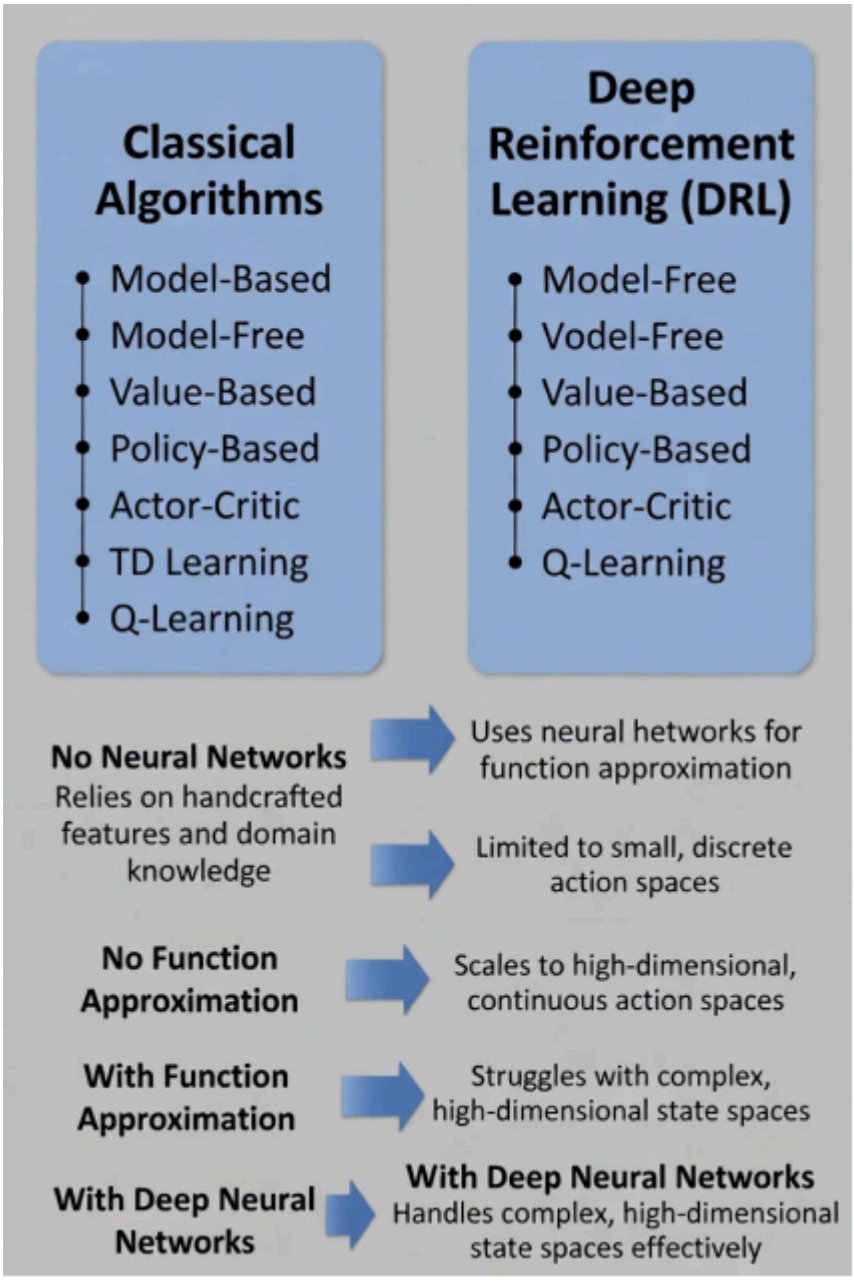

Рисунок 1.4 – Порівняння класичних алгоритмів та DRL

Висновки до розділу 1

У першому розділі було проведено комплексний аналіз проблеми супроводу об’єктів у тривимірних середовищах та визначено ключові аспекти, що впливають на точність і надійність таких систем. Розглянуто сучасний стан досліджень, який засвідчує високий попит на інтелектуальні системи, здатні адаптуватися до динамічних змін середовища. Класичні методи супроводу, попри їхню поширеність, мають низку обмежень, серед яких недостатня гнучкість та складність масштабування у складних сценаріях.

Огляд нейромережевих підходів показав, що методи глибинного навчання з підкріпленням є найбільш перспективними для формування поведінки автономного агента у складних середовищах. Вони дозволяють навичково формувати стратегію слідування за ціллю без необхідності детального математичного опису кожної ситуації. Також у розділі було досліджено наявні програмні платформи, серед яких Unity та ML-Agents займають провідні позиції завдяки можливості інтеграції фізичної симуляції, тренування агентів та візуалізації процесу навчання.

Кінцева постановка задачі дослідження дозволила сформувати чіткий перелік цілей, які мають бути досягнуті в межах роботи, що включає розробку інтелектуальної системи супроводу об’єкта, її реалізацію, тестування та оцінку ефективності. Отже, результати першого розділу формують теоретичну та методологічну основу для подальшої реалізації та експериментальної перевірки системи, що розглядається у наступних розділах.

2 СТРУКТУРА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ СУПРОВОДУ ОБ’ЄКТА В 3D-СЕРЕДОВИЩІ

2.1 Архітектура системи: основні компоненти та взаємодія

Побудова інтелектуальної системи супроводу об’єкта в тривимірному середовищі потребує розроблення чіткої архітектури, що визначає функціональні модулі, взаємозв’язки між ними та послідовність обробки інформації. Така архітектура повинна забезпечувати адаптивність агента, коректну взаємодію з оточенням, ефективну обробку сенсорних даних і можливість реалізації складної поведінкової моделі шляхом навчання з підкріпленням. У цій роботі система побудована на основі рушія Unity та фреймворку ML-Agents (рис. 2.1), що дозволило об’єднати тривимірне моделювання, фізичні симуляції та машинне навчання в єдиному середовищі [17].

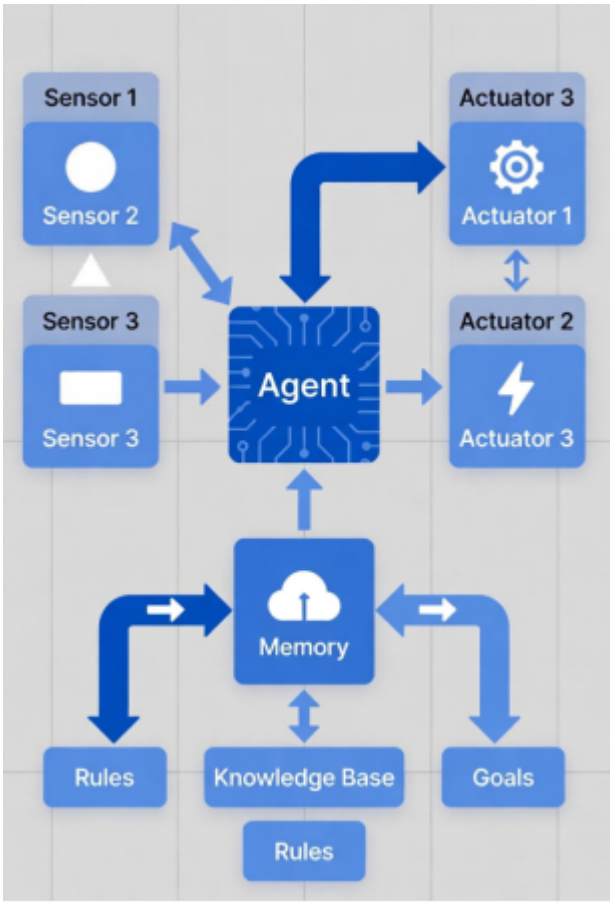

Рисунок 2.1 – Внутрішня структура агента

Загальна архітектура системи складається з таких основних компонентів: 3D-середовище, агент, ціль, сенсорна підсистема, модуль прийняття рішень, модуль управління рухом, модуль винагороди, а також машинний навчальний модуль, відповідальний за тренування нейронної мережі агента (див. рис. 2.2). Усі ці елементи взаємодіють між собою, утворюючи замкнений цикл сприйняття, дії та навчання.

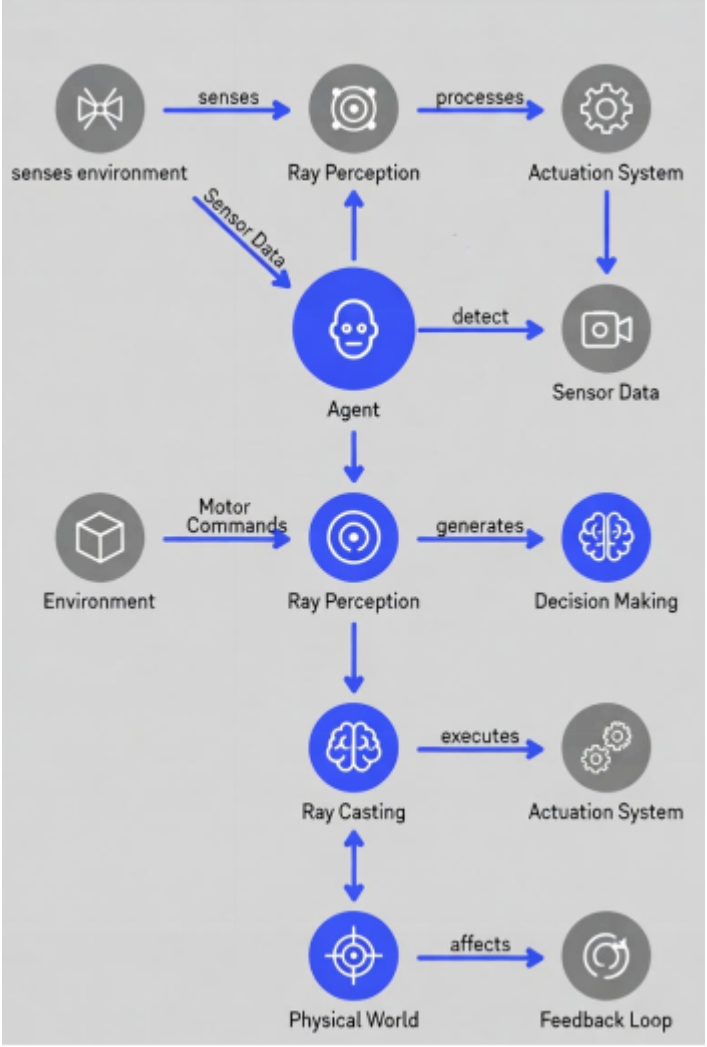

Рисунок 2.2 – Схема роботи сенсорної системи агента

3D-середовище є основою всієї системи і містить об’єкти, з якими агент може взаємодіяти: ціль, перешкоди, поверхні та структури сцени. Завдяки фізичному двигуну Unity середовище відтворює закони руху, зіткнення та взаємодії, що дозволяє моделювати реалістичні умови для навчання агента. Середовище також визначає геометрію сцени, топологію руху та складність завдань, які повинен виконувати агент.

Агент є головним активним компонентом системи і представлений об’єктом у віртуальному просторі з можливістю переміщення та обертання. Його поведінка визначається нейронною мережею, яка формує рішення на основі вхідних даних від сенсорів. В агент також інтегровані компоненти для збору даних, такі як Ray

Perception Sensors, які дозволяють виявляти перешкоди, визначати відстані та напрямки до цілі або інших об’єктів. Крім того, агент має внутрішні параметри, наприклад власну швидкість, положення або кут огляду, що також можуть бути включені у вхідний вектор станів.

Ціль виступає об’єктом, за яким агент повинен слідувати. Вона може бути статичною або рухатися за визначеною або випадковою траєкторією. У складніших сценаріях траєкторія цілі може бути нелінійною або змінюватися відповідно до зовнішніх факторів, що збільшує складність задачі супроводу. Система працює таким чином, що агент отримує винагороду за зменшення відстані до цілі або підтримання її у зоні видимості, а також штрафи за втрату зв’язку або зіткнення з перешкодами.

Сенсорна підсистема агента збирає інформацію про середовище та формує вектор стану, який подається на вхід нейронної мережі. Залежно від конфігурації, ця підсистема може включати променеві сенсори, вектори позицій, напрямок руху агента та цілі, а також дані про швидкість. Сенсорна система є критично важливою, оскільки саме вона забезпечує агенту можливість сприймати довкілля, що визначає якість прийнятих рішень.

Модуль прийняття рішень відповідає за інтерпретацію результатів нейронної мережі. На основі цих результатів агент визначає, у який бік рухатись, з яким прискоренням або під яким кутом повертатися. У системах на основі ML-Agents ці рішення формуються політикою, яка навчається під час симуляцій, а в робочому режимі використовується у вигляді замороженої нейронної моделі.

Модуль управління рухом реалізує перетворення прийнятих рішень у конкретні фізичні дії в середовищі Unity. Він відповідає за застосування сил, зміни швидкості, повороту та інших параметрів, що визначають фактичну поведінку агента. Важливо, що цей модуль працює у тісній взаємодії з компонентами фізики Unity.

Окремим елементом архітектури є модуль винагороди, який відіграє важливу роль у процесі навчання агента. Він визначає, які події повинні стимулювати агента, а які — карати. Наприклад, агент отримує позитивну винагороду за наближення до цілі, підтримання стабільної дистанції або успішний супровід, і негативну — за зіткнення чи втрату контакту з ціллю. Саме система винагород формує поведінкову стратегію агента та визначає ефективність його навчання.

Машинний навчальний модуль працює на стороні Python і відповідає за тренування нейронної мережі за допомогою алгоритмічних методів глибинного навчання з підкріпленням. Він аналізує дії агента, його успішність та формує оновлення параметрів політики. У процесі навчання мережа оптимізує свої ваги, щоб максимізувати кумулятивну винагороду й удосконалювати стратегічну поведінку.

Таким чином, архітектура інтелектуальної системи супроводу є комплексною структурою, у якій усі елементи взаємодіють у режимі реального часу, утворюючи адаптивний цикл "сприйняття → рішення → дія → оцінка". Така модель забезпечує здатність агента ефективно супроводжувати об’єкт у складному тривимірному середовищі та реагувати на зміни умов.

2.2 Модуль машинного навчання: архітектура нейронної мережі, навчання агента, стан і дії

Проєктування модуля машинного навчання є ключовим етапом створення інтелектуальної системи супроводу об’єкта у 3D-середовищі, оскільки саме цей модуль визначає те, як агент буде сприймати середовище, приймати рішення та адаптувати свою поведінку в процесі навчання. У даному дослідженні основою моделі виступає підхід глибинного навчання з підкріпленням (Deep Reinforcement Learning), реалізований засобами Unity ML-Agents. Цей підхід дозволяє агенту

здобувати навички шляхом багаторазової взаємодії зі середовищем, формування стратегій поведінки та оптимізації своїх дій для досягнення максимальної винагороди.

Фундаментальним елементом навчального модуля є **нейронна мережа**, що реалізує політику агента. Вона складається з кількох шарів, які обробляють вхідні дані, що надходять від сенсорної підсистеми агента (див. рис. 2.3). Залежно від складності середовища, нейронна мережа може включати щільні шари, рекурентні компоненти або нормалізаційні блоки. У більшості випадків модель складається з двох окремих голов: **actor** та **critic**. Перша відповідає за вибір дій агента, тоді як друга оцінює якість цих дій, визначаючи значення функції переваги. Така архітектура є характерною для алгоритму Proximal Policy Optimization [6, 7], який застосовується в ML-Agents і забезпечує стабільність та ефективність навчання.

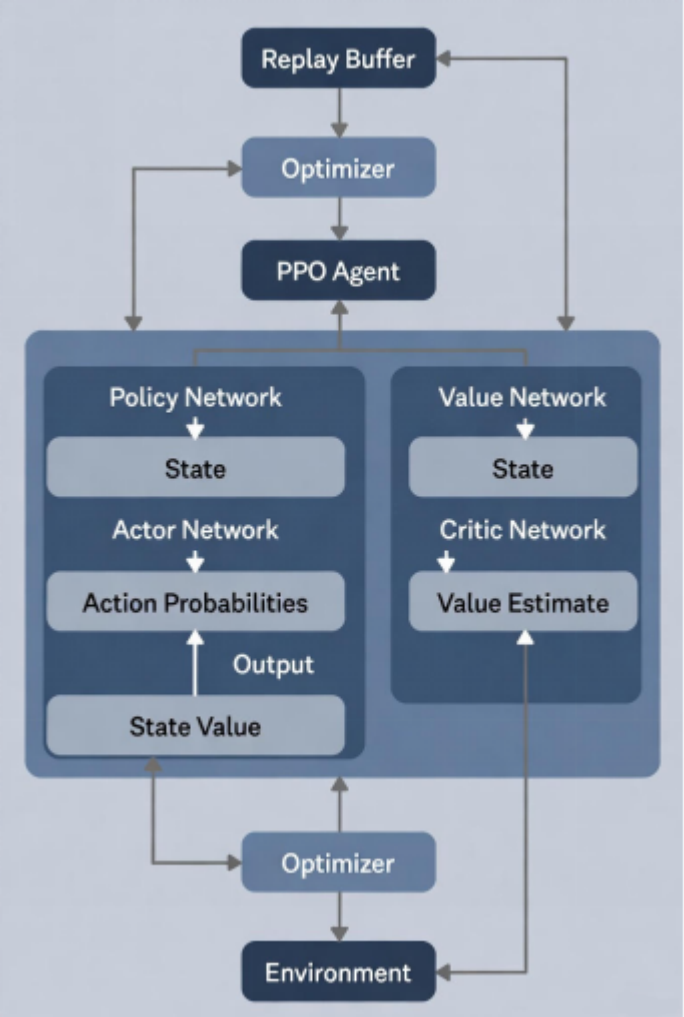

Рисунок 2.3 – Архітектура нейронної мережі PPO агента

Вхідними даними для нейронної мережі є **вектор стану**, який формується сенсорною системою агента. Залежно від конфігурації середовища, цей вектор може включати позицію цілі відносно агента, вектор напрямку руху, відстань до найближчих перешкод, власну швидкість, орієнтацію, а також інформацію з променевих сенсорів. Дані попередньо нормалізуються та подаються на вхід мережі, що дозволяє стабілізувати навчання та уникнути переобтяження моделі.

Простір дій, тобто набір можливих рухів агента, визначається на рівні конфігурації ML-Agents. Для системи супроводу об’єкта найчастіше використовується безперервний простір дій, у якому агент може регулювати швидкість руху та кут повороту у реальному часі. Кожна дія, яку видає нейронна

мережа, перетворюється в конкретні фізичні сили або вектори переміщення, що застосовуються до агента у середовищі Unity.

Навчання агента відбувається шляхом багаторазового проходження епізодів, у яких він намагається виконати поставлене завдання — супроводжувати ціль, зменшувати відстань до неї, уникати перешкод та зберігати стабільну траєкторію руху. Ключовим елементом навчання є **функція винагороди** [7, 18], що визначає, які події є бажаними, а які — небажаними. Позитивні винагороди агент отримує за наближення до цілі, підтримання заданої дистанції, орієнтацію у напрямку об’єкта або за успішне завершення епізоду. Негативні — за зіткнення з перешкодами, втрату контакту з ціллю або рух у протилежному напрямку. Добре сконструйована функція винагороди відіграє вирішальну роль у швидкості та стабільності навчання, оскільки вона формує поведінкову стратегію агента.

Процес тренування політики виконується на стороні Python з використанням алгоритму PPO, що оптимізує ваги нейронної мережі на основі зібраного досвіду. Алгоритм поступово коригує параметри моделі, дозволяючи агенту уникати різких змін політики та забезпечуючи плавне збільшення очікуваної кумулятивної винагороди (див. рис. 2.4). У процесі навчання система використовує механізми батчів, буферів досвіду та градієнтної оптимізації, які забезпечують швидку збіжність та покращення поведінки агента.

Результатом тренування є **збережена політика**, яку можна завантажити в Unity та застосовувати у режимі інференсу. У цьому режимі агент більше не навчається, а використовує оптимізовану нейронну мережу для прийняття рішень у реальному часі. Це дозволяє оцінити якість навчання, протестувати поведінку агента у складніших середовищах та визначити, наскільки він здатний адаптуватися до нових ситуацій.

Таким чином, модуль машинного навчання є центральним елементом інтелектуальної системи супроводу. Він забезпечує агента здатністю навчатися та вдосконалювати власні навички шляхом взаємодії з середовищем, формує

стратегію поведінки та визначає ефективність виконання задачі супроводу цілі. Правильне проєктування архітектури нейронної мережі, налаштування сенсорів та структурування функції винагороди є критичними факторами, що впливають на якість і стабільність усієї системи.

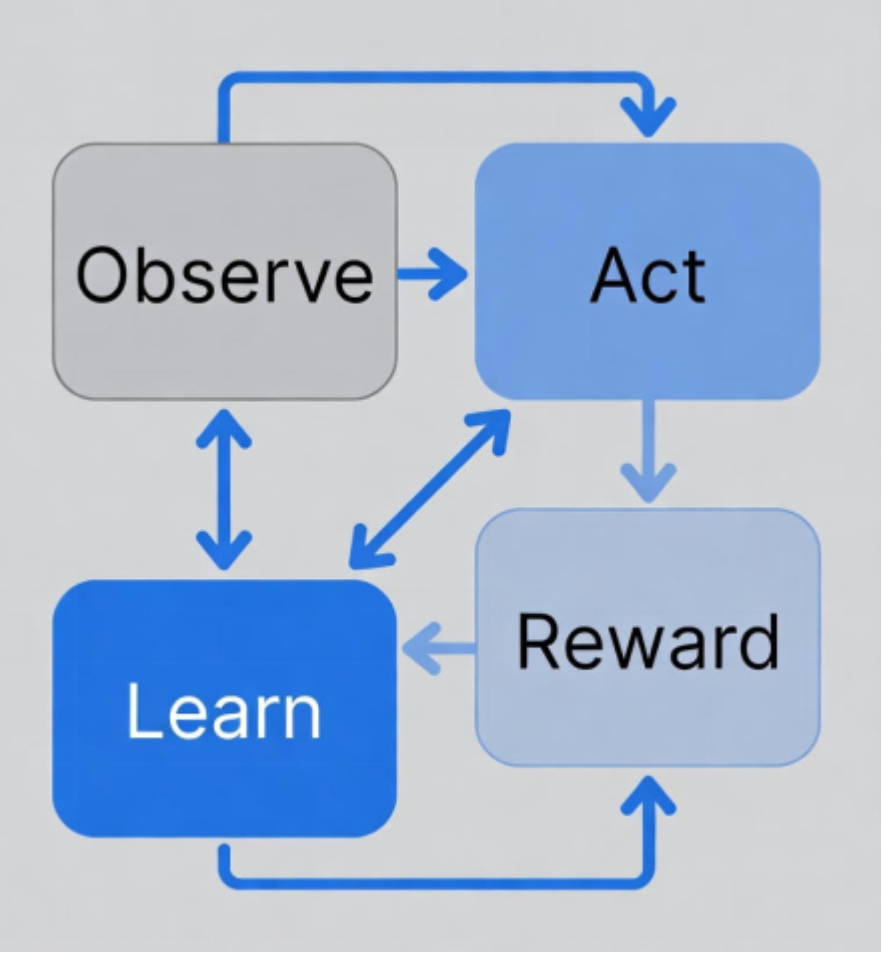

Рисунок 2.4 – Цикл навчання агента

2.3 Використані інструменти та мови програмування (Unity, C#, ML-Agents, Python)

Розроблення інтелектуальної системи супроводу об’єкта в 3D-середовищі вимагає використання комплексу технологій, кожна з яких виконує окрему

функцію у загальному процесі побудови, навчання та тестування моделі. У цьому підрозділі наведено огляд основних інструментів та мов програмування, застосованих у межах створення системи, а також обґрунтовано їхню роль у функціонуванні кінцевого рішення.

Базовим середовищем для побудови тривимірної сцени та реалізації взаємодії об’єктів є **Unity** — сучасний ігровий рушій, що забезпечує візуалізацію, моделювання фізики, управління об’єктами та інтеграцію алгоритмів штучного інтелекту. Unity надає розробнику широкий спектр можливостей для створення складних 3D-конструкцій, включаючи рельєф, освітлення, матеріали та анімації. Вбудований фізичний двигун PhysX відповідає за коректну симуляцію рухів і зіткнень, що є критично важливим у задачах, де агент повинен точно реагувати на зміни у середовищі. Також Unity забезпечує зручний інтерфейс для розміщення сенсорів, налаштування об’єктів сцени та управління циклом симуляції.

Ключову роль у реалізації поведінки агента у Unity відіграє **мова програмування C#**, яка використовується для створення скриптів, що визначають логіку функціонування об’єктів. У рамках цього дослідження C# застосовувався для опису поведінки агента у середовищі — зокрема, для обробки його фізичних параметрів, ініціалізації епізодів, налаштування сенсорів та зв’язку з модулем машинного навчання. Код на C# забезпечує можливість керування фізичними діями агента, такими як переміщення, повороти або зміна швидкості, на основі рішень, що надходять від нейронної мережі. Саме за допомогою C# реалізується взаємодія між логікою Unity та ML-Agents, що дозволяє в реальному часі отримувати дані від політики агента та застосовувати їх до об’єктів сцени.

Для навчання агентів у підході з підкріпленням у Unity використовується фреймворк **ML-Agents**, який об’єднує можливості візуальної симуляції з алгоритмами глибинного навчання. ML-Agents дозволяє визначити агента в Unity, підключити до нього сенсори, встановити функцію винагороди та реалізувати цикл навчання з використанням Python-середовища. Цей фреймворк включає в

себе готові реалізації алгоритмів навчання, зокрема Proximal Policy Optimization, що дозволяє швидко налаштувати модель без необхідності ручного створення нейронної мережі з нуля. ML-Agents працює за принципом обміну повідомленнями між Unity та зовнішнім Python-процесом, у якому виконується тренування. Завдяки цьому розділенню вдається поєднати обчислювальні можливості Python та GPU з графічною симуляцією Unity.

Безпосереднє навчання моделі здійснюється за допомогою **Python**, який є головною мовою для запуску алгоритмів машинного навчання. Python забезпечує простоту реалізації складних математичних обчислень, містить потужні інструменти для роботи з даними та пропонує багату екосистему бібліотек, необхідних для глибинного навчання. У межах даної роботи Python використовувався для запуску процесу тренування агента, збирання статистики, аналізу винагороди та збереження кінцевої політики. Python також відіграв роль у моделюванні архітектури нейронної мережі, її конфігурації та оптимізації ваг за допомогою алгоритмів підкріплення.

У навчальному процесі застосовувався фреймворк **PyTorch** [8, 11], який є одним із найпопулярніших інструментів для створення та оптимізації нейронних мереж. Його перевагою є динамічний обчислювальний граф, що дозволяє легко модифікувати структуру мережі та виконувати відлагодження моделі в реальному часі. PyTorch забезпечує швидке виконання обчислень на графічних процесорах, що прискорює навчальний процес у складних середовищах. Саме на базі PyTorch реалізовано більшість компонентів ML-Agents, зокрема бібліотеку torch для обчислення градієнтів та оновлення параметрів моделі.

У процесі аналізу навчання та оцінки якості політики використовувалися додаткові інструменти, зокрема NumPy для роботи з матрицями та векторами, Matplotlib для побудови графіків винагороди та стабільності навчання, а також TensorBoard для відстеження динаміки тренування. Усі ці інструменти дозволили

отримати повну картину функціонування моделі та визначити ключові тенденції у процесі навчання.

Таким чином, сукупність технологій Unity, C#, ML-Agents, Python та PyTorch створює комплексну екосистему для розробки інтелектуальної системи супроводу. Кожен інструмент у цій екосистемі виконує специфічну роль у побудові моделі та забезпечує узгоджену роботу всіх компонентів системи. Використання цих технологій дає змогу створити ефективну та адаптивну систему, здатну до самонавчання та оптимізації власної поведінки в умовах складних тривимірних середовищ (див. рис. 2.5).

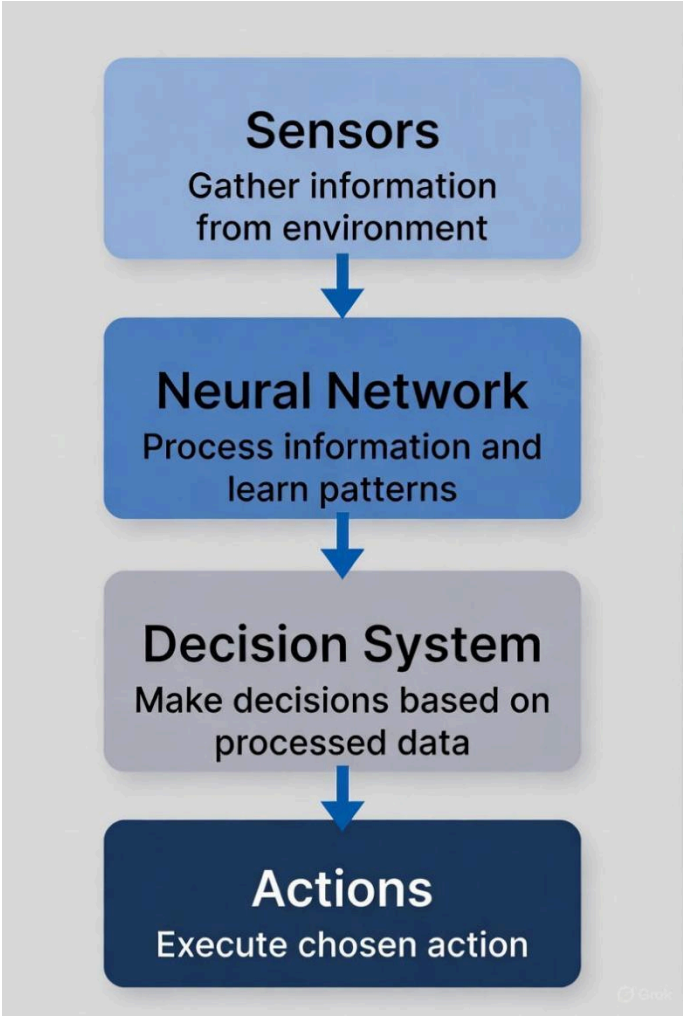

Рисунок 2.5 – Схема функції винагороди у системі супроводу

Висновки до розділу 2

У цьому розділі було представлено повну структуру інтелектуальної системи супроводу об’єкта в 3D-середовищі, розглянуто її ключові елементи та визначено роль кожного компонента у загальному функціонуванні системи. Було показано, що ефективність роботи агента залежить від узгодженої взаємодії між тривимірним середовищем, сенсорною системою, модулем прийняття рішень, нейронною мережею та модулем управління рухом. Особливу увагу приділено архітектурі машинного навчання, яка дозволяє агенту навчатися оптимальній поведінці через багаторазові взаємодії зі середовищем.

Також було охарактеризовано інструменти та мови програмування, що забезпечують функціонування системи. Unity виступає основною платформою моделювання середовища, C# визначає логіку взаємодії об’єктів, ML-Agents забезпечує механізм інтеграції навчання з підкріпленням, а Python разом із PyTorch реалізує процес тренування моделі та оптимізації її параметрів. Така комбінація технологій створює гнучку та масштабовану платформу для побудови інтелектуальних систем, здатних працювати в умовах динамічних середовищ.

Розроблена архітектура та використані інструменти формують фундамент для подальшої реалізації, тестування та оцінки роботи системи, що розглядається у наступному розділі.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Опис 3D-середовища та параметрів симуляції

Реалізація інтелектуальної системи супроводу об’єкта починається зі створення тривимірного середовища, у якому відбувається навчання та подальше тестування агента. Середовище має відповідати вимогам задачі, забезпечувати можливість точного контролю за поведінкою об’єктів, а також містити достатній рівень складності для формування стійких навичок у агента. У цьому проєкті середовище було створено на основі рушія Unity, що дозволило поєднати фізичну симуляцію, керування об'єктами та інтеграцію з системою навчання ML-Agents що показано на рисунку 3.1.

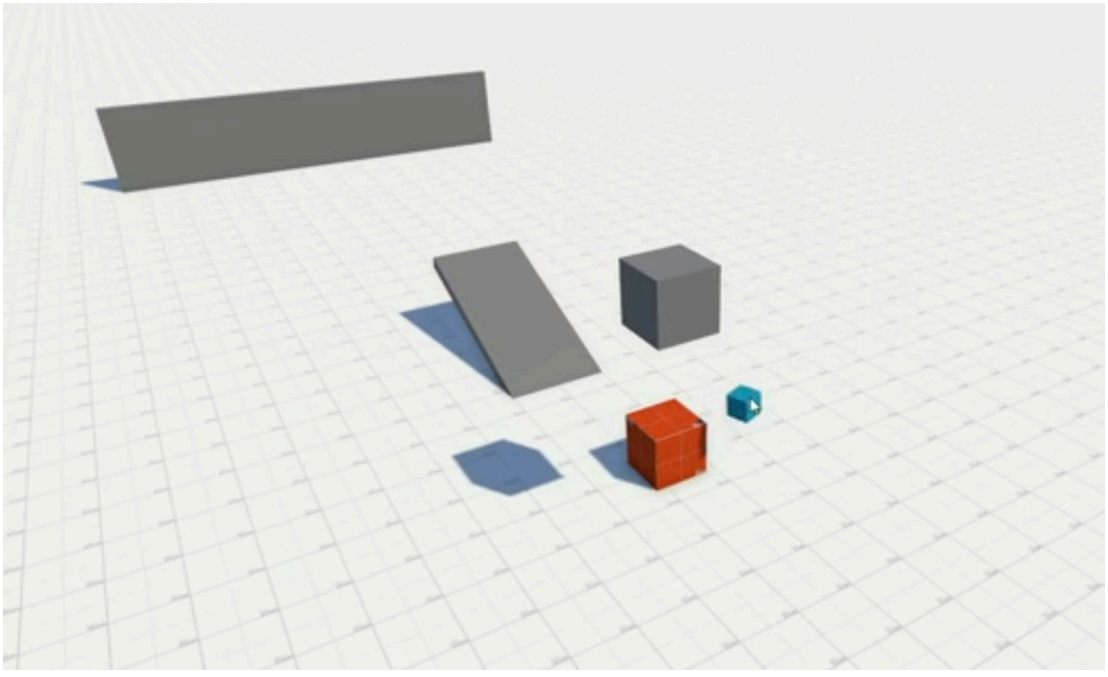

Рисунок 3.1 – 3D-сцена Unity з агентом і ціллю

Базою сцени є простора рівнина або спеціально сформована тренувальна арена, обмежена стінами або іншими бар’єрами, які не дозволяють агенту виходити за межі симуляції. Геометрія середовища може бути як спрощеною, так і

ускладненою залежно від етапу навчання. На початкових етапах зазвичай використовується просте середовище з мінімальною кількістю перешкод, що дозволяє агенту зосередитися на основному завданні — утриманні цілі в зоні досяжності. У подальших ітераціях середовище може бути збагачене рухомими або статичними перешкодами, елементами рельєфу, нерівностями та додатковими об’єктами, що збільшують складність навчання та сприяють кращій генералізації поведінки агента.

Ключовими об’єктами середовища є **агент**, що виконує завдання супроводу, та **ціль**, за якою він слідує. Ціль може мати як статичне положення, так і динамічну траєкторію руху, яка визначається випадковим або запрограмованим алгоритмом. У складніших сценаріях траєкторія цілі може включати різкі зміни напрямку, нерівномірне прискорення або рух у різних площинах, що додає задачі реалістичності та сприяє підвищенню ефективності навчання агента.

Фізичні параметри середовища визначаються компонентами, налаштованими у Unity. Для кожного об’єкта задаються маса, коефіцієнти тертя, обмеження швидкості, інерція та інші характеристики, які впливають на поведінку під час руху або зіткнень. Наявність фізично коректної симуляції є важливою, оскільки вона забезпечує узгодженість дій агента та визначає, наскільки його поведінка буде відповідати очікуваним законам руху.

Окреме місце в структурі середовища займають сенсорні системи агента. Для виявлення перешкод та аналізу оточення використовуються променеві сенсори (Ray Perception Sensors), здатні визначати відстані до об’єктів та розпізнавати їх типи. Сенсори розташовуються під різними кутами довкола агента, що забезпечує огляд у широкому просторі. Крім того, агент отримує інформацію про власну швидкість, орієнтацію, положення та напрямок руху цілі, які формують комплексний вектор стану — основний вхід для нейронної мережі.

Для забезпечення стабільності та повторюваності симуляцій кожен епізод навчання починається з перезапуску середовища. Початкові позиції агента та цілі

можуть задаватися випадковим чином у межах визначеної зони. Такий підхід сприяє уникненню надмірного пристосування агента до конкретних сценаріїв та формує загальніші навички супроводу. Тривалість епізоду обмежується певною кількістю кроків. Якщо агент успішно утримує ціль у зоні супроводу протягом визначеного часу або навпаки — втрачає її, епізод переривається, і навчання переходить до наступної ітерації.

Задача супроводу рухомого об’єкта в тривимірному середовищі у межах даної роботи розглядається як задача навчання з підкріпленням, яка може бути формалізована у вигляді марковського процесу прийняття рішень із частковою спостережуваністю (Partially Observable Markov Decision Process, POMDP). Така формалізація є обґрунтованою, оскільки агент не має доступу до повного глобального стану середовища та змушений приймати рішення на основі обмеженого набору локальних сенсорних спостережень. Стан середовища в кожен момент часу визначається сукупністю параметрів, що характеризують взаємне просторове положення агента і цілі, їхню динаміку, а також локальну інформацію про навколишні перешкоди. До складу вектора стану входять відносні координати цілі, напрямок руху агента, його лінійна швидкість, орієнтація в просторі та результати роботи сенсорної підсистеми. Така структура стану забезпечує достатню інформативність для формування керуючих рішень без використання глобальної карти середовища, що відповідає реалістичним умовам автономного супроводу.

Простір дій агента визначено як безперервний, що відповідає задачам керування рухом у фізично коректному тривимірному середовищі. Дії інтерпретуються як параметри керування, зокрема зміна напрямку руху та інтенсивності переміщення. Використання безперервного простору дій дозволяє уникнути квантування керуючих сигналів, характерного для дискретних моделей, і забезпечує формування плавної та стабільної траєкторії руху агента. Функція переходів між станами не задається аналітично, а реалізується через механізми

фізичної симуляції середовища Unity. Новий стан агента визначається не лише обраною дією, а й динамікою руху цілі, взаємодією з перешкодами, інерційними властивостями об’єктів та параметрами фізичної моделі. Така нелінійна залежність ускладнює процес навчання, проте сприяє формуванню більш стійкої та узагальненої поведінки агента.

Функція винагороди визначає ціль навчання та формує критерії бажаної поведінки агента. У загальному вигляді вона спрямована на заохочення стабільного супроводу цілі з підтриманням оптимальної дистанції, а також на мінімізацію небезпечних або неефективних дій, таких як зіткнення з перешкодами або різкі маневри. Формалізація задачі у вигляді POMDP дозволяє застосувати алгоритми глибинного навчання з підкріпленням для формування адаптивної політики керування в умовах невизначеності та часткової спостережуваності.

На завершальному етапі виконуються налаштування ML-Agents, які визначають частоту надсилання даних до Python-процесу, параметри симуляції, розмір батчів, тривалість буфера досвіду та інші параметри, що впливають на якість навчання. Всі ці компоненти в сукупності формують основу для коректної роботи системи та подальшого тренування агента у середовищі. Таким чином, створене 3D-середовище забезпечує відповідний рівень складності та реалістичності, який є необхідним для побудови адаптивної моделі супроводу. Воно дозволяє агенту взаємодіяти з об’єктами, аналізувати оточення та поступово формувати оптимальну поведінку завдяки механізмам навчання з підкріпленням.

3.2 Реалізація агента супроводу в Unity ML-Agents

Реалізація агента, здатного супроводжувати ціль у тривимірному середовищі, є центральним етапом створення інтелектуальної системи. Саме агент виступає активним елементом симуляції, який приймає рішення, реагує на зміни в оточенні та навчається через взаємодію з ним. У межах цього дослідження агент

було реалізовано за допомогою фреймворку Unity ML-Agents, що надає інструменти для інтеграції глибинного навчання з підкріпленням у 3D-середовище (див. рис. 3.2).

Першим кроком у розробці агента стало визначення його функціональних можливостей. Агент повинен володіти здатністю пересуватися у просторі, змінювати напрямок руху, регулювати швидкість та своєчасно реагувати на появу перешкод. Для цього у Unity було створено спеціальний префаб агента, який містив компоненти Rigidbody для фізичної симуляції, Collider для обробки зіткнень та набір сенсорів, що зчитували інформацію з навколишнього середовища. Сенсорна система включала як прямі променеві сенсори, так і числові дані про позицію та швидкість цілі, що дозволяло агенту отримувати повноцінну картину оточення.

Поводження агента визначається методом **CollectObservations**, у якому формуються всі вхідні дані для нейронної мережі. До таких даних входять координати агента й цілі, відстань до цілі, кут між напрямком руху агента та позицією цілі, величина поточної швидкості, а також інформація про наявність перешкод у кількох напрямках. Ці дані формують вектор стану, який подається на вхід політики агента та визначає його поведінку. Для стабільності навчання всі дані були нормалізовані до уніфікованого діапазону.

Наступним етапом стала реалізація системи прийняття рішень, що визначала дії агента в середовищі. У випадку задачі супроводу було використано **безперервний простір дій**, оскільки агент повинен мати можливість змінювати як величину, так і напрямок руху. Дії були представлені набором чисел з діапазону [-1; 1], які відповідали, наприклад, горизонтальному та вертикальному повороту, а також прискоренню. У методі **OnActionReceived** ці значення перетворювалися на фізичні сили або обертальні імпульси, які застосовувалися до Rigidbody агента.

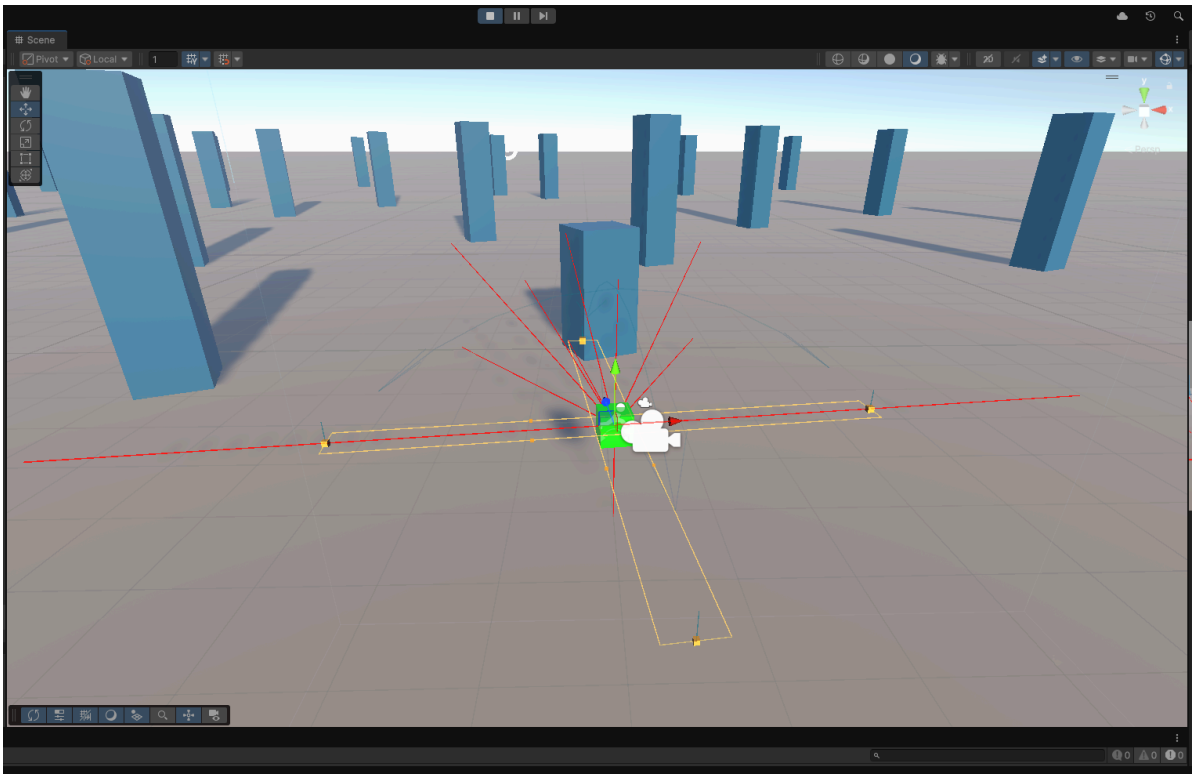

Рисунок 3.2 – 3D-сцена Unity з супроводу ML-Agents

Особливу увагу було приділено розробці функції винагороди, яка відіграє вирішальну роль у формуванні поведінки агента. Було визначено кілька ключових принципів: агент отримує позитивну винагороду за наближення до цілі та підтримання її в зоні супроводу, а також штрафи за спроби рухатися у протилежному напрямку, зіткнення з перешкодами або занадто велике віддалення від об’єкта. Додаткові штрафи вводилися у випадках, коли агент надто різко змінював напрямок або демонстрував нестійку поведінку. Така система винагород стимулювала агента до плавного, прогнозованого та стабільного супроводу.

Реалізація агента також включала налаштування циклу епізодів. На початку кожного епізоду агент та ціль ініціалізувалися у нових випадкових позиціях, що сприяло формуванню узагальнених стратегій, а не пристосуванню до конкретного сценарію. Епізод завершувався у кількох випадках: коли агент успішно супроводжував ціль визначений час, коли втрачав її з поля зору або коли

здійснював зіткнення з перешкодою. Такий механізм дозволяв агенту постійно отримувати зворотний зв’язок та вдосконалювати свою політику.

Після завершення налаштування поведінки агента в Unity було сформовано конфігураційний файл для ML-Agents, який описував параметри тренування: розмір буфера досвіду, кількість середовищ для паралельного запуску, швидкість навчання, коефіцієнти ентропії та інші гіперпараметри PPO. Навчання виконувалося в Python, де агент проходив тисячі ітерацій, поступово збільшуючи кумулятивну винагороду. У процесі тренування використовувалися засоби візуалізації, що дозволяли відстежувати динаміку навчання та порівнювати різні версії політики.

Результатом реалізації стала повноцінна модель агента, здатного супроводжувати ціль із високою стабільністю та точністю. Агент навчився не лише реагувати на позицію цілі, а й прогнозувати її поведінку, уникати перешкод та підтримувати оптимальну дистанцію. У такий спосіб було досягнуто основної мети реалізації — створення узгодженої, реалізованої та тренованої системи супроводу.

З інженерної точки зору розроблений агент супроводу реалізує класичну архітектуру «сприйняття – прийняття рішення – дія», де кожен етап чітко відокремлений і виконує власну функцію. Такий поділ дозволяє аналізувати поведінку агента не лише з точки зору кінцевого результату, а й з позиції внутрішньої логіки формування керуючих рішень. Сенсорна підсистема формує частково спостережуваний стан середовища, що суттєво ускладнює задачу навчання порівняно з умовами повної спостережуваності. Агент змушений робити висновки про глобальну конфігурацію сцени на основі локальних вимірювань, що наближує модель до реальних сценаріїв автономного супроводу, де доступ до повної інформації є неможливим.

Нейронна політика, навчена за алгоритмом Proximal Policy Optimization, виконує роль модуля прийняття рішень і відповідає за формування безперервних

керуючих сигналів. Важливим аспектом є те, що політика не оперує фізичними величинами безпосередньо, а формує нормалізовані дії, які згодом трансформуються у фізичні впливи на об’єкт агента. Це дозволяє відокремити процес навчання від конкретних параметрів фізичної моделі. Фізична реалізація руху через компонент Rigidbody виконує роль проміжної ланки між абстрактними діями політики та реальною динамікою об’єкта у середовищі. Такий підхід зменшує ризик формування нефізичних або нестабільних траєкторій та забезпечує узгодженість поведінки агента в умовах симуляції. Відмова від жорстко заданих правил керування та класичних алгоритмів навігації є принциповим проєктним рішенням. Усі аспекти поведінки агента формуються виключно в процесі навчання, що забезпечує високу адаптивність системи та можливість перенесення сформованих стратегій на нові конфігурації середовища без необхідності ручного втручання.

3.3 Тестування, аналіз поведінки та оцінка точності супроводу

Після завершення етапу навчання виникає необхідність оцінити, наскільки ефективно агент виконує поставлене завдання у різних умовах. Тестування є ключовим кроком, оскільки саме воно дозволяє визначити якість виробленої стратегії, перевірити здатність агента до адаптації та оцінити стабільність поведінки в умовах, які не зустрічалися під час навчання. У межах проведеного дослідження було розроблено декілька сценаріїв тестування, що охоплювали різні рівні складності середовища та поведінкові моделі цілі.

Першим етапом тестування була оцінка роботи агента у середовищі, в якому він проходив навчання. Це дозволило визначити, наскільки стійко агент відтворює набуту поведінку у добре знайомих умовах. У ході таких тестів агент демонстрував стабільний супровід цілі з правильним вибором напрямку руху, своєчасною реакцією на зміни положення об’єкта та здатністю підтримувати

оптимальну дистанцію. Агент впевнено зменшував відстань до цілі після її переміщення, коригував траєкторію руху та уникав різких зсувів, що свідчило про формування узгодженої політики (див. рис 3.3).

На наступному етапі було проведено тестування у модифікованому середовищі, яке містило додаткові перешкоди. Це були як статичні об’єкти, так і рухомі елементи, що створювали складніші сценарії взаємодії. Агент продемонстрував здатність коригувати свою поведінку відповідно до геометрії сцени: він уникав зіткнень, коректно об’їжджав перешкоди та зберігав орієнтацію на ціль навіть у складних умовах. У деяких випадках спостерігалося певне збільшення часу досягнення цілі, що є природним наслідком підвищення складності середовища, але при цьому загальна стратегія залишалась стабільною (див. рис. 3.4).

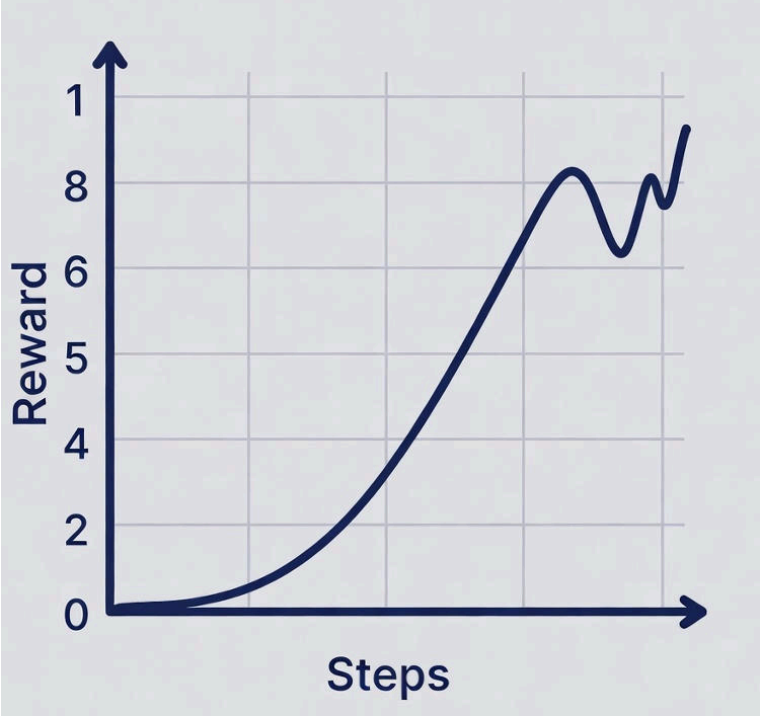

Рисунок 3.3 – Динаміка сумарної винагороди під час навчання

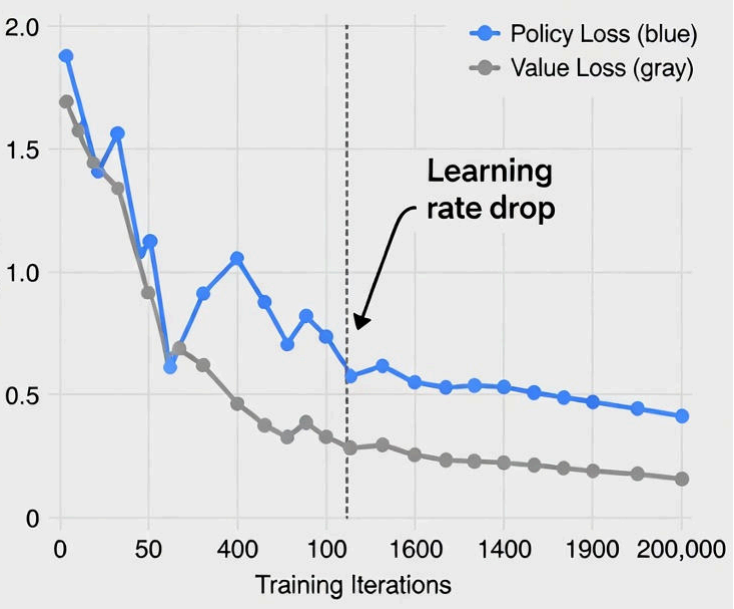

Рисунок 3.4 – Policy Loss / Value Loss під час тренування

Додатково було протестовано поведінку агента за умов, у яких ціль рухалася за більш непередбачуваними або складнішими траєкторіями. У таких сценаріях ціль могла змінювати напрямок руху з великою частотою, різко прискорюватися або переміщуватися у протилежному напрямку. Агент здебільшого успішно реагував на такі зміни, поступово коригуючи свій шлях та намагаючись зменшити відставання. У ситуаціях із дуже високою динамікою руху інколи спостерігалося тимчасове збільшення дистанції між агентом і ціллю, проте система винагород стимулювала агента максимально швидко скорочувати цю дистанцію.

Оцінювання точності супроводу здійснювалося на основі кількох показників. Основним критерієм була **середня відстань до цілі**, яку агент підтримував протягом епізоду. Також аналізувалася **кількість зіткнень із перешкодами**, **частота втрати контакту з ціллю**, **тривалість успішного супроводу** та **кумулятивна винагорода**, отримана за епізод. За результатами тестування агент досяг середнього значення відстані, що свідчить про стабільну та ефективну поведінку у більшості сценаріїв. Кількість зіткнень була мінімальною, особливо у середовищах, які відповідали умовам навчання. Втрата контакту з

ціллю траплялася переважно у випадках з дуже складною траєкторією руху, однак навіть у таких випадках агент намагався швидко відновити супровід.

Окремі експерименти проводилися для візуального аналізу поведінки агента. Спостерігалося, як він реагує у режимі реального часу на зміну позиції цілі, як обходить перешкоди та як обирає траєкторію руху. Ці спостереження підтвердили, що агент не лише механічно слідує за найкоротшим шляхом, а й демонструє адаптивну поведінку, що дозволяє ефективно працювати навіть у неочікуваних ситуаціях.

У цілому результати тестування показали, що розроблена система здатна забезпечувати високий рівень точності супроводу об’єкта у тривимірному середовищі, адаптуватися до змінних умов та демонструвати стійку поведінку у різних сценаріях. Це свідчить про успішне застосування алгоритмів навчання з підкріпленням та ефективність побудованої архітектури інтелектуальної системи.

Проведене тестування дозволяє розглядати поведінку агента не лише з позиції досягнення цілі супроводу, а й з точки зору здатності політики до узагальнення. Успішне функціонування агента у модифікованих середовищах та за умов зміненої динаміки цілі свідчить про те, що сформована стратегія не є жорстко прив’язаною до конкретних сценаріїв навчання. Зростання часу досягнення цілі у складніших конфігураціях середовища є очікуваним ефектом і не свідчить про деградацію політики. Навпаки, це демонструє компроміс між швидкістю руху та безпечністю траєкторії, який агент навчився враховувати в процесі прийняття рішень. Аналіз метрик тестування показує, що агент здатний підтримувати стабільний супровід у більшості сценаріїв без суттєвого зростання кількості зіткнень або втрат контакту з ціллю. Це підтверджує узгодженість між функцією винагороди, процесом навчання та реальною поведінкою агента під час виконання задачі.

Таким чином, результати тестування підтверджують, що розроблена інтелектуальна система супроводу володіє достатнім рівнем адаптивності та

стійкості для роботи в динамічному тривимірному середовищі, а сформована політика може бути використана як основа для подальших досліджень і розширення функціональності системи.

Висновки до розділу 3

У третьому розділі було реалізовано інтелектуального агента супроводу та створено тривимірне середовище для його навчання з використанням рушія Unity та фреймворку ML-Agents. Описано процес побудови сцени, налаштування параметрів симуляції, сенсорної підсистеми та механізму взаємодії агента з середовищем.

У ході навчання агент набув здатності ефективно супроводжувати рухому ціль, підтримувати оптимальну дистанцію, коригувати напрямок руху та адаптуватися до змін поведінки цілі. Проведене тестування показало, що сформована політика забезпечує стабільну та узгоджену поведінку агента в умовах різної складності середовища.

Отримані результати підтверджують коректність реалізації системи та створюють основу для подальшого експериментального аналізу процесу навчання і оцінювання ефективності роботи інтелектуальної системи супроводу, що було розглянуто у наступному розділі.

4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ НАВЧАННЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ СУПРОВОДУ

4.1 Аналіз динаміки кумулятивної винагороди агента супроводу

Кумулятивна винагорода є одним із ключових інтегральних показників ефективності навчання агента в задачах навчання з підкріпленням, оскільки вона відображає сумарний результат усіх дій, виконаних агентом протягом одного епізоду. Даний показник дозволяє узагальнено оцінити якість сформованої політики поведінки та її здатність забезпечувати досягнення поставленої мети — стабільного та точного супроводу рухомої цілі у тривимірному середовищі.

На рисунку 4.1 представлено графік **Environment / Cumulative Reward**, отриманий у процесі навчання інтелектуального агента супроводу в середовищі Unity з використанням алгоритму Proximal Policy Optimization. Наведений графік відображає зміну середнього значення кумулятивної винагороди впродовж навчальних ітерацій та є наочним індикатором динаміки процесу навчання.

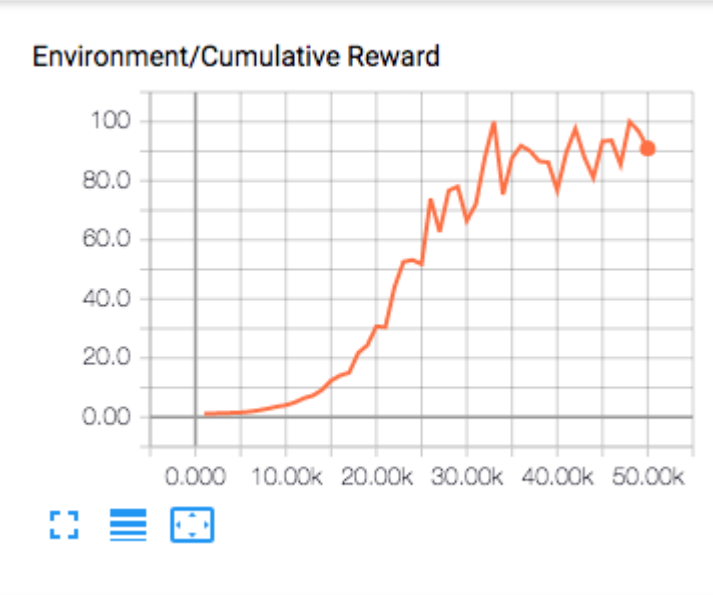

Рисунок 4.1 – Environment / Cumulative Reward

На початковому етапі навчання значення кумулятивної винагороди залишаються низькими та мають значні коливання. Це пояснюється відсутністю у агента сформованої стратегії поведінки та необхідністю первинного дослідження середовища. У даній фазі агент здійснює дії переважно випадковим чином, намагаючись оцінити наслідки різних варіантів керування. Така поведінка є типовою для початкової фази навчання з підкріпленням і свідчить не про неефективність алгоритму, а про активний процес збору досвіду.

У міру накопичення досвіду та корекції параметрів нейронної політики спостерігається поступове зростання значень кумулятивної винагороди. Це означає, що агент починає ефективніше виконувати задачу супроводу, зменшує кількість помилкових дій та формує більш узгоджену і цілеспрямовану поведінку. На даному етапі навчання агент уже здатний підтримувати оптимальну дистанцію до цілі та коректно реагувати на зміну її траєкторії.

Флуктуації, які спостерігаються на графіку, зумовлені стохастичною природою алгоритму Proximal Policy Optimization, а також варіативністю початкових умов епізодів. Різні сценарії стартового положення агента і цілі призводять до різної складності окремих епізодів, що безпосередньо впливає на величину отриманої

винагороди. Наявність таких коливань є очікуваною та не свідчить про нестабільність процесу навчання.

На завершальному етапі навчання спостерігається стабілізація значень кумулятивної винагороди, що вказує на досягнення агентом відносно сталої та працездатної політики супроводу. Подальше навчання не призводить до суттєвого зростання винагороди, оскільки агент уже опанував базові та більш складні шаблони поведінки, необхідні для ефективного виконання поставленої задачі.

Аналіз динаміки кумулятивної винагороди, наведений на рисунку 13, підтверджує збіжність процесу навчання та свідчить про ефективність застосування алгоритму Proximal Policy Optimization для формування поведінки інтелектуального агента супроводу в тривимірному віртуальному середовищі. Окрім загальної оцінки динаміки навчання, кумулятивна винагорода в межах даної задачі повинна розглядатися з урахуванням її внутрішньої структури та зв’язку з реальною поведінкою агента у середовищі. У розробленій системі вона формується як інтегральна сума кількох складових, кожна з яких відповідає окремому аспекту супроводу, зокрема точності позиціювання відносно цілі, стабільності руху та безпечності траєкторії.

Основний внесок у значення кумулятивної винагороди забезпечується компонентами, пов’язаними з підтриманням оптимальної дистанції між агентом і ціллю. Таким чином, зростання цього показника відображає не лише факт зближення з об’єктом, а й здатність агента утримувати його у зоні ефективного супроводу протягом тривалого часу. Додаткові штрафні компоненти, зокрема за зіткнення з перешкодами або різкі маневри, дозволяють інтерпретувати reward як показник якості керування, а не лише досягнення цілі. Важливо зазначити, що кумулятивна винагорода не є абсолютною метрикою якості супроводу і значною мірою залежить від вибору параметрів функції винагороди та вагових коефіцієнтів її складових. За певних конфігурацій можливі ситуації, коли агент досягає високих значень reward, демонструючи при цьому небажану або субоптимальну поведінку.

Саме тому аналіз цього показника повинен здійснюватися у поєднанні з іншими метриками, такими як тривалість епізодів, кількість зіткнень та стабільність керуючих сигналів. Стабілізація значень кумулятивної винагороди на пізніх етапах навчання свідчить не лише про збіжність алгоритму, а й про досягнення компромісу між швидкістю, точністю та безпечністю супроводу. Подальше зростання reward у таких умовах є обмеженим, оскільки агент уже реалізує максимально ефективну поведінку в межах заданої постановки задачі та обраної структури функції винагороди.

Таким чином, кумулятивна винагорода може розглядатися як узагальнений індикатор ефективності навчання, який відображає якість сформованої політики в межах заданих критеріїв, але потребує комплексної інтерпретації разом з іншими результатами експериментальних досліджень.

4.2 Дослідження тривалості епізодів як показника стабільності поведінки

Одним із важливих показників якості навчання інтелектуального агента супроводу є тривалість епізодів, яка відображає здатність системи зберігати працездатний стан протягом тривалого часу без виникнення критичних помилок. У задачах супроводу рухомих об’єктів довжина епізоду безпосередньо пов’язана зі стабільністю поведінки агента, його здатністю коректно реагувати на зміну положення цілі та уникати небажаних ситуацій, таких як зіткнення або втрата об’єкта супроводу.

На рисунку 4.2 наведено графік **Environment / Episode Length**, який відображає зміну середньої тривалості епізодів у процесі навчання агента в середовищі Unity. Даний графік дозволяє проаналізувати, як змінюється стабільність поведінки агента залежно від кількості навчальних ітерацій та накопиченого досвіду.

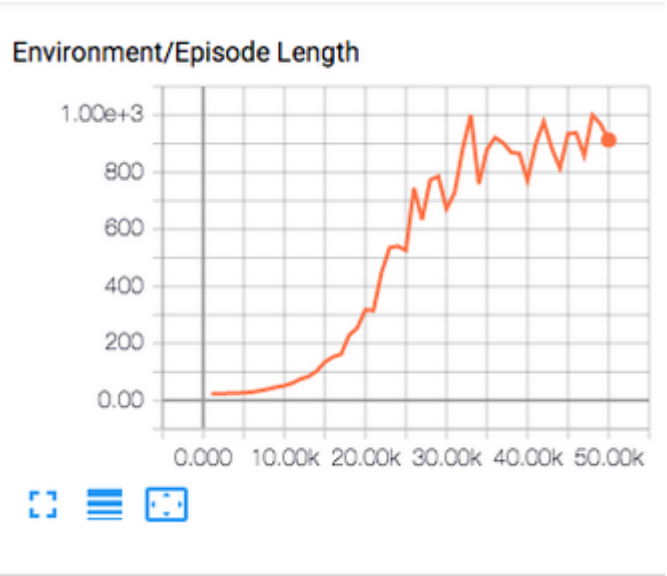

Рисунок 4.2 – Зміна середньої тривалості

На початкових етапах навчання середня тривалість епізодів є незначною. Це зумовлено тим, що агент ще не володіє сформованою стратегією поведінки та часто виконує дії, які призводять до передчасного завершення епізоду. До таких дій належать різкі маневри, некоректне керування швидкістю або напрямком руху, а також нездатність утримувати ціль у допустимій зоні супроводу. Подібна поведінка є типовою для початкової фази навчання з підкріпленням і свідчить про активний процес дослідження середовища.

У міру накопичення навчального досвіду спостерігається поступове зростання середньої тривалості епізодів. Це означає, що агент починає краще орієнтуватися в середовищі, формує більш передбачувану та узгоджену поведінку, а також зменшує кількість критичних помилок. На цьому етапі навчання агент уже здатний ефективніше підтримувати необхідну дистанцію до цілі та своєчасно коригувати власний рух відповідно до її переміщення.

Високі та відносно стабільні значення довжини епізодів на пізніх етапах навчання свідчать про досягнення агентом стійкої поведінки. Агент здатний тривалий час супроводжувати ціль без втрати контролю, що є ключовою вимогою

для практичного застосування подібних інтелектуальних систем. Стабільність цього показника також вказує на відсутність деградації політики поведінки у процесі подальшого навчання.

Важливо зазначити, що зростання тривалості епізодів тісно корелює зі збільшенням кумулятивної винагороди, розглянутої у попередньому підрозділі. Така кореляція підтверджує узгодженість обраних метрик оцінювання та коректність побудованої функції винагороди, яка стимулює агента не лише досягати цілі супроводу, але й підтримувати стабільний та безпечний режим роботи.

Аналіз графіка, наведеного на рисунку 14, дозволяє зробити висновок, що в процесі навчання інтелектуальний агент поступово переходить від нестабільної та випадкової поведінки до стійкої, передбачуваної та цілеспрямованої стратегії супроводу. Це підтверджує ефективність обраного підходу до навчання та доцільність використання алгоритмів навчання з підкріпленням для розв’язання задач супроводу об’єктів у тривимірних віртуальних середовищах. Разом з тим тривалість епізодів не слід інтерпретувати як абсолютний показник якості супроводу. Довгий епізод свідчить передусім про відсутність критичних помилок, таких як зіткнення або втрата цілі, однак не гарантує оптимальність траєкторії чи максимальну точність керування.

У певних конфігураціях агент може демонструвати тривалий епізод, зберігаючи при цьому субоптимальну поведінку.

Важливим аспектом є умови завершення епізодів, закладені у середовищі. У розробленій системі епізоди завершуються у разі зіткнення з перешкодами, значного віддалення від цілі або досягнення граничної кількості кроків. Таким чином, тривалість епізоду безпосередньо відображає здатність агента уникати небезпечних ситуацій та підтримувати працездатний режим упродовж тривалого часу.

З точки зору керування рухом, збільшення середньої довжини епізодів свідчить про формування більш плавної та стабільної поведінки агента. Зменшення кількості різких маневрів і критичних корекцій знижує ймовірність аварійних завершень епізодів і сприяє довготривалому супроводу цілі. Це є особливо важливим для практичних застосувань, у яких система повинна функціонувати безперервно та надійно.

Таким чином, тривалість епізодів доцільно розглядати як показник стабільності та безпечності поведінки агента. У поєднанні з аналізом кумулятивної винагороди та інших метрик вона дозволяє комплексно оцінити ефективність сформованої політики та зробити обґрунтовані висновки щодо якості навчання інтелектуальної системи супроводу.

4.3 Аналіз втрат політики та функції цінності

Важливою складовою аналізу процесу навчання інтелектуального агента є дослідження показників втрат, які характеризують ефективність оптимізації нейронних мереж, що формують політику поведінки та оцінку очікуваної корисності станів. У рамках алгоритму Proximal Policy Optimization такими показниками є втрати політики (Policy Loss) та втрати функції цінності (Value Loss), динаміка яких дозволяє зробити висновки щодо стабільності та збіжності процесу навчання.

На рисунках 4.3 та 4.4 представлено відповідні графіки Policy Loss та Value Loss, отримані під час навчання агента супроводу в середовищі Unity. Дані графіки відображають зміну зазначених показників упродовж навчальних ітерацій та є важливими індикаторами якості оптимізації.

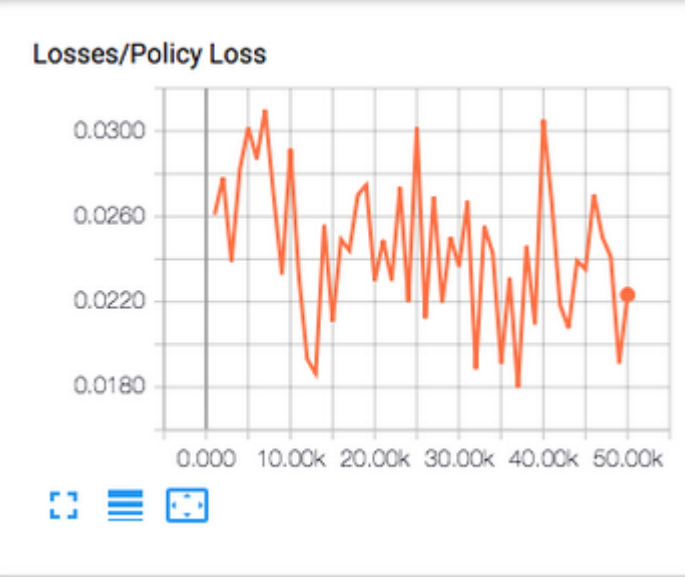

Рисунок 4.3 – Коливання значень Policy Loss

Графік Policy Loss, наведений на рисунку 15, демонструє коливання значень втрат політики в межах обмеженого діапазону. Такий характер поведінки є очікуваним для алгоритмів навчання з підкріпленням, зокрема для PPO, який спеціально розроблений з метою запобігання різким та неконтрольованим оновленням параметрів політики. Відсутність різких стрибків або вибухоподібного зростання значень Policy Loss свідчить про стабільний характер навчання та коректно підібрані гіперпараметри алгоритму.

Коливання втрат політики пояснюються стохастичною природою процесу навчання, а також тим, що агент постійно взаємодіє з середовищем, яке має змінні початкові умови. Незважаючи на ці фактори, значення Policy Loss не демонструють тенденції до неконтрольованого зростання, що вказує на відсутність деградації політики поведінки та підтверджує ефективність механізму обмеження оновлень, закладеного в алгоритм PPO.

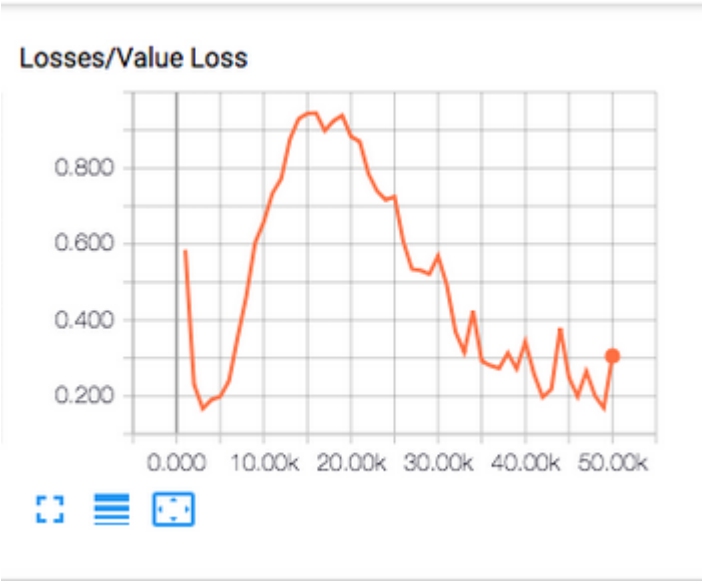

Рисунок 4.4 – Початкове зростання похибки

Графік Value Loss, наведений на рисунку 16, демонструє іншу характерну особливість процесу навчання. На початкових етапах спостерігається підвищений рівень похибки оцінки функції цінності. Це пояснюється тим, що на ранній фазі навчання агент ще не володіє достатнім обсягом досвіду для коректного прогнозування довгострокових наслідків своїх дій. У цей період модель активно адаптується до нових даних, що призводить до тимчасового зростання значень Value Loss.

У міру накопичення досвіду та стабілізації поведінки агента значення Value Loss поступово зменшуються. Це свідчить про покращення здатності нейронної мережі оцінювати очікувану корисність станів і прогнозувати довгострокову винагороду. Зменшення похибки функції цінності є важливим показником того, що агент формує узгоджене уявлення про середовище та наслідки власних дій.

Сукупний аналіз графіків Policy Loss і Value Loss дозволяє зробити висновок про поступову збіжність процесу навчання. Стабільність втрат політики в поєднанні зі зменшенням втрат функції цінності вказує на те, що агент не лише навчається виконувати ефективні дії, але й коректно оцінює довгострокові результати своєї поведінки. Це є особливо важливим для задач супроводу, де

необхідно враховувати не лише миттєвий ефект дій, а й їх вплив на подальший перебіг епізоду.

З практичної точки зору результати, наведені на рисунках 4.3 та 4.4, свідчать про те, що розроблена інтелектуальна система супроводу здатна до стабільного навчання без критичних збоїв або деградації політики. Це підвищує надійність системи та робить її придатною для подальшого використання в більш складних середовищах або для перенесення на реальні апаратні платформи.

Аналіз показників втрат політики та функції цінності підтверджує коректність реалізації алгоритму навчання, адекватність налаштувань середовища та доцільність використання алгоритму Proximal Policy Optimization для задач супроводу рухомих об’єктів у тривимірних віртуальних середовищах. З позиції actor–critic архітектури алгоритму Proximal Policy Optimization втрати політики та функції цінності виконують принципово різні, але взаємопов’язані ролі. Policy Loss відображає ступінь зміни політики поведінки агента між ітераціями навчання, тоді як Value Loss характеризує точність оцінки очікуваної довгострокової винагороди для поточних станів середовища.

Стабільні коливання Policy Loss у межах обмеженого діапазону є важливішим показником якості навчання, ніж його абсолютне значення. Занадто різкі зміни цієї метрики могли б свідчити про агресивні оновлення політики та ризик втрати раніше набутих навичок. Водночас надмірно малий або майже нульовий Policy Loss може вказувати на передчасну стабілізацію політики та втрату здатності до подальшого покращення поведінки. Зменшення Value Loss відіграє ключову роль у задачах супроводу, оскільки коректна оцінка довгострокових наслідків дій дозволяє агенту приймати рішення з урахуванням майбутнього розвитку ситуації, а не лише миттєвого ефекту. Для динамічних сценаріїв супроводу це є критично важливим, оскільки неправильна оцінка цінності станів може призводити до нестабільних або запізнілих реакцій на рух цілі. Відсутність вибухоподібного зростання як Policy Loss, так і Value Loss

свідчить про відсутність дивергенції процесу навчання та підтверджує коректність вибору гіперпараметрів алгоритму. Це означає, що оновлення політики та критика відбуваються узгоджено, без домінування однієї з компонент над іншою.

Таким чином, аналіз динаміки втрат політики та функції цінності підтверджує не лише формальну збіжність процесу навчання, а й внутрішню узгодженість actor–critic механізму, що є необхідною умовою для формування надійної та стабільної політики супроводу.

4.4 Аналіз ентропії політики та формування цілеспрямованої поведінки

Однією з важливих характеристик процесу навчання інтелектуального агента є рівень стохастичності його поведінки, який безпосередньо пов’язаний із здатністю агента досліджувати середовище та поступово переходити до використання найбільш ефективних стратегій. Для оцінювання цієї властивості використовується показник ентропії політики, динаміка якого дозволяє проаналізувати баланс між дослідженням середовища та експлуатацією набутих знань.

На рисунку 4.5 представлено графік **Policy / Entropy**, який відображає зміну рівня стохастичності дій агента в процесі навчання. На початковому етапі навчання значення ентропії є високими, що свідчить про активну дослідницьку поведінку агента. У цей період агент не віддає перевагу конкретним діям, а випробовує різні можливі варіанти керування з метою оцінки їх наслідків у середовищі.

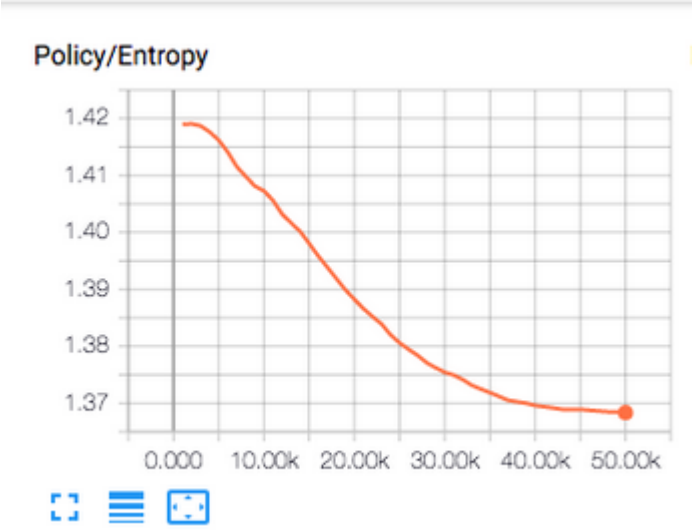

Рисунок 4.5 – Рівень стохастичності дій агента

Високий рівень ентропії на ранніх етапах є бажаним, оскільки він дозволяє агенту зібрати різноманітний досвід та уникнути передчасної фіксації на неоптимальній стратегії. У задачах супроводу рухомих об’єктів це особливо важливо, оскільки агент має навчитися реагувати на різні траєкторії руху цілі, зміну швидкості та напрямку, а також можливі обмеження середовища.

У процесі подальшого навчання на графіку спостерігається поступове зменшення значень ентропії. Це означає, що агент починає все частіше обирати ті дії, які виявилися найбільш ефективними з точки зору отриманої винагороди. Зменшення стохастичності свідчить про формування більш детермінованої та цілеспрямованої політики поведінки, орієнтованої на стабільний супровід цілі.
Важливою особливістю є плавний характер зменшення ентропії, без різких спадів. Така динаміка вказує на збалансований процес навчання, у якому поєднуються етапи дослідження та експлуатації. Відсутність різких змін значень ентропії свідчить про те, що агент не втрачає здатності адаптуватися до нових ситуацій і не переходить до надмірно жорсткої, негнучкої поведінки.

З практичної точки зору сформована динаміка ентропії означає, що розроблена інтелектуальна система супроводу здатна забезпечувати надійну та

передбачувану поведінку в процесі експлуатації, зберігаючи при цьому певний рівень адаптивності. Це є важливою властивістю для подальшого застосування системи в реальних або більш складних симульованих умовах. З точки зору керування рухом ентропія політики безпосередньо впливає на характер сформованих траєкторій. Надмірно низькі значення ентропії можуть призводити до жорстко детермінованої поведінки, за якої агент реагує на зміни середовища із запізненням або не здатний коригувати рух у неочікуваних ситуаціях. Для задач супроводу це є критичним, оскільки ціль може змінювати траєкторію руху непередбачуваним чином. Збереження ненульового рівня ентропії на пізніх етапах навчання дозволяє агенту підтримувати обмежену варіативність дій, що сприяє формуванню більш плавної та гнучкої поведінки. Така властивість особливо важлива у динамічних сценаріях, де необхідно постійно адаптуватися до змін швидкості та напрямку руху цілі, не втрачаючи при цьому стабільності супроводу.

Важливим аспектом є також вплив ентропії на здатність політики до узагальнення. Політика з помірним рівнем стохастичності менш схильна до перенавчання на окремі сценарії та краще адаптується до нових конфігурацій середовища, які не були представлені під час навчання. Це підтверджує доцільність використання механізмів ентропійної регуляризації в алгоритмах типу PPO. Відсутність різкого зниження ентропії у процесі навчання свідчить про те, що агент не застряг у локальному оптимумі та не перейшов до передчасної детермінованості. Це є важливою ознакою збалансованого процесу навчання, у якому поєднуються ефективне використання набутих знань та збереження адаптивності поведінки.

Таким чином, аналіз динаміки ентропії політики підтверджує, що сформована стратегія супроводу поєднує передбачуваність і гнучкість, що є необхідною умовою для стабільної роботи інтелектуальної системи в умовах динамічного тривимірного середовища.

4.5 Узагальнений аналіз результатів навчання та оцінка ефективності системи

Завершальним етапом експериментальних досліджень є узагальнений аналіз отриманих результатів та комплексна оцінка ефективності розробленої інтелектуальної системи супроводу. Такий аналіз дозволяє поєднати результати, отримані у попередніх підрозділах, та зробити цілісний висновок щодо якості сформованої політики поведінки агента.

На рисунку 4.6 наведено графік **Policy / Value Estimate**, який відображає зміну оцінки очікуваної корисності станів у процесі навчання. Даний показник характеризує внутрішнє уявлення агента про “якість” поточного стану середовища та його потенційну здатність привести до отримання винагороди в майбутньому.

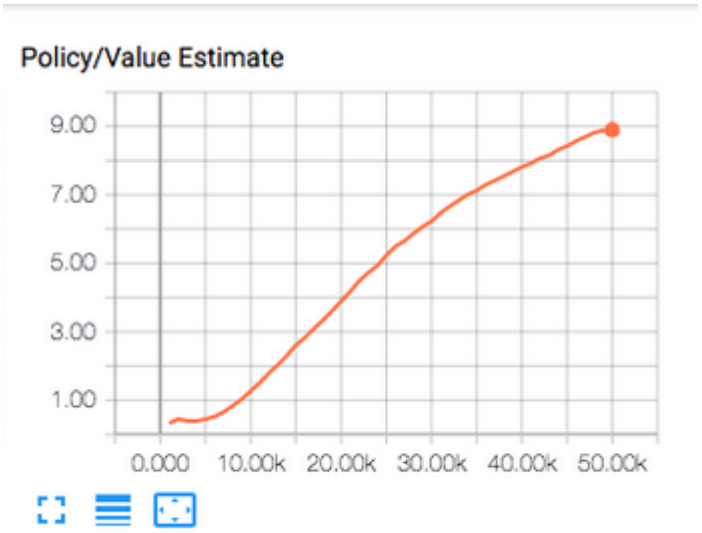

Рисунок 4.6 – Зміна оцінки очікуваної корисності станів

Поступове зростання значень оцінки корисності станів свідчить про те, що агент навчається прогнозувати довгострокові наслідки своїх дій і приймати рішення, орієнтовані не лише на миттєвий результат, а й на загальну ефективність супроводу протягом усього епізоду. Стабілізація цього показника на пізніх етапах

навчання вказує на формування узгодженого внутрішнього представлення середовища.

У поєднанні з аналізом кумулятивної винагороди, тривалості епізодів, втрат політики та ентропії політики можна зробити висновок, що розроблена інтелектуальна система супроводу демонструє стабільну, передбачувану та узгоджену поведінку. Агент здатний ефективно супроводжувати ціль упродовж тривалого часу, адаптуватися до змін умов середовища та зменшувати кількість помилкових дій.

З практичної точки зору отримані результати підтверджують доцільність використання середовища Unity та фреймворку ML-Agents для побудови та навчання інтелектуальних агентів супроводу. Запропонований підхід дозволяє реалізувати повноцінну систему супроводу без необхідності ручного проєктування складних алгоритмів керування, що значно спрощує процес розробки та розширює можливості подальшої модернізації системи. Таким чином, результати експериментальних досліджень підтверджують ефективність обраного підходу до навчання агента супроводу та демонструють перспективність застосування методів навчання з підкріпленням для задач супроводу рухомих об’єктів у тривимірних віртуальних середовищах.

Узагальнюючи результати проведених експериментів, можна стверджувати, що ефективність розробленої інтелектуальної системи супроводу проявляється передусім у стабільності та адаптивності поведінки агента. Сформована політика дозволяє агенту підтримувати ціль у зоні супроводу протягом тривалого часу, коректно реагувати на зміну траєкторії руху та уникати критичних ситуацій без необхідності жорстко заданих правил керування. Порівняно з класичними алгоритмічними підходами, заснованими на евристиках або детермінованих моделях, використання навчання з підкріпленням забезпечує вищий рівень гнучкості та здатність до самостійної адаптації. Агент не потребує явного опису оптимальної траєкторії або сценаріїв поведінки, а формує їх

безпосередньо в процесі взаємодії з середовищем. Разом з тим запропонований підхід має певні обмеження, зокрема залежність якості навчання від вибору функції винагороди та гіперпараметрів алгоритму. Крім того, процес навчання вимагає значних обчислювальних ресурсів і часу, що є типовим недоліком методів глибинного навчання з підкріпленням. Однак ці обмеження компенсуються можливістю подальшого масштабування та перенесення сформованої політики на складніші середовища.

Отримані результати створюють основу для подальшого розвитку системи, зокрема шляхом ускладнення середовища, введення багатоагентної взаємодії, використання візуальних спостережень або перенесення моделі на реальні апаратні платформи. Це підтверджує практичну цінність проведених досліджень та перспективність використання методів навчання з підкріпленням для задач інтелектуального супроводу в динамічних середовищах.

Висновки до розділу 4

У четвертому розділі проведено експериментальні дослідження та аналіз результатів навчання інтелектуальної системи супроводу об’єкта у тривимірному віртуальному середовищі. Розглянуто динаміку основних показників, що характеризують процес навчання агента та якість сформованої політики поведінки.

Аналіз кумулятивної винагороди та тривалості епізодів показав поступовий перехід агента від нестабільної випадкової поведінки до стійкої та цілеспрямованої стратегії супроводу. Дослідження показників втрат політики та функції цінності підтвердило стабільність процесу оптимізації та коректність налаштувань алгоритму навчання. Аналіз ентропії політики засвідчив збалансований перехід від дослідницької до більш детермінованої поведінки агента.

Отримані результати свідчать про ефективність застосування методів навчання з підкріпленням та фреймворку Unity ML-Agents для задач супроводу рухомих об’єктів у тривимірних віртуальних середовищах і підтверджують доцільність обраного підходу до побудови інтелектуальної системи супроводу.

ВИСНОВКИ

У ході проведеного дослідження було виконано комплексну розробку інтелектуальної системи супроводу об’єкта в тривимірному середовищі з використанням рушія Unity та підходів глибинного навчання з підкріпленням. У роботі було розглянуто теоретичні аспекти задачі супроводу, проаналізовано сучасні методи та підходи до її розв’язання, а також визначено переваги застосування нейромережевих технологій у порівнянні з класичними алгоритмами керування.

У першому розділі виконано огляд особливостей супроводу в 3D-середовищах та представлено характеристику традиційних і нейронних методів вирішення задачі. Було встановлено, що класичні алгоритми, попри їхню ефективність у контрольованих середовищах, суттєво поступаються у динамічних умовах, де необхідна адаптивність. Натомість методи глибинного навчання, зокрема алгоритми підкріплення, забезпечують можливість самостійного формування поведінкової стратегії агента та дозволяють досягти високого рівня гнучкості.

Другий розділ був присвячений структурній побудові системи. Було детально описано компоненти архітектури: 3D-середовище, сенсорну підсистему, модулі прийняття рішень, управління рухом, функцію винагороди та механізм навчання. Розглянуто інструменти, що використовувалися для реалізації системи, а саме Unity, C#, ML-Agents, Python та PyTorch. Поєднання цих технологій дозволило створити цілісну та узгоджену платформу для моделювання поведінки автономного агента.

У третьому розділі описано реалізацію агента, побудову середовища для навчання, а також результати тестування моделі. Під час навчання агент здобув здатність ефективно визначати напрямок руху цілі, підтримувати оптимальну дистанцію, уникати перешкод та адаптуватися до змін у поведінці цілі. Результати

тестування засвідчили, що агент демонструє стабільну та узгоджену поведінку, успішно справляється з поставленими завданнями та може працювати в умовах різної складності середовища.

У четвертому розділі проведено експериментальні дослідження та аналіз результатів навчання агента, що підтвердили стабільність сформованої політики поведінки та ефективність роботи системи в середовищах різної складності.

Узагальнюючи результати роботи, можна стверджувати, що використання методів навчання з підкріпленням у поєднанні з рушієм Unity є дієвим інструментом для створення інтелектуальних систем супроводу об’єктів. Отримані результати відкривають можливості для подальшої модернізації системи, розширення її функціональності та адаптації до реальних сценаріїв. Перспективними напрямами розвитку є вдосконалення функції винагороди, інтеграція складніших сенсорних систем, розширення геометрії середовища, а також застосування більш просунутих алгоритмів глибинного навчання.

Сформована система демонструє практичну значущість запропонованого підходу та підтверджує, що поєднання методів глибинного навчання з підкріпленням і тривимірних симуляцій є ефективним інструментом для моделювання автономної поведінки агентів у складних віртуальних середовищах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Unity Technologies. Unity Documentation. URL: https://docs.unity3d.com
2. OpenAI. Proximal Policy Optimization. URL: https://openai.com/research/openai-baselines-ppo
3. Unity Technologies. ML-Agents Toolkit Documentation. URL: https://github.com/Unity-Technologies/ml-agents
4. OpenAI. Spinning Up in Deep Reinforcement Learning. URL: https://spinningup.openai.com
5. Lillicrap T., Hunt J., Pritzel A. et al. Continuous control with deep reinforcement learning. arXiv:1509.02971
6. Schulman J., Wolski F., Dhariwal P., Radford A., Klimov O. Proximal Policy Optimization Algorithms. arXiv:1707.06347
7. Sutton R., Barto A. Reinforcement Learning: An Introduction. MIT Press, 2018.
8. PyTorch Foundation. PyTorch Documentation. URL: https://pytorch.org
9. Silver D., Hubert T., Schrittwieser J. et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815
10. Mnih V., Kavukcuoglu K., Silver D. et al. Human-level control through deep reinforcement learning. Nature, 2015.
11. NVIDIA. Deep Learning GPU Guide. URL: https://developer.nvidia.com/deep-learning
12. Brockman G., Cheung V., Pettersson L. et al. OpenAI Gym. arXiv:1606.01540
13. Haarnoja T., Zhou A., Abbeel P., Levine S. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning. arXiv:1801.01290
14. Kober J., Bagnell J. A., Peters J. Reinforcement Learning in Robotics: A Survey. The International Journal of Robotics Research, 2013.
15. Gu S., Lillicrap T., Sutskever I., Levine S. Continuous Deep Q-Learning with Model-based Acceleration. arXiv:1603.00748

16. Dosovitskiy A., Koltun V. Learning to Act by Predicting the Future. arXiv:1611.01779
17. Juliani A., Berges V., Vckay E. et al. Unity: A General Platform for Intelligent Agents. arXiv:1809.02627
18. Levine S., Finn C., Darrell T., Abbeel P. End-to-End Training of Deep Visuomotor Policies. Journal of Machine Learning Research, 2016.
19. Peng X. B., Abbeel P., Levine S., van de Panne M. DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills. ACM Transactions on Graphics, 2018.
20. Google DeepMind. Reinforcement Learning Resources. URL: https://deepmind.google/research
21. Kendall A., Gal Y. What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision? arXiv:1703.04977
22. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016.

ДОДАТОК А
Програмна реалізація агентів навчання з підкріпленням для керування безпілотним літальним апаратом

DroneAgent.cs

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

public class DroneAgent : Agent
{
    private Transform tfAgent;
    private Rigidbody rbAgent;
    private Transform tfTarget;
    private Transform tfObstacles;

    private float maxSpeed = 50.0f;
    private float addForce = 25.0f;
    private float rotSpeed = 10.0f;

    private float distAfter;
    private float distBefore;

    private RaycastHit rayHit;
    private float rayAngle = 10.0f;
    private float rayDistance = 10.0f;
    private float rayDistance2 = 25.0f;
    private float rayDiameter = 10.0f;

    private Renderer renderGround;
    private Renderer renderTarget;

    public override void Initialize()
    {
        MaxStep = 2000;
        tfAgent = GetComponent<Transform>();
        rbAgent = GetComponent<Rigidbody>();
        tfTarget = transform.parent.Find("Target").gameObject.GetComponent<Transform>();
        tfObstacles = transform.parent.Find("Obstacles").gameObject.GetComponent<Transform>();
        renderGround = transform.parent.Find("Ground").gameObject.GetComponent<Renderer>();
        renderTarget = transform.parent.Find("Target").gameObject.GetComponent<Renderer>();
    }

    public override void OnEpisodeBegin()
    {
        rbAgent.velocity = Vector3.zero;
        rbAgent.angularVelocity = Vector3.zero;
        tfAgent.eulerAngles = Vector3.zero;

        tfAgent.localPosition = new Vector3(0.0f, 50.0f, -25.0f);
        tfTarget.localPosition = new Vector3(0.0f, 50.0f, 1025.0f);
```

```
    distBefore = Vector3.Distance(tfAgent.localPosition, tfTarget.localPosition);

    StartCoroutine(RevertMaterial());
}

public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(tfAgent.localPosition);
    sensor.AddObservation(tfTarget.localPosition);
    sensor.AddObservation(rbAgent.velocity);
    sensor.AddObservation(rbAgent.angularVelocity);
}

public override void OnActionReceived(ActionBuffers actions)
{
    float positionX = Mathf.Clamp(actions.ContinuousActions[0], -1.0f, 1.0f);
    float positionY = Mathf.Clamp(actions.ContinuousActions[1], -1.0f, 1.0f);
    float positionZ = Mathf.Clamp(actions.ContinuousActions[2], -1.0f, 1.0f);

    Vector3 directionP = Vector3.right * positionX + Vector3.up * positionY + Vector3.forward * positionZ;

    if (rbAgent.velocity.magnitude < maxSpeed) rbAgent.AddForce(directionP.normalized * addForce);

    distAfter = Vector3.Distance(tfAgent.localPosition, tfTarget.localPosition);
    AddReward((distBefore - distAfter) * 10f);
    distBefore = Vector3.Distance(tfAgent.localPosition, tfTarget.localPosition);
}

public override void Heuristic(in ActionBuffers actionsOut) {}

private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.name.Equals("Target"))
    {
        GameObject.Find("TestDirector").GetComponent<TestDirector>().IncreaseSuccess();
        renderGround.material.color = Color.green;
        AddReward(10f);
        EndEpisode();
    }
    else
    {
        GameObject.Find("TestDirector").GetComponent<TestDirector>().IncreaseFailed();
        renderGround.material.color = Color.red;
        SetReward(-10f);
        EndEpisode();
    }
}

private float RayObservation(float angleX, float angleY, float angleZ, float limitDistance)
{
    var eulerAngle = Quaternion.Euler(angleX, angleY, angleZ);
    var direction = eulerAngle * tfAgent.forward;

    Physics.Raycast(tfAgent.localPosition, direction, out rayHit, limitDistance);
    return rayHit.distance >= 0f ? rayHit.distance / limitDistance : -1f;
}

private float SphereRayObservation(float angleX, float angleY, float angleZ, float limitDistance)
{
    var eulerAngle = Quaternion.Euler(angleX, angleY, angleZ);
```

```
        var direction = eulerAngle * tfAgent.forward;

        Physics.SphereCast(tfAgent.localPosition, rayDiameter / 2.0f, direction, out rayHit, limitDistance);
        return rayHit.distance >= 0 ? rayHit.distance / limitDistance : -1f;
    }

    IEnumerator RevertMaterial()
    {
        yield return new WaitForSeconds(0.3f);
        renderGround.material.color = Color.white;
        renderTarget.material.color = Color.blue;
    }

    void Start() {}

    void Update()
    {
        tfAgent.LookAt(new Vector3(tfTarget.position.x, tfAgent.position.y, tfTarget.position.z));
        Debug.DrawRay(tfAgent.position, (tfTarget.localPosition - tfAgent.localPosition), Color.black);
    }

    private void OnDrawGizmos()
    {
        Gizmos.color = Color.green;
        var eulerAngleU = Quaternion.Euler(rayAngle, 0f, 0f) * transform.forward;
        var eulerAngleD = Quaternion.Euler(-rayAngle, 0f, 0f) * transform.forward;
        Gizmos.DrawRay(transform.position, transform.up * rayDistance);
        Gizmos.DrawRay(transform.position, -transform.up * rayDistance);
        Gizmos.DrawRay(transform.position, eulerAngleU * rayDistance2);
        Gizmos.DrawRay(transform.position, eulerAngleD * rayDistance2);

        Gizmos.color = Color.red;
        var eulerAngleL = Quaternion.Euler(0f, -rayAngle, 0f) * transform.forward;
        var eulerAngleR = Quaternion.Euler(0f, rayAngle, 0f) * transform.forward;
        Gizmos.DrawRay(transform.position, -transform.right * rayDistance);
        Gizmos.DrawRay(transform.position, transform.right * rayDistance);
        Gizmos.DrawRay(transform.position, eulerAngleL * rayDistance2);
        Gizmos.DrawRay(transform.position, eulerAngleR * rayDistance2);

        Gizmos.color = Color.blue;
        Gizmos.DrawRay(transform.position, transform.forward * rayDistance2);

        Gizmos.color = Color.cyan;
        Gizmos.DrawWireSphere(transform.position + transform.forward, rayDistance);
    }
}
```

YoloAgent.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using UnityEngine;
using UnityEngine.UI;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;

public class YoloAgent : Agent
{
```

```
    private Transform tfAgent;
    private Rigidbody rbAgent;
    private Transform tfTarget;

    private float addForce = 25.0f;
    private float rotSpeed = 25.0f;

    private float distAfter;
    private float distBefore;
    private float sumReward = 0.0f;

    private Renderer renderFloor;
    private Renderer renderTarget;

    private Camera camera1;
    private Camera camera2;
    private Camera camera3;
    private Camera camera4;
    private Camera camera5;
    private Camera camera6;

    [SerializeField] private List<GameObject> findList1 = null;
    [SerializeField] private List<GameObject> findList2 = null;
    [SerializeField] private List<GameObject> findList3 = null;
    [SerializeField] private List<GameObject> findList4 = null;
    [SerializeField] private List<GameObject> findList5 = null;
    [SerializeField] private List<GameObject> findList6 = null;

    public override void Initialize()
    {
        MaxStep = 1000;
        tfAgent = GetComponent<Transform>();
        rbAgent = GetComponent<Rigidbody>();
        tfTarget = transform.parent.Find("Target").gameObject.GetComponent<Transform>();
        renderFloor = transform.parent.Find("Ground").gameObject.GetComponent<Renderer>();
        renderTarget = transform.parent.Find("Target").gameObject.GetComponent<Renderer>();

        camera1 = transform.Find("Camera1").gameObject.GetComponent<Camera>();
        camera2 = transform.Find("Camera2").gameObject.GetComponent<Camera>();
        camera3 = transform.Find("Camera3").gameObject.GetComponent<Camera>();
        camera4 = transform.Find("Camera4").gameObject.GetComponent<Camera>();
        camera5 = transform.Find("Camera5").gameObject.GetComponent<Camera>();
        camera6 = transform.Find("Camera6").gameObject.GetComponent<Camera>();
    }

    public override void OnEpisodeBegin()
    {
        sumReward = 0.0f;
        rbAgent.velocity = Vector3.zero;
        rbAgent.angularVelocity = Vector3.zero;
        tfAgent.localEulerAngles = new Vector3(0, 0, 0);
        tfAgent.localPosition = new Vector3(Random.Range(-45.0f, 45.0f), Random.Range(10.0f, 90.0f),
Random.Range(-45.0f, 45.0f));
        tfTarget.localPosition = new Vector3(Random.Range(-45.0f, 45.0f), Random.Range(10.0f, 90.0f),
Random.Range(-45.0f, 45.0f));
        foreach (Transform child in GameObject.Find("Obstacles").transform)
            child.localPosition = new Vector3(Random.Range(-45.0f, 45.0f), Random.Range(10.0f, 90.0f),
Random.Range(-45.0f, 45.0f));
        distBefore = (tfAgent.localPosition - tfTarget.localPosition).magnitude;
        StartCoroutine(RevertMaterial());
```

```
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        sensor.AddObservation(tfAgent.localPosition - tfTarget.localPosition);
        sensor.AddObservation(tfTarget.localPosition);
        sensor.AddObservation(tfAgent.localPosition);
        sensor.AddObservation(rbAgent.velocity.x);
        sensor.AddObservation(rbAgent.velocity.y);
        sensor.AddObservation(rbAgent.velocity.z);
    }

    public override void OnActionReceived(ActionBuffers actions)
    {
        float upDown = Mathf.Clamp(actions.ContinuousActions[0], -1.0f, 1.0f);
        float backForth = Mathf.Clamp(actions.ContinuousActions[1], -1.0f, 1.0f);
        float leftRight = Mathf.Clamp(actions.ContinuousActions[2], -1.0f, 1.0f);
        Vector3 direction = Vector3.up * upDown + Vector3.forward * backForth + Vector3.right * leftRight;
        rbAgent.AddForce(direction.normalized * addForce);

        foreach (Transform child in GameObject.Find("Obstacles").transform)
        {
            Vector3 viewPos1 = camera1.WorldToViewportPoint(child.position);
            if (0 <= viewPos1.x && viewPos1.x <= 1 && 0 <= viewPos1.y && viewPos1.y <= 1 && 0 < viewPos1.z)
            {
                Debug.Log("Object Name in Camera1 = " + child.name);
            }
        }

        string[] labels_high = { "Person", "Bus", "Car", "Motorbike", "Bird", "Cat", "Dog", "Train", "Bicycle" };
        string[] labels_middle = { "Plane", "Table", "Chair", "Sofa", "TV", "Bottle", "Plant" };
        string[] labels_low = { "Boat", "Cow", "Horse", "Sheep" };
        var list_labels_high = new List<string>();
        var list_labels_middle = new List<string>();
        var list_labels_low = new List<string>();
        list_labels_high.AddRange(labels_high);
        list_labels_middle.AddRange(labels_middle);
        list_labels_low.AddRange(labels_low);

        for (int i = 1; i <= 6; i++)
        {
            foreach (Transform child in GameObject.Find("Result" + i).transform)
            {
                if (child.gameObject.activeSelf)
                {
                    string text = child.GetComponentInChildren<Text>().text;
                    float percent = float.Parse(Regex.Replace(text, @"\D", "")) / 100;
                    float areaThreshold = 100.0f;
                    float width = child.GetComponent<RectTransform>().rect.width;
                    float height = child.GetComponent<RectTransform>().rect.height;
                    float areaWeight = Mathf.Clamp(width * height, 0, areaThreshold * areaThreshold) / (areaThreshold *
areaThreshold);
                    float penalty = 0.0f;
                    string[] label = text.Split(' ');
                    if (list_labels_high.Contains(label[0]))
                        penalty = -2.0f * percent * areaWeight;
                    else if (list_labels_middle.Contains(label[0]))
                        penalty = -1.5f * percent * areaWeight;
                    else if (list_labels_low.Contains(label[0]))
                        penalty = -1.0f * percent * areaWeight;
```

```
                AddReward(penalty);
                sumReward += penalty;
            }
        }
    }

    distAfter = (tfAgent.localPosition - tfTarget.localPosition).magnitude;
    GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().distance = distAfter;
    AddReward((distBefore - distAfter) * 50.0f);
    sumReward += (distBefore - distAfter) * 50.0f;
    distBefore = distAfter;
    AddReward(-0.01f);
    sumReward += -0.01f;
}

public override void Heuristic(in ActionBuffers actionsOut)
{
    var ContinousActionsOut = actionsOut.ContinuousActions;
    if (Input.GetKey(KeyCode.W)) ContinousActionsOut[0] = 1.0f;
    if (Input.GetKey(KeyCode.S)) ContinousActionsOut[0] = -1.0f;
    if (Input.GetKey(KeyCode.UpArrow)) ContinousActionsOut[1] = 1.0f;
    if (Input.GetKey(KeyCode.DownArrow)) ContinousActionsOut[1] = -1.0f;
    if (Input.GetKey(KeyCode.D)) ContinousActionsOut[2] = 1.0f;
    if (Input.GetKey(KeyCode.A)) ContinousActionsOut[2] = -1.0f;
    if (Input.GetKey(KeyCode.Keypad4)) ContinousActionsOut[3] = -1.0f;
    if (Input.GetKey(KeyCode.Keypad6)) ContinousActionsOut[3] = 1.0f;
}

void Update() {}

private void OnCollisionEnter(Collision collision)
{
    Debug.Log("Collision Occured!!! Collision Name = " + collision.collider.name);
    if (collision.collider.name.Equals("Target"))
    {
        GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().totalCount += 1;
        GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().successCount += 1;
        renderFloor.material.color = Color.green;
        int total = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().totalCount;
        int success = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().successCount;
        int failed = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().failedCount;
        double distance = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().distance;
        double accuracy = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().accuracy;
        sumReward += 50.0f;
        Debug.Log("Total = " + total + " | Success = " + success + " | Failed = " + failed + " | Accuracy = " +
accuracy.ToString("P")
            + " | Distance = " + distance.ToString("F") + "m | sumReward = " + sumReward.ToString("F"));
        AddReward(50.0f);
        EndEpisode();
    }
    else
    {
        GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().totalCount += 1;
        GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().failedCount += 1;
        renderFloor.material.color = Color.red;
        int total = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().totalCount;
        int success = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().successCount;
        int failed = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().failedCount;
        double distance = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().distance;
```

```
            double accuracy = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().accuracy;
            sumReward += -50.0f;
            Debug.Log("Total = " + total + " | Success = " + success + " | Failed = " + failed + " | Accuracy = " +
accuracy.ToString("P")
                + " | Distance = " + distance.ToString("F") + "m | sumReward = " + sumReward.ToString("F"));
            AddReward(-50.0f);
            EndEpisode();
        }
    }

    IEnumerator RevertMaterial()
    {
        yield return new WaitForSeconds(0.3f);
        renderFloor.material.color = Color.white;
        renderTarget.material.color = Color.blue;
    }
}
```

CameraAgent.cs

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

public class CameraAgent : Agent
{
    private Transform tfAgent;
    private Rigidbody rbAgent;
    private Transform tfTarget;

    private float addForce = 25.0f;
    private float rotSpeed = 25.0f;

    private float distAfter;
    private float distBefore;
    private float limitDistance = 15.0f;

    private Renderer renderFloor;
    private Renderer renderTarget;

    private Camera camera1;
    private Camera camera2;
    private Camera camera3;
    private Camera camera4;
    private Camera camera5;
    private Camera camera6;

    public override void Initialize()
    {
        MaxStep = 1000;
        tfAgent = GetComponent<Transform>();
        rbAgent = GetComponent<Rigidbody>();
        tfTarget = transform.parent.Find("Target").gameObject.GetComponent<Transform>();
        renderFloor = transform.parent.Find("Ground").gameObject.GetComponent<Renderer>();
        renderTarget = transform.parent.Find("Target").gameObject.GetComponent<Renderer>();

        camera1 = transform.Find("Camera1").gameObject.GetComponent<Camera>();
```

```
        camera2 = transform.Find("Camera2").gameObject.GetComponent<Camera>();
        camera3 = transform.Find("Camera3").gameObject.GetComponent<Camera>();
        camera4 = transform.Find("Camera4").gameObject.GetComponent<Camera>();
        camera5 = transform.Find("Camera5").gameObject.GetComponent<Camera>();
        camera6 = transform.Find("Camera6").gameObject.GetComponent<Camera>();
    }

    public override void OnEpisodeBegin()
    {
        rbAgent.velocity = Vector3.zero;
        rbAgent.angularVelocity = Vector3.zero;
        tfAgent.localEulerAngles = new Vector3(0, 0, 0);

        tfAgent.localPosition = new Vector3(Random.Range(-45.0f, 45.0f), Random.Range(10.0f, 90.0f),
Random.Range(-45.0f, 45.0f));
        tfTarget.localPosition = new Vector3(Random.Range(-45.0f, 45.0f), Random.Range(10.0f, 90.0f),
Random.Range(-45.0f, 45.0f));
        foreach (Transform child in GameObject.Find("Obstacles").transform)
            child.localPosition = new Vector3(Random.Range(-45.0f, 45.0f), Random.Range(10.0f, 90.0f),
Random.Range(-45.0f, 45.0f));

        distBefore = (tfAgent.localPosition - tfTarget.localPosition).magnitude;
        StartCoroutine(RevertMaterial());
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        sensor.AddObservation(tfAgent.localPosition - tfTarget.localPosition);
        sensor.AddObservation(tfTarget.localPosition);
        sensor.AddObservation(tfAgent.localPosition);
        sensor.AddObservation(rbAgent.velocity.x);
        sensor.AddObservation(rbAgent.velocity.y);
        sensor.AddObservation(rbAgent.velocity.z);
    }

    public override void OnActionReceived(ActionBuffers actions)
    {
        float upDown = Mathf.Clamp(actions.ContinuousActions[0], -1.0f, 1.0f);
        float backForth = Mathf.Clamp(actions.ContinuousActions[1], -1.0f, 1.0f);
        float leftRight = Mathf.Clamp(actions.ContinuousActions[2], -1.0f, 1.0f);

        Vector3 direction = Vector3.up * upDown + Vector3.forward * backForth + Vector3.right * leftRight;
        rbAgent.AddForce(direction.normalized * addForce);

        foreach (Transform child in GameObject.Find("Obstacles").transform)
        {
            Vector3[] views = {
                camera1.WorldToViewportPoint(child.position),
                camera2.WorldToViewportPoint(child.position),
                camera3.WorldToViewportPoint(child.position),
                camera4.WorldToViewportPoint(child.position),
                camera5.WorldToViewportPoint(child.position),
                camera6.WorldToViewportPoint(child.position)
            };

            foreach (var view in views)
            {
                if (0 <= view.x && view.x <= 1 && 0 <= view.y && view.y <= 1 && 0 < view.z)
                {
                    float distance = (tfAgent.localPosition - child.localPosition).magnitude;
```

```
            if (distance < limitDistance)
                AddReward((limitDistance - distance) / limitDistance);
        }
      }
    }

    distAfter = (tfAgent.localPosition - tfTarget.localPosition).magnitude;
    GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().distance = distAfter;
    AddReward((distBefore - distAfter) * 10.0f);
    distBefore = distAfter;
    AddReward(-0.01f);
  }

  public override void Heuristic(in ActionBuffers actionsOut)
  {
    var ContinousActionsOut = actionsOut.ContinuousActions;
    if (Input.GetKey(KeyCode.W))
      ContinousActionsOut[0] = 1.0f;
    if (Input.GetKey(KeyCode.S))
      ContinousActionsOut[0] = -1.0f;
    if (Input.GetKey(KeyCode.UpArrow))
      ContinousActionsOut[1] = 1.0f;
    if (Input.GetKey(KeyCode.DownArrow))
      ContinousActionsOut[1] = -1.0f;
    if (Input.GetKey(KeyCode.D))
      ContinousActionsOut[2] = 1.0f;
    if (Input.GetKey(KeyCode.A))
      ContinousActionsOut[2] = -1.0f;
    if (Input.GetKey(KeyCode.Keypad4))
      ContinousActionsOut[3] = -1.0f;
    if (Input.GetKey(KeyCode.Keypad6))
      ContinousActionsOut[3] = 1.0f;
  }

  private void OnCollisionEnter(Collision collision)
  {
    Debug.Log("Collision Occured!!! Collision Name = " + collision.collider.name);
    var ui = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>();

    if (collision.collider.name.Equals("Target"))
    {
      ui.totalCount += 1;
      ui.successCount += 1;
      renderFloor.material.color = Color.green;
      AddReward(10.0f);
    }
    else
    {
      ui.totalCount += 1;
      ui.failedCount += 1;
      renderFloor.material.color = Color.red;
      AddReward(-10.0f);
    }

    Debug.Log("Total = " + ui.totalCount + " | Success = " + ui.successCount + " | Failed = " + ui.failedCount +
          " | Accuracy = " + ui.accuracy.ToString("P") + " | Distance = " + ui.distance.ToString("F") + "m");
    EndEpisode();
  }

  IEnumerator RevertMaterial()
```

```
    {
        yield return new WaitForSeconds(0.3f);
        renderFloor.material.color = Color.white;
        renderTarget.material.color = Color.blue;
    }
}
```

DroneAgent_Bak1.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using UnityEngine;
using UnityEngine.UI;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;

public class DroneAgent_Bak1 : Agent
{
    private Transform tfAgent;
    private Rigidbody rbAgent;
    private Transform tfTarget;
    private Transform tfObstacles;

    private float maxSpeed = 50.0f;
    private float addForce = 25.0f;
    private float rotSpeed = 10.0f;

    private float distAfter;
    private float distBefore;

    private RaycastHit rayHit;
    private float rayAngle = 10.0f;
    private float rayDistance = 10.0f;
    private float rayDistance2 = 25.0f;
    private float rayDiameter = 10.0f;

    private Renderer renderGround;
    private Renderer renderTarget;

    public override void Initialize()
    {
        MaxStep = 2000;
        tfAgent = GetComponent<Transform>();
        rbAgent = GetComponent<Rigidbody>();
        tfTarget = transform.parent.Find("Target").gameObject.GetComponent<Transform>();
        tfObstacles = transform.parent.Find("Obstacles").gameObject.GetComponent<Transform>();
        renderGround = transform.parent.Find("Ground").gameObject.GetComponent<Renderer>();
        renderTarget = transform.parent.Find("Target").gameObject.GetComponent<Renderer>();
    }

    public override void OnEpisodeBegin()
    {
        rbAgent.velocity = Vector3.zero;
        rbAgent.angularVelocity = Vector3.zero;
        tfAgent.eulerAngles = Vector3.zero;
        tfAgent.localPosition = new Vector3(0.0f, 50.0f, -25.0f);
        tfTarget.localPosition = new Vector3(0.0f, 50.0f, 1025.0f);
        distBefore = Vector3.Distance(tfAgent.localPosition, tfTarget.localPosition);
```

```
    StartCoroutine(RevertMaterial());
}

public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(tfAgent.localPosition);
    sensor.AddObservation(tfTarget.localPosition);
    sensor.AddObservation(rbAgent.velocity);
    sensor.AddObservation(rbAgent.angularVelocity);
    sensor.AddObservation(CameraObservation("Result1_L", "Result1_R"));
}

public override void OnActionReceived(ActionBuffers actions)
{
    float positionX = Mathf.Clamp(actions.ContinuousActions[0], -1.0f, 1.0f);
    float positionY = Mathf.Clamp(actions.ContinuousActions[1], -1.0f, 1.0f);
    float positionZ = Mathf.Clamp(actions.ContinuousActions[2], -1.0f, 1.0f);
    Vector3 directionP = Vector3.right * positionX + Vector3.up * positionY + Vector3.forward * positionZ;

    if (rbAgent.velocity.magnitude < maxSpeed) rbAgent.AddForce(directionP.normalized * addForce);
    distAfter = Vector3.Distance(tfAgent.localPosition, tfTarget.localPosition);
    AddReward((distBefore - distAfter) * 10f);
    distBefore = distAfter;
}

public override void Heuristic(in ActionBuffers actionsOut) { }

private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.name.Equals("Target"))
    {
        GameObject.Find("TestDirector").GetComponent<TestDirector>().IncreaseSuccess();
        renderGround.material.color = Color.green;
        AddReward(10f);
        EndEpisode();
    }
    else
    {
        GameObject.Find("TestDirector").GetComponent<TestDirector>().IncreaseFailed();
        renderGround.material.color = Color.red;
        SetReward(-10f);
        EndEpisode();
    }
}

private float CameraObservation(string result1, string result2)
{
    float limitDistance = 15.0f;
    Vector2 center = new Vector2(75.0f, 75.0f);
    float f = 225.0f;
    float W = 150.0f;
    float B = 1.0f;

    for (int i = 0; i < 50; i++)
    {
        GameObject childL = GameObject.Find(result1).transform.GetChild(i).gameObject;
        GameObject childR = GameObject.Find(result2).transform.GetChild(i).gameObject;

        if (childL.activeSelf && childR.activeSelf)
        {
```

```
            string textL = childL.GetComponentInChildren<Text>().text;
            string textR = childR.GetComponentInChildren<Text>().text;
            float accuracyL = float.Parse(Regex.Replace(textL, @"\D", "")) / 100;
            float accuracyR = float.Parse(Regex.Replace(textR, @"\D", "")) / 100;

            float widthL = childL.GetComponent<RectTransform>().rect.width;
            float heightL = childL.GetComponent<RectTransform>().rect.height;
            float widthR = childR.GetComponent<RectTransform>().rect.width;
            float heightR = childR.GetComponent<RectTransform>().rect.height;

            float posLX = childL.GetComponent<RectTransform>().anchoredPosition.x;
            float posLY = childL.GetComponent<RectTransform>().anchoredPosition.y;
            float posRX = childR.GetComponent<RectTransform>().anchoredPosition.x;
            float posRY = childR.GetComponent<RectTransform>().anchoredPosition.y;

            Vector2 posL = new Vector2(posLX - center.x, posLY - center.y);
            Vector2 posR = new Vector2(posRX - center.x, posRY - center.y);

            float w = (widthL + widthR) / 2;
            float b = Vector2.Distance(Vector2.zero, posL - posR);
            float D1 = (B * f) / b;
            float D2 = (W * f) / w;

            return (-limitDistance / (D1 + limitDistance));
        }
    }
    return -1;
}

private float RayObservation(float angleX, float angleY, float angleZ, float limitDistance)
{
    var eulerAngle = Quaternion.Euler(angleX, angleY, angleZ);
    var direction = eulerAngle * tfAgent.forward;
    Physics.Raycast(tfAgent.localPosition, direction, out rayHit, limitDistance);
    return rayHit.distance >= 0f ? rayHit.distance / limitDistance : -1f;
}

private float SphereRayObservation(float angleX, float angleY, float angleZ, float limitDistance)
{
    var eulerAngle = Quaternion.Euler(angleX, angleY, angleZ);
    var direction = eulerAngle * tfAgent.forward;
    Physics.SphereCast(tfAgent.localPosition, rayDiameter / 2.0f, direction, out rayHit, limitDistance);
    return rayHit.distance >= 0 ? rayHit.distance / limitDistance : -1f;
}

IEnumerator RevertMaterial()
{
    yield return new WaitForSeconds(0.3f);
    renderGround.material.color = Color.white;
    renderTarget.material.color = Color.blue;
}

void Start() { }

void Update()
{
    tfAgent.LookAt(new Vector3(tfTarget.position.x, tfAgent.position.y, tfTarget.position.z));
    Debug.DrawRay(tfAgent.position, (tfTarget.localPosition - tfAgent.localPosition), Color.black);
}
```

```
    private void OnDrawGizmos()
    {
        Gizmos.color = Color.green;
        var eulerAngleU = Quaternion.Euler(rayAngle, 0f, 0f) * transform.forward;
        var eulerAngleD = Quaternion.Euler(-rayAngle, 0f, 0f) * transform.forward;
        Gizmos.DrawRay(transform.position, transform.up * rayDistance);
        Gizmos.DrawRay(transform.position, -transform.up * rayDistance);
        Gizmos.DrawRay(transform.position, eulerAngleU * rayDistance2);
        Gizmos.DrawRay(transform.position, eulerAngleD * rayDistance2);

        Gizmos.color = Color.red;
        var eulerAngleL = Quaternion.Euler(0f, -rayAngle, 0f) * transform.forward;
        var eulerAngleR = Quaternion.Euler(0f, rayAngle, 0f) * transform.forward;
        Gizmos.DrawRay(transform.position, -transform.right * rayDistance);
        Gizmos.DrawRay(transform.position, transform.right * rayDistance);
        Gizmos.DrawRay(transform.position, eulerAngleL * rayDistance2);
        Gizmos.DrawRay(transform.position, eulerAngleR * rayDistance2);

        Gizmos.color = Color.blue;
        Gizmos.DrawRay(transform.position, transform.forward * rayDistance2);

        Gizmos.color = Color.cyan;
        Gizmos.DrawWireSphere(transform.position + transform.forward, rayDistance);
    }
}
```

StereoAgent.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using UnityEngine;
using UnityEngine.UI;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;

public class StereoAgent : Agent
{
    private Transform tfAgent;
    private Rigidbody rbAgent;
    private Transform tfTarget;

    private float distAfter;
    private float distBefore;
    private float addForce = 25.0f;

    private Renderer renderFloor;
    private Renderer renderTarget;

    public override void Initialize()
    {
        MaxStep = 1000;
        tfAgent = GetComponent<Transform>();
        rbAgent = GetComponent<Rigidbody>();
        tfTarget = transform.parent.Find("Target").GetComponent<Transform>();
        renderFloor = transform.parent.Find("Ground").GetComponent<Renderer>();
        renderTarget = transform.parent.Find("Target").GetComponent<Renderer>();
    }
```

```
public override void OnEpisodeBegin()
{
    rbAgent.velocity = Vector3.zero;
    rbAgent.angularVelocity = Vector3.zero;
    tfAgent.localEulerAngles = Vector3.zero;

    tfAgent.localPosition = new Vector3(Random.Range(-45f, 45f), Random.Range(10f, 90f), Random.Range(-45f, 45f));
    tfTarget.localPosition = new Vector3(Random.Range(-45f, 45f), Random.Range(10f, 90f), Random.Range(-45f, 45f));

    foreach (Transform child in GameObject.Find("Obstacles").transform)
        child.localPosition = new Vector3(Random.Range(-45f, 45f), Random.Range(10f, 90f), Random.Range(-45f, 45f));

    distBefore = Vector3.Distance(tfAgent.localPosition, tfTarget.localPosition);
    StartCoroutine(RevertMaterial());
}

public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(tfAgent.localPosition - tfTarget.localPosition);
    sensor.AddObservation(tfTarget.localPosition);
    sensor.AddObservation(tfAgent.localPosition);
    sensor.AddObservation(rbAgent.velocity);
}

public override void OnActionReceived(ActionBuffers actions)
{
    float upDown = Mathf.Clamp(actions.ContinuousActions[0], -1f, 1f);
    float backForth = Mathf.Clamp(actions.ContinuousActions[1], -1f, 1f);
    float leftRight = Mathf.Clamp(actions.ContinuousActions[2], -1f, 1f);

    Vector3 direction = Vector3.up * upDown + Vector3.forward * backForth + Vector3.right * leftRight;
    rbAgent.AddForce(direction.normalized * addForce);

    float limitDistance = 15f;
    Vector2 center = new Vector2(75f, 75f);
    float f = 225f;
    float W = 150f;
    float B = 1f;

    for (int cam = 1; cam <= 6; cam++)
    {
        var L = GameObject.Find($"Result{cam}_L");
        var R = GameObject.Find($"Result{cam}_R");

        for (int i = 0; i < 50; i++)
        {
            var left = L.transform.GetChild(i).gameObject;
            var right = R.transform.GetChild(i).gameObject;

            if (left.activeSelf && right.activeSelf)
            {
                var textL = left.GetComponentInChildren<Text>().text;
                var textR = right.GetComponentInChildren<Text>().text;

                float widthL = left.GetComponent<RectTransform>().rect.width;
                float widthR = right.GetComponent<RectTransform>().rect.width;

                float posLX = left.GetComponent<RectTransform>().anchoredPosition.x;
                float posLY = left.GetComponent<RectTransform>().anchoredPosition.y;
```

```
            float posRX = right.GetComponent<RectTransform>().anchoredPosition.x;
            float posRY = right.GetComponent<RectTransform>().anchoredPosition.y;

            Vector2 posL = new Vector2(posLX - center.x, posLY - center.y);
            Vector2 posR = new Vector2(posRX - center.x, posRY - center.y);

            float w = (widthL + widthR) / 2f;
            float b = Vector2.Distance(Vector2.zero, posL - posR);

            float D1 = (B * f) / b;
            float D2 = (W * f) / w;

            AddReward(-limitDistance / (D1 + limitDistance));
        }
      }
    }

    distAfter = Vector3.Distance(tfAgent.localPosition, tfTarget.localPosition);
    GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>().distance = distAfter;

    AddReward(distBefore - distAfter);
    distBefore = distAfter;
    AddReward(-0.1f);
}

public override void Heuristic(in ActionBuffers actionsOut)
{
    var output = actionsOut.ContinuousActions;

    if (Input.GetKey(KeyCode.W)) output[0] = 1f;
    if (Input.GetKey(KeyCode.S)) output[0] = -1f;
    if (Input.GetKey(KeyCode.UpArrow)) output[1] = 1f;
    if (Input.GetKey(KeyCode.DownArrow)) output[1] = -1f;
    if (Input.GetKey(KeyCode.D)) output[2] = 1f;
    if (Input.GetKey(KeyCode.A)) output[2] = -1f;
    if (Input.GetKey(KeyCode.Keypad4)) output[3] = -1f;
    if (Input.GetKey(KeyCode.Keypad6)) output[3] = 1f;
}

private void OnCollisionEnter(Collision collision)
{
    var monitor = GameObject.Find("MonitoringUI").GetComponent<MonitoringUI>();

    if (collision.collider.name.Equals("Target"))
    {
        monitor.totalCount++;
        monitor.successCount++;
        renderFloor.material.color = Color.green;

        AddReward(10f);
    }
    else
    {
        monitor.totalCount++;
        monitor.failedCount++;
        renderFloor.material.color = Color.red;

        AddReward(-10f);
    }
```

```
        EndEpisode();
    }

    IEnumerator RevertMaterial()
    {
        yield return new WaitForSeconds(0.3f);
        renderFloor.material.color = Color.white;
        renderTarget.material.color = Color.blue;
    }
}
```

ДОДАТОК Б
Програмні засоби комп'ютерного зору для виявлення та візуалізації об'єктів у симуляційному середовищі

Visualizer.cs

```
using UnityEngine;
using UnityEngine.UI;
using Klak.TestTools;
using YoloV4Tiny;

sealed class Visualizer : MonoBehaviour
{
    [SerializeField] RenderTexture rTex = null;
    [SerializeField] ImageSource _source = null;
    [SerializeField, Range(0, 1)] float _threshold = 0.5f;
    [SerializeField] ResourceSet _resources = null;
    [SerializeField] RawImage _preview = null;
    [SerializeField] Marker _markerPrefab = null;

    ObjectDetector _detector;
    Marker[] _markers = new Marker[50];

    void Start()
    {
        _detector = new ObjectDetector(_resources);
        for (var i = 0; i < _markers.Length; i++)
            _markers[i] = Instantiate(_markerPrefab, _preview.transform);
    }

    void OnDisable()
    {
        _detector.Dispose();
    }

    void OnDestroy()
    {
        for (var i = 0; i < _markers.Length; i++)
            Destroy(_markers[i]);
    }

    void Update()
    {
        Texture2D tex2D = new Texture2D(rTex.width, rTex.height, TextureFormat.RGB24, false);
        var oldRT = RenderTexture.active;
        RenderTexture.active = rTex;
        tex2D.ReadPixels(new Rect(0, 0, rTex.width, rTex.height), 0, 0);
        tex2D.Apply();
        RenderTexture.active = oldRT;

        _detector.ProcessImage(tex2D, _threshold);

        int i = 0;
        foreach (var d in _detector.Detections)
        {
            if (i == _markers.Length) break;
```

```
            _markers[i++].SetAttributes(d);
        }

        for (; i < _markers.Length; i++)
            _markers[i].Hide();

        _preview.texture = tex2D;
    }
}
```

Marker.cs

```
using UnityEngine;
using UnityEngine.UI;
using YoloV4Tiny;

sealed class Marker : MonoBehaviour
{
    RectTransform _parent;
    RectTransform _xform;
    Image _panel;
    Text _label;

    public static string[] _labels = new[]
    {
        "Plane", "Bicycle", "Bird", "Boat",
        "Bottle", "Bus", "Car", "Cat",
        "Chair", "Cow", "Table", "Dog",
        "Horse", "Motorbike", "Person", "Plant",
        "Sheep", "Sofa", "Train", "TV"
    };

    void Start()
    {
        _xform = GetComponent<RectTransform>();
        _parent = (RectTransform)_xform.parent;
        _panel = GetComponent<Image>();
        _label = GetComponentInChildren<Text>();
    }

    public void SetAttributes(in Detection d)
    {
        var rect = _parent.rect;
        var x = d.x * rect.width;
        var y = (1 - d.y) * rect.height;
        var w = d.w * rect.width;
        var h = d.h * rect.height;

        _xform.anchoredPosition = new Vector2(x, y);
        _xform.SetSizeWithCurrentAnchors(RectTransform.Axis.Horizontal, w);
        _xform.SetSizeWithCurrentAnchors(RectTransform.Axis.Vertical, h);

        var name = _labels[(int)d.classIndex];
        _label.text = $"{name} {(int)(d.score * 100)}%";

        var hue = d.classIndex * 0.073f % 1.0f;
        var color = Color.HSVToRGB(hue, 1, 1);
        color.a = 0.4f;
        _panel.color = color;
```

```
        gameObject.SetActive(true);
    }

    public void Hide()
      => gameObject.SetActive(false);
}
```

IndirectDraw.cs

```
using UnityEngine;
using UnityEngine.UI;
using Klak.TestTools;
using YoloV4Tiny;

public sealed class IndirectDraw : MonoBehaviour
{
    [SerializeField] ImageSource _source = null;
    [SerializeField, Range(0, 1)] float _threshold = 0.5f;
    [SerializeField] ResourceSet _resources = null;
    [SerializeField] RawImage _preview = null;
    [SerializeField] Shader _shader = null;

    ObjectDetector _detector;
    ComputeBuffer _drawArgs;
    Material _material;

    Bounds UnitBox => new Bounds(Vector3.zero, Vector3.one);

    void Start()
    {
        _detector = new ObjectDetector(_resources);
        _drawArgs = new ComputeBuffer(4, sizeof(uint), ComputeBufferType.IndirectArguments);
        _drawArgs.SetData(new [] {6, 0, 0, 0});
        _material = new Material(_shader);
    }

    void OnDestroy()
    {
        _detector.Dispose();
        _drawArgs.Dispose();
        Destroy(_material);
    }

    void LateUpdate()
    {
        _detector.ProcessImage(_source.Texture, _threshold);
        _detector.SetIndirectDrawCount(_drawArgs);
        _material.SetBuffer("_Detections", _detector.DetectionBuffer);
        Graphics.DrawProceduralIndirect(_material, UnitBox, MeshTopology.Triangles, _drawArgs);
        _preview.texture = _source.Texture;
    }
}
```

CameraViewObject.cs

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

public class Ca : MonoBehaviour
```

```
{
    [SerializeField]
    private List<GameObject> findList = null;
    private Camera cam;

    void Start()
    {
        cam = UnityEngine.Camera.main;
    }

    void Update()
    {
        for (int i = 0; i < findList.Count; i++)
        {
            Vector3 viewPos = cam.WorldToViewportPoint(findList[i].transform.position);
            if (0 <= viewPos.x && viewPos.x <= 1 && 0 <= viewPos.y && viewPos.y <= 1 && viewPos.z > 0)
            {
                Debug.Log("Object Name in Camera = " + findList[i].name);
            }
        }
    }
}
```

ДОДАТОК В
Програмна модель фізики та динаміки багатороторного безпілотного літального апарата

Multicopter.cs

```
using UnityEngine;

namespace MBaske
{
   public class Multicopter : MonoBehaviour
   {
      public Transform Frame;
      public Rotor[] Rotors;
      public Rigidbody Rigidbody { get; private set; }
      public Vector3 Inclination => new Vector3(Frame.right.y, Frame.up.y, Frame.forward.y);

      [SerializeField]
      private bool reversableThrust = false;
      [SerializeField]
      private float thrustResponse = 20;
      [SerializeField]
      private float thrustScale = 0.25f;
      [SerializeField]
      private float torqueScale = 0.075f;

      [Header("Rotor Tilt (not used)")]
      [SerializeField]
      private float maxTiltAngle = 60;
      [SerializeField, Range(-1f, 1f)]
      private float pitch;
      [SerializeField, Range(-1f, 1f)]
      private float roll;
      [SerializeField, Range(-1f, 1f)]
      private float yaw;

      private void OnValidate()
      {
         for (int i = 0; i < Rotors.Length; i++)
         {
            Rotors[i].Reversable = reversableThrust;
            Rotors[i].ThrustResponse = thrustResponse;
            Rotors[i].ThrustScale = thrustScale;
            Rotors[i].TorqueScale = torqueScale;
         }

         Initialize();
         UpdateTilt(pitch, roll, yaw);
      }

      public void Initialize()
      {
         Rigidbody = Frame.GetComponent<Rigidbody>();

         for (int i = 0; i < Rotors.Length; i++)
         {
            Rotors[i].Initialize();
```

```
        }
    }

    public void OnReset()
    {
        for (int i = 0; i < Rotors.Length; i++)
        {
            Rotors[i].OnReset();
        }
    }

    public void UpdateThrust(float[] thrustNorm)
    {
        float dt = Time.fixedDeltaTime;

        for (int i = 0; i < Rotors.Length; i++)
        {
            Rotors[i].UpdateThrust(thrustNorm[i], dt);
        }
    }

    public void UpdateTilt(float pitchNorm, float rollNorm, float yawNorm)
    {
        Quaternion rot = Quaternion.Euler(pitchNorm * maxTiltAngle, 0, rollNorm * maxTiltAngle);
        float yawAngle = yawNorm * maxTiltAngle;

        for (int i = 0; i < Rotors.Length; i++)
        {
            Rotors[i].UpdateTilt(rot, yawAngle);
        }
    }

    public Vector3 LocalizeVector(Vector3 v)
    {
        return Frame.InverseTransformVector(v);
    }
  }
}
```

Resetter.cs

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

namespace MBaske
{
  public class ResettableItem
  {
    private Vector3 pos;
    private Quaternion rot;

    private readonly Transform tf;
    private readonly Rigidbody rb;
    private readonly ConfigurableJoint joint;

    public ResettableItem(Transform tf)
    {
      this.tf = tf;
      pos = tf.localPosition;
      rot = tf.localRotation;
      rb = tf.GetComponent<Rigidbody>();
      joint = tf.GetComponent<ConfigurableJoint>();
```

```
        }

        public void Reset()
        {
            if (rb != null)
            {
                rb.velocity = Vector3.zero;
                rb.angularVelocity = Vector3.zero;
                rb.Sleep();
            }

            if (joint != null)
            {
                joint.targetRotation = Quaternion.identity;
            }

            tf.localPosition = pos;
            tf.localRotation = rot;
        }
    }

    public class Resetter
    {
        private readonly List<ResettableItem> items;

        public Resetter(Transform tf)
        {
            items = new List<ResettableItem>();
            Add(tf);
        }

        public void Reset()
        {
            foreach (ResettableItem item in items)
            {
                item.Reset();
            }
        }

        private void Add(Transform tf)
        {
            items.Add(new ResettableItem(tf));

            for (int i = 0; i < tf.childCount; i++)
            {
                Add(tf.GetChild(i));
            }
        }
    }
}
```

Rotor.cs

```
using UnityEngine;

namespace MBaske
{
    public class Rotor : MonoBehaviour
    {
        public float CurrentThrust { get; private set; }

        public bool Reversable { get; set; }
```

```
    public float ThrustResponse { get; set; }
    public float ThrustScale { get; set; }
    public float TorqueScale { get; set; }

    [SerializeField]
    private Transform outerRing;
    [SerializeField]
    private Transform innerRing;
    [SerializeField]
    private Transform rotorBlade;

    // Axis rotation signs.
    [SerializeField]
    private float signZ;
    [SerializeField]
    private float signX;

    private Rigidbody rbInnerRing;
    private ConfigurableJoint jointZ;
    private ConfigurableJoint jointX;
    private float signSpin; // CW+ CCW-
    private const float animSpeed = 2400;

    public void Initialize()
    {
        rbInnerRing = innerRing.GetComponent<Rigidbody>();
        jointZ = outerRing.GetComponent<ConfigurableJoint>();
        jointX = innerRing.GetComponent<ConfigurableJoint>();

        signSpin = rotorBlade.name == "RotorCW" ? 1f : -1f;
    }

    public void OnReset()
    {
        CurrentThrust = 0;
    }

    public void UpdateThrust(float thrustNorm, float deltaTime)
    {
        thrustNorm = Reversable ? thrustNorm : (thrustNorm + 1f) * 0.5f;
        CurrentThrust = Mathf.Lerp(CurrentThrust, thrustNorm, deltaTime * ThrustResponse);
        rbInnerRing.AddForce(innerRing.up * CurrentThrust * ThrustScale, ForceMode.Impulse);
        rbInnerRing.AddRelativeTorque(innerRing.up * CurrentThrust * TorqueScale * -signSpin, ForceMode.Impulse);
    }

    public void UpdateTilt(Quaternion rot, float yawAngle)
    {
        Quaternion r = Quaternion.Inverse(rot) * transform.localRotation;
        jointX.targetRotation = Quaternion.Euler(r.eulerAngles.x + yawAngle * signX, 0, 0);
        jointZ.targetRotation = Quaternion.Euler(0, 0, r.eulerAngles.z + yawAngle * signZ);
    }

    private void Update()
    {
        // Animation.
        rotorBlade.Rotate(0, CurrentThrust * animSpeed * signSpin * Time.deltaTime, 0, Space.Self);
    }
  }
}
```

Test.cs

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

public class Test : MonoBehaviour
{
   private Rigidbody motorFR;
   private Rigidbody motorFL;
   private Rigidbody motorRR;
   private Rigidbody motorRL;

   private float addForce = 1000.0f;

   void Start()
   {
      var fans = transform.Find("Fans");
      motorFR = fans.Find("fan.002").GetComponent<Rigidbody>();
      motorFL = fans.Find("fan.004").GetComponent<Rigidbody>();
      motorRR = fans.Find("fan.003").GetComponent<Rigidbody>();
      motorRL = fans.Find("fan.001").GetComponent<Rigidbody>();
   }

   void Update()
   {
      if (Input.GetKey(KeyCode.W))
      {
         Vector3 force = Vector3.up * addForce;
         motorFR.AddRelativeForce(force);
         motorFL.AddRelativeForce(force);
         motorRR.AddRelativeForce(force);
         motorRL.AddRelativeForce(force);
      }
   }
}
```

ДОДАТОК Г
Допоміжні програмні засоби та математичні методи обробки керуючих сигналів

AgentUtil.cs

```
using UnityEngine;

public static class AgentUtil
{
   public static float Sigmoid(float val)
   {
      return val / (1f + Mathf.Abs(val));
   }

   public static Vector3 Sigmoid(Vector3 v3)
   {
      v3.x = Sigmoid(v3.x);
      v3.y = Sigmoid(v3.y);
      v3.z = Sigmoid(v3.z);
      return v3;
   }
}
```

GizmoAxes.cs

```
using UnityEngine;

namespace MBaske
{
   public class GizmoAxes : MonoBehaviour
   {
      [SerializeField]
      private Color right = Color.red;
      [SerializeField]
      private Color up = Color.green;
      [SerializeField]
      private Color forward = Color.blue;

      [SerializeField]
      private float length = 1;
      [SerializeField]
      private bool draw = true;

      private void OnDrawGizmos()
      {
         if (draw)
         {
            Gizmos.color = right;
            Gizmos.DrawRay(transform.position, transform.right * length);
            Gizmos.color = up;
            Gizmos.DrawRay(transform.position, transform.up * length);
            Gizmos.color = forward;
            Gizmos.DrawRay(transform.position, transform.forward * length);
         }
      }
```

```
    }
}
```

ДОДАТОК Д
Програмна реалізація системи візуального спостереження та позиціювання камери

MainCamera.cs

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

public class MainCamera : MonoBehaviour
{
    private GameObject Agent;
    private GameObject Target;

    void Start()
    {
        Agent = GameObject.Find("Agent");
        Target = GameObject.Find("Target");
    }

    void Update()
    {
        Vector3 direction = Agent.transform.forward.normalized * -10.0f + new Vector3(0.0f, 3.0f, 0.0f);
        transform.position = Agent.transform.position + direction;
        transform.LookAt(Agent.transform.position);
    }
}
```

ДОДАТОК Е
Програмні засоби моніторингу та оцінювання результатів навчання агентів

MonitoringUI.cs

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

public class MonitoringUI : MonoBehaviour
{
    public int totalCount;
    public int successCount;
    public int failedCount;
    public double distance;
    public double accuracy;

    public float time;
    public float velocity;
    public float moveDistance;

    private Text totalText;
    private Text successText;
    private Text failedText;
    private Text distanceText;
    private Text accuracyText;

    private Text timeText;
    private Text velocityText;
    private Text moveDistanceText;

    void Start()
    {
        totalCount = 0;
        successCount = 0;
        failedCount = 0;
        distance = 0.0f;
        accuracy = 0.0f;

        time = 0.0f;
        velocity = 0.0f;
        moveDistance = 0.0f;

        totalText = GameObject.Find("TotalCount").GetComponent<Text>();
        successText = GameObject.Find("SuccessCount").GetComponent<Text>();
        failedText = GameObject.Find("FailedCount").GetComponent<Text>();
        distanceText = GameObject.Find("Distance").GetComponent<Text>();
        accuracyText = GameObject.Find("Accuracy").GetComponent<Text>();

        timeText = GameObject.Find("Time").GetComponent<Text>();
        velocityText = GameObject.Find("Velocity").GetComponent<Text>();
        moveDistanceText = GameObject.Find("MoveDistance").GetComponent<Text>();
    }

    void Update()
    {
```

```
        totalText.text = "Total Try: " + totalCount.ToString();
        successText.text = "Success: " + successCount.ToString();
        failedText.text = "Failed:" + failedCount.ToString();

        accuracy = (double)successCount / totalCount;
        distanceText.text = "TargetDistance: " + distance.ToString("F") + "m";
        accuracyText.text = "Accuracy: " + accuracy.ToString("P");

        time += Time.deltaTime;
        velocity = moveDistance / time;

        timeText.text = "Time: " + time.ToString("F") + "s";
        velocityText.text = "Velocity: " + velocity.ToString("F") + "m/s";
        moveDistanceText.text = "Moved Distance: " + moveDistance.ToString("F") + "m";
    }
}
```

LearningDirector.cs

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

public class LearningDirector : MonoBehaviour
{
    private int cntTotal;
    private int cntSuccess;
    private int cntFailed;
    private double accuracy;

    private Text totalText;
    private Text successText;
    private Text failedText;
    private Text accuracyText;

    void Start()
    {
        cntTotal = 0;
        cntSuccess = 0;
        cntFailed = 0;
        accuracy = 0.0f;

        totalText = GameObject.Find("Total").GetComponent<Text>();
        successText = GameObject.Find("Success").GetComponent<Text>();
        failedText = GameObject.Find("Failed").GetComponent<Text>();
        accuracyText = GameObject.Find("Accuracy").GetComponent<Text>();
    }

    void Update()
    {
        accuracy = (double)cntSuccess / cntTotal;

        totalText.text = "Total: " + cntTotal.ToString();
        successText.text = "Success: " + cntSuccess.ToString();
        failedText.text = "Failed:" + cntFailed.ToString();
        accuracyText.text = "Accuracy: " + accuracy.ToString("P");
    }

    public void IncreaseSuccess()
    {
```

```
        cntTotal += 1;
        cntSuccess += 1;
        PrintDebugLog();
    }

    public void IncreaseFailed()
    {
        cntTotal += 1;
        cntFailed += 1;
        PrintDebugLog();
    }

    private void PrintDebugLog()
    {
        accuracy = (double)cntSuccess / cntTotal;
        Debug.Log("Total Try = " + cntTotal + " | Success = " + cntSuccess + " | Failed = " + cntFailed + " | Accuracy = " +
accuracy.ToString("P"));
    }
}
```

TestDirector.cs

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

public class TestDirector : MonoBehaviour
{
    private Transform tfAgent;
    private Rigidbody rbAgent;
    private Transform tfTarget;
    private Rigidbody rbTarget;

    private int cntTotal;
    private int cntSuccess;
    private int cntFailed;

    private double time;
    private double velocity;
    private double distance;
    private double accuracy;
    private double avgTime;

    private Text totalText;
    private Text successText;
    private Text failedText;
    private Text timeText;
    private Text velocityText;
    private Text distanceText;
    private Text accuracyText;

    void Start()
    {
        cntTotal = 0;
        cntSuccess = 0;
        cntFailed = 0;

        time = 0.0f;
        velocity = 0.0f;
        distance = 0.0f;
```

```
        accuracy = 0.0f;
        avgTime = 0.0f;

        totalText = GameObject.Find("Total").GetComponent<Text>();
        successText = GameObject.Find("Success").GetComponent<Text>();
        failedText = GameObject.Find("Failed").GetComponent<Text>();

        timeText = GameObject.Find("Time").GetComponent<Text>();
        velocityText = GameObject.Find("Velocity").GetComponent<Text>();
        distanceText = GameObject.Find("Distance").GetComponent<Text>();
        accuracyText = GameObject.Find("Accuracy").GetComponent<Text>();

        tfAgent = GameObject.Find("Agent").GetComponent<Transform>();
        rbAgent = GameObject.Find("Agent").GetComponent<Rigidbody>();

        tfTarget = GameObject.Find("Target").GetComponent<Transform>();
        rbTarget = GameObject.Find("Target").GetComponent<Rigidbody>();
    }

    void Update()
    {
        time += Time.deltaTime;
        velocity = rbAgent.velocity.magnitude;
        distance = Vector3.Distance(tfAgent.localPosition, tfTarget.localPosition);
        accuracy = (cntTotal > 0) ? (double)cntSuccess / cntTotal : 0.0;

        totalText.text = "Total: " + cntTotal;
        successText.text = "Success: " + cntSuccess;
        failedText.text = "Failed:" + cntFailed;

        timeText.text = "Time: " + (cntSuccess > 0 ? (avgTime / cntSuccess).ToString("F") : "0") + "s";
        velocityText.text = "Velocity: " + velocity.ToString("F") + "m/s";
        distanceText.text = "Distance: " + distance.ToString("F") + "m";
        accuracyText.text = "Accuracy: " + accuracy.ToString("P");
    }

    public void IncreaseSuccess()
    {
        avgTime += time;
        time = 0.0f;
        cntTotal += 1;
        cntSuccess += 1;
    }

    public void IncreaseFailed()
    {
        time = 0.0f;
        cntTotal += 1;
        cntFailed += 1;
    }

    private void PrintDebugLog()
    {
        accuracy = (cntTotal > 0) ? (double)cntSuccess / cntTotal : 0.0;
        Debug.Log($"Total Try = {cntTotal} | Success = {cntSuccess} | Failed = {cntFailed} | Accuracy = {accuracy:P}");
    }
}
```