НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук
(назва кафедри)

_____/ Голуб Б.Л. /
(підпис) (ПІБ)

"___" _____ 2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему

«ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ КЕРУВАННЯ ВІРТУАЛЬНИМИ ОБ'ЄКТАМИ НА ОСНОВІ БАГАТОШАРОВИХ НЕЙРОМЕРЕЖ»

Спеціальність 121 – «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент _____ Вайганг Ганна Олексіївна .
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи
д.е.н., професор _____ Руденський Роман Анатолійович
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Тищенко Михайло Вячеславович
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ
УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук
(назва кафедри)

к.т.н., доцент Голуб Б.Л.
(науковий ступінь, вчене звання) (підпис) (ПІБ)
"___" _____

2025 р.

З А В Д А Н Н Я
на виконання бакалаврської кваліфікаційної роботи студенту
Тищенко Михайло Вячеславович

Спеціальність 121 – «Інженерія програмного забезпечення»

1. Тема бакалаврської кваліфікаційної роботи «Програмне забезпечення системи керування віртуальними об'єктами на основі багатошарових нейромереж» затверджена наказом ректора НУБіП України від 16.12.2024 № 2248 "С"
2. Термін подання завершеної роботи на кафедру_____
(рік, місяць, число)

3. Вихідні дані: симуляційне середовище Unity, ML-Agents Toolkit, архітектури нейронних мереж різної глибини та ширини.
4. Перелік питань, що розглядаються:

- Аналіз проблемної області
- Моделювання предметної області
- Проєктування програмної системи
- Впровадження та експлуатація системи

plex scenarios.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ML (Machine Learning) - машинне навчання

AI (Artificial Intelligence) - штучний інтелект

NN (Neural Network) - нейронна мережа

RL (Reinforcement Learning) - навчання з підкріпленням

MLP (Multi-Layer Perceptron) - багатошаровий перцептрон

API (Application Programming Interface) - інтерфейс програмування додатків

PPO (Proximal Policy Optimization) - проксимальна оптимізація політики

GPU (Graphics Processing Unit) - графічний процесор

CPU (Central Processing Unit) - центральний процесор

FPS (Frames Per Second) - кадри за секунду

UI (User Interface) - користувацький інтерфейс

3D - тривимірний

SDK (Software Development Kit) - набір засобів розробки

C# - мова програмування C Sharp

ВСТУП

У сучасному світі машинного навчання та штучного інтелекту спостерігається значне зростання популярності методів глибокого навчання, які успішно застосовуються для вирішення різноманітних завдань у багатьох галузях. Особливо перспективним є використання нейронних мереж для створення систем автономного керування, здатних адаптуватися до динамічних середовищ та приймати рішення в реальному часі. Одним з актуальних напрямків є розробка інтелектуальних агентів для керування віртуальними об'єктами, що має практичне застосування в ігровій індустрії, симуляційних системах, робототехніці та автономному транспорті.

Проблема вибору оптимальної архітектури нейронної мережі залишається відкритим питанням у сфері машинного навчання. Різні конфігурації мереж демонструють різну ефективність залежно від характеру задачі, складності середовища та доступних обчислювальних ресурсів. Систематичне дослідження впливу архітектурних параметрів на продуктивність навчання є важливим для оптимізації процесу розробки інтелектуальних систем.

Метою дипломної роботи є розробка програмного забезпечення для створення, навчання та порівняльного аналізу багатошарових нейронних мереж різної архітектури для задачі керування віртуальними об'єктами в симуляційному середовищі.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

проаналізувати існуючі підходи до навчання агентів для керування віртуальними об'єктами;

спроектувати та реалізувати симуляційне середовище на базі Unity;

розробити систему для навчання агентів з використанням ML-Agents;

реалізувати п'ять різних архітектур нейронних мереж для порівняльного аналізу;

провести експериментальні дослідження ефективності різних архітектур;

створити інтерактивний інтерфейс для візуалізації результатів та взаємодії з навченими моделями.

Об'єктом дослідження є процес навчання багатошарових нейронних мереж для автономного керування віртуальними об'єктами.

Предметом дослідження є методи, алгоритми та архітектури нейронних мереж для задач керування віртуальними об'єктами в симуляційному середовищі.

Методи дослідження включають аналіз літературних джерел, математичне та комп'ютерне моделювання, методи машинного навчання з підкріпленням, експериментальні дослідження, статистичний аналіз результатів.

Наукова новизна роботи полягає в систематичному порівняльному аналізі впливу архітектурних параметрів нейронних мереж на ефективність навчання в контексті задачі автономного керування віртуальними автомобілями.

Практична цінність роботи визначається розробленим програмним забезпеченням, яке може використовуватися для навчальних цілей, демонстрації принципів навчання з підкріпленням, а також як основа для подальших досліджень у галузі машинного навчання та автономних систем.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Огляд методів машинного навчання для керування віртуальними об'єктами

Керування віртуальними об'єктами є однією з ключових задач у сфері штучного інтелекту та машинного навчання. Протягом останніх десятиліть спостерігається значний прогрес у розробці алгоритмів, здатних навчати агентів приймати рішення в динамічних середовищах. Еволюція цих методів пройшла шлях від простих правил поведінки до складних систем на основі глибокого навчання.

На ранніх етапах розвитку для керування віртуальними об'єктами використовувалися класичні алгоритми, такі як скінченні автомати та дерева рішень. Ці методи базувалися на жорстко запрограмованих правилах і не могли адаптуватися до нових ситуацій. З появою теорії навчання з підкріпленням (Reinforcement Learning, RL) у 1980-х роках ситуація почала змінюватися.

Основоположники теорії навчання з підкріпленням Річард Саттон та Ендрю Барто визначили базові принципи взаємодії агента з середовищем через систему винагород та покарань. Класичні алгоритми RL, такі як Q-learning та SARSA, стали основою для багатьох систем керування, але мали обмеження при роботі з високорозмірними просторами станів[1].

Прорив у галузі відбувся з появою Deep Reinforcement Learning - поєднання глибоких нейронних мереж з методами навчання з підкріпленням. Робота DeepMind з алгоритмом DQN (Deep Q-Network) у 2013 році продемонструвала можливість навчання агентів грати в класичні відеоігри Atari на рівні людини-

експерта. Це відкрило нові горизонти для застосування методів глибокого навчання в задачах керування[9].

Сучасні підходи до навчання агентів включають:

1. **Policy Gradient методи** - прямо оптимізують політику агента без необхідності оцінювати функцію цінності. Алгоритми REINFORCE та Actor-Critic належать до цієї категорії.
2. **Proximal Policy Optimization (PPO)** - один з найпопулярніших алгоритмів, що забезпечує баланс між стабільністю навчання та швидкістю збіжності[7]. Саме PPO використовується в Unity ML-Agents за замовчуванням.
3. **Soft Actor-Critic (SAC)** - алгоритм, що максимізує не тільки очікувану винагороду, але й ентропію політики, забезпечуючи більш стійке та різноманітне навчання.
4. **Curiosity-driven Exploration** - методи, що стимулюють агента досліджувати нові стани середовища через внутрішню мотивацію.

Важливим аспектом сучасних систем є використання паралельного навчання, коли кілька агентів одночасно взаємодіють з різними копіями середовища, прискорюючи процес збору досвіду. Це особливо актуально для задач керування віртуальними об'єктами, де потрібно багато ітерацій для досягнення оптимальної поведінки[21].

1.2. Архітектури нейронних мереж в задачах керування

Архітектура нейронної мережі є критичним фактором, що визначає здатність агента навчатися та адаптуватися до складних середовищ[22]. У контексті керування віртуальними об'єктами вибір правильної архітектури

може суттєво вплинути на швидкість навчання, якість виконання завдань та стабільність роботи системи.

Базовою архітектурою для задач керування є багатошаровий перцептрон (MLP)[25]. Він складається з вхідного шару, одного або декількох прихованих шарів та вихідного шару. Кожен нейрон у шарі з'єднаний з усіма нейронами наступного шару, що забезпечує повну зв'язність мережі.

Ключові параметри архітектури включають:

1. **Глибина мережі** - кількість прихованих шарів. Глибші мережі здатні вивчати більш складні представлення, але можуть страждати від проблем зникаючого градієнта та потребувати більше даних для навчання.
2. **Ширина мережі** - кількість нейронів у кожному шарі. Ширші мережі мають більшу ємність для зберігання інформації, але можуть призводити до перенавчання.
3. **Функції активації** - нелінійні перетворення, що застосовуються до виходів нейронів. ReLU (Rectified Linear Unit) є найпоширенішим вибором для прихованих шарів, тоді як вихідний шар може використовувати tanh для безперервних дій або softmax для дискретних.
4. **Нормалізація** - техніки як batch normalization або layer normalization можуть покращити стабільність навчання та прискорити збіжність.

У контексті задач керування важливим є баланс між глибиною та шириною мережі. Дослідження показують, що для простих середовищ часто достатньо неглибоких, але широких мереж, тоді як складні середовища з багатьма залежностями можуть вимагати глибших архітектур[28].

Специфічні архітектурні рішення для задач керування включають:

- **Dual-stream архітектури** - окремі потоки для обробки різних типів вхідних даних (наприклад, візуальних та сенсорних)
- **Recurrent архітектури** - використання LSTM або GRU для врахування історії станів
- **Attention механізми** - для фокусування на важливих елементах вхідних даних

Критерії оцінки ефективності архітектури включають:

- Швидкість збіжності до оптимального рішення
- Стабільність навчання (відсутність різких падінь продуктивності)
- Здатність до узагальнення на нові ситуації
- Обчислювальна ефективність (час виконання одного кроку)

1.3. Unity ML-Agents як платформа для дослідження

Unity ML-Agents є потужним фреймворком для розробки та дослідження систем машинного навчання в контексті 3D симуляцій. Випущений компанією Unity Technologies у 2017 році, цей інструментарій надає можливість інтегрувати алгоритми навчання з підкріпленням безпосередньо в Unity Engine, створюючи міст між світом розробки ігор та штучного інтелекту[8,10].

Основні компоненти ML-Agents включають[11]:

1. **Learning Environment** - середовище Unity, де агенти взаємодіють з віртуальним світом

2. **Python API** - інтерфейс для з'єднання Unity з бібліотеками машинного навчання
3. **Trainers** - реалізації алгоритмів навчання (PPO, SAC, тощо)
4. **Gym wrapper** - сумісність з OpenAI Gym для використання стандартних алгоритмів

Ключові можливості платформи:

- **Реалістична фізика** - використання PhysX engine для точного моделювання фізичних взаємодій
- **Візуальне спостереження** - агенти можуть використовувати камери як джерело вхідних даних
- **Паралельне навчання** - можливість запуску багатьох екземплярів середовища одночасно
- **Curriculum Learning** - поступове ускладнення завдань для покращення навчання
- **Imitation Learning** - навчання на демонстраціях експерта

Процес розробки агента в ML-Agents включає:

1. Створення середовища в Unity
2. Визначення спостережень (observations) та дій (actions)
3. Розробка системи винагород
4. Налаштування конфігурації навчання
5. Запуск процесу навчання
6. Оцінка та оптимізація результатів

Переваги використання ML-Agents для дослідження[12]:

- Інтуїтивно зрозумілий візуальний редактор Unity
- Широкі можливості налаштування середовища
- Вбудовані інструменти для моніторингу навчання
- Активна спільнота розробників
- Регулярні оновлення та покращення

Обмеження платформи:

- Залежність від екосистеми Unity
- Обмежений вибір алгоритмів навчання
- Потреба в потужному апаратному забезпеченні для складних симуляцій
- Крива навчання для розробників без досвіду в Unity

Для задачі дослідження архітектур нейронних мереж ML-Agents надає ідеальне середовище, оскільки дозволяє легко змінювати конфігурацію мережі через YAML файли та проводити експерименти з однаковими умовами для різних архітектур.

1.4. Постановка задачі дослідження

На основі проведеного аналізу предметної області можна сформулювати основну проблему дослідження: відсутність систематичного порівняльного аналізу впливу архітектурних параметрів нейронних мереж на ефективність навчання в контексті керування віртуальними об'єктами. Хоча існує багато

досліджень окремих архітектур, бракує комплексного підходу до оцінки їх ефективності в однакових умовах.

Мета дослідження: розробити програмне забезпечення для створення, навчання та порівняльного аналізу багатошарових нейронних мереж різної архітектури для задачі керування віртуальними об'єктами в симуляційному середовищі.

Основні завдання дослідження:

1. **Розробити симуляційне середовище** на базі Unity, що моделює гоночний трек з реалістичною фізикою для навчання агентів керуванню автомобілем.
2. **Реалізувати систему навчання** з використанням Unity ML-Agents, що дозволяє тренувати агентів з різними архітектурами нейронних мереж.
3. **Провести експериментальне дослідження** п'яти архітектур:
    - 1×512 (один шар з 512 нейронами)
    - 2×256 (два шари по 256 нейронів)
    - 4×128 (чотири шари по 128 нейронів)
    - 8×64 (вісім шарів по 64 нейрони)
    - 16×32 (шістнадцять шарів по 32 нейрони)
4. **Розробити систему збору та аналізу статистики**, що включає метрики швидкості навчання, стабільності результатів, якості виконання завдань.
5. **Створити інтерактивний інтерфейс користувача** для візуалізації результатів, перегляду найкращих заїздів та змагання з навченими агентами.

6. **Провести порівняльний аналіз** ефективності різних архітектур та сформулювати рекомендації щодо їх використання.

Очікувані результати:

1. Програмний комплекс для дослідження архітектур нейронних мереж у задачах керування
2. Експериментальні дані про ефективність різних архітектур
3. Рекомендації щодо вибору оптимальної архітектури для конкретних сценаріїв
4. Інтерактивна система для демонстрації результатів навчання

Критерії успішності дослідження:

- Успішне навчання агентів для всіх досліджуваних архітектур
- Отримання статистично значущих результатів порівняння
- Функціональність усіх компонентів програмного забезпечення
- Можливість відтворення результатів експериментів

Практична значимість роботи полягає в створенні інструменту для дослідників та розробників у галузі машинного навчання, який дозволяє швидко експериментувати з різними архітектурами нейронних мереж та оцінювати їх ефективність у стандартизованому середовищі.

2. ПРОЕКТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

2.1. Архітектура програмного забезпечення

Розроблювана система представляє собою інтегроване програмне рішення, що поєднує симуляційне середовище Unity з фреймворком машинного навчання ML-Agents для керування віртуальними об'єктами на основі багатошарових нейромереж. Архітектура системи спроектована за модульним принципом, що забезпечує гнучкість, масштабованість та можливість незалежного тестування окремих компонентів.

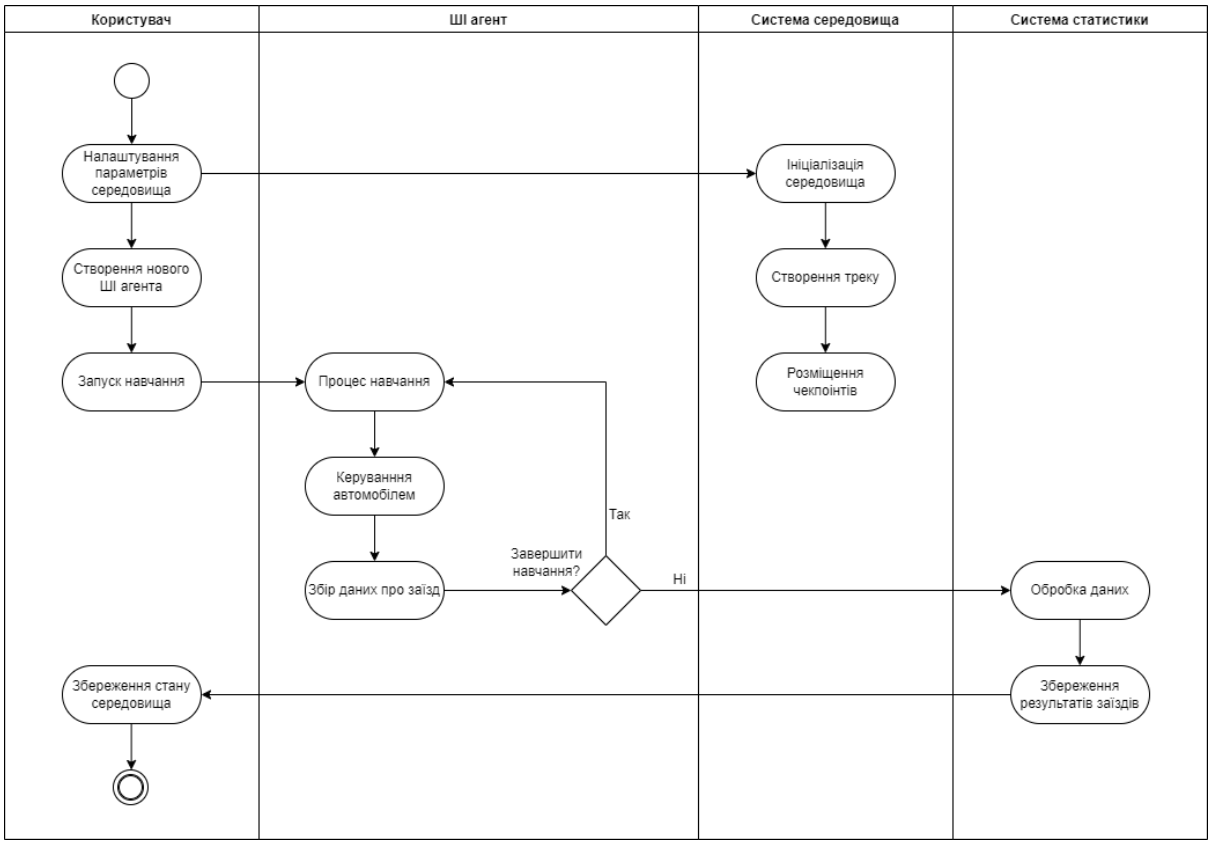


Рис. 1 Діаграма активності

Загальна структура системи

Система поділяється на чотири основні архітектурні шари:

1. **Рівень представлення** (Presentation Layer) - відповідає за користувацький інтерфейс, візуалізацію результатів навчання та взаємодію з користувачем
2. **Рівень бізнес-логіки** (Business Logic Layer) - містить основну логіку симуляції, управління агентами та обробку ігрових подій
3. **Рівень даних** (Data Layer) - забезпечує збереження та доступ до даних навчання, результатів експериментів та конфігурацій нейромереж
4. **Рівень інтеграції з ML** (ML Integration Layer) - забезпечує зв'язок між Unity середовищем та Python-based ML-Agents

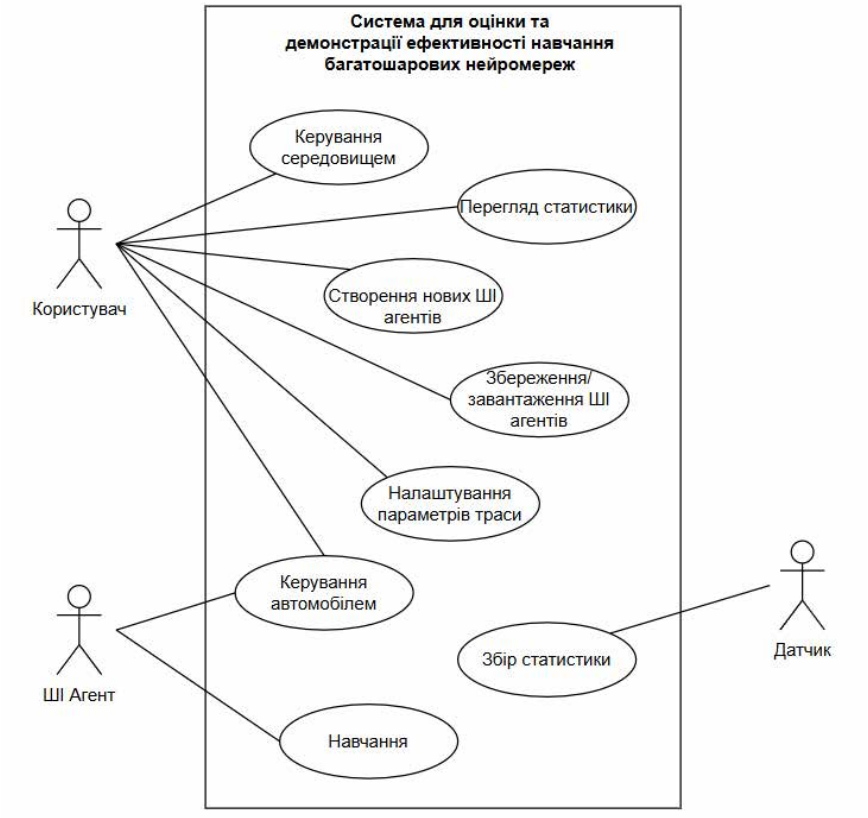


Рис. 2 Use-Case діаграма

Діаграма компонентів

Ключові компоненти системи та їх взаємодія представлені на діаграмі компонентів:

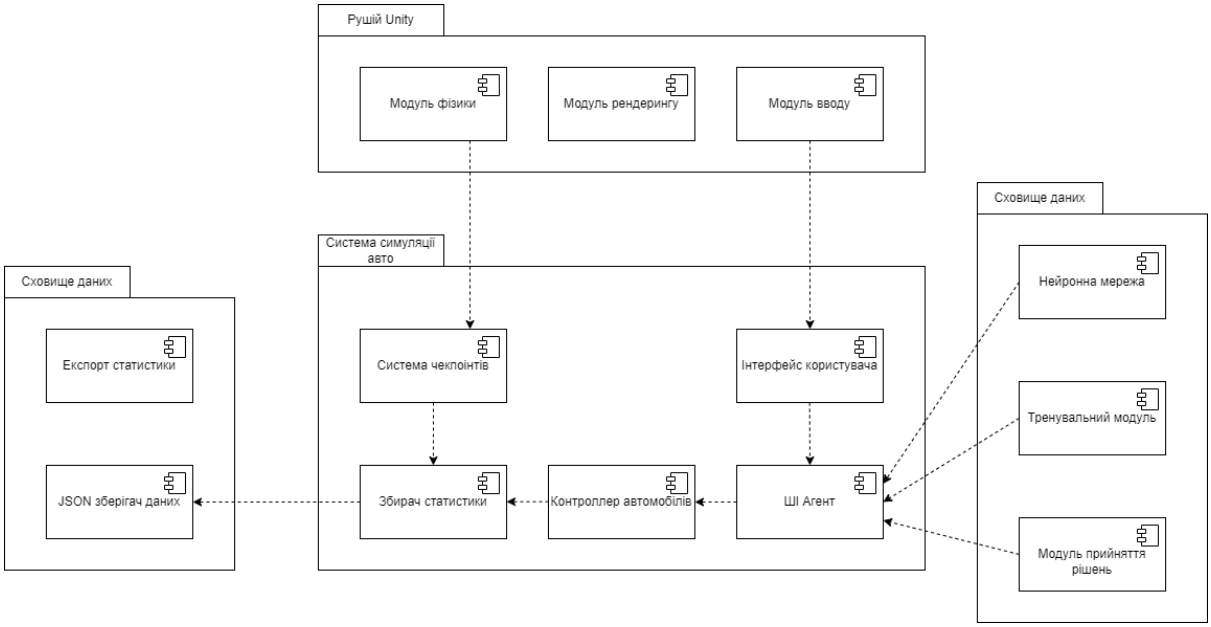


Рис. 3 Діаграма компонентів

1. Компоненти симуляційного середовища Unity:

- **CarController** - відповідає за фізичне моделювання автомобіля та базову систему керування
- **CarAgent** - імплементує агента ML-Agents, який керує автомобілем, використовуючи глибоку нейронну мережу
- **Tracker** - відстежує прогрес автомобіля на трасі, перевіряє проходження контрольних точок
- **LapDataSaver** - зберігає дані про пройдені кола, включаючи траєкторію руху, час, швидкість
- **TrajectorySimulate** - відтворює записані траєкторії руху для візуалізації і аналізу результатів
- **UIEngine** - забезпечує взаємодію з користувачем через інтерфейс

2. Компоненти ML-Agents:

- **Training Module** - керує процесом навчання нейронних мереж
- **Brain Component** - реалізує різні архітектури нейронних мереж (1×512, 2×256, 4×128, тощо)
- **Policy Component** - визначає поведінку агента на основі вихідних даних нейромережі
- **Reward System** - обчислює винагороди на основі дій агента та стану середовища

3. Компоненти даних та аналітики:

- **Statistics Collector** - збирає метрики продуктивності під час навчання
- **Model Repository** - зберігає навчені моделі нейромереж
- **Experiment Manager** - керує проведенням експериментів з різними архітектурами

Інтерфейси взаємодії модулів

Для забезпечення гнучкості та розширюваності системи, взаємодія між компонентами здійснюється через чітко визначені інтерфейси:

```
// Інтерфейс для контролерів автомобіля
public interface ICarController
{
    void InputAIParams(int forwardMove, int rightMove);
    float GetSpeed();
    void Brakes();
}
```

```
// Інтерфейс для роботи з агентами машинного навчання
public interface IAgent
{
    void OnEpisodeBegin();
    void CollectObservations(VectorSensor sensor);
    void OnActionReceived(ActionBuffers actions);
    float GetTotalReward();
    void EndOfRace(bool isFinished, LapData lapData);
}

// Інтерфейс для відстеження прогресу на трасі
public interface ITracker
{
    void ResetCheckpoints();
    float CalculateCompletionPercentage();
    Vector3 GetDirectionToNextCheckpoint();
    float GetDistanceToNextCheckpoint();
    float CalculateAverageSpeed();
    int GetCurrentLapNumber();
    float GetCurrentLapTime();
    FrameData[] GetCurrentFrames();
```

```
}
```

Для імплементації гнучкої архітектури були використані наступні патерни проектування:

- **Observer** - для сповіщення про проходження контрольних точок та зіткнення
- **Strategy** - для реалізації різних підходів до розрахунку винагород
- **Factory** - для створення агентів з різними архітектурами нейромереж
- **Singleton** - для глобальних сервісів, таких як LapDataSaver
- **Command** - для відокремлення логіки керування автомобілем від логіки прийняття рішень

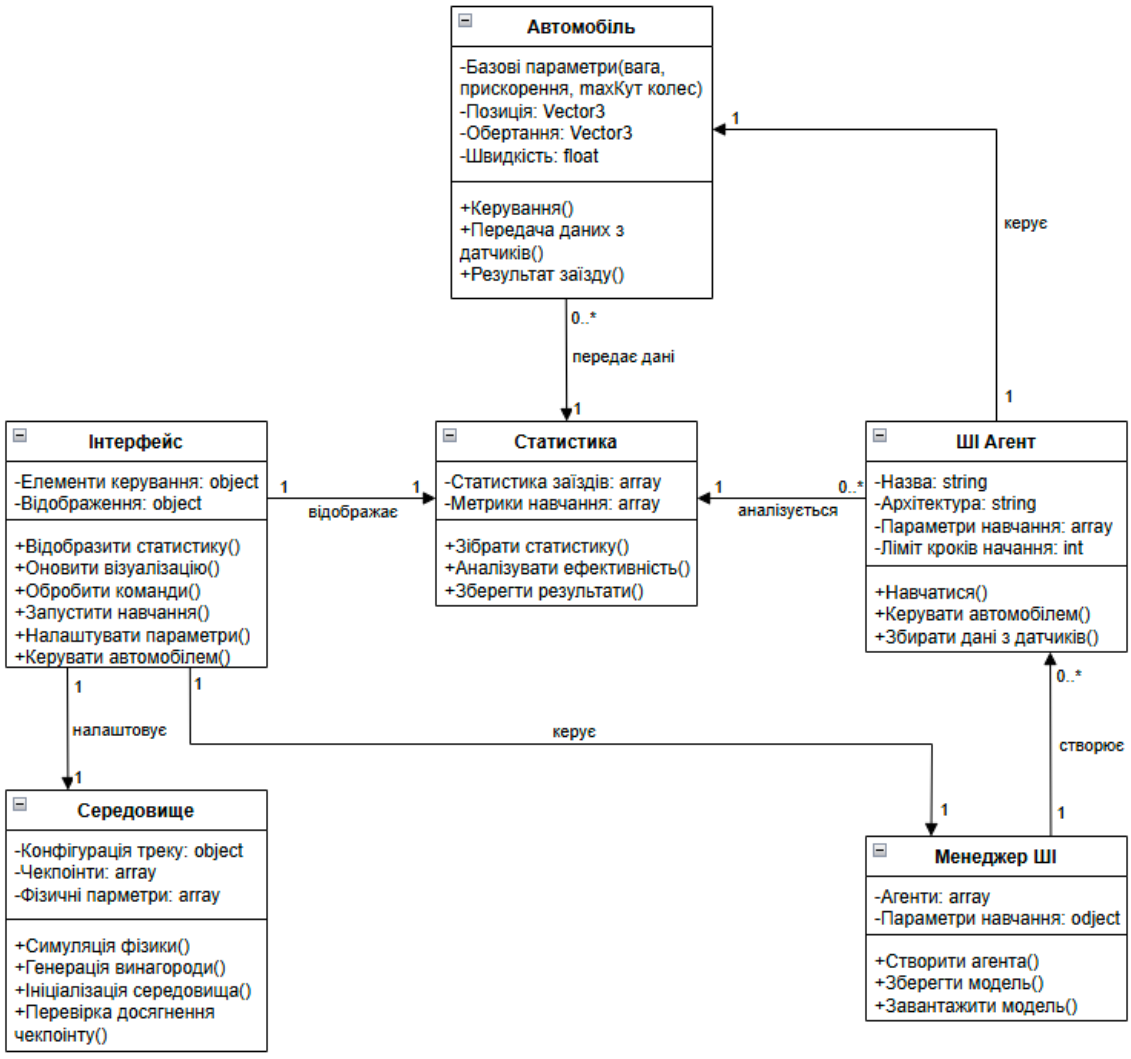


Рис. 4 Діаграма класів

2.2. Моделювання віртуального середовища

Віртуальне середовище є ключовим компонентом системи, що забезпечує реалістичну симуляцію для навчання нейромережевих агентів. Проектування середовища включає створення фізичної моделі автомобіля, гоночного треку та системи сенсорів для збору інформації.

Проектування гоночного треку

Гоночний трек спроектований як модульна система, що дозволяє налаштовувати рівень складності для оцінки адаптивності різних архітектур нейромереж:

Базові елементи треку:

- Прямі сегменти різної довжини
- Повороти з різними радіусами кривизни
- Складні секції (S-подібні повороти)
- Перепади висот

Параметри конфігурації:

- Ширина треку: 10-15 метрів
- Загальна довжина: ~800 метрів
- Кількість поворотів: 12
- Складність маршруту: регульована

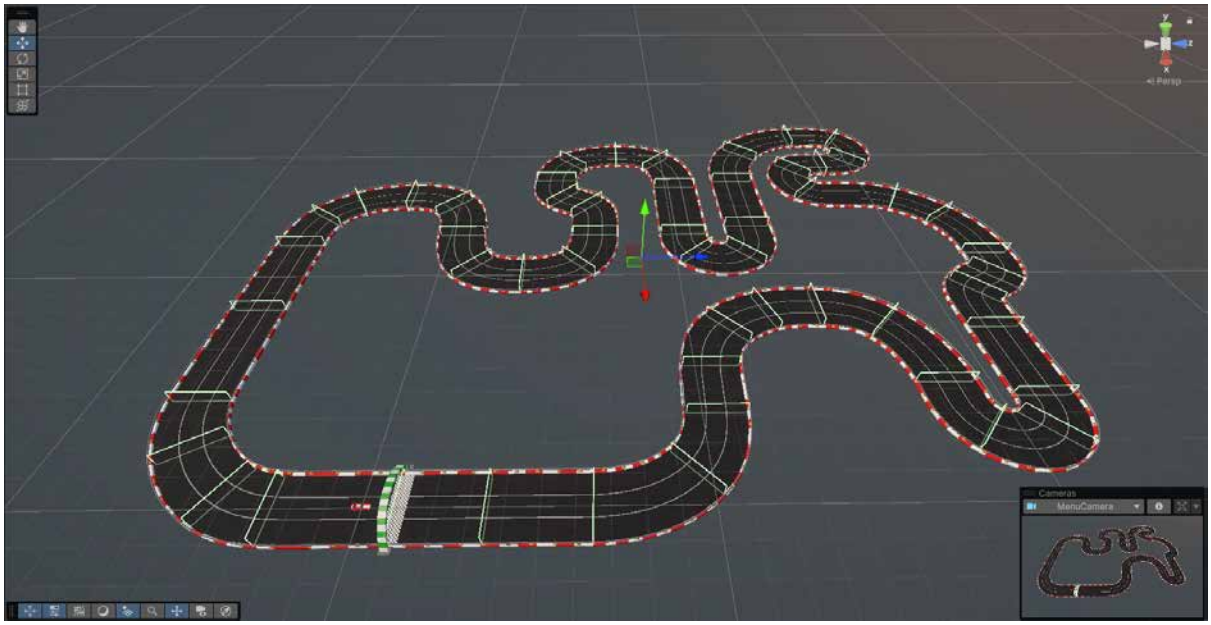


Рис. 5 Структура треку

Система контрольних точок:

- Розміщення checkpoint'ів кожні 5-10 метрів вздовж треку
- Використання для розрахунку прогресу та винагород
- Визначення напрямку руху для оптимальної траєкторії

Фізична модель автомобіля

Фізична модель автомобіля реалізована з використанням компонентів Unity Physics з наступними характеристиками:

Фізичні параметри:

- Маса: 1200 кг
- Аеродинамічний опір: 0.1
- Кутовий опір: 0.05
- Центр мас: (0, -0.5, 0.2)

Компоненти автомобіля:

- Rigidbody для фізичних взаємодій з середовищем
- Wheel Colliders для реалістичної поведінки коліс
- Box Collider для виявлення зіткнень з елементами треку
- Система підвіски з налаштовуваними параметрами

Параметри керування:

- Максимальний кут повороту: 27°
- Максимальна швидкість: 90 км/год
- Максимальна швидкість заднього ходу: 45 км/год
- Прискорення: налаштовується параметром accelerationMultiplier
- Гальмівна сила: 350 Н

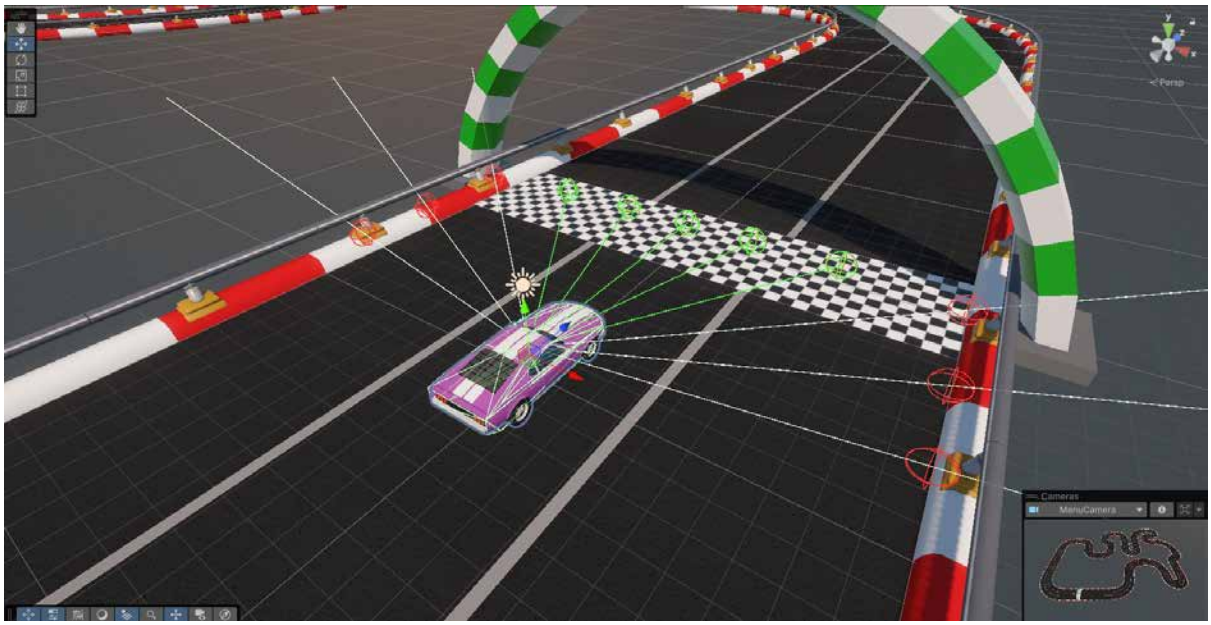


Рис. 6 Модель автомобіля

Система сенсорів та спостережень

Для забезпечення нейромережевого агента інформацією про навколишнє середовище реалізовано систему збору даних:

Основні спостереження, що передаються в нейромережу:

- Поточне положення автомобіля (Vector3)
- Кут повороту автомобіля (Quaternion)
- Поточна швидкість автомобіля (float)
- Напрямок до наступної контрольної точки (Vector3)
- Відстань до наступної контрольної точки (float)

Формат вхідних даних для нейронної мережі:

Input Vector Size: 9

- Position: 3 values (x, y, z)

- Rotation: 4 values (quaternion)

- Speed: 1 value (normalized 0-1)

- Direction to checkpoint: 3 values (normalized vector)

- Distance to checkpoint: 1 value (normalized 0-1)

2.3. Проектування експериментального модуля

Експериментальний модуль призначений для систематичного дослідження впливу різних архітектур нейронних мереж на ефективність керування віртуальним автомобілем. Він забезпечує автоматизацію процесу навчання, збір даних та аналіз результатів.

Архітектура модуля для порівняння нейромереж

Модуль включає компоненти для конфігурації, навчання та оцінки різних архітектур нейронних мереж:

Компоненти експериментального модуля:

- Configuration Manager - управляє параметрами навчання
- Experiment Controller - керує проведенням експериментів
- Training Pipeline - реалізує процес навчання
- Model Evaluator - оцінює продуктивність навчених моделей

Реалізація архітектур нейронних мереж:

```
# Архітектура 1×512
network_settings:
  hidden_units: 512
  num_layers: 1

# Архітектура 2×256
network_settings:
  hidden_units: 256
  num_layers: 2

# Архітектура 4×128
network_settings:
```

```
  hidden_units: 128
  num_layers: 4

# Архітектура 8×64
network_settings:
  hidden_units: 64
  num_layers: 8

# Архітектура 16×32
network_settings:
  hidden_units: 32
  num_layers: 16
```

Процес навчання:

- Загальна кількість кроків навчання: 4,000,000
- Розмір пакета (batch size): 1024
- Розмір буфера (buffer size): 10240
- Частота збереження моделей: кожні 50,000 кроків
- Частота оцінки: кожні 10,000 кроків

Система збору та аналізу статистики

Для ефективного порівняння різних архітектур нейромереж розроблено систему збору та аналізу статистичних даних:

Ключові метрики продуктивності:

- Сумарна винагорода за епізод (TotalReward)
- Відсоток завершення траси (CompletionPercentage)
- Час проходження кола (LapTime)
- Середня швидкість (AverageSpeed)
- Кількість зіткнень зі стінами (wallPenalties)
- Стабільність результатів (стандартне відхилення)

Реалізація збору даних:

```
// Фрагмент коду для збору даних про заїзд
LapData lapData = new LapData
{
    LapNumber = tracker.GetCurrentLapNumber(),
    CompletionPercentage = tracker.CalculateCompletionPercentage(),
    NetworkEffectiveness = 0f,
    Frames = tracker.GetCurrentFrames(),
    LapTime = tracker.GetCurrentLapTime(),
    AverageSpeed = tracker.CalculateAverageSpeed(),
    TotalReward = totalRewardAccumulated,
    IsCompleted = isFinished
};
```

```
// Збереження даних для подальшого аналізу
LapDataSaver.SaveLapToFile(lapData, reason);
```

Структура збережених даних:

```
json
{
  "LapNumber": 10,
  "CompletionPercentage": 100.0,
  "NetworkEffectiveness": 0.78,
  "LapTime": 45.2,
  "AverageSpeed": 67.4,
  "TotalReward": 150.5,
  "IsCompleted": true,
  "Frames": [
    {
      "Position": {"x": 0.0, "y": 0.0, "z": 0.0},
      "Rotation": {"x": 0.0, "y": 0.0, "z": 0.0, "w": 1.0},
      "Speed": 0.0
    },
    // ... інші кадри траєкторії
  ]
}
```

Візуалізація результатів навчання

Для аналізу результатів навчання різних архітектур нейромереж розроблено систему візуалізації:

Компоненти візуалізації:

- Графіки навчання (Learning Curves)
- Візуалізація траєкторій руху автомобіля
- Порівняльні діаграми продуктивності
- Таблиці з ключовими метриками

Режими візуалізації:

- Real-time візуалізація під час навчання
- Відтворення записаних траєкторій найкращих заїздів
- Порівняльний аналіз різних архітектур

2.4. Проектування користувацького інтерфейсу

Користувацький інтерфейс спроектований для забезпечення зручної взаємодії з системою та ефективної презентації результатів експериментів.

Головне меню

Головне меню системи забезпечує доступ до всіх основних функцій системи:

Основні розділи меню:

- Training Mode - запуск процесу навчання з вибором архітектури
- Race AI - режим змагання з навченими агентами
- Statistics - перегляд статистики навчання

- AI Replay - відтворення записаних заїздів
- Settings - налаштування системи

Рис. 7 Головне меню

Екрани статистики

Екрани статистики надають інформацію про результати навчання різних архітектур нейромереж:

Елементи екранів статистики:

- Selector для вибору архітектури
- Графіки продуктивності
- Таблиці з ключовими метриками
- Інструменти порівняльного аналізу

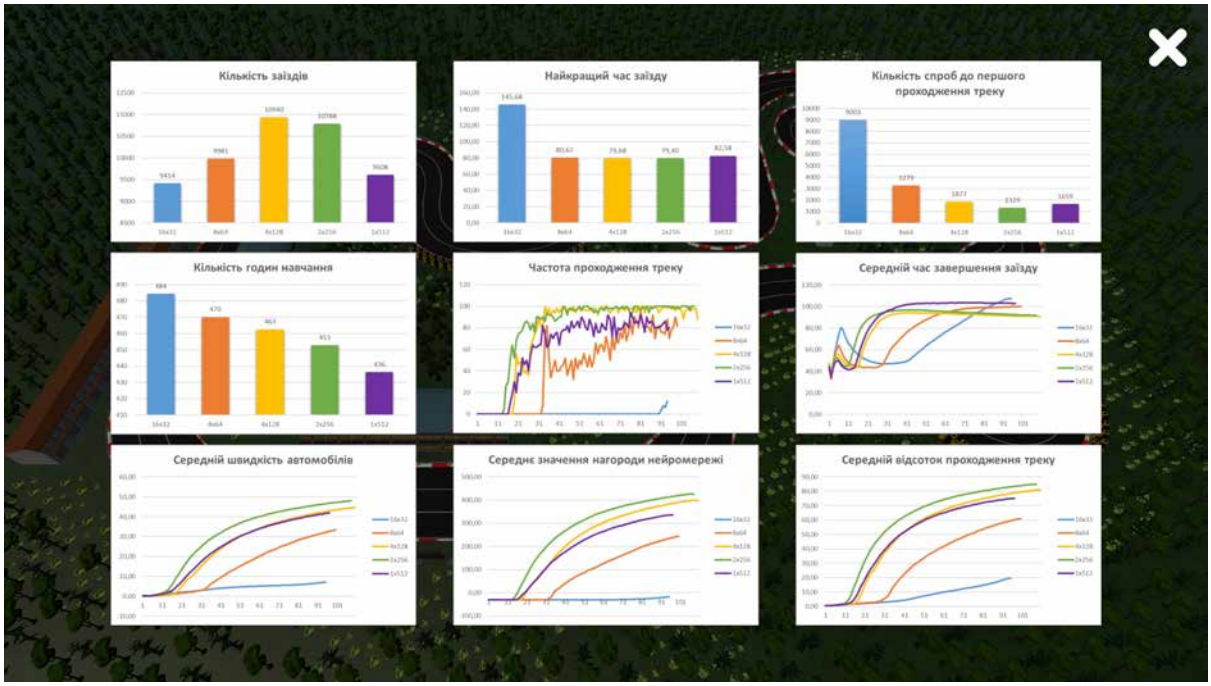


Рис. 8 Меню статистики

Ключові метрики для відображення:

- Average Reward - середня винагорода
- Completion Rate - відсоток завершених заїздів
- Average Lap Time - середній час проходження кола
- Collisions per Lap - кількість зіткнень за коло
- Learning Efficiency - ефективність навчання

Режими симуляції

Система підтримує різні режими симуляції для тестування та демонстрації навчених агентів:

Training Mode:

- Візуалізація процесу навчання

- Відображення поточних винагород
- Інформаційна панель з метриками

Replay Mode:

- Відтворення записаних траєкторій
- Керування швидкістю відтворення
- Можливість порівняння декількох заїздів

Race Mode:

- Змагання з навченими агентами
- Відображення статистики в реальному часі
- Порівняння результатів людини та ШІ

Рис. 9 Режим заїзду

Реалізація режиму відтворення:

```
// Фрагмент коду TrajectorySimulate для відтворення записаних траєкторій
private void UpdateObjectPosition(ObjectMovementData objData)
{
    // Перевіряємо, чи минув час стартової затримки
    if (objData.startDelaySeconds > 0)
    {
        objData.elapsedDelay += Time.fixedDeltaTime;
        if (objData.elapsedDelay >= objData.startDelaySeconds)
        {
            objData.startDelaySeconds = 0;
            objData.elapsedDelay = 0;
        }
        return;
    }

    // Визначаємо кількість кадрів для пропуску в залежності від швидкості
    int framesToSkip = Mathf.Max(1, Mathf.RoundToInt(objData.playbackSpeed));

    // Просуваємось вперед на відповідну кількість кадрів
    objData.currentFrameIndex += framesToSkip;
```

```
        // Перевіряємо, чи досягли кінця анімації
        if (objData.currentFrameIndex >= objData.loadedLapData.Frames.Length)
        {
            if (objData.loopPlayback)
            {
                // Перезапуск анімації, якщо включено зациклювання
                objData.currentFrameIndex = objData.currentFrameIndex % objData.loadedLapData.Frames.Length;
            }
            else
            {
                // Зупинка анімації
                objData.playing = false;
                return;
            }
        }

        // Застосовуємо позицію та обертання з даних кадру
        FrameData frame = objData.loadedLapData.Frames[objData.currentFrameIndex];
        objData.objectToMove.transform.position = frame.Position;
```

```
        objData.objectToMove.transform.rotation = frame.Rotation;
    }
```

3. РЕАЛІЗАЦІЯ ТА ДОСЛІДЖЕННЯ СИСТЕМИ

3.1. Розробка симуляційного середовища

Реалізація симуляційного середовища є ключовим етапом у створенні системи керування віртуальними об'єктами на основі багатошарових нейромереж. Середовище розроблено з використанням платформи Unity 2021.3 LTS, яка забезпечує необхідні інструменти для фізичного моделювання, візуалізації та інтеграції з ML-Agents.

3.1.1. Реалізація гоночного треку в Unity

Гоночний трек реалізовано як замкнуту трасу з різними типами поворотів та прямих ділянок. Основні елементи реалізації:

- Модульна структура: Траса складається з окремих сегментів, що дозволяє легко налаштовувати складність та конфігурацію
- Система контрольних точок: Вздовж треку розміщено серію невидимих тригерів, які використовуються для відстеження прогресу автомобіля та підрахунку винагород
- Обмежувачі (colliders): По обидва боки траси розміщено бар'єри, що визначають межі дозволеної зони руху

Ключовим компонентом треку є система контрольних точок, реалізована через клас Tracker:

```
private void OnTriggerEnter(Collider other)
{
    // Перевіряємо тільки для чекпоінтів
    if (!other.CompareTag("Checkpoint"))
    {
        return; // Ігноруємо не-чекпоінти
    }

    if (IsValidNextCheckpoint(other.gameObject))
    {
        // Код для проходження звичайного чекпоінту
        currentCheckpointIndex++;
        if (currentCheckpointIndex >= checkpoints.Length)
        {
            currentCheckpointIndex = 0;
        }

        // Викликаємо метод OnCheckpointEnter для нагороди за чекпоінт
        carAgent.OnCheckpointEnter();
```

```
    }
}
```

Реалізована система чекпоінтів дозволяє ефективно відстежувати прогрес автомобіля на трасі, розраховувати відсоток проходження та ідентифікувати завершення кола. Ця інформація є ключовою для формування сигналів винагороди під час навчання нейронних мереж.

Рис. 10 Реалізація гоночного треку

3.1.2. Налаштування фізичної моделі автомобіля

Для реалізації точної та реалістичної фізичної моделі автомобіля використано вбудований фізичний рушій Unity та спеціалізовані компоненти для моделювання поведінки транспортних засобів:

- Базова фізична модель: Використано компоненти Rigidbody та колайдери для реалістичного моделювання маси, інерції та зіткнень

- Система керування: Реалізовано через компоненти PrometeoCarController та PlayerCarController, що відповідають за керування рухом автомобіля
- Налаштування параметрів: Тонко налаштовано параметри фізичної моделі, включаючи:
    - Максимальна швидкість: 90 км/год
    - Максимальна швидкість заднього ходу: 45 км/год
    - Прискорення: регульоване через параметр accelerationMultiplier
    - Кути повороту керма: до 27°
    - Гальмівна сила: 350 Н

Ключова частина реалізації фізичної моделі зосереджена в класі PrometeoCarController, який керує всіма аспектами руху автомобіля. Основні методи та функціональність цього класу забезпечують реалістичну поведінку транспортного засобу, включаючи прискорення, гальмування та керування. Детальна реалізація цього класу наведена в Додатку А.

Для агентів машинного навчання реалізовано окремий інтерфейс керування через клас CarAgent, який трансформує вихідні сигнали нейронної мережі в команди керування автомобілем:

```
public override void OnActionReceived(ActionBuffers actions)
{
    int forwardMove = actions.DiscreteActions[0] - 1;
    int rightMove = actions.DiscreteActions[1] - 1;
    controller.InputAIParams(forwardMove, rightMove);
```

```
        float speedFactor = Mathf.Clamp01(controller.carSpeed / 120f);
        float timeReward = timeRewardPerSecond * (1.0f - speedFactor) *
Time.deltaTime;

        AddReward(timeReward);
        totalRewardAccumulated += timeReward;
        timeRewards += timeReward;

        // Перевірка порогу винагороди після кожної дії
        CheckRewardThreshold();
    }
```

3.1.3. Імплементація системи детекції зіткнень

Для ефективного навчання агентів критично важливою є система детекції зіткнень, яка дозволяє визначати помилки керування та формувати відповідні сигнали винагороди. Система включає:

- Детекція зіткнень зі стінами: Визначає зіткнення автомобіля з бар'єрами та нараховує штрафи
- Перевірка сходження з траси: Відстежує, чи знаходиться автомобіль в межах дозволеної зони
- Обробка інтенсивності зіткнень: Розраховує силу зіткнення для пропорційного штрафу

Реалізація системи детекції зіткнень відбувається через обробники подій OnCollisionEnter у класі CarAgent:

```
public void OnWallEnter()
{
    // Ігноруємо зіткнення, якщо минуло менше 0.2 секунди з моменту останнього
    if (Time.time - lastWallHitTime < 0.2f)
        return;

    lastWallHitTime = Time.time;
    float wallReward = wallPenaltyValue;
    AddReward(wallReward);
    totalRewardAccumulated += wallReward;
    wallPenalties += wallReward;

    Debug.Log($"Зіткнення зі стіною! Штраф: {wallReward}. Поточна загальна винагорода: {totalRewardAccumulated:F2}");

    // Перевірка порогу винагороди після отримання штрафу
    CheckRewardThreshold();
}
```

Використання системи детекції зіткнень дозволяє ефективно навчати нейронні мережі уникати перешкод та дотримуватися безпечної траєкторії руху.

3.2. Реалізація агентів з різними архітектурами

Для дослідження впливу архітектури нейронних мереж на ефективність керування віртуальними об'єктами реалізовано п'ять різних конфігурацій глибоких нейронних мереж. Всі архітектури використовують однаковий набір вхідних параметрів та вихідних дій, але відрізняються кількістю шарів та нейронів у кожному шарі.

3.2.1. Загальна структура нейромережевого агента

Основою для всіх експериментальних архітектур є клас CarAgent, який наслідує базовий клас Agent з фреймворку ML-Agents. Ключова функціональність агента включає:

- Збір спостережень: Отримання інформації про стан автомобіля та навколишнє середовище
- Обробка дій: Інтерпретація виходів нейронної мережі в конкретні команди керування
- Система винагород: Розрахунок винагород на основі прогресу та помилок
- Управління епізодами: Початок, завершення та скидання стану епізоду навчання

Система збору спостережень включає наступні параметри:

```
    public override void CollectObservations(VectorSensor sensor)
    {
        sensor.AddObservation(transform.position);       // Позиція автомобіля (3 значення)
        sensor.AddObservation(transform.rotation);       // Обертання автомобіля (4 значення)
        sensor.AddObservation(controller.carSpeed);      // Швидкість автомобіля (1 значення)

        // Додатково додаємо спостереження за напрямком до наступного чекпоінту
        if (tracker != null)
        {
            Vector3 directionToNextCheckpoint = tracker.GetDirectionToNextCheckpoint();
            sensor.AddObservation(directionToNextCheckpoint);   // Напрямок до наступного чекпоінту (3 значення)

            // Також додаємо відстань до наступного чекпоінту
            float distanceToNextCheckpoint = tracker.GetDistanceToNextCheckpoint();
            sensor.AddObservation(distanceToNextCheckpoint);    // Відстань (1 значення)
        }
```

```
}
```

Загальна розмірність вектора спостережень становить 12 значень, що забезпечує достатню інформацію для прийняття рішень нейронною мережею.

3.2.2. Архітектури досліджуваних нейронних мереж

У рамках дослідження реалізовано п'ять різних архітектур нейронних мереж, що відрізняються глибиною (кількістю шарів) та шириною (кількістю нейронів у шарі):

Архітектура	Кількість шарів	Нейронів на шар	Загальна кількість нейронів	Загальна кількість параметрів *
1×512	1	512	512	6,656
2×256	2	256	512	6,912
4×128	4	128	512	6,784
8×64	8	64	512	6,720
16×32	16	32	512	6,688

*з урахуванням вхідного шару (12 нейронів) та вихідного шару (2 дії)

Всі архітектури налаштовані з однаковими гіперпараметрами навчання для забезпечення об'єктивного порівняння:

```
# Спільні гіперпараметри для всіх архітектур
hyperparameters:
  batch_size: 1024          # Розмір пакета для навчання
  buffer_size: 10240        # Розмір буфера досвіду
  learning_rate: 3.0e-4     # Швидкість навчання
```

```
  beta: 5.0e-3                # Параметр регуляризації ентропії
  epsilon: 0.2                # Параметр PPO-відсікання
  lambd: 0.95                 # Параметр компромісу зміщення/дисперсії
  num_epoch: 3                # Кількість епох на кожній ітерації
  learning_rate_schedule: linear  # Графік зміни швидкості навчання
```

Кількість епох навчання обрана як 3 на основі попередніх експериментів, які показали, що збільшення до 5-10 епох не покращує якість навчання суттєво, але збільшує час тренування[11].

Архітектури відрізняються лише налаштуваннями нейронної мережі, як показано в прикладі конфігурації для архітектури 2×256:

```
# Приклад конфігурації для архітектури 2×256
network_settings:
  normalize: false
  hidden_units: 256
  num_layers: 2
  vis_encode_type: simple
```

Використання фреймворку ML-Agents дозволяє легко змінювати конфігурацію нейронної мережі без зміни коду агента, що забезпечує гнучкість та швидке проведення експериментів.

3.3. Проведення експериментів та збір даних

3.3.1. Методика експерименту

Для систематичного дослідження впливу архітектури нейронних мереж на ефективність керування віртуальними об'єктами розроблено методику експерименту, що забезпечує об'єктивне порівняння різних конфігурацій. План експерименту включає:

1. Підготовчий етап:
    - Налаштування однакового гоночного треку для всіх архітектур
    - Калібрування системи винагород для забезпечення порівнянності результатів
    - Встановлення однакових початкових умов та параметрів середовища
2. Навчання:
    - Незалежне навчання кожної архітектури в ідентичних умовах
    - Загальна тривалість навчання: 1 мільйон кроків для кожної архітектури
    - Збереження контрольних точок моделі кожні 100,000 кроків
    - Фіксація метрик продуктивності кожні 10,000 кроків
3. Тестування:
    - Оцінка навчених моделей на тестовій трасі (100 заїздів для кожної архітектури)
    - Запис детальної статистики для кожного заїзду

- Порівняльний аналіз ключових метрик продуктивності

Для забезпечення статистичної значущості результатів кожен експеримент проводився з трьома різними значеннями seed, що дозволяє оцінити стабільність навчання та варіативність результатів.

3.3.2. Процес навчання агентів

Процес навчання реалізовано з використанням алгоритму Proximal Policy Optimization (PPO), який є одним з найефективніших алгоритмів навчання з підкріпленням для керування безперервними діями. Навчання проводилося в паралельному режимі з використанням декількох копій середовища для прискорення збору досвіду.

Для автоматизації процесу навчання різних архітектур розроблено спеціальний менеджер експериментів, який забезпечує:

- Послідовний запуск навчання для кожної архітектури
- Моніторинг процесу навчання та збір статистики
- Збереження результатів в структурованому форматі для подальшого аналізу

Процес збору та аналізу даних автоматизовано за допомогою класу LapDataSaver, який зберігає детальну інформацію про кожен заїзд. Детальна реалізація методів збереження даних наведена в Додатку А.

Для кожного заїзду зберігається наступна інформація:

- Номер заїзду та причина завершення
- Відсоток проходження траси

- Час заїзду та середня швидкість
- Загальна отримана винагорода
- Детальна траєкторія руху (позиція, обертання, швидкість)

Ці дані дозволяють провести детальний аналіз ефективності різних архітектур нейронних мереж та визначити оптимальну конфігурацію для задачі керування віртуальним автомобілем.

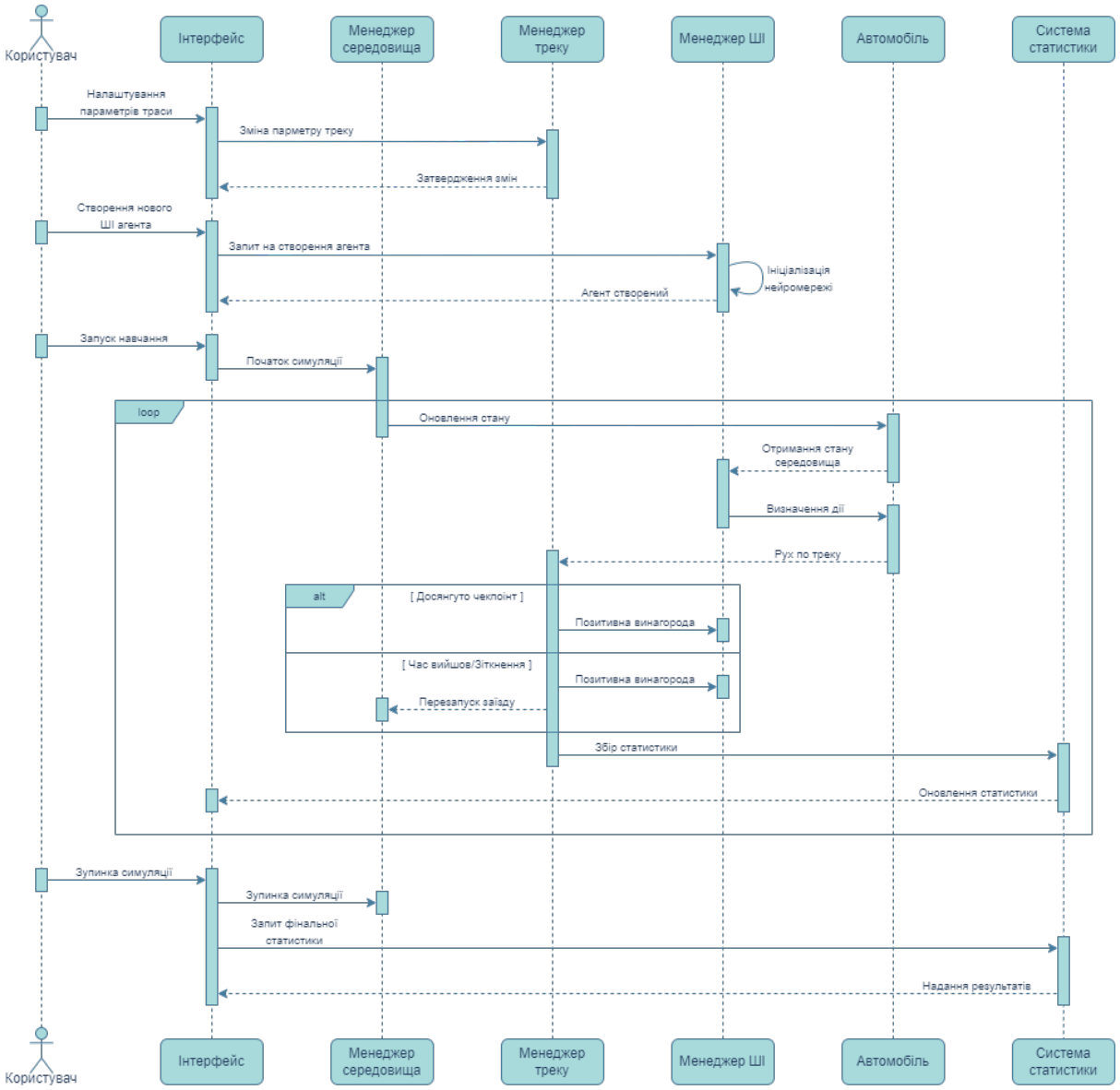


Рис. 11 Діаграма послідовності

3.3.3. Збір метрик продуктивності

Для оцінки ефективності різних архітектур нейронних мереж розроблено систему збору та аналізу ключових метрик продуктивності. Зібрані метрики включають:

1. Прямі показники навчання:
    - Кумулятивна винагорода за епізод
    - Рівень успішності (відсоток завершених кіл)
    - Кількість ітерацій до досягнення певного рівня продуктивності
2. Показники якості керування:
    - Час проходження кола (найкращий, середній, варіативність)
    - Траєкторія руху (відхилення від оптимальної лінії)
    - Плавність управління (частота та амплітуда рухів керма)
3. Стабільність:
    - Кількість зіткнень за коло
    - Частота виїздів за межі треку
    - Консистентність часу проходження кола

Для зберігання та аналізу метрик розроблено спеціалізовану базу даних, що дозволяє структуровано зберігати та ефективно аналізувати великі обсяги даних експериментів:

```
// Структура для зберігання даних про епізод навчання
public class EpisodeMetrics
{
```

```
    public string AgentArchitecture;  // Архітектура нейронної мережі
    public string TrackName;          // Назва траси
    public int SeedValue;             // Значення seed для відтворюваності
    public int EpisodeId;             // Ідентифікатор епізоду
    public float Duration;            // Тривалість епізоду в секундах
    public float FinalReward;         // Підсумкова винагорода
    public bool Successful;           // Чи було коло завершено успішно
    public int CheckpointsPassed;     // Кількість пройдених контрольних точок
    public int CollisionCount;        // Кількість зіткнень
    public int OffTrackCount;         // Кількість виїздів за межі траси
    public float LapTime;             // Час проходження кола (якщо завершено)
}
```

Для візуалізації та аналізу зібраних метрик розроблено систему генерації звітів, яка створює графіки, діаграми та таблиці з порівняльним аналізом різних архітектур.

3.3.4. Аналіз стабільності та швидкості навчання

Для аналізу стабільності та швидкості навчання використано спеціалізовані алгоритми обробки зібраних метрик. Ключовими показниками виступали:

- Швидкість збіжності - кількість ітерацій, необхідних для досягнення заданого рівня продуктивності (наприклад, успішне проходження кола без зіткнень)
- Варіабельність навчання - стандартне відхилення винагороди між різними запусками з різними seed-значеннями
- Стійкість до перенавчання - динаміка продуктивності на тренувальній та валідаційній трасах

Для аналізу цих показників розроблено спеціальний компонент LearningAnalyzer, детальна реалізація якого наведена в Додатку А.

Для візуалізації процесу навчання розроблено інтерактивні графіки, що відображають динаміку ключових метрик протягом часу. Особлива увага приділена візуалізації learning curves для різних архітектур, що дозволяє наочно порівняти швидкість та стабільність навчання.

Результати аналізу показують значні відмінності у швидкості та стабільності навчання різних архітектур, що детально обговорюється в розділі 4 "Аналіз результатів дослідження".

3.4. Реалізація інтерактивних режимів

Для наочної демонстрації результатів навчання та порівняння різних архітектур нейронних мереж розроблено інтерактивні режими взаємодії з системою.

3.4.1. Режим перегляду найкращих заїздів

Режим перегляду найкращих заїздів дозволяє користувачу спостерігати за поведінкою навчених агентів та порівнювати різні архітектури. Основні функції класів ReplayManager та TrajectorySimulate наведені в Додатку А.

Для запису найкращих заїздів використовується спеціальна система, що фіксує стан автомобіля та середовища кожен кадр.

Режим перегляду надає наступні можливості:

- Відтворення записаних траєкторій з можливістю регулювання швидкості
- Одночасне відтворення декількох траєкторій для порівняння різних архітектур
- Відображення додаткової інформації про заїзд (швидкість, винагорода, тощо)
- Експорт та імпорт записів для обміну та подальшого аналізу

3.4.2. Режим змагання з навченими агентами

Режим змагання дозволяє користувачу керувати автомобілем та змагатися з навченими нейронними мережами різних архітектур. Цей режим забезпечує:

- Управління автомобілем користувачем за допомогою клавіатури
- Одночасну присутність на трасі автомобілів, керованих різними архітектурами нейромереж

- Відстеження та порівняння продуктивності людини та ШІ в реальному часі

Реалізація режиму змагання базується на класі RaceManager, який відповідає за:

- Налаштування гонки з вибраними архітектурами
- Створення та розміщення автомобілів на стартових позиціях
- Відстеження прогресу та визначення переможця

Для забезпечення управління автомобілем користувачем реалізовано спеціальний контролер, що трансформує вхідні дані клавіатури в команди керування автомобілем.

3.4.3. Система відображення статистики

Для візуалізації зібраних даних та порівняння продуктивності різних архітектур розроблено інтерактивну систему відображення статистики.

Система включає ряд графіків та діаграм для порівняння різних аспектів продуктивності:

- Learning Curves - відображення прогресу навчання для кожної архітектури
- Success Rate Comparison - порівняння відсотку успішних заїздів
- Lap Time Distribution - розподіл часу проходження кола для різних архітектур

- Convergence Speed - порівняння швидкості збіжності різних архітектур

Система надає можливості для:

- Інтерактивного дослідження результатів експериментів
- Фільтрації даних за різними параметрами (архітектура, траса, метрика)
- Експорту результатів в різних форматах для подальшого аналізу
- Генерації комплексних звітів з ключовими висновками

4. АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

4.1. Порівняльний аналіз ефективності архітектур

Проведене дослідження різних архітектур нейронних мереж для задачі керування віртуальними об'єктами дозволило виявити суттєві відмінності в їх ефективності. Аналіз метрик продуктивності продемонстрував чіткі закономірності та залежності між архітектурними параметрами та результатами навчання.

4.1.1. Кількість заїздів та ефективність тренування

Рис. 12

Аналізуючи дані про кількість заїздів, виконаних різними архітектурами (Рис. 12), можна відзначити, що архітектури з середньою кількістю шарів та нейронів (4×128 та 2×256) виконали найбільшу кількість заїздів – 10940 та 10788 відповідно. Це свідчить про оптимальний баланс між швидкістю обчислень та здатністю до навчання у цих архітектур.

Архітектури з крайніми параметрами (16×32 та 1×512) показали найнижчі результати за кількістю заїздів – 9414 та 9608 відповідно. Для найглибшої архітектури (16×32) це може бути пояснено обчислювальними витратами на проходження сигналу через численні шари, що сповільнює процес прийняття рішень. Для найпростішої архітектури (1×512) причиною можуть бути обмеження у здатності вивчати складні залежності, що призводить до частіших помилок і перезапусків заїздів.

Архітектура 8×64 продемонструвала проміжний результат (9981 заїздів), що вказує на баланс між глибиною мережі та її обчислювальною ефективністю.

4.1.2. Аналіз часу проходження треку

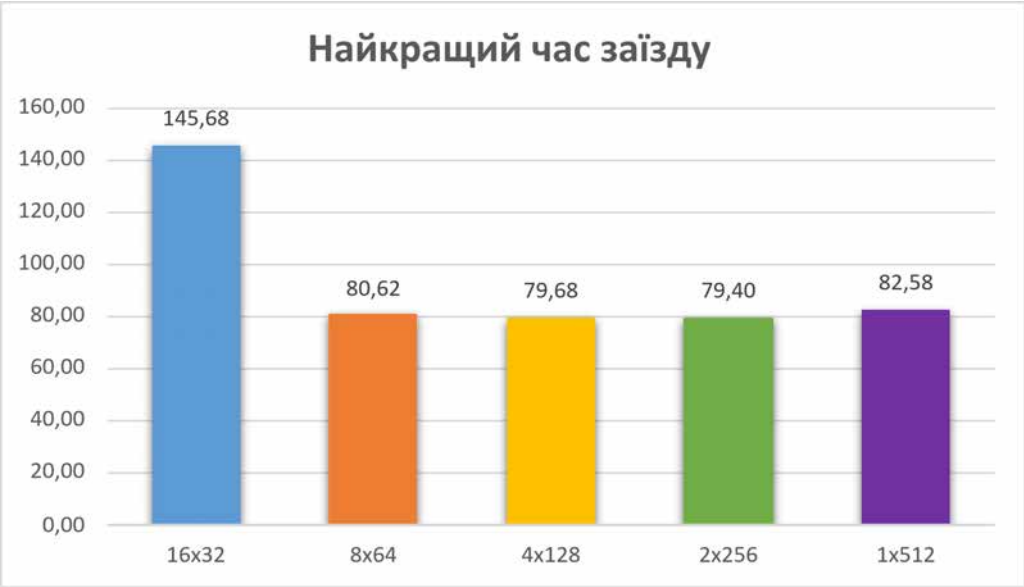


Рис. 13

Найкращий час заїзду (Рис. 13) демонструє значні відмінності між архітектурами. Найшвидший час показала архітектура 2×256 (79,40 с), що свідчить про її оптимальну здатність вивчати ефективні стратегії керування. Архітектури 4×128 та 8×64 показали близькі результати (79,68 с та 80,62 с відповідно).

Архітектура 1×512 продемонструвала дещо гірший результат (82,58 с), що може вказувати на обмежену здатність вивчати складні послідовності дій через відсутність глибини.

Особливої уваги заслуговує найглибша архітектура 16×32, яка показала значно гірший час (145,68 с) порівняно з іншими. Це може бути пояснено проблемою зникаючого градієнта, що ускладнює ефективне навчання глибоких мереж, а також можливим перенавчанням на специфічних патернах без здатності до узагальнення.

4.1.3. Швидкість навчання та адаптації

Рис. 14

Аналіз кількості спроб до першого проходження треку (Рис. 14) демонструє драматичні відмінності у швидкості навчання різних архітектур. Архітектура 2×256 показала найкращий результат, потребуючи лише 1329 спроб для успішного проходження треку. Це свідчить про її оптимальну здатність до швидкого навчання.

Архітектури 1×512 та 4×128 показали дещо гірші, але все ще конкурентоспроможні результати (1659 та 1877 спроб відповідно).

Архітектура 8×64 потребувала значно більше спроб (3279), що вказує на повільніше навчання через більшу глибину мережі.

Найбільш разючий результат показала архітектура 16×32, якій знадобилося 9003 спроби – майже в 7 разів більше, ніж архітектурі 2×256. Це підтверджує, що надмірна глибина мережі при малій кількості нейронів на шар суттєво ускладнює процес навчання для даної задачі.

4.1.4. Стабільність результатів

Рис. 15

Аналіз частоти проходження треку (Рис. 15) демонструє динаміку зміни успішності проходження з часом. Архітектури 2×256 та 4×128 показали найшвидшу конвергенцію до високих показників (близько 100% успішних заїздів) і найстабільніші результати протягом усього періоду навчання.

Архітектура 1×512 досягла стабільного рівня близько 80% успішних заїздів, що дещо нижче, ніж у лідерів, але все ж прийнятно.

Архітектура 8×64 продемонструвала повільнішу конвергенцію, досягнувши прийнятного рівня лише після 80 епох навчання.

Найглибша архітектура 16×32 показала найгірші результати, досягнувши лише близько 10% успішних заїздів наприкінці періоду навчання, що свідчить про суттєві проблеми у навчанні та стабільності для надто глибоких мереж з малою кількістю нейронів на шар.

4.2. Вплив глибини та ширини мережі на продуктивність

4.2.1. Аналіз кривих навчання

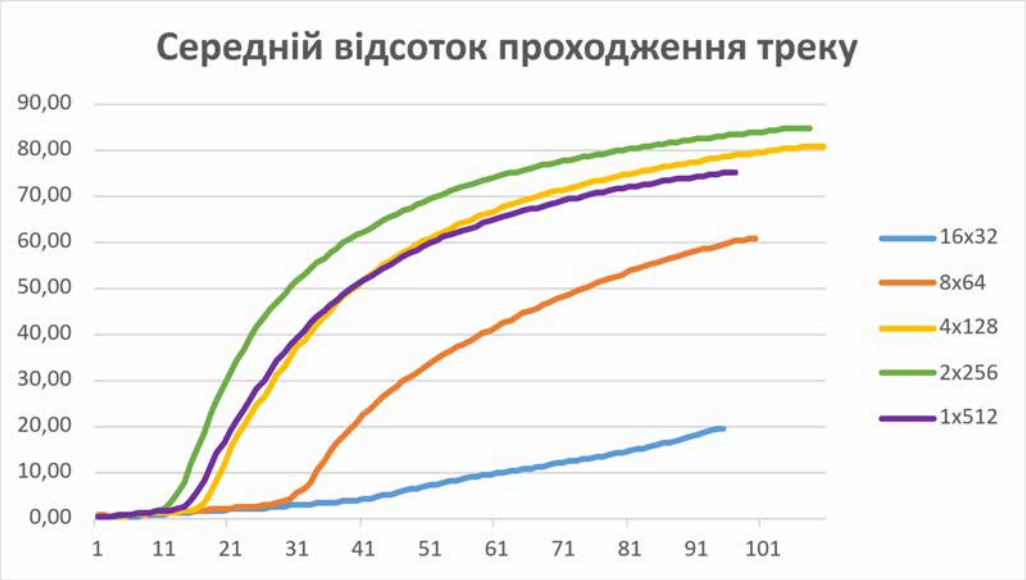


Рис. 16

Аналіз середнього відсотка проходження треку (Рис. 16) дозволяє оцінити швидкість прогресу навчання для різних архітектур. Крива архітектури 2×256 демонструє найшвидше зростання та досягає найвищого фінального значення (близько 85%), що підтверджує її оптимальність для даної задачі.

Архітектури 4×128 та 1×512 показують схожі криві навчання, але з повільнішою конвергенцією та нижчим кінцевим результатом (близько 75-80%).

Архітектура 8×64 демонструє значно повільніше навчання та досягає лише близько 60% наприкінці тренування.

Найглибша архітектура 16×32 показує мінімальний прогрес, досягаючи лише 20% проходження треку після 100 епох навчання, що свідчить про

серйозні обмеження у здатності вивчати складні залежності при такій конфігурації.

4.2.2. Аналіз часових характеристик

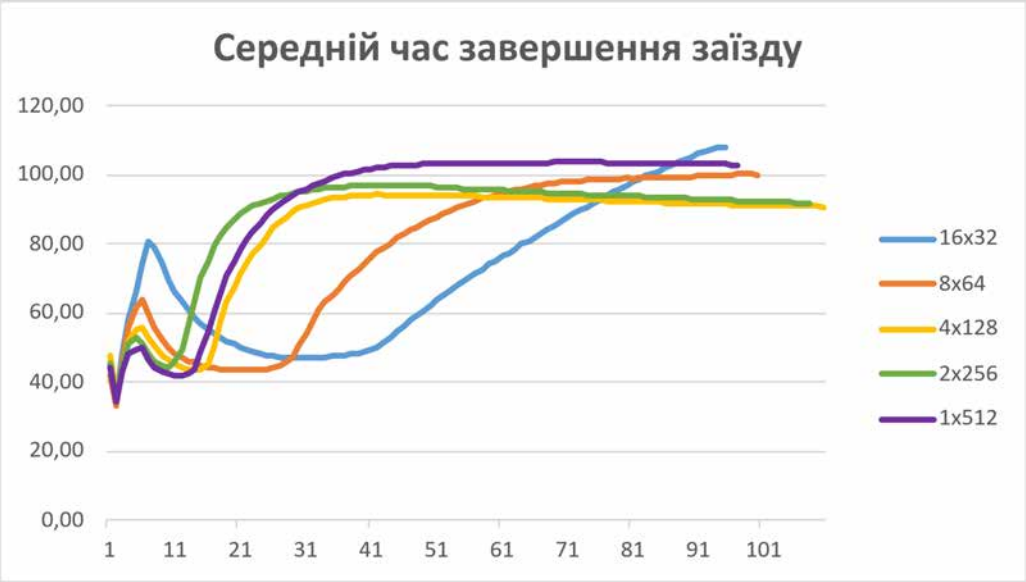


Рис. 17

Середній час завершення заїзду (Рис. 17) показує, як змінювалася ефективність керування протягом навчання. Архітектури 2×256 та 4×128 швидко досягли оптимальних показників (близько 90 секунд) і зберігали їх стабільними.

Архітектура 1×512 демонструє повільнішу конвергенцію та гірший кінцевий результат (близько 100-105 секунд).

Архітектури 8×64 та 16×32 показують найгірші результати, з нестабільною динамікою та високими фінальними значеннями часу заїзду (понад 100 секунд), що свідчить про субоптимальні стратегії керування.

4.2.3. Аналіз динаміки винагороди

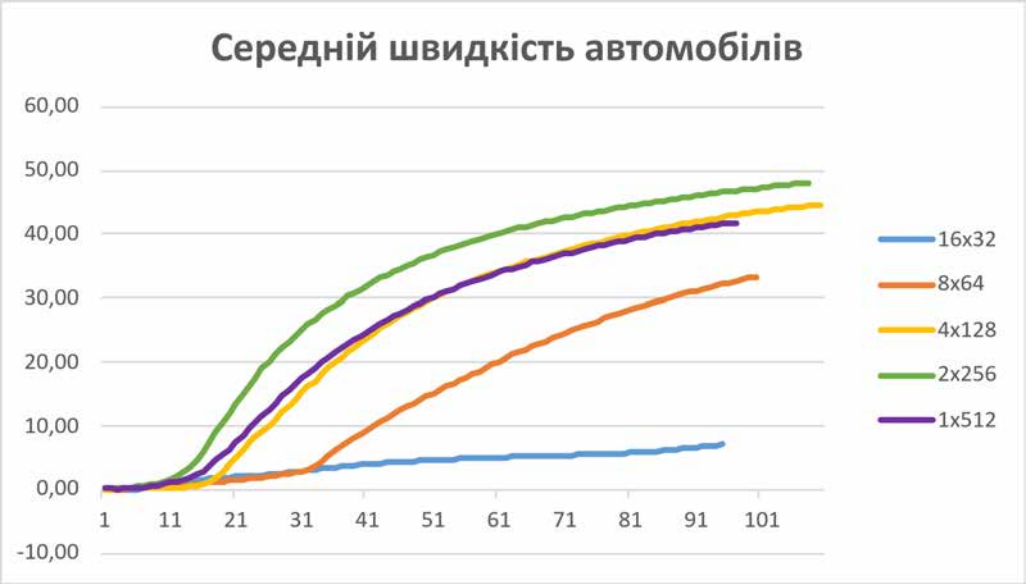


Рис. 18

Середнє значення нагороди нейромережі (Рис. 18) є інтегральним показником ефективності навчання. Архітектура 2×256 демонструє найшвидше зростання та найвищі фінальні значення винагороди (понад 400), що підтверджує її перевагу над іншими архітектурами.

Архітектура 4×128 показує подібну, але дещо повільнішу динаміку, досягаючи винагороди близько 400.

Архітектура 1×512 демонструє помірне зростання, досягаючи значення близько 350.

Архітектура 8×64 показує повільне зростання з кінцевим значенням близько 250.

Найглибша архітектура 16×32 демонструє мінімальний прогрес, не перевищуючи винагороди 50 навіть після 100 епох навчання, що свідчить про фундаментальні проблеми з навчанням такої глибокої мережі для даної задачі.

4.2.4. Аналіз швидкісних характеристик

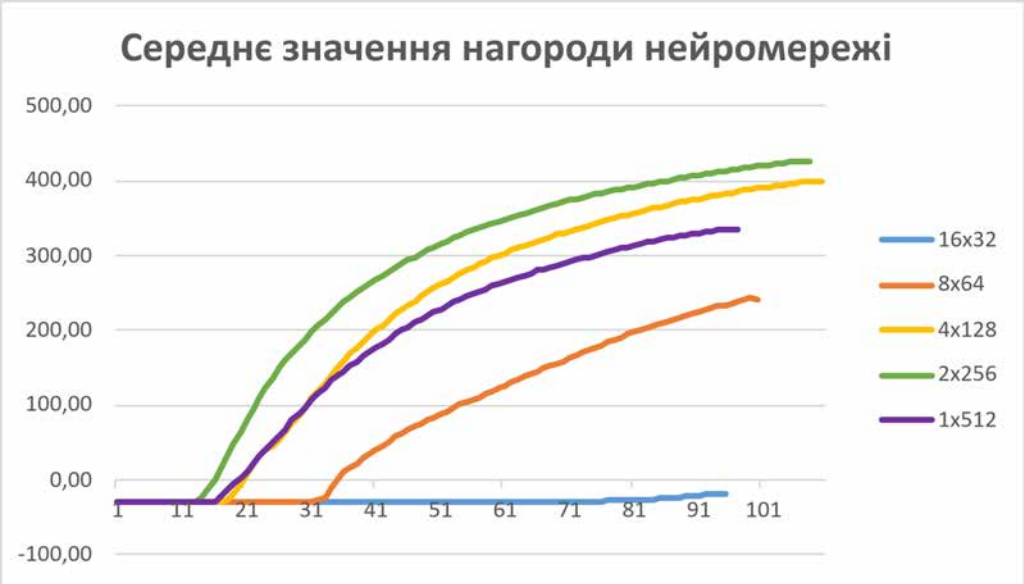


Рис. 19

Середня швидкість автомобілів (Рис. 19) є показником якості керування. Архітектура 2×256 демонструє найкращі результати, досягаючи найвищої середньої швидкості (близько 48 одиниць), що свідчить про ефективне керування та оптимальне проходження треку.

Архітектури 4×128 та 1×512 показують дещо нижчі, але близькі результати (43-45 одиниць).

Архітектура 8×64 демонструє значно нижчу середню швидкість (близько.33 одиниць), що вказує на субоптимальне керування.

Найглибша архітектура 16×32 показує мінімальну середню швидкість (менше 10 одиниць), що підтверджує її нездатність ефективно керувати автомобілем на треку.

4.3. Рекомендації щодо вибору архітектури

На основі проведеного дослідження можна сформулювати наступні рекомендації щодо вибору архітектури нейронної мережі для задач керування віртуальними об'єктами:

4.3.1. Оптимальний баланс глибини та ширини

Результати дослідження чітко демонструють, що найкращим варіантом з досліджених архітектур є архітектура з помірною глибиною та достатньою шириною – 2×256. Така конфігурація забезпечує:

1. Найшвидше навчання (мінімальна кількість спроб до першого успішного проходження)
2. Найвищу частоту успішних заїздів (близько 100%)
3. Найкращий час проходження треку
4. Найвищу середню швидкість на треку
5. Максимальну кумулятивну винагороду

Ця архітектура забезпечує оптимальний компроміс між здатністю вивчати складні залежності (завдяки двом шарам) та обчислювальною ефективністю (завдяки помірній загальній кількості параметрів).

4.3.2. Критерії вибору архітектури для різних сценаріїв

При виборі архітектури для конкретних застосувань рекомендується керуватися наступними критеріями:

1. **Для простих трас з передбачуваним рельєфом:**

- Рекомендована архітектура: 1×512 або 2×256
- Обґрунтування: Такі архітектури забезпечують швидке навчання та ефективне виконання для простих сценаріїв
- Переваги: Менша обчислювальна складність, швидше навчання, більша енергоефективність

2. **Для трас середньої складності:**
   - Рекомендована архітектура: 2×256 або 4×128
   - Обґрунтування: Баланс між здатністю до вивчення складних патернів та ефективністю
   - Переваги: Стабільна робота, висока якість керування, хороша здатність до узагальнення
3. **Для складних трас з непередбачуваними елементами:**
   - Рекомендована архітектура: 4×128
   - Обґрунтування: Достатня глибина для вивчення складних залежностей при збереженні прийнятної швидкості навчання
   - Переваги: Краща адаптивність до нових умов, стійкість до варіацій параметрів середовища
4. **Для ресурсообмежених систем:**
   - Рекомендована архітектура: 2×256 (або 4×128 з меншою частотою оновлення)
   - Обґрунтування: Оптимальний баланс між якістю керування та обчислювальними вимогами
   - Переваги: Менші вимоги до пам'яті, менше енергоспоживання

4.3.3. Практичні рекомендації

1. **Уникати надмірної глибини:** Результати дослідження чітко показують, що надто глибокі мережі (особливо 16×32) не підходять для задач керування віртуальними об'єктами в реальному часі. Вони страждають від проблеми зникаючого градієнта, повільно навчаються та показують низьку продуктивність.
2. **Віддавати перевагу збалансованим архітектурам:** Мережі з 2-4 шарами та 128-256 нейронами на шар демонструють найкращий баланс між швидкістю навчання, якістю керування та обчислювальною ефективністю.
3. **Враховувати складність задачі:** Для простіших задач керування достатньо мережі з 1-2 шарами. Для складніших сценаріїв можна збільшити глибину до 4 шарів, але не більше, щоб уникнути проблем з навчанням.
4. **Оптимізувати загальну кількість параметрів:** Результати показують, що для ефективного керування віртуальним автомобілем оптимальна кількість параметрів становить близько 130-260 тисяч (мережі 2×256 та 4×128), що забезпечує достатню ємність моделі без надмірної складності.
5. **Застосовувати техніки регуляризації:** Для запобігання перенавчанню рекомендується використовувати dropout, L2-регуляризацію та нормалізацію батчів, особливо для ширших мереж (1×512 та 2×256).

Ці рекомендації дозволять розробникам систем керування віртуальними об'єктами ефективно вибирати архітектуру нейронної мережі відповідно до конкретних вимог задачі, обчислювальних ресурсів та бажаного балансу між швидкістю навчання та якістю керування.

ВИСНОВКИ

У даній бакалаврській кваліфікаційній роботі розроблено програмне забезпечення системи керування віртуальними об'єктами на основі багатошарових нейромереж та проведено порівняльний аналіз ефективності різних архітектур нейронних мереж для задачі автономного керування віртуальним автомобілем.

Основні результати роботи:

1. **Розроблено програмний комплекс** на базі Unity та ML-Agents, що включає симуляційне середовище з гоночним треком, систему керування віртуальним автомобілем, модуль для навчання агентів з різними архітектурами нейронних мереж, систему збору та аналізу статистики, а також інтерактивний інтерфейс для візуалізації результатів та взаємодії з навченими моделями.

2. **Проведено систематичне дослідження** впливу архітектурних параметрів нейронних мереж на ефективність навчання та якість керування. Порівняльний аналіз п'яти різних архітектур (1×512, 2×256, 4×128, 8×64, 16×32) дозволив виявити чіткі закономірності та залежності між структурою мережі та її продуктивністю.

3. **Визначено найефективнішу архітектуру серед досліджених** для задачі керування віртуальним автомобілем. Архітектура з двома прихованими шарами по 256 нейронів (2×256) продемонструвала найкращі результати за більшістю метрик: найшвидше навчання, найкращий час проходження треку, найвищу частоту успішних заїздів та найвищу середню швидкість.

4. **Виявлено обмеження надто глибоких мереж** для задач керування в реальному часі. Архітектура з 16 шарами по 32 нейрони (16×32) показала найгірші результати за всіма метриками, що підтверджує наявність проблеми

зникаючого градієнта та складність навчання надто глибоких мереж для даної задачі.

5. **Встановлено рекомендований баланс між глибиною та шириною мережі**. Результати показують, що для ефективного керування віртуальним автомобілем оптимальною є архітектура з помірною глибиною (2-4 шари) та достатньою шириною (128-256 нейронів на шар), що забезпечує необхідну ємність моделі без надмірної обчислювальної складності.

6. **Розроблено рекомендації** щодо вибору архітектури нейронної мережі для різних сценаріїв застосування, враховуючи складність траси, обчислювальні обмеження та вимоги до якості керування.

7. **Створено інтерактивну систему** для демонстрації результатів навчання, що дозволяє переглядати найкращі заїзди, аналізувати статистику та змагатися з навченими агентами.

Практична цінність роботи полягає у створенні програмного комплексу, який може використовуватися як для подальших досліджень у галузі машинного навчання та автономних систем, так і для навчальних цілей для демонстрації принципів навчання з підкріпленням.

Результати дослідження можуть бути використані розробниками систем керування віртуальними об'єктами для ефективного вибору архітектури нейронних мереж відповідно до конкретних вимог задачі. Запропоновані рекомендації дозволяють знаходити оптимальний баланс між швидкістю навчання, якістю керування та обчислювальною ефективністю.

Подальші напрямки досліджень включають:

- Вивчення впливу інших архітектурних параметрів (функцій активації, алгоритмів оптимізації) на ефективність навчання

- Розширення експериментів на більш складні сценарії керування з динамічними перешкодами
- Дослідження трансферного навчання між різними трасами
- Порівняння ефективності рекурентних та згорткових архітектур для задач керування віртуальними об'єктами

Розроблений програмний комплекс та отримані результати створюють основу для майбутніх досліджень у галузі машинного навчання для керування автономними системами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sutton R. S., Barto A. G. Reinforcement Learning: An Introduction. 2nd ed. MIT Press, 2018. 548 p.
2. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016. 800 p.
3. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 4th ed. Pearson, 2020. 1136 p.
4. Bishop C. M. Pattern Recognition and Machine Learning. Springer, 2006. 738 p.
5. Mitchell T. M. Machine Learning. McGraw-Hill, 1997. 414 p.
6. Haykin S. Neural Networks and Learning Machines. 3rd ed. Pearson, 2008. 936 p.
7. Schulman J., Wolski F., Dhariwal P., Radford A., Klimov O. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347 [Електронний ресурс]. URL: https://arxiv.org/abs/1707.06347 (дата звернення: 15.05.2025).
8. Juliani A., Berges V.-P., Vckay E., Gao Y., Henry H., Mattar M., Lange D. Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627 [Електронний ресурс]. URL: https://arxiv.org/abs/1809.02627 (дата звернення: 15.05.2025).
9. Mnih V., Kavukcuoglu K., Silver D., Rusu A. A., Veness J., Bellemare M. G., Graves A., Riedmiller M., Fidjeland A. K., Ostrovski G., Petersen S. Human-level control through deep reinforcement learning. Nature. 2015. Vol. 518, No. 7540. P. 529-533 [Електронний ресурс]. URL: https://www.nature.com/articles/nature14236 (дата звернення: 15.05.2025).
10. Unity Technologies. Unity ML-Agents Toolkit Documentation [Електронний ресурс]. URL: https://unity-technologies.github.io/ml-agents/ (дата звернення: 15.05.2025).

11. Unity Technologies. ML-Agents Overview [Електронний ресурс]. URL: https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/ (дата звернення: 15.05.2025).
12. Unity Manual: ML Agents [Електронний ресурс]. URL: https://docs.unity3d.com/6000.0/Documentation/Manual/com.unity.ml-agents.html (дата звернення: 15.05.2025).
13. PPO Implementation Details [Електронний ресурс]. URL: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/ (дата звернення: 15.05.2025).
14. OpenAI Spinning Up: Proximal Policy Optimization [Електронний ресурс]. URL: https://spinningup.openai.com/en/latest/algorithms/ppo.html (дата звернення: 15.05.2025).
15. Papers with Code: DQN Explained [Електронний ресурс]. URL: https://paperswithcode.com/method/dqn (дата звернення: 15.05.2025).
16. Papers with Code: PPO Explained [Електронний ресурс]. URL: https://paperswithcode.com/method/ppo (дата звернення: 15.05.2025).
17. Google DeepMind: Deep Reinforcement Learning [Електронний ресурс]. URL: https://deepmind.google/discover/blog/deep-reinforcement-learning/ (дата звернення: 15.05.2025).
18. Yang Z., Wang Z., Liu H., Chen Y. C., Wang Z. A Theoretical Analysis of Deep Q-Learning. arXiv preprint arXiv:1901.00137 [Електронний ресурс]. URL: https://arxiv.org/abs/1901.00137 (дата звернення: 15.05.2025).
19. Schaul T., Quan J., Antonoglou I., Silver D. Prioritized Experience Replay. arXiv preprint arXiv:1511.05952 [Електронний ресурс]. URL: https://arxiv.org/abs/1511.05952 (дата звернення: 15.05.2025).

20. Wang Z., Schaul T., Hessel M., van Hasselt H., Lanctot M., de Freitas N. Dueling Network Architectures for Deep Reinforcement Learning. arXiv preprint arXiv:1511.06581 [Електронний ресурс]. URL: https://arxiv.org/abs/1511.06581 (дата звернення: 15.05.2025).

21. Liu W., Wang Z., Liu X., Zeng N., Liu Y., Alsaadi F. E. A survey of deep neural network architectures and their applications. Neurocomputing. 2017. Vol. 234. P. 11-26 [Електронний ресурс]. URL: https://www.sciencedirect.com/science/article/abs/pii/S0925231216315533 (дата звернення: 15.05.2025).

22. Madhiarasan M., Louzazni S. Analysis of Artificial Neural Network: Architecture, Types, and Forecasting Applications. Journal of Electrical and Computer Engineering. 2022 [Електронний ресурс]. URL: https://onlinelibrary.wiley.com/doi/10.1155/2022/5416722 (дата звернення: 15.05.2025).

23. NVIDIA Developer: Isaac Sim - Robotics Simulation [Електронний ресурс]. URL: https://developer.nvidia.com/isaac/sim (дата звернення: 15.05.2025).

24. NVIDIA: Autonomous Vehicle Sensor Simulation [Електронний ресурс]. URL: https://www.nvidia.com/en-us/use-cases/autonomous-vehicle-simulation/ (дата звернення: 15.05.2025).

25. DataCamp: Multilayer Perceptrons in Machine Learning [Електронний ресурс]. URL: https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning (дата звернення: 15.05.2025).

26. SciKit-Learn: Neural network models (supervised) [Електронний ресурс]. URL: https://scikit-learn.org/stable/modules/neural_networks_supervised.html (дата звернення: 15.05.2025).

27. GeeksforGeeks: Multi-Layer Perceptron Learning in Tensorflow [Електронний ресурс]. URL: https://www.geeksforgeeks.org/multi-layer-perceptron-learning-in-tensorflow/ (дата звернення: 15.05.2025).

28. Yu Y., Zhang Y. Multi-layer Perceptron Trainability Explained via Variability. arXiv preprint arXiv:2105.08911 [Електронний ресурс]. URL: https://arxiv.org/abs/2105.08911 (дата звернення: 15.05.2025).

29. V7 Labs: The Essential Guide to Neural Network Architectures [Електронний ресурс]. URL: https://www.v7labs.com/blog/neural-network-architectures-guide (дата звернення: 15.05.2025).

30. Google Research: Transformer: A Novel Neural Network Architecture for Language Understanding [Електронний ресурс]. URL: https://research.google/blog/transformer-a-novel-neural-network-architecture-for-language-understanding/ (дата звернення: 15.05.2025).

ДОДАТОК А

Експорт даних в Excel

Вигляд структури таблиці для запису інформації про заїзд:

	A	B	C	D	E	F	G
1	LapNumber	CompletionPercentage	LapTime	AverageSpeed	TotalReward	IsCompleted	Timestamp
2	1	0,00	25,02	0,00	-25,00	FALSE	2025-04-12 15:48:55
3	1	0,00	25,02	0,00	-35,00	FALSE	2025-04-12 15:48:55
4	1	0,00	25,08	0,00	-25,00	FALSE	2025-04-12 15:48:55
5	1	0,00	25,08	0,00	-35,00	FALSE	2025-04-12 15:48:55
6	1	0,00	25,10	0,00	-25,01	FALSE	2025-04-12 15:48:55
7	1	0,00	25,10	0,00	-35,01	FALSE	2025-04-12 15:48:55
8	1	0,00	25,10	0,00	-25,02	FALSE	2025-04-12 15:48:55
9	1	0,00	25,10	0,00	-35,02	FALSE	2025-04-12 15:48:55
10	1	0,00	25,12	0,00	-25,01	FALSE	2025-04-12 15:48:55
11	1	0,00	25,12	0,00	-35,01	FALSE	2025-04-12 15:48:55
12	1	0,00	25,12	0,00	-25,00	FALSE	2025-04-12 15:48:55
13	1	0,00	25,12	0,00	-35,00	FALSE	2025-04-12 15:48:55
14	1	0,00	25,14	0,00	-25,01	FALSE	2025-04-12 15:48:55
15	1	0,00	25,14	0,00	-35,01	FALSE	2025-04-12 15:48:55
16	1	0,00	25,14	0,00	-25,02	FALSE	2025-04-12 15:48:55
17	1	0,00	25,14	0,00	-35,02	FALSE	2025-04-12 15:48:55
18	1	0,00	25,14	0,00	-25,01	FALSE	2025-04-12 15:48:55
19	1	0,00	25,14	0,00	-35,01	FALSE	2025-04-12 15:48:55
20	1	0,00	25,16	0,00	-25,01	FALSE	2025-04-12 15:48:55
21	1	0,00	25,16	0,00	-35,01	FALSE	2025-04-12 15:48:55
22	1	0,00	25,16	0,00	-25,02	FALSE	2025-04-12 15:48:55
23	1	0,00	25,16	0,00	-35,02	FALSE	2025-04-12 15:48:55
24	1	0,00	25,16	0,00	-25,02	FALSE	2025-04-12 15:48:55
25	1	0,00	25,16	0,00	-35,02	FALSE	2025-04-12 15:48:55
26	1	0,00	25,20	0,00	-25,00	FALSE	2025-04-12 15:48:55
27	1	0,00	25,20	0,00	-35,00	FALSE	2025-04-12 15:48:55
28	1	0,00	25,20	0,00	-25,01	FALSE	2025-04-12 15:48:55
29	1	0,00	25,20	0,00	-35,01	FALSE	2025-04-12 15:48:55
30	1	0,00	25,20	0,00	-25,02	FALSE	2025-04-12 15:48:55
31	1	0,00	25,20	0,00	-35,02	FALSE	2025-04-12 15:48:55
32	1	0,00	25,20	0,00	-25,00	FALSE	2025-04-12 15:48:55
33	1	0,00	25,20	0,00	-35,00	FALSE	2025-04-12 15:48:55
34	1	0,00	25,22	0,00	-25,00	FALSE	2025-04-12 15:48:56
35	1	0,00	25,22	0,00	-35,00	FALSE	2025-04-12 15:48:56
36	1	0,00	25,22	0,00	-25,02	FALSE	2025-04-12 15:48:56
37	1	0,00	25,22	0,00	-35,02	FALSE	2025-04-12 15:48:56
38	1	0,00	25,24	0,00	-25,01	FALSE	2025-04-12 15:48:56
39	1	0,00	25,24	0,00	-35,01	FALSE	2025-04-12 15:48:56
40	1	0,00	25,24	0,00	-25,02	FALSE	2025-04-12 15:48:56
41	1	0,00	25,24	0,00	-35,02	FALSE	2025-04-12 15:48:56

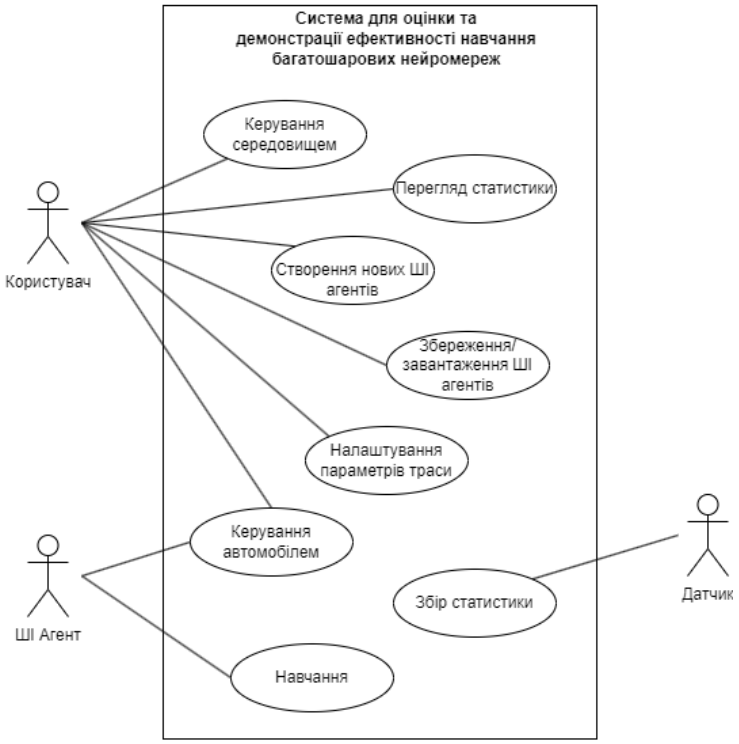
Підрахунок інших параметрів:

H	I	J	K	L	M	N	O
Перший 100%	Кількість заїздів	Min час	array[% прох.]	array[час]	array[швид.]	array[нагорода]	array[прох.]
9003	9414	145,68	0,48	44,52	-0,05	-30,01	0
			0,48	34,37	-0,05	-30,01	0
			0,38	48,07	0,00	-30,01	0
			0,40	57,90	0,01	-30,01	0
			0,44	66,35	0,09	-30,01	0
			0,47	74,48	0,14	-30,01	0
			0,53	80,33	0,21	-30,02	0
			0,64	78,94	0,39	-30,04	0
			0,78	74,07	0,63	-30,10	0
			0,90	69,71	0,81	-30,13	0
			1,05	65,97	0,96	-30,14	0
			1,17	63,50	1,10	-30,16	0
			1,30	60,98	1,23	-30,16	0
			1,40	58,72	1,37	-30,18	0
			1,50	56,76	1,51	-30,20	0
			1,59	55,16	1,62	-30,21	0
			1,70	53,93	1,76	-30,22	0
			1,78	52,81	1,82	-30,23	0
			1,84	51,71	1,88	-30,23	0
			1,91	50,76	1,95	-30,23	0
			1,99	49,93	2,04	-30,25	0
			2,08	49,40	2,12	-30,25	0
			2,13	48,74	2,17	-30,26	0
			2,19	48,25	2,22	-30,26	0
			2,28	47,84	2,31	-30,26	0
			2,35	47,47	2,39	-30,26	0
			2,45	47,23	2,49	-30,27	0
			2,57	47,12	2,60	-30,27	0
			2,69	47,08	2,71	-30,28	0
			2,81	47,04	2,85	-30,28	0
			2,92	47,07	2,96	-30,28	0
			3,03	47,02	3,08	-30,28	0
			3,15	47,12	3,21	-30,28	0
			3,26	47,22	3,32	-30,29	0
			3,38	47,30	3,44	-30,30	0
			3,50	47,48	3,56	-30,30	0
			3,64	47,72	3,68	-30,31	0
			3,76	47,99	3,77	-30,31	0
			3,88	48,24	3,86	-30,31	0
			4,02	48,63	3,94	-30,31	0
			4,22	49,29	4,04	-30,31	0
			4,42	50,09	4,11	-30,31	0
			4,75	51,61	4,19	-30,31	0
			5,04	52,94	4,26	-30,30	0

ДОДАТОК Б

UML-діаграми

Use-case діаграма:



Абстракції:

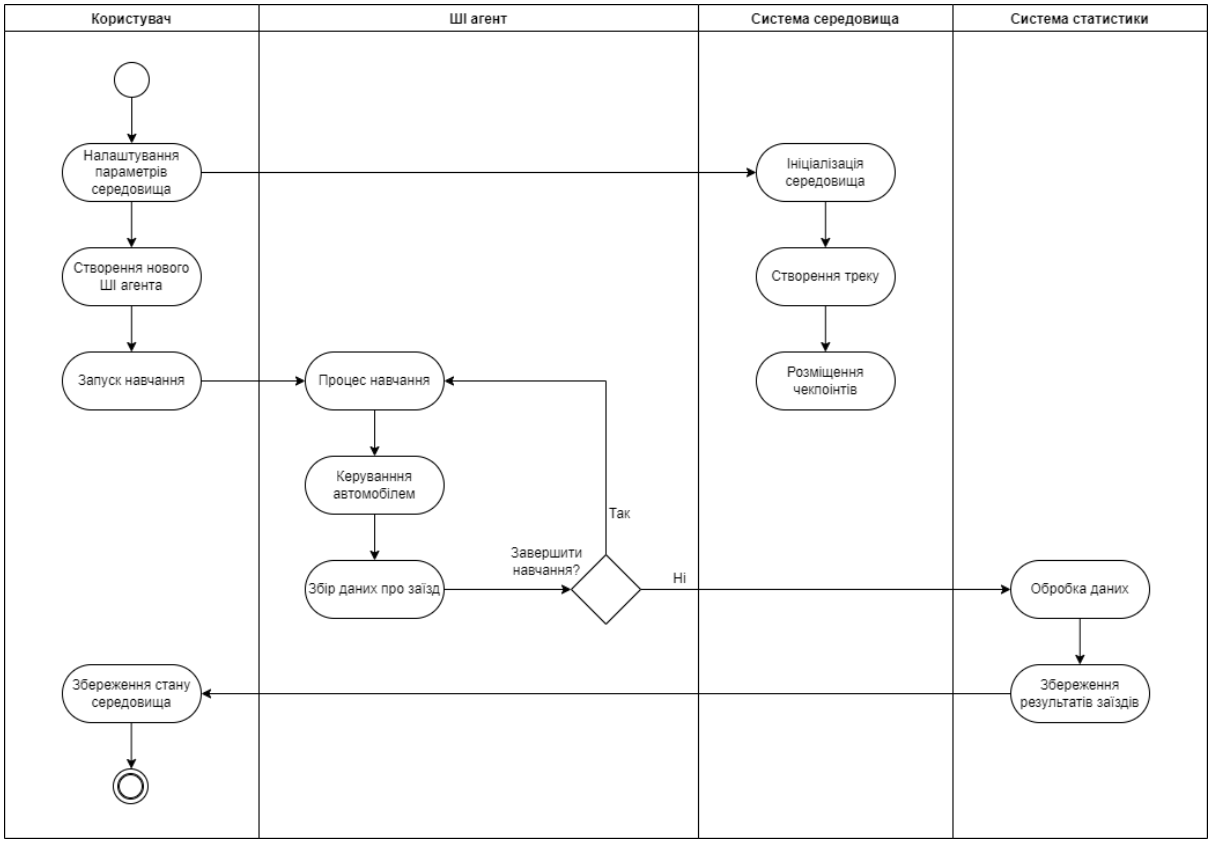
ШІ агент
- Назва - Архітектура - Параметри навчання - Статус
- Керування автомобілем - Навчання - Збір інформації датчиків

Статистика навчання
- Агент - № заїзду - Час заїзду - Пройдені чекпоінти - Метрики ефективності - Траєкторія
- Збір метрик - Аналіз ефективності

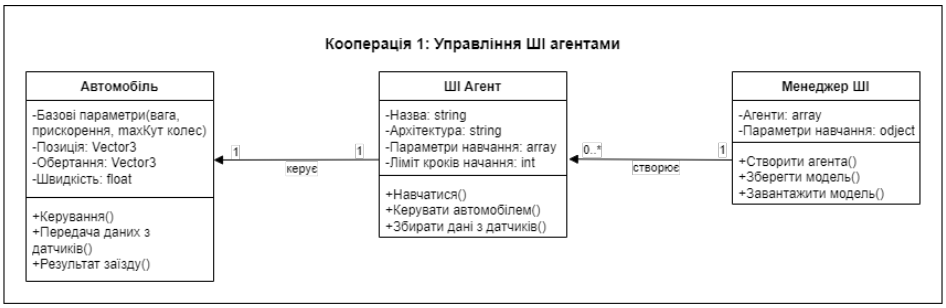
Середовище
- Конфігуратор траси - Чекпоінти - Старт - Фініш
- Симуляція фізики - Генерація винагород

Автомобіль
- Положення - Швидкість - Стан
- Рух по трасі - Взаємодія з середовищем - Передача даних з датчиків

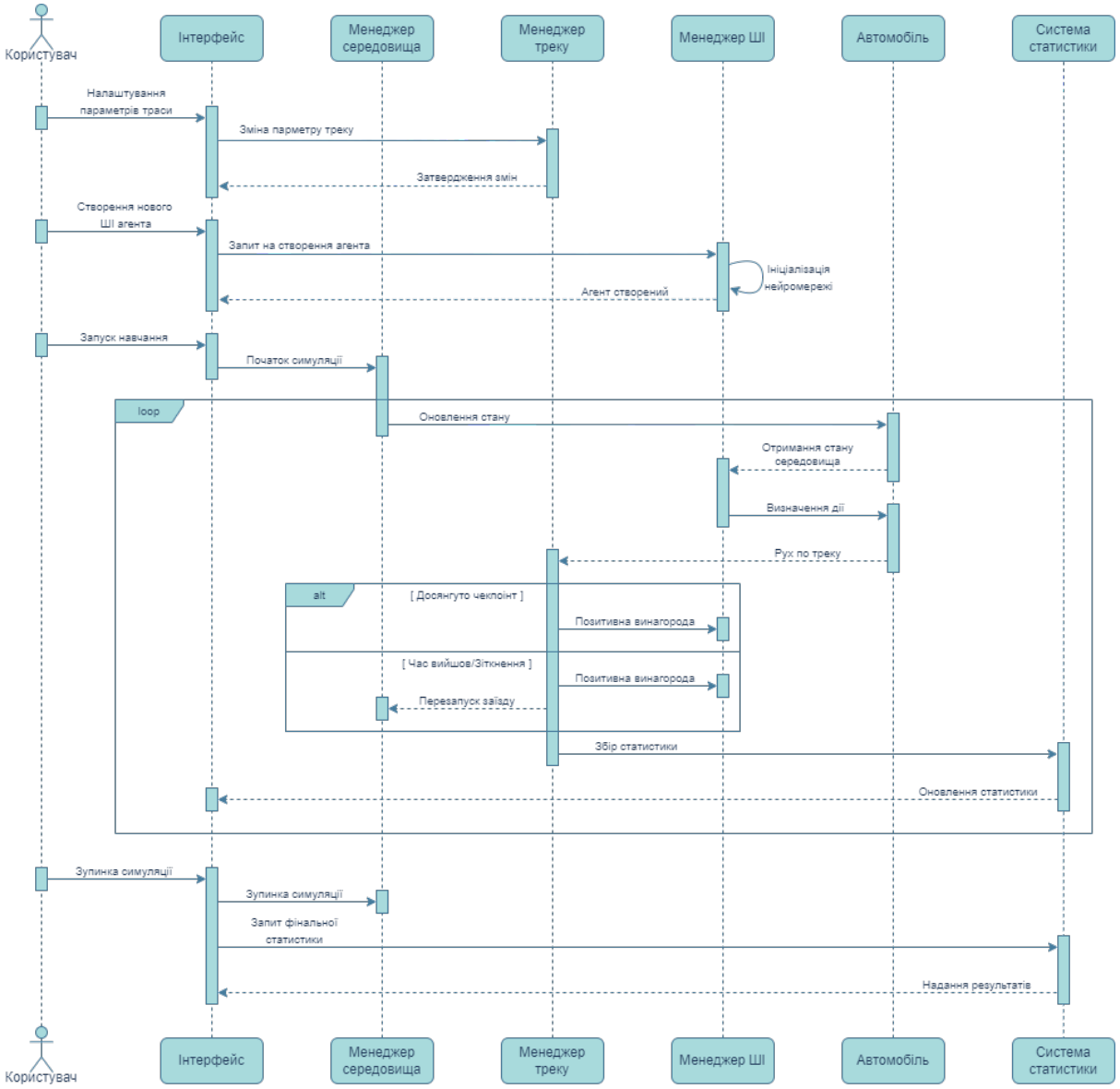
Діаграма активності:



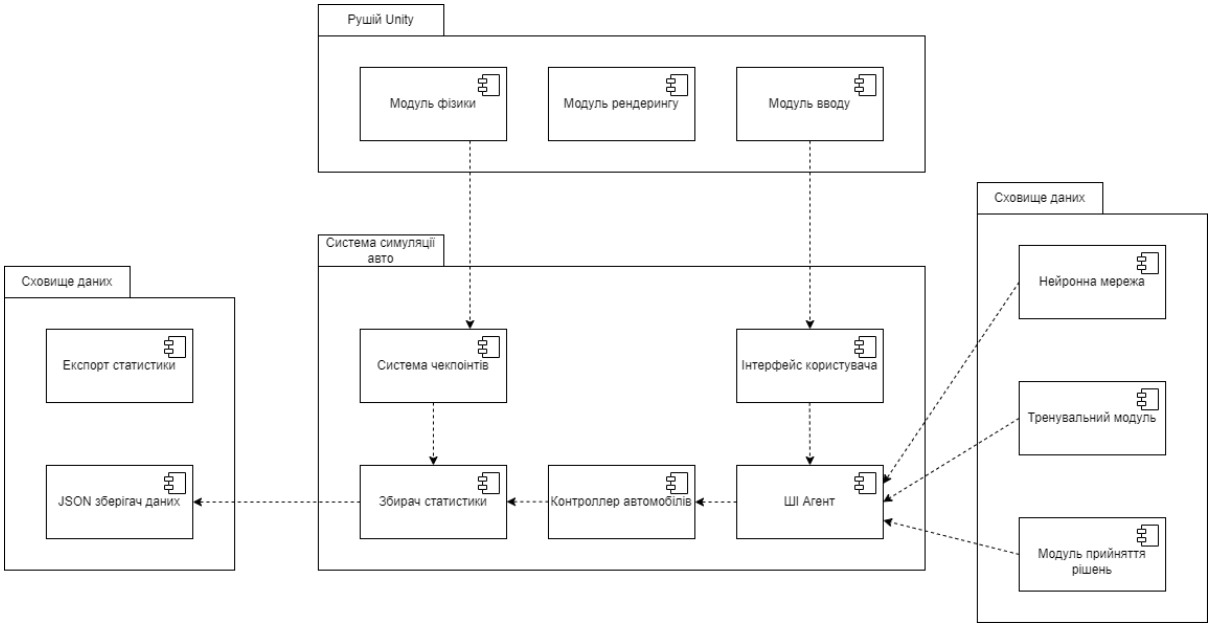
Діаграма кооперацій:



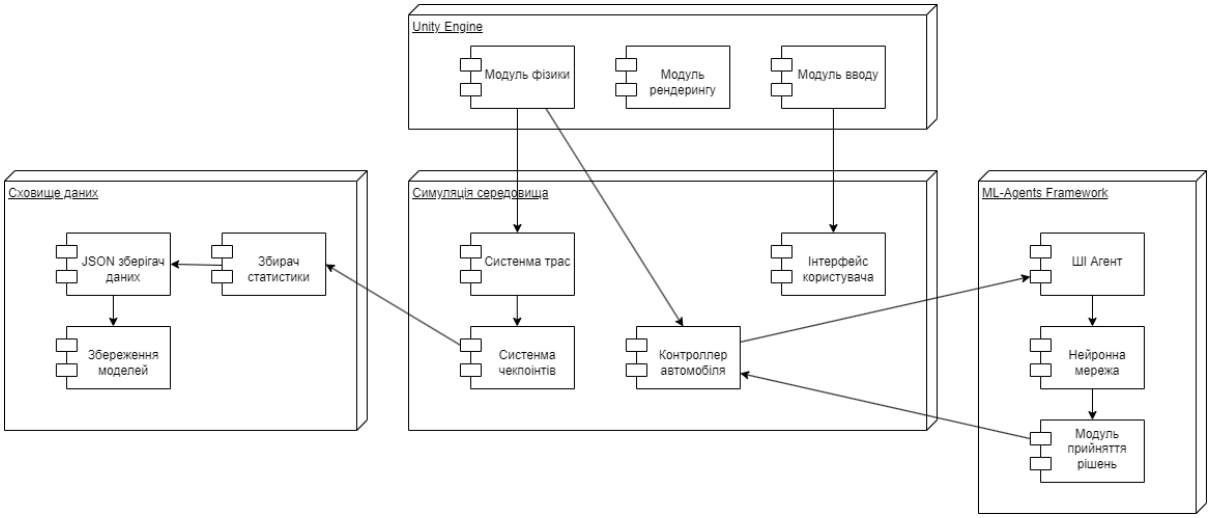
Діаграма послідовності:



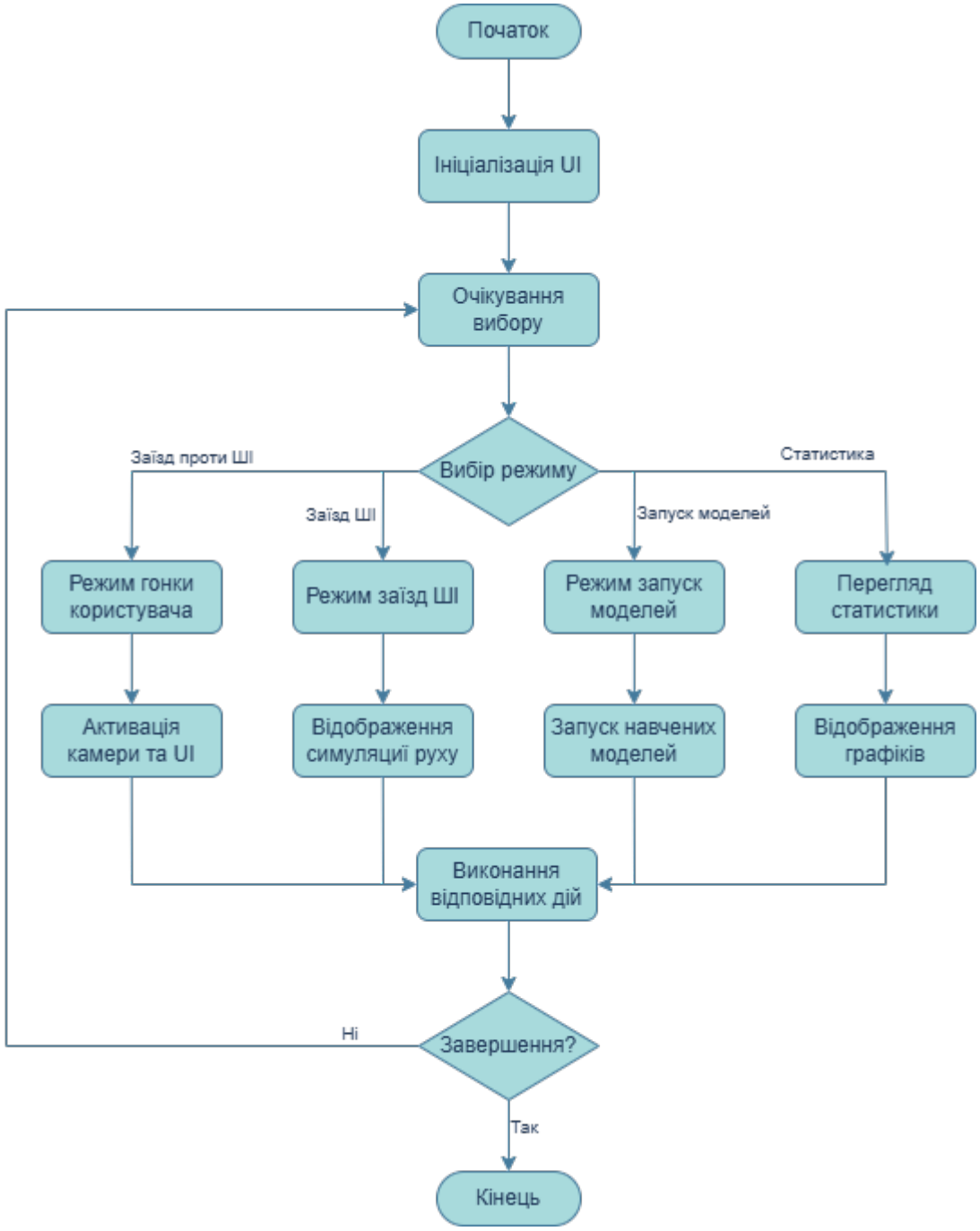
Діаграма компонентів:



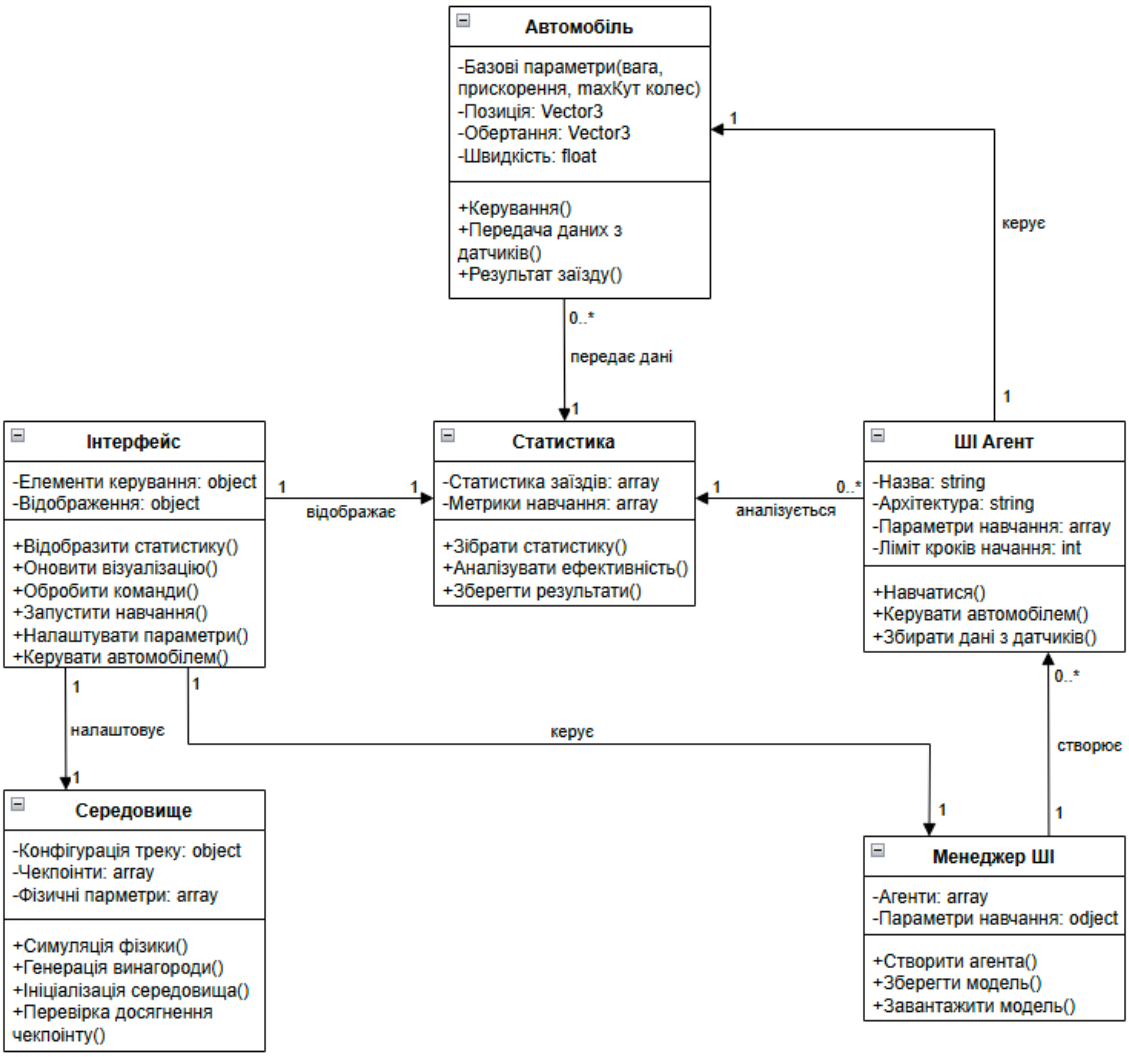
Діаграма розгортання:



Блок-схема:



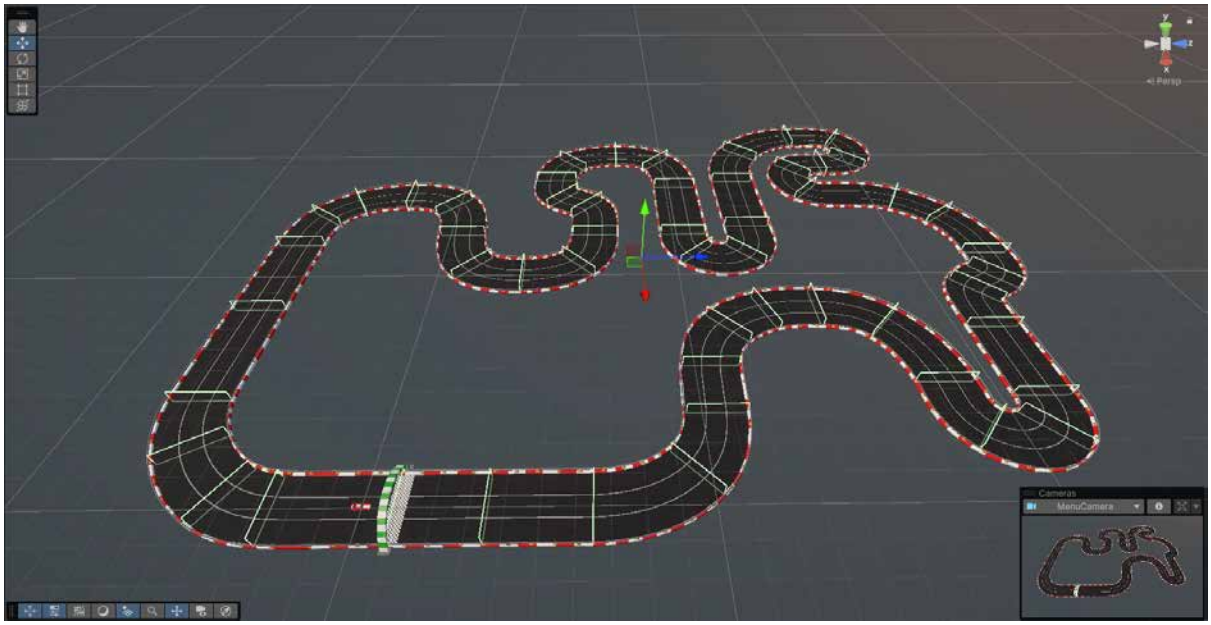
Діаграма класів:


Автомобіль
-Базові параметри(вага, прискорення, maxКут колес)
-Позиція: Vector3
-Обертання: Vector3
-Швидкість: float
+Керування()
+Передача даних з датчиків()
+Результат заїзду()
керує
0..*
передає дані
Інтерфейс
-Елементи керування: object
-Відображення: object
+Відобразити статистику()
+Оновити візуалізацію()
+Обробити команди()
+Запустити навчання()
+Налаштувати параметри()
+Керувати автомобілем()
відображає
Статистика
-Статистика заїздів: array
-Метрики навчання: array
+Зібрати статистику()
+Аналізувати ефективність()
+Зберегти результати()
аналізується
ШІ Агент
-Назва: string
-Архітектура: string
-Параметри навчання: array
-Ліміт кроків начання: int
+Навчатися()
+Керувати автомобілем()
+Збирати дані з датчиків()
налаштовує
керує
створює
Середовище
-Конфігурація треку: object
-Чекпоінти: array
-Фізичні парметри: array
+Симуляція фізики()
+Генерація винагороди()
+Ініціалізація середовища()
+Перевірка досягнення чекпоінту()
Менеджер ШІ
-Агенти: array
-Параметри навчання: odject
+Створити агента()
+Зберегти модель()
+Завантажити модель()

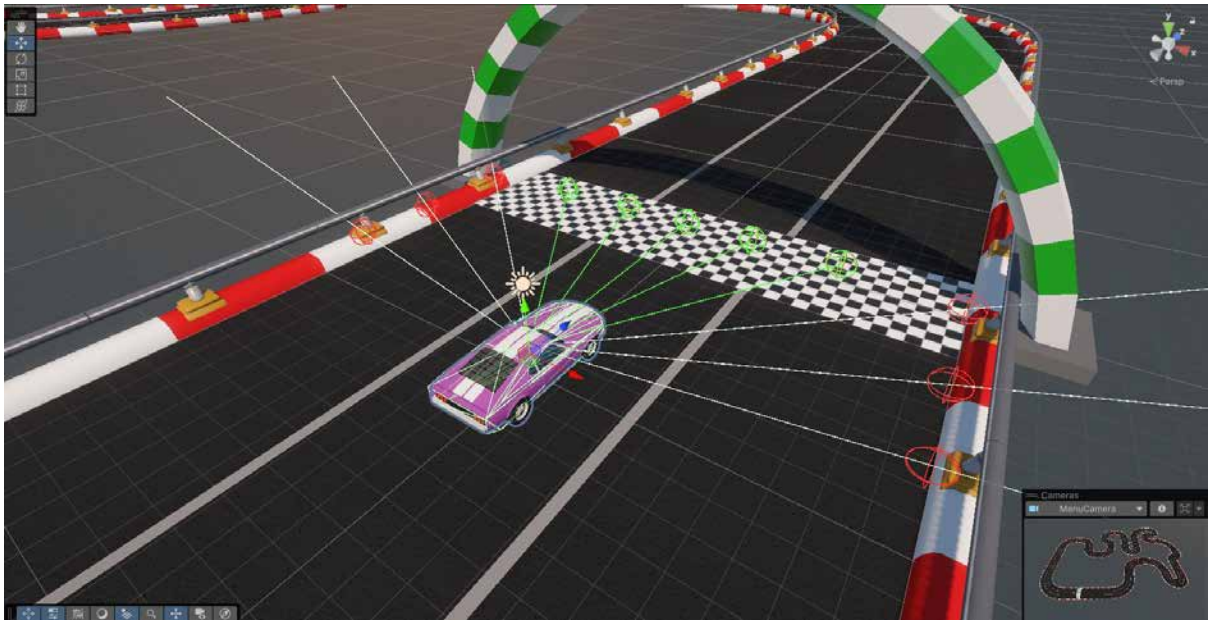
ДОДАТОК В

Конфігурація компонентів в Unity

Гоночний трек і чекпоінти:



Автомобіль і датчики:



ДОДАТОК Г

Реалізація основних класів

Car Agent:

```
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;

public class CarAgent : Agent
{
    PrometeoCarController controller;
    Tracker tracker;

    // Змінні для відстеження винагороди
    private float totalRewardAccumulated = 0f;
    private float lastLoggedReward = 0f;
    private float logRewardInterval = 1.0f; // Інтервал виведення в консоль (секунди)
    private float lastLogTime = 0f;

    // Змінні для відстеження окремих компонентів винагороди
    private float wallPenalties = 0f;
    private float checkpointRewards = 0f;
    private float timeRewards = 0f;
    private float finishRewards = 0f;

    // Налаштування винагород
    [SerializeField] private float timeRewardPerSecond = -1f; // Набагато менший штраф за час
    [SerializeField] private float checkpointRewardValue = 10f;
    [SerializeField] private float wallPenaltyValue = -5f;
    [SerializeField] private float finishRewardValue = 50f;
    [SerializeField] private float failPenaltyValue = -10f;
    [SerializeField] private float minRewardThreshold = -1000f; // Мінімальний поріг винагороди

    private void Awake()
    {
        controller = this.gameObject.GetComponent<PrometeoCarController>();
```

```
        tracker = this.gameObject.GetComponent<Tracker>();
        lastLogTime = Time.time;
    }

    private void Update()
    {
        // Виводимо інформацію про винагороду кожен logRewardInterval секунд
        if (Time.time - lastLogTime >= logRewardInterval)
        {
            // Виводимо тільки якщо винагорода змінилася
            if (Mathf.Abs(totalRewardAccumulated - lastLoggedReward) > 0.01f)
            {
                Debug.Log($"Поточна винагорода: {totalRewardAccumulated:F2} " +
                          $"(Стіни: {wallPenalties:F2}, " +
                          $"Чекпоінти: {checkpointRewards:F2}, " +
                          $"Час: {timeRewards:F2}, " +
                          $"Фініш: {finishRewards:F2})");

                lastLoggedReward = totalRewardAccumulated;
            }
            lastLogTime = Time.time;
        }

        // Перевірка, чи винагорода стала менше мінімального порогу
        CheckRewardThreshold();
    }

    // Метод для перевірки, чи винагорода впала нижче порогу
    private void CheckRewardThreshold()
    {
        if (totalRewardAccumulated < minRewardThreshold)
        {
            Debug.Log($"Заїзд завершено через низьку винагороду: {totalRewardAccumulated:F2} < {minRewardThreshold:F2}");

            // Отримуємо посилання на Tracker
            if (tracker != null)
            {
                // Підготовка даних заїзду для запису
                LapData lapData = new LapData
                {
```

```
                    LapNumber = tracker.GetCurrentLapNumber(),
                    CompletionPercentage =
tracker.CalculateCompletionPercentage(),
                    NetworkEffectiveness = 0f,
                    Frames = tracker.GetCurrentFrames(),
                    LapTime = tracker.GetCurrentLapTime(),
                    AverageSpeed = tracker.CalculateAverageSpeed(),
                    TotalReward = totalRewardAccumulated,
                    IsCompleted = false // Заїзд не завершено успішно
                };

                // Додатковий крок - спробувати зберегти дані перед
завершенням епізоду
                bool saveSuccess = LapDataSaver.SaveLapToFile(lapData,
"low_reward");
                Debug.Log($"Результат збереження даних заїзду при
низькій винагороді: {(saveSuccess ? "Успішно" : "Помилка")}");

                // Потім завершуємо заїзд
                EndOfRace(false, lapData);
            }
            else
            {
                // Якщо Tracker недоступний, продовжуємо як раніше
                LapData lapData = new LapData
                {
                    TotalReward = totalRewardAccumulated,
                    IsCompleted = false
                };

                EndOfRace(false, lapData);
            }
        }
    }

    public override void Initialize()
    {
        SetResetParemeters();
    }

    public override void OnEpisodeBegin()
    {
        SetResetParemeters();
        tracker.ResetCheckpoints();
```

```
        // Скидаємо накопичену винагороду та компоненти
        totalRewardAccumulated = 0f;
        wallPenalties = 0f;
        checkpointRewards = 0f;
        timeRewards = 0f;
        finishRewards = 0f;
        lastLoggedReward = 0f;
    }

    public void EndOfRace(bool isFinished, LapData lapData)
    {
        float reward = 0f;

        if (isFinished)
        {
            reward = finishRewardValue;
            finishRewards += reward;
            Debug.Log($"Фініш! Додано винагороду: +{reward}");
        }
        else
        {
            reward = failPenaltyValue;
            finishRewards += reward;
            Debug.Log($"Не вдалося завершити заїзд. Штраф: {reward}");
        }

        ResetCarSpeed();
        AddReward(reward);
        totalRewardAccumulated += reward;

        // Встановлюємо загальну винагороду в LapData перед
закінченням епізоду
        if (lapData != null)
        {
            lapData.TotalReward = totalRewardAccumulated;
            Debug.Log($"Завершення заїзду: Загальна винагорода =
{totalRewardAccumulated:F2} " +
                $"(Стіни: {wallPenalties:F2}, " +
                $"Чекпоінти: {checkpointRewards:F2}, " +
                $"Час: {timeRewards:F2}, " +
                $"Фініш: {finishRewards:F2})");

            // Спроба зберегти дані, якщо вони ще не збережені
```

```
            // Використовуємо різні причини збереження для діагностики
            string saveReason = isFinished ? "finish" : "fail";
            bool saveSuccess = LapDataSaver.SaveLapToFile(lapData,
saveReason);
            Debug.Log($"Результат збереження даних заїзду при
EndOfRace: {(saveSuccess ? "Успішно" : "Помилка")}");
        }

        EndEpisode();
    }

    private void ResetCarSpeed()
    {
        Debug.Log("Скидаємо швидкість автомобіля після завершення
заїзду");

        // Отримуємо Rigidbody автомобіля
        Rigidbody carRigidbody = controller.GetComponent<Rigidbody>();
        if (carRigidbody != null)
        {
            // Встановлюємо швидкість у нуль
            carRigidbody.linearVelocity = Vector3.zero;
            carRigidbody.angularVelocity = Vector3.zero;
        }

        // Також можна викликати метод для поступового гальмування
з контролера
        controller.Brakes();
    }

    public override void OnActionReceived(ActionBuffers actions)
    {
        //Debug.Log("Ai = " + actions.DiscreteActions[0] + " " +
actions.DiscreteActions[1]);
        int forwardMove = actions.DiscreteActions[0] - 1;
        int rightMove = actions.DiscreteActions[1] - 1;
        controller.InputAIParams(forwardMove, rightMove);

        float speedFactor = Mathf.Clamp01(controller.carSpeed / 120f);
        float timeReward = timeRewardPerSecond * (1.0f - speedFactor) *
Time.deltaTime;

        AddReward(timeReward);
        totalRewardAccumulated += timeReward;
```

```
            timeRewards += timeReward;

            // Перевірка порогу винагороди після кожної дії
            CheckRewardThreshold();
        }

        // Змінна для запобігання частим спрацюванням колізій зі стінами
        private float lastWallHitTime = 0f;

        public void OnCheckpointEnter()
        {
            float checkpointReward = checkpointRewardValue;
            AddReward(checkpointReward);
            totalRewardAccumulated += checkpointReward;
            checkpointRewards += checkpointReward;

            Debug.Log($"Чекпоінт пройдено! Додано винагороду:
+{checkpointReward}");
        }

        public void OnWallEnter()
        {
            // Ігноруємо зіткнення, якщо минуло менше 0.2 секунди з
моменту останнього
            if (Time.time - lastWallHitTime < 0.2f)
                return;

            lastWallHitTime = Time.time;
            float wallReward = wallPenaltyValue;
            AddReward(wallReward);
            totalRewardAccumulated += wallReward;
            wallPenalties += wallReward;

            Debug.Log($"Зіткнення зі стіною! Штраф: {wallReward}. Поточна
загальна винагорода: {totalRewardAccumulated:F2}");

            // Перевірка порогу винагороди після отримання штрафу
            CheckRewardThreshold();
        }

        public override void CollectObservations(VectorSensor sensor)
        {
            sensor.AddObservation(transform.position);
            sensor.AddObservation(transform.rotation);
```

```
            sensor.AddObservation(controller.carSpeed);

            // Додатково можна додати спостереження за напрямком до наступного чекпоінту
            if (tracker != null)
            {
                Vector3 directionToNextCheckpoint = tracker.GetDirectionToNextCheckpoint();
                sensor.AddObservation(directionToNextCheckpoint);

                // Також можна додати відстань до наступного чекпоінту
                float distanceToNextCheckpoint = tracker.GetDistanceToNextCheckpoint();
                sensor.AddObservation(distanceToNextCheckpoint);
            }
        }

        private void SetResetParemeters()
        {
            transform.position = Vector3.zero;
            transform.rotation = Quaternion.Euler(0f, 0f, 0f);
        }

        // Метод для отримання поточної загальної винагороди
        public float GetTotalReward()
        {
            return totalRewardAccumulated;
        }
    }
```

LapDataSaver:

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

public static class LapDataSaver
{
    private const int MaxTopRecordsToKeep = 10; // Обмежуємо до 10 найкращих заїздів
    private const float MinCompletionPercentage = 50.0f; // Мінімальний відсоток проходження для збереження траєкторії
    private const string StatisticsFileName = "AllRacesStatistics.json";
    private const string TopRecordsFileName = "TopRecords.json";
    // Використовуємо декілька шляхів для запису з резервуванням
    private static readonly string[] PossiblePaths = new string[] {
        @"E:\AIresults",
        Path.Combine(Application.persistentDataPath, "AIresults"),
        Path.Combine(Application.dataPath, "AIresults"),
        @"C:\AIresults"
    };
    // Змінна для збереження обраного шляху
    private static string selectedPath = null;

    // Додаємо лічильник для кількості записаних заїздів
    private static int savedLapsCount = 0;
    private static int failedSavesCount = 0;
    private static int filteredOutLapsCount = 0; // Лічильник для відфільтрованих заїздів

    // Додаємо буфер для тимчасового зберігання незаписаних даних
    private static List<LapData> pendingLapData = new List<LapData>();

    // Топ-записи для збереження
    private static List<LapData> topRecords = new List<LapData>();

    [System.Serializable]
    private class RaceStatistics
    {
        public List<RaceStatEntry> Entries = new List<RaceStatEntry>();
    }
```

```
    [System.Serializable]
    private class TopRecordsContainer
    {
        public List<LapData> Records = new List<LapData>();
    }

    [System.Serializable]
    public class RaceStatEntry
    {
        public int LapNumber;
        public float CompletionPercentage;
        public float LapTime;
        public float AverageSpeed;
        public float TotalReward;
        public bool IsCompleted;
        public string Timestamp;
        public string SaveReason;
    }

    // Ініціалізація шляху для запису
    static LapDataSaver()
    {
        // Виводимо всі можливі шляхи для діагностики
        Debug.Log("Ініціалізація LapDataSaver з можливими шляхами
для запису:");
        foreach (string path in PossiblePaths)
        {
            Debug.Log($"  - {path}");
        }

        // Спробуємо знайти доступний шлях при першому запуску
        FindWorkingPath();

        // Логуємо обраний шлях
        Debug.Log($"LapDataSaver обрав шлях: {selectedPath ?? "Шлях
не знайдено!"}");

        // Завантажуємо існуючі топ-записи, якщо вони є
        LoadTopRecords();

        // Спробуємо записати тестовий файл для перевірки
        if (selectedPath != null)
        {
            try
```

```
            {
                string testPath = Path.Combine(selectedPath, "test_write.txt");
                File.WriteAllText(testPath, "Test write at " + DateTime.Now);
                Debug.Log($"Тестовий запис успішний: {testPath}");
            }
            catch (Exception e)
            {
                Debug.LogError($"Тестовий запис не вдався: {e.Message}");
                selectedPath = null;
            }
        }
    }

    // Метод для пошуку доступного шляху
    private static void FindWorkingPath()
    {
        foreach (string path in PossiblePaths)
        {
            try
            {
                // Якщо директорія не існує, спробуємо створити
                if (!Directory.Exists(path))
                {
                    Directory.CreateDirectory(path);
                    Debug.Log($"Створено директорію: {path}");
                }

                // Тестовий запис
                string testFile = Path.Combine(path, "test.txt");
                File.WriteAllText(testFile, "Test " + DateTime.Now);
                File.Delete(testFile);

                // Шлях знайдено
                selectedPath = path;
                Debug.Log($"Знайдено робочий шлях для запису: {path}");
                return;
            }
            catch (Exception e)
            {
                Debug.LogWarning($"Шлях {path} недоступний:
{e.Message}");
                // Продовжуємо перевірку наступного шляху
            }
        }
```

```
            // Якщо не знайдено жодного доступного шляху
            Debug.LogError("Не знайдено жодного доступного шляху для запису даних!");
        }

        // Завантаження існуючих топ-записів
        private static void LoadTopRecords()
        {
            if (selectedPath == null) return;

            string topRecordsPath = Path.Combine(selectedPath, TopRecordsFileName);

            if (File.Exists(topRecordsPath))
            {
                try
                {
                    string jsonData = File.ReadAllText(topRecordsPath);
                    TopRecordsContainer container = JsonUtility.FromJson<TopRecordsContainer>(jsonData);

                    if (container != null && container.Records != null)
                    {
                        topRecords = container.Records;
                        Debug.Log($"Завантажено {topRecords.Count} існуючих топ-записів");
                    }
                }
                catch (Exception e)
                {
                    Debug.LogError($"Помилка при завантаженні топ-записів: {e.Message}");
                    topRecords = new List<LapData>();
                }
            }
        }

        // Збереження оновлених топ-записів
        private static void SaveTopRecords()
        {
            if (selectedPath == null || topRecords == null) return;
```

```
            string topRecordsPath = Path.Combine(selectedPath,
TopRecordsFileName);

            try
            {
                TopRecordsContainer container = new TopRecordsContainer
                {
                    Records = topRecords
                };

                string jsonData = JsonUtility.ToJson(container, true);
                File.WriteAllText(topRecordsPath, jsonData);

                Debug.Log($"Збережено {topRecords.Count} топ-записів");
            }
            catch (Exception e)
            {
                Debug.LogError($"Помилка при збереженні топ-записів:
{e.Message}");
            }
        }

        // Основний метод збереження
        public static bool SaveLapToFile(LapData lapData, string reason =
"standard")
        {
            if (lapData == null)
            {
                Debug.LogError("Спроба зберегти null-об'єкт LapData!");
                failedSavesCount++;
                return false;
            }

            Debug.Log($"Початок збереження даних заїзду
#{lapData.LapNumber}, TotalReward: {lapData.TotalReward}, " +
                      $"Проходження: {lapData.CompletionPercentage}%, Час:
{lapData.LapTime}с, Причина: {reason}");

            // Перевіряємо, чи є доступний шлях
            if (selectedPath == null)
            {
                Debug.LogWarning("Немає доступного шляху для запису.
Спробуємо знайти інший...");
                FindWorkingPath();
```

```
            // Якщо шлях все ще не знайдено - додаємо дані в буфер і виходимо
            if (selectedPath == null)
            {
                Debug.LogError("Не вдалося знайти шлях для запису. Дані додано в буфер.");
                pendingLapData.Add(lapData);
                failedSavesCount++;
                return false;
            }
        }

        try
        {
            // Перевіряємо відсоток проходження (мінімум 50%)
            bool hasMinimumCompletion = (lapData.CompletionPercentage >= MinCompletionPercentage);

            // Перевіряємо, чи заїзд входить до топ-10 (тільки для заїздів з мінімальним відсотком)
            bool isTopRecord = hasMinimumCompletion && IsTopRecord(lapData);

            // Створюємо копію запису для статистики (без траєкторії)
            RaceStatEntry entryCopy = new RaceStatEntry
            {
                LapNumber = lapData.LapNumber,
                CompletionPercentage = lapData.CompletionPercentage,
                LapTime = lapData.LapTime,
                AverageSpeed = lapData.AverageSpeed,
                TotalReward = lapData.TotalReward,
                IsCompleted = lapData.IsCompleted,
                Timestamp = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"),
                SaveReason = reason
            };

            // Завжди зберігаємо статистику для всіх заїздів
            bool statSuccess = SaveStatisticsRecord(entryCopy, selectedPath);

            // Зберігаємо детальні дані тільки для топ-10 записів і якщо відсоток >= 50%
```

```
            bool detailSuccess = true;
            if (!hasMinimumCompletion)
            {
                filteredOutLapsCount++;
                Debug.Log($"Заїзд #{lapData.LapNumber} має відсоток проходження {lapData.CompletionPercentage}% < {MinCompletionPercentage}%. " +
                          $"Траєкторію не буде збережено. Всього відфільтровано: {filteredOutLapsCount}");
            }
            else if (isTopRecord)
            {
                Debug.Log($"Заїзд #{lapData.LapNumber} має достатній % проходження та увійшов до топ-10! Зберігаємо детальні дані.");
                detailSuccess = SaveIndividualLapFile(lapData, selectedPath, reason);

                // Оновлюємо топ-записи
                UpdateTopRecords(lapData);
            }
            else
            {
                Debug.Log($"Заїзд #{lapData.LapNumber} має достатній % проходження, але не потрапив до топ-10. Зберігаємо тільки в статистику.");
            }

            // Оновлюємо лічильник
            savedLapsCount++;
            Debug.Log($"Збережено статистику заїзду #{lapData.LapNumber}. Всього збережено заїздів: {savedLapsCount}");

            // Спробуємо записати буфер, якщо є успішний запис
            if (statSuccess && pendingLapData.Count > 0)
            {
                TryFlushPendingData();
            }

            return statSuccess && detailSuccess;
        }
        catch (Exception e)
        {
            failedSavesCount++;
```

```
            Debug.LogError($"Помилка в SaveLapToFile:
{e.Message}\n{e.StackTrace}");

            // Додаємо дані в буфер для подальших спроб
            pendingLapData.Add(lapData);
            return false;
        }
    }

    // Перевірка, чи заїзд входить до топ-10
    private static bool IsTopRecord(LapData newLap)
    {
        // Якщо у нас менше 10 записів з >50% проходження, завжди
зберігаємо
        if (topRecords.Count < MaxTopRecordsToKeep)
            return true;

        // Сортуємо записи за відсотком проходження (спадання) і
часом (зростання)
        var sortedRecords = topRecords
            .OrderByDescending(r => r.CompletionPercentage)
            .ThenBy(r => r.LapTime)
            .ToList();

        // Перевіряємо, чи новий запис кращий за останній у
відсортованому списку
        var worstRecord = sortedRecords.Last();

        // Якщо новий запис має більший відсоток, він кращий
        if (newLap.CompletionPercentage >
worstRecord.CompletionPercentage)
            return true;

        // Якщо відсотки рівні, але час менший, він кращий
        if (newLap.CompletionPercentage ==
worstRecord.CompletionPercentage &&
            newLap.LapTime < worstRecord.LapTime)
            return true;

        // В іншому випадку не входить до топ-10
        return false;
    }

    // Оновлення списку топ-10 записів
```

```
        private static void UpdateTopRecords(LapData newLap)
        {
            try
            {
                // Клонуємо новий запис перед додаванням до колекції
                LapData clonedLap = CloneLapData(newLap);

                // Додаємо новий запис
                topRecords.Add(clonedLap);

                // Сортуємо за відсотком проходження (спадання) і часом
(зростання)
                topRecords = topRecords
                    .OrderByDescending(r => r.CompletionPercentage)
                    .ThenBy(r => r.LapTime)
                    .ToList();

                // Залишаємо тільки топ-10
                if (topRecords.Count > MaxTopRecordsToKeep)
                {
                    int countToRemove = topRecords.Count -
MaxTopRecordsToKeep;
                    topRecords.RemoveRange(MaxTopRecordsToKeep,
countToRemove);
                }

                // Зберігаємо оновлений список
                SaveTopRecords();

                // Виводимо інформацію про топ-записи
                Debug.Log("Оновлений список топ-10 заїздів:");
                for (int i = 0; i < topRecords.Count; i++)
                {
                    var record = topRecords[i];
                    Debug.Log($"  {i + 1}. Проходження:
{record.CompletionPercentage}%, Час: {record.LapTime}с");
                }
            }
            catch (Exception e)
            {
                Debug.LogError($"Помилка при оновленні топ-записів:
{e.Message}");
            }
        }
```

```
        // Метод для збереження загальної статистики
        private static bool SaveStatisticsRecord(RaceStatEntry entry, string
folderPath)
        {
            try
            {
                string statisticsPath = Path.Combine(folderPath,
StatisticsFileName);
                RaceStatistics statistics;

                // Завантажуємо існуючу статистику або створюємо нову,
якщо файл не існує
                if (File.Exists(statisticsPath))
                {
                    string jsonData = File.ReadAllText(statisticsPath);
                    try
                    {
                        statistics = JsonUtility.FromJson<RaceStatistics>(jsonData);
                        if (statistics == null)
                        {
                            Debug.LogWarning($"Не вдалося десеріалізувати
JSON, створено новий об'єкт статистики: {jsonData}");
                            statistics = new RaceStatistics();
                        }
                    }
                    catch (Exception jsonEx)
                    {
                        Debug.LogError($"Помилка при розборі JSON:
{jsonEx.Message}. Створено новий об'єкт статистики.");
                        statistics = new RaceStatistics();
                    }
                }
                else
                {
                    statistics = new RaceStatistics();
                }

                // Додаємо новий запис до статистики
                statistics.Entries.Add(entry);

                // Зберігаємо оновлену статистику
                string updatedJsonData = JsonUtility.ToJson(statistics, true);
                File.WriteAllText(statisticsPath, updatedJsonData);
```

```
            Debug.Log($"Збережено статистику заїзду
#{entry.LapNumber} у JSON файл. TotalReward: {entry.TotalReward}");
            return true;
        }
        catch (Exception e)
        {
            Debug.LogError($"Помилка при збереженні статистики:
{e.Message}");
            return false;
        }
    }

    // Метод для збереження детальних даних заїзду
    private static bool SaveIndividualLapFile(LapData lapData, string
folderPath, string reason = "standard")
    {
        try
        {
            DateTime now = DateTime.Now;
            string lapTimeStr = lapData.LapTime.ToString("F2").Replace(",",
".");
            string completionStr =
lapData.CompletionPercentage.ToString("F1").Replace(",", ".");
            string speedStr =
lapData.AverageSpeed.ToString("F1").Replace(",", ".");
            string rewardStr =
lapData.TotalReward.ToString("F1").Replace(",", ".");

            // Додаємо причину збереження в ім'я файлу
            string fileName =
$"Top_Lap_{lapData.LapNumber}_{completionStr}%_{lapTimeStr}s_{speed
Str}kmh_r{rewardStr}_{reason}_{now:MM-dd_HH-mm-ss}.json";
            string filePath = Path.Combine(folderPath, fileName);

            // Клонуємо дані для серіалізації
            LapData lapDataCopy = CloneLapData(lapData);

            // Серіалізуємо та записуємо дані
            string jsonData = JsonUtility.ToJson(lapDataCopy, true);
            File.WriteAllText(filePath, jsonData);

            Debug.Log($"Збережено детальну інформацію про заїзд
#{lapData.LapNumber} у файл: {filePath}");
```

```
            return true;
        }
        catch (Exception e)
        {
            Debug.LogError($"Помилка при збереженні окремого файлу заїзду: {e.Message}\n{e.StackTrace}");
            return false;
        }
    }

    // Метод для клонування даних заїзду
    private static LapData CloneLapData(LapData original)
    {
        // Створюємо новий об'єкт з основними даними
        LapData clone = new LapData
        {
            LapNumber = original.LapNumber,
            CompletionPercentage = original.CompletionPercentage,
            NetworkEffectiveness = original.NetworkEffectiveness,
            LapTime = original.LapTime,
            AverageSpeed = original.AverageSpeed,
            TotalReward = original.TotalReward,
            IsCompleted = original.IsCompleted
        };

        // Копіюємо кадри, якщо вони є
        if (original.Frames != null && original.Frames.Length > 0)
        {
            try
            {
                clone.Frames = new FrameData[original.Frames.Length];
                for (int i = 0; i < original.Frames.Length; i++)
                {
                    if (original.Frames[i] != null)
                    {
                        clone.Frames[i] = new FrameData
                        {
                            Position = original.Frames[i].Position,
                            Rotation = original.Frames[i].Rotation,
                            Speed = original.Frames[i].Speed
                        };
                    }
                    else
                    {
```

```
                        // Якщо кадр null, створюємо пустий кадр
                        clone.Frames[i] = new FrameData
                        {
                            Position = Vector3.zero,
                            Rotation = Quaternion.identity,
                            Speed = 0f
                        };
                    }
                }
                Debug.Log($"Скопійовано {original.Frames.Length} кадрів
для заїзду #{original.LapNumber}");
            }
            catch (Exception e)
            {
                Debug.LogError($"Помилка при клонуванні кадрів:
{e.Message}. Створено пустий масив кадрів.");
                clone.Frames = new FrameData[0];
            }
        }
        else
        {
            Debug.LogWarning($"Заїзд #{original.LapNumber} не має
кадрів (Frames == null або Frames.Length == 0)");
            clone.Frames = new FrameData[0];
        }

        return clone;
    }

    // Метод для спроби збереження накопичених даних із буфера
    public static void TryFlushPendingData()
    {
        if (pendingLapData.Count == 0)
            return;

        Debug.Log($"Спроба зберегти {pendingLapData.Count}
накопичених записів...");

        List<LapData> successfullySaved = new List<LapData>();

        foreach (var lapData in pendingLapData)
        {
            if (SaveLapToFile(lapData, "buffer_flush"))
            {
```

```
                successfullySaved.Add(lapData);
            }
        }

        // Видаляємо успішно збережені записи з буфера
        foreach (var lapData in successfullySaved)
        {
            pendingLapData.Remove(lapData);
        }

        Debug.Log($"Збережено {successfullySaved.Count} записів із буфера. Залишилось: {pendingLapData.Count}");
    }

    // Метод для отримання статистики збереження
    public static string GetSavingStatistics()
    {
        return $"Збережено заїздів: {savedLapsCount}, невдалих спроб: {failedSavesCount}, " +
               $"відфільтровано <{MinCompletionPercentage}%: {filteredOutLapsCount}, " +
               $"у буфері: {pendingLapData.Count}, шлях: {selectedPath ?? "не знайдено"}, " +
               $"топ-записів: {topRecords.Count}/{MaxTopRecordsToKeep}";
    }

    // Метод для примусового тестування запису
    public static bool TestSaving()
    {
        Debug.Log("Виконується тест запису...");

        // Пробуємо знайти робочий шлях, якщо поточний шлях неробочий
        if (selectedPath == null)
        {
            FindWorkingPath();
        }

        if (selectedPath == null)
        {
            Debug.LogError("Не вдається знайти шлях для запису. Тест провалено.");
            return false;
        }
```

```
        try
        {
            // Створюємо базовий тестовий файл
            string testFilePath = Path.Combine(selectedPath, "test_save_" +
DateTime.Now.Ticks + ".txt");
            File.WriteAllText(testFilePath, "Тестовий запис: " +
DateTime.Now);

            // Перевіряємо, що файл дійсно створено
            bool fileExists = File.Exists(testFilePath);
            Debug.Log($"Тестовий файл створено: {fileExists}, шлях:
{testFilePath}");

            // Створюємо тестові заїзди з різним відсотком проходження

            // 1. Заїзд з низьким відсотком (не повинен зберігатися з
траєкторією)
            LapData testLapLow = new LapData
            {
                LapNumber = 991,
                CompletionPercentage = 30.0f, // Нижче мінімуму
                LapTime = 60.0f,
                AverageSpeed = 30.0f,
                TotalReward = 10.0f,
                IsCompleted = false,
                Frames = new FrameData[1] { new FrameData {
                    Position = Vector3.zero,
                    Rotation = Quaternion.identity,
                    Speed = 0f
                }}
            };

            // 2. Заїзд з достатнім відсотком (повинен зберігатися з
траєкторією)
            LapData testLapHigh = new LapData
            {
                LapNumber = 992,
                CompletionPercentage = 70.0f, // Вище мінімуму
                LapTime = 55.0f,
                AverageSpeed = 35.0f,
                TotalReward = 15.0f,
                IsCompleted = false,
                Frames = new FrameData[1] { new FrameData {
```

```
                Position = Vector3.one,
                Rotation = Quaternion.identity,
                Speed = 50f
            }}
        };

        // Зберігаємо обидва заїзди
        bool saveLowResult = SaveLapToFile(testLapLow, "test_low");
        bool saveHighResult = SaveLapToFile(testLapHigh, "test_high");

        Debug.Log($"Результати тестового збереження: " +
            $"Низький % ({testLapLow.CompletionPercentage}%): {saveLowResult}, " +
            $"Високий % ({testLapHigh.CompletionPercentage}%): {saveHighResult}");

        return fileExists && saveLowResult && saveHighResult;
      }
      catch (Exception e)
      {
        Debug.LogError($"Помилка під час тесту запису: {e.Message}");
        return false;
      }
    }

    // Метод для отримання списку топ-записів
    public static List<LapData> GetTopRecords()
    {
      return topRecords.OrderByDescending(r => r.CompletionPercentage)
        .ThenBy(r => r.LapTime)
        .ToList();
    }
  }
```