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

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук, проф.
____________________Ю. П. Кондратенко
«____»___________202___ р.

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

ІНТЕЛЕКТУАЛЬНА СИСТЕМА КЕРУВАННЯ СЦЕНАРІЯМИ ТА МОДЕЛЮВАННЯ РУХУ В UNITY 3D

Спеціальність 122 «Комп'ютерні науки»

122 – КРМ – 601.21810321

Виконав студент 6-го курсу, групи 601
_______________Б. О. Супрун
«20» лютого 2024 р.

Керівник: д-р техн. наук, проф.
_______________О. П. Гожий
«20» лютого 2024 р.

Миколаїв – 2024

Чорноморський національний університет ім. Петра Могили
Факультет комп’ютерних наук
Кафедра інтелектуальних інформаційних систем

Освітньо-кваліфікаційний рівень **магістр**
Галузь знань **12 «Інформаційні технології»**
(шифр і назва)
Спеціальність **122 «Комп’ютерні науки»**
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
____________________ Ю. П. Кондратенко
«____» _____________20___ р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Супруну Богдану Олександровичу
(прізвище, ім’я, по батькові)

1. Тема кваліфікаційної роботи магістра «Інтелектуальна система керування сценаріями та моделювання руху в Unity 3D».

Керівник роботи Гожий Олександр Петрович, д-р техн. наук, професор.

Затв. наказом Ректора ЧНУ ім. Петра Могили від «01» лютого 2024 р. № 20

2. Строк подання студентом роботи 20 лютого 2024 р.

3. Вхідні (початкові) дані до роботи: середовище створеної гри в Unity із системою управління автомобілем. Очікуваний результат: система моделювання руху автомобіля у середовищі.

4. Зміст пояснювальної записки (перелік питань, які потрібно розглянути):

– аналіз сучасного стану задачі моделювання руху в автомобіля в комп’ютерних іграх;

– огляд існуючих методів штучного інтелекту для управління агентом в ігровому середовищі;

– опис розробленої системи моделювання руху автомобіля у грі, створеній на основі Unity.

5. Перелік графічного матеріалу: презентація.

6. Завдання до спеціальної частини: __Охорона праці та безпека у надзвичайних ситуаціях.____

7. Консультанти:

Розділ	Прізвище, ініціали та посада консультанта	Підпис
Спеціальна частина з охорони праці	**д-р біол. наук, професор Григор'єва Л. І.**	
Методична частина	**д-р техн. наук, професор Гожий О. П.**	

Керівник роботи __д-р техн. наук, проф. Гожий О. П._______

(наук. ступінь, вчене звання, прізвище та ініціали)

____

(підпис)

Завдання прийнято до виконання ___Супрун Б. О.____________

(прізвище та ініціали)

____

(підпис)

Дата видачі завдання «__31__» ______жовтня____ 2023 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи магістра

Тема: Інтелектуальна система керування сценаріями та моделювання руху в Unity 3D

№	Найменування роботи	Початок	Закінчення	Примітки
1	Визначення керівника і теми КРМ. Подання заяви на затвердження теми та керівників КРМ	01.09.2023	10.10.2023	
2	Отримання завдання на виконання КРМ	11.10.2023	01.11.2023	
3	Складання календарного плану роботи на період виконання КРМ	08.11.2023	10.11.2023	
4	Огляд літератури за темою дослідження	20.11.2023	26.11.2023	
5	Проходження передатестаційної практики, збір та аналіз матеріалів до КРМ	27.11.2023	23.12.2023	
6	Аналіз предметної області та розробка технічного завдання	28.12.2023	12.01.2024	
7	Опис фахової частини КРМ, зокрема аналіз інструментів для розробки системи, реалізація змодельованої системи з аналізом отриманих результатів	13.01.2024	25.01.2024	
8	Розробка спеціальної частини з охорони праці та методичної частини	26.01.2024	02.02.2024	
9	Перший попередній захист КРМ на засіданні комісії кафедри	29.01.2024	29.01.2024	
10	Доробка та остаточне оформлення КРМ	07.02.2024	11.02.2024	
11	Другий попередній захист КРМ на засіданні комісії кафедри	12.02.2024	12.02.2024	
12	Подання КРМ, її електронної копії та інших документів (відгуку, рецензії) до захисту	19.02.2024	20.02.2024	
13	Захист КРМ перед екзаменаційною комісією (ЕК)	26.02.2024	27.02.2024	

Розробив студент Б. О. Супрун ____________
(прізвище та ініціали) (*підпис*)

Керівник роботи д-р техн. наук, проф. Гожий О. П. ____________
(наук. ступінь, вчене звання, прізвище та ініціали) *(підпис)*

«____» ______________ 2024 р.

АНОТАЦІЯ

до кваліфікаційної роботи магістра
студента групи 601 ЧНУ ім. Петра Могили

Супруна Богдана Олександровича

на тему: “**ІНТЕЛЕКТУАЛЬНА СИСТЕМА КЕРУВАННЯ СЦЕНАРІЯМИ ТА МОДЕЛЮВАННЯ РУХУ В UNITY 3D**”

Актуальність даного дослідження полягає необхідності підвищення ефективності створення сценаріїв розробленого ігрового середовища з моделюванням руху автомобіля, який проходить дорожній поворот по максимально бажаній траєкторії.

Об’єктом дослідження є процес моделювання руху автомобіля у ігровому середовищі.

Предметом дослідження є методи і алгоритми машинного навчання для моделювання руху в комп’ютерних іграх.

Метою дослідження є підвищення ефективності створення сценаріїв з моделюванням руху автомобіля у комп’ютерних іграх.

В результаті виконання було досліджено метод машинного навчання, який використовує генетичні алгоритми для еволюції нейронних мереж, розвиваючи їх структуру та ваги згідно з вимогами задачі, а також розроблено програмне забезпечення для моделювання руху автомобіля та реалізації відповідного методу.

Дана робота складається з чотирьох розділів. Кожен розділ відповідно присвячений: аналізу предметної області, алгоритмам та методам, використаним у магістерській роботі, аналізу інструментів для розробки, розробці системи та її тестуванню. Загальний обсяг роботи – 110 сторінок. Кваліфікаційна робота магістра містить три додатки, 31 рисунок і посилання на 45 літературних джерел.

Ключові слова: дрифт, Unity, симуляція руху автомобіля, нейроеволюція, NEAT.

ABSTRACT

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

Bohdan Suprun

"INTELLIGENT SCENARIO CONTROL AND MOTION SIMULATION SYSTEM IN UNITY 3D"

A relevance of this study is the need to increase the efficiency of creating scenarios for the developed game environment with modeling the movement of a car that passes a road turn along the most desirable trajectory.

An object of research is the process of modeling the movement of a car in a game environment.

A subject of the study is machine learning methods and algorithms for modeling motion in computer games.

A purpose of the study is to increase the efficiency of creating scenarios with car motion modeling in computer games.

As a result of this work, a machine learning method was investigated that uses genetic algorithms to evolve neural networks, developing their structure and weights according to the requirements of the task, and software was developed to model car movement and implement the corresponding method.

This consists consists of four sections. Each of them is devoted to: analysis of the subject area, algorithms and methods used in the master's thesis, analysis of development tools, system development and testing.

The overall scope of the work is 110 pages. Thesis contains 3 applications and 45 sources in it.

Key words: drift, Unity, car simulation, neuroevolution, NEAT.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ....4
ВСТУП....5
1 АНАЛІЗ ПРЕДМЕТНОЇ СФЕРИ. ПОСТАНОВКА ЗАДАЧІ....7
1.1 Кат-сцена....7
1.2 Дрифт....9
1.3 Ігровий рушій....11
1.4 Штучний інтелект в іграх....12
1.5 Аналіз аналогів....14
1.6 Постановка задачі....15
Висновки до розділу 1....17
2 АЛГОРИТМИ ТА МЕТОДИ ДЛЯ МОДЕЛЮВАННЯ РУХУ В СЕРЕДОВИЩІ UNITY....18
2.1 Навчання з підкріпленням....18
2.2 Генетичний алгоритм....22
Висновки до розділу 2....30
3 АНАЛІЗ ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ ТА МОДЕЛЮВАННЯ СИСТЕМИ....31
3.1 Unity 3D....31
3.2 NEAT-Python....37
3.3 Модуль Pickle....43
3.4 Мережеві бібліотеки....44
3.5 Моделювання системи....46
Висновки до розділу 3....51
4 РОЗРОБКА СИСТЕМИ ТА ТЕСТУВАННЯ....52
4.1 Створення сцени у Unity....52
4.2 Налаштування автомобіля....54
4.3 Написання логіки середовища....56

4.4 Написання скрипта для агента 57
4.5 Написання програми NEAT алгоритму 65
4.6 Тестування системи та аналіз результатів 67
Висновки до розділу 4 70
ВИСНОВКИ 71
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 72
ДОДАТОК А Лістинг коду програми середовища 75
ДОДАТОК Б Лістинг коду програми алгоритму NEAT 80

ПЕРЕЛІК СКОРОЧЕНЬ

ШІ – штучний інтелект

ГА – генетичний алгоритм

НМ – нейронна мережа

MDP – Марковський процес вирішування

API – application programming interface

RL – Reinforcement learning

NE – neuro-evolution

ВСТУП

Штучний інтелект у відеоіграх давно вийшов за рамки простих анімованих персонажів та статичних опонентів, перетворюючись на ключовий елемент, що визначає глибину та захопленість геймерами. Сьогодні він виступає не просто як програмний код, але як віртуальний супротивник, який вміє вчитися, адаптуватися та взаємодіяти з гравцем на рівні, що надто складно відрізнити віртуальну реальність від реальної.

Ігрова індустрія виявилася ідеальним полем для застосування штучного інтелекту. Сьогоднішні боти віддзеркалюють не лише задані алгоритми, але й імітують людське мислення, навчаються на власних помилках та розвивають унікальні стратегії, сприяючи виникненню великого різноманіття геймплею.

В цьому віртуальному світі штучний інтелект виступає не лише як механізм, який розуміє гравця, але і як виклик, що ставить перед ним завдання, аналізує його рішення та вдосконалює свою тактику. Від цього залежить не лише результат поєдинку, але й емоційний досвід гравця, який переходить на новий рівень завдяки непередбачуваній та інтелектуальній поведінці віртуальних опонентів.

Штучний інтелект у відеоіграх стає тим самим необхідним елементом, який формує нові реалії геймінгу та розширює можливості взаємодії з віртуальним світом. У цьому контексті, розгляд штучного інтелекту не просто як технічного вдосконалення, але як ключового актора, що збагачує та трансформує геймплей, відкриває нові горизонти для творчості розробників і створює захоплюючий, а часом і непередбачуваний, світ віртуальної реальності.

ШІ відкриває можливості реалізовувати поведінку акторів гри, чи то змодельованої сцени адаптуючись до середовища з усіма її змінними, чого іноді неможливо досягти використовуючи звичайні алгоритми.

Ігри автомобільних симуляторів у сучасному світі визнані не лише розважальним засобом, але й могутнім інструментом для навчання та

вдосконалення водійських навичок. Вони дозволяють гравцям експериментувати з різними автомобілями, налаштуваннями та стратегіями водіння.

При створенні і не тільки такого типу ігор існує задача створення ботів для імітації віртуального супротивника. Також є потреба у створенні кат-сцен, в яких імітується рух автомобіля. Штучний інтелект відкриває можливості реалізовувати поведінку акторів гри, адаптуючись до середовища з усіма її змінними, чого іноді неможливо досягти використовуючи звичайні алгоритми.

Об’єктом дослідження є процес моделювання руху автомобіля у ігровому середовищі.

Предметом дослідження є методи і алгоритми машинного навчання для моделювання руху в комп’ютерних іграх.

Мета дослідження – підвищення ефективності створення сценаріїв з моделюванням руху автомобіля у комп’ютерних іграх.

1 АНАЛІЗ ПРЕДМЕТНОЇ СФЕРИ. ПОСТАНОВКА ЗАДАЧІ

1.1 Кат-сцена

Внутрішньоігрове відео (англ. Cutscene; дослівно кат-сцена, також сінематика) – епізод у відеогрі у якому гравець продовжує контролювати події у супроводі інтерактивного фону, або не контролює дії на екрані взагалі. Кат-сцени роблять вимушену зупинку в геймплеї. Можуть бути як у вигляді анімації (зазвичай звуться cinematics (англ. сінематика) так і у вигляді кліпів, відзнятих з акторами.

Сінематика (cinematics) у відеоіграх представляє собою елемент геймдизайну, що включає в себе створення та використання кінематографічних елементів для розкриття сюжету [1], передачі настрою, а також поглиблення імерсивності геймерського досвіду. Це може бути у вигляді проміжних відеороликів, анімаційних вставок, інтерактивних сцен або будь-яких інших спеціально створених секцій, які виводять гравця з основного геймплею для подачі історії чи ключових подій.

Основні аспекти сінематики в іграх.

1. Сюжетна вставка. Сінематика дозволяє розробникам ефективно розповідати історію гри. Вона може включати в себе розкішно оформлені відеоролики з героями, які висловлюють діалоги, або інтерактивні сцени, де гравець має можливість взаємодіяти з подіями.

2. Підсилення настрою. Завдяки використанню кінематографічних прийомів, сінематика може ефективно підсилювати емоційний вплив гри. Вона дозволяє створювати неповторні атмосфери, відчуття напруження або радості, що поглиблює враження від ігрового процесу.

3. Розвиток персонажів. Сінематика відмінно підходить для розкриття характерів та їхнього розвитку протягом гри. Вона дозволяє показати багатогранність персонажів, їхні мотивації та зміни в ході історії.

4. Інтерактивність. В деяких випадках сінематика може бути інтерактивною, де гравець має можливість впливати на хід подій, приймаючи рішення або взаємодіючи з об'єктами навколишнього світу.

5. Подача ключових подій. Сінематика дозволяє виділити та підкреслити ключові моменти гри, такі як епічні битви, рішення важливих завдань чи розкриття секретів ігрового світу.

6. Графічна якість. Сінематика часто відрізняється високою якістю візуальних ефектів, анімацією та звуковим оформленням, що додає загальній графічній привабливості гри.

7. Підвищення імерсивності. Завдяки сінематичним елементам гравець відчуває себе більш втягнутим у світ гри, адже він стає учасником історії, а не лише спостерігачем.

Види сінематики:

– попередньо згенеровані сцени (Pre-rendered Cutscenes) – відео, що були попередньо обчислені та записані, часто використовуються для важливих моментів сюжету, оскільки дозволяють високу якість графіки та розрізняються від ігрового двигуна;

– ін-гейм сцени (In-engine Cutscenes) – сцени, що відтворюються за допомогою графічного двигуна гри, адаптовані під реальний ігровий світ, дозволяють безшовно переходити від сцени до геймплею.

Використання сінематики в іграх є важливим інструментом для розробників, оскільки вона сприяє глибшому розумінню гравцем ігрового світу. Сінематика у відеоіграх є не тільки засобом подачі історії, але і способом взаємодії з гравцем на більш особистому та емоційному рівні, роблячи ігровий досвід більш насиченим та глибоким.

1.1.1 Сінематика в іграх-гонках

Сінематика в іграх-гонках відіграє важливу роль у створенні захоплюючого геймплею, підвищенні імерсивності та підкресленні динаміки гонок. Цей елемент

геймдизайну використовується для введення гравців в атмосферу гоночного світу, підкреслення ключових моментів, а також для розвитку сюжету та персональності гонщика.

Аспекти сінематики з використанням авто в гонках.

1. Введення в гоночний світ, запальна вступна сцена: сінематика може починати гру з захопливого відеоролика, вводячи гравця в атмосферу світу гонок.

2. Представлення гоночного транспорту: гарно оформлені сцени можуть висвітлити різноманітні види гоночного транспорту та його можливості.

2. Епічні моменти гонок, запальні сцени спеціальних обгонів: відзначення особливих моментів гонок через вражаючі сцени, де гравець може виконати захоплюючі обгони або уникати аварій, чи просто проходити ефектно круті повороти на високій швидкості чи у заносі. Фінішні лінії та перемоги: сінематичні сцени під час перетинання фінішної лінії та святкування перемоги.

1.2 Дрифт

Дрифт – це захоплюючий і мистецький стиль керування автомобілем, який виник із світу гоночного спорту та перетворився в самостійну дисципліну. Це не лише технічне водіння, але й вираз особистого стилю та водійської майстерності.

Дрифт – це керування автомобілем під час руху, при якому задні колеса втрачають зчеплення з дорогою, і автомобіль починає пробуксовувати задніми колесами, одночасно тримаючись в певному куті відносно напрямку руху [2]. Це технічне водіння, яке вимагає від водія високого рівня вправності та контролю над автомобілем.

Він відрізняється від звичайних автоперегонів, які використовують максимальну швидкість або мінімальний час як показник продуктивності [3].

Дрифт останнім часом набув популярності як у професійних, так і в аматорських колах автоперегонів. Поява цього виду спорту значною мірою пояснюється заслугою японського автогонщика Кейічі Цучія, який вперше почав дрифтувати на регіональних гонках, щоб продемонструвати своє керування

автомобілем. Цучія продемонстрував здатність частково роз’єднувати курс і орієнтацію свого транспортного засобу, що часто призводило до великого ковзання або кутів дрифту.

Привабливість дрифту полягає в складності збереження контролю та бажаної траєкторії серед великих коливань керованості та стабільності автомобіля, обидва з яких частково залежать від кута бокового ковзання. Інші фактори, важливі для дрифту – це гоночна лінія, швидкість і в деяких випадках дим.

Для успішного дрифту необхідно обмежене зчеплення задніх коліс з дорогою. Це досягається використанням спеціальних шин із зменшеним коефіцієнтом зчеплення або регулюванням тиску у шинах.

Автомобілі з заднім приводом є ідеальними для дрифту, оскільки вони дозволяють водію легше керувати задніми колесами та сприяють ковзанню. Розуміння динаміки автомобіля щодо центру мас, розподілу ваги та інших аспектів є важливим для успішного дрифту.

Водій повертає передні колеса у напрямок, протилежний руху, щоб викликати ковзання задніх коліс. Раціональне використання гальма та активне керування газом допомагають утримати автомобіль в дрифті та контролювати кут повороту.

Дрифт може бути ініційований через обертання руля, роботу гальма та активне керування газом. Важливо мати правильний розгін для дрифту, а також визначити час для перемикання передач.

1.2.1 Фізика дрифту в Іграх

Фізика дрифту в іграх є ключовим елементом, що визначає реалістичність та задоволення від водіння віртуальних автомобілів.

Вірний рендеринг зчеплення автомобільних шин з дорогою є основою для реалістичного дрифту. Моделі зчеплення повинні враховувати такі фактори, як тип шин, стан асфальту, тиск у шинах та інші параметри. А якщо глибше, то може

знадобитись моделювання взаємодії автомобіля з різними типами покриття (асфальт, гравій, трава), щоб врахувати різницю в зчепленні та динаміці руху.

Механізм передачі сили на дорогу враховується через віртуальні шини [4], які реагують на різні умови, такі як мокрий асфальт, крутий кут нахилу та різні покриття. Застосування моделей фізики, що враховують силу тертя та ковзання, дозволяє реалістично відтворити динаміку керування в режимі дрифту.

Для вірного відтворення дрифту важливо враховувати положення центру мас автомобіля. Зрушення центру мас при надто гострому куті дрифту може викликати втрату контролю.

Щодо системи управління:

– для реалістичного дрифту важливо враховувати вплив керування на кут дрифту та нахил автомобіля;

– високий ступінь повороту передніх коліс сприяє легшому входженню в дрифт та управлінню автомобілем під час нього;

Також, з більш візуальних факторів, якщо розглядати дрифт у контексті сінематики, важливою механікою являється кут повороту та перекидування. Кут повороту вказує на те, наскільки автомобіль повернувся відносно його руху. Це реалістично впливає на відтворення ефекту дрифту. Перекидування визначає, наскільки автомобіль змінює свій кут під час руху, що може додати більше реалізму при дрифті.

Фізика дрифту в іграх є складним завданням, яке вимагає детального моделювання численних фізичних аспектів. Реалістичне відтворення зчеплення, динаміки руху та взаємодії з поверхнею є важливим для того, щоб гравець відчував автентичність та задоволення від дрифту в іграх.

1.3 Ігровий рушій

Ігровий рушій – це програмне забезпечення, спроектоване переважно для створення відеоігор і, зазвичай, включає відповідні бібліотеки та програми підтримки, такі як редактор рівнів [5].

Розробники можуть використовувати ігрові механізми для створення ігор для консолей, комп'ютерів, смартфонів тощо. Основні функції ігрового двигуна включають механізм візуалізації ("рендеринг") для 2D або 3D-графіки, фізичний двигун, звук, сценарії, анімацію, штучний інтелект, мережеву підтримку, потокову передачу, управління пам'яттю, потоки, підтримку локалізації, графіку сцени та моделювання сінематиків чи створення відео для кінематографії. Розробники ігрових двигунів часто раціоналізують процес розробки ігор, використовуючи той самий двигун для створення різних ігор.

Розробники ігрових рушіїв активно працюють над тим, щоб задовольнити потреби розробників, створюючи надійні пакети програмного забезпечення, які включають багато необхідних елементів для розробки гри.

Одною з ключових функціональностей ігрового двигуна являється фізика комп’ютерної анімації або фізика гри. Це закони фізики, визначені в симуляції чи відеогрі, а також логіка програмування, яка використовується для реалізації цих законів. Ігрова фізика значно відрізняється за ступенем подібності до фізики реального світу. Іноді фізика гри може бути розроблена так, щоб імітувати фізику реального світу настільки точно, наскільки це можливо, щоб виглядати реалістичною для гравця або спостерігача. В інших випадках ігри можуть навмисно відхилятися від фактичної фізики для ігрових цілей. Поширені приклади в іграх на платформі включають можливість почати рух горизонтально або змінити напрямок у повітрі та здатність подвійного стрибка, яку можна знайти в деяких іграх. Встановлення значень фізичних параметрів, таких як рівень присутньої сили тяжіння, також є частиною визначення фізики гри в конкретній грі.

1.4 Штучний інтелект в іграх

У відеоіграх штучний інтелект (AI) використовується для генерації адаптивної або інтелектуальної поведінки головним чином у неігрових персонажів (NPC), подібних до людського інтелекту [6]. Штучний інтелект у відеоіграх є окремою підсферою.

Термін «ігровий ШІ» використовується для позначення широкого набору алгоритмів, які також включають методи з теорії управління, робототехніки, комп'ютерної графіки та інформатики загалом, тому штучний інтелект у відеоіграх часто не є «справжнім ШІ», оскільки такі методи не обов'язково сприяють комп'ютерному навчанню чи іншим стандартним критеріям, вони лише становлять «автоматизоване обчислення» або заздалегідь визначений та обмежений набір відповідей на заздалегідь визначений та обмежений набір вхідних даних.

Як раз люди у сфері штучного інтелекту і стверджують, що штучний інтелект у відеоіграх – це не справжній інтелект, а скоріше абревіатура, яка використовується для опису комп'ютерних програм, що використовують алгоритми, щоб створити ілюзію розумної поведінки, даруючи програмне забезпечення з оманливою аурою наукової чи технологічної складності та прогресу. Хоча зараз це і не зовсім так, ігрова індустрія та внутрішньо-ігрові середовища являються широким полігоном для впровадження нових технологій машинного навчання.

Методи штучного інтелекту та машинного навчання використовуються у відеоіграх для широкого спектру додатків, таких як керування неігровим персонажем (NPC) і процедурна генерація (PCG). Машинне навчання – це підмножина штучного інтелекту, яка використовує дані для створення прогнозних і аналітичних моделей.

Найвідомішим застосуванням машинного навчання в іграх є використання агентів глибокого навчання, які конкурують із професійними гравцями в складних стратегічних іграх. Було застосовано машинне навчання в таких іграх, як Atari/ALE, Doom, Minecraft, StarCraft, автомобільних гонках, Dota 2 тощо. В інших іграх, які спочатку не існували як відеоігри, такі як шахи, також було використано агентів-супротивників із застосуванням машинного навчання.

1.5 Аналіз аналогів

AI-гонщик GT Sophy від Sony здобув перемогу над найкращими гравцями Gran Turismo 7 і тепер освоїв навички дрифту. У відео із заходу Gran Turismo World Series 2023 видно, як керований штучним інтелектом автомобіль з легкістю та мистецтвом проходить дрифт трасою. Це чудове досягнення, враховуючи, що більшість людей зазнають труднощів з вдалим дрифтом навіть у менш реалістичних іграх, таких як Mario Kart.

GT Sophy, розроблений Sony AI та Sony Interactive Entertainment протягом шести років, використовує методи глибокого навчання із підкріпленням. Ці методи дозволяють ефективно керувати цифровим гоночним автомобілем у рамках структури та обмежень гри. Спочатку бот володів навичками гонок, такими як підтримка в потоці, обгін та блокування, а ось дрифт став новим доповненням до його репертуару.

Sony заявила, що GT Sophy тут, щоб залишитися, плануючи зробити цю систему постійною частиною гри.

Щоб провести оновлення, Sony довелося внести значні покращення у базову технологію гри. Sophy отримала доступ до 340 автомобілів на 9 різних трасах. При цьому гравець може вибрати трасу на свій смак, а Sophy вибере автомобіль на основі гравця, що вже є в гаражі. Тобто ШІ-суперник не опиниться у невигідному для себе положенні на трасі, а зможе конкурувати з гравцем на високому рівні.

Хоч і Gran Turismo Sophy створена за принципами машинного навчання, та в цьому плані вона не новатор. Раніше розробники гоночних ігор вже застосовували цю технологію або експериментували з її використанням. Один із найвдаліших і найяскравіших прикладів – система Drivatar у серії ігор Forza. Вона використовується в ній з 2005 року і вважається однією з довготривалих систем машинного навчання в ігровій індустрії.

Якщо коротко, ШІ навчається у гравця, як керувати автомобілем. Але особливість системи Drivatar полягає в тому, що вона не просто повторює за

гравцем його дії. На основі знань, отриманих про ваше вміння їздити, вона робить реалістичні припущення про те, як би ви повелися в ситуації, в якій раніше ніколи не виявлялися. Грубо кажучи, ШІ може проїхатися незнайомою вам трасою так, як це зробили б ви.

Ця система працює на основі нейромережі – вона обробляє всі можливі рішення, які міг прийняти гравець на трасі, і призначає певну вагу тим зв'язкам нейронів, які найчастіше відповідають поведінці гравця.

А щоб нейромережа не починала повторювати той самий сценарій поведінки, замість звичайних нейромереж розробники використовували так звані Байєсовські мережі. Якщо сильно спростити, в них зв'язки між нейронами не представлені конкретними числами, що позначають вагу цього зв'язку, а ймовірностями цих цифр.

1.6 Постановка задачі

Моделювання руху автомобіля в комп’ютерних іграх являється досить важкою задачею. Особливо з’являється потреба у моделюванні руху авто для відтворення кінематографічної сцени у грі.

Для відтворення реалістичних рухів персонажів чи акторів у грі досить часто замість розробки анімацій вручну, використовують технології захоплення руху (motion capture), тим паче використання технології стає все більш доступним. Але що стосовно моделювання руху авто по заданій мапі на сцені, все ж неможливо знайти ідеальну локацію, схожу на змодельовану. Також становить проблемою підготувати обладнання, авто, найняти гонщика тощо.

Зазвичай у відеоіграх саме для катсцен використовують інструменти для анімації руху авто по заданій траєкторії у сцені. Також деякі розробники вже використовують алгоритми штучного інтелекту, але вони зазвичай мають універсальну поведінку, та частіше ШІ використовують для створення супротивника для гравця, а не для ефектного заїзду по локації.

Рух авто у заносі (дрифт) має більш складні механіки, та для його реалізації потрібно враховувати більше фізичних властивостей автомобіля та взаємодії його з дорогою. Ігровий рушій має свої змодельовані у ньому закони чи механіки для реалізації чи то звичайного руху авто, чи у заносі. Ефектний рух авто у заносі для відтворення кінематографічної сцени у грі вимагає проходження автомобілем повороту на максимально можливій швидкості з максимально можливим поворотом автомобіля відносно його напрямку руху та мінімальною відстанню від зовнішньої границі повороту дороги, по якій авто здійснює маневр, зазвичай такі параметри і оцінюються у змаганнях дрифту в автоспорті, так як виглядають найбільш видовищно.

Для реалізації такого маневру у ігровому середовищі зі змодельованою віртуальною фізикою можна використати статичну анімацію, тобто прокладання траєкторії руху автомобіля, але не можливо урахувати усіх механік чи фізичних властивостей, які були змодельовані, тому рух може бути не достатньо реалістичним. Також є можливість передати управління, яке буде використовуватись у грі, людині, якій знадобиться невідома кількість спроб для отримання бажаного результату, та зберегти усі параметри її вдалого руху і відтворити при генерації кат-сцени. Але для кожної нової подібної задачі з іншими параметрами (характеристики авто, локація дороги) очевидно, що подібна техніка буде повторюватись.

Постає потреба у створенні універсальної системи ШІ для управління автомобілем у розробленому ігровому середовищі, для проходження дорожніх поворотів у заносі по максимально бажаній треєкторії.

Задача полягає у створенні ШІ агента для моделювання руху автомобіля у певній ігровій локації. Вхідними параметрами системи являються обертаючий момент автомобіля, його швидкість та поворот у просторі відносно вектору його руху, поворот колісної бази та “сенсори” границь треку, по якому може рухатись авто. Для створення кат-сцени та моделювання фізики руху авто використовується ігровий рушій Unity, скриптовою мовою якого являється C#. На основі мови Python

виконується реалізація інтелектуальної системи. Її основою являється поєднання Генетичного алгоритму та методу машинного навчання – Навчання з підкріпленням. Комбінація генетичного алгоритму та навчання з підкріпленням можлива тому, що ці два підходи не є взаємовиключними. Точно також, як два природні принципи, з яких вони випливають, можуть співіснувати, можуть співіснувати ці підходи.

Висновки до розділу 1

Було проаналізовано принципи побудови кат-сцени в іграх-гонках. Було запропоновано реалізацію інтелектуальної системи для моделювання руху автомобіля у середовищі ігрового рушія. Було досліджено принципи руху автомобіля у заносі та саме у іграх. Проаналізовано систему ШІ компанії Sony, розробленої для гри Grand Turismo 7, компанії Polyphony Digital.

2 АЛГОРИТМИ ТА МЕТОДИ ДЛЯ МОДЕЛЮВАННЯ РУХУ В СЕРЕДОВИЩІ UNITY

2.1 Навчання з підкріпленням

Навчання з підкріпленням – це один із основних способів машинного навчання, в якому агент, що самонавчається, повинен приймати рішення в певному середовищі таким чином, щоб отримувати найчастіше нагороду за вчинені правильні дії.

Навчання з підкріпленням складається із ключових елементів (див. рис. 2.1):

– агент (agent) – програма, яку тренують для виконання певної роботи (здійснює дії, які впливають на навколишнє середовище);

– середовище – реальний або віртуальний світ, в якому діє агент (усі дії, які здійснює агент навчання з підкріпленням, безпосередньо впливають на середовище, воно приймає поточний стан і дію агента як інформацію та повертає винагороду агенту з новим станом);

– дія (action) – це набір усіх можливих операцій/ходів, які може зробити агент, він приймає рішення про те, яку дію виконати з набору дискретних дій;

– нагорода – оцінка правильності дії, можливо, як позитивної, і негативної (середовище дає зворотній зв’язок, за допомогою якого визначається обґрунтованість дій агента в кожному стані, це має вирішальне значення в сценарії Reinforcement Learning, коли потрібно, щоб машина вчилася сама, і єдиним критиком, який допоможе їй у навчанні, є зворотній зв’язок/винагорода, яку вона отримує);

– стан (state) – це конкретна ситуація, в якій знаходиться агент.

Мета навчання з підкріпленням полягає в тому, щоб агент вивчив оптимальну чи майже оптимальну політику, яка максимізує «функцію винагороди» або інший сигнал, що надається користувачем підкріплення, що накопичується внаслідок негайної винагороди.

Це схоже на процеси, які, відбуваються в психології тварин. Наприклад, біологічний мозок налаштований інтерпретувати такі сигнали, як біль і голод, як негативні підкріплення, і інтерпретувати задоволення та споживання їжі як позитивні підкріплення. За деяких обставин тварини можуть навчитися брати участь у поведінці, яка оптимізує ці винагороди. Це свідчить про те, що тварини здатні до навчання з підкріпленням [7].

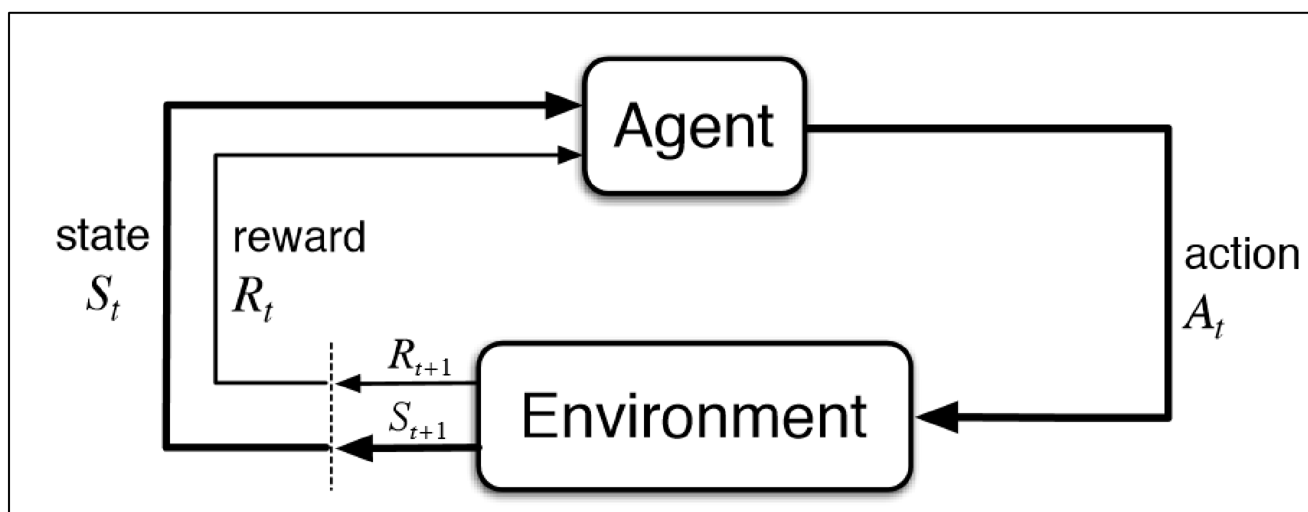

Рисунок 2.1 – навчання з підкріпленням

Навчання з підкріпленням особливо добре підходить для вирішення проблем, які включають компроміс між довгостроковою і короткостроковою винагородою.

Агент навчається досягати мети в нерозвіданому, потенційно складному середовищі, яким може бути, наприклад, тривимірне віртуальне середовище в ігровому рушії зі своїми змінними. При навчанні з підкріпленням штучний інтелект стикається з ігровою ситуацією та використовує метод спроб і помилок, щоб знайти рішення задачі. Заохочення правильних дій чи покарання за неправильні відбувається за рахунок відгуків середовища, що є сигналами підкріплення агента. Головною метою агента є отримання якнайбільшої нагороди.

Основна мета того, хто створює систему, це не дати моделі жодних підказок як вирішити поставлене завдання, а лише встановити політику нагороджень та покарань. А вже сама модель намагатиметься знаходити способи, щоб вирішити

завдання або, іншими словами, максимізувати нагороду, починаючи з абсолютно випадкових методів проб і поступово переходячи до складніших тактик.

Використовуючи можливості пошуку та безліч ітерацій випробувань, навчання з підкріпленням в даний час є найефективнішим способом демонстрації творчого потенціалу машини. На відміну від людей, штучний інтелект може навчатися, запускаючи паралельно безліч симуляцій, якщо звичайно виконується на дуже потужних комп'ютерах. При паралельному навчанні можна як залишати можливість агентам взаємодіяти один з одним (якщо це входить у рішення поставленого завдання), так і зробити їх незалежними один від одного.

2.1.1 Алгоритми навчання на основі моделі та без моделі

Існує два основних типи алгоритмів навчання з підкріпленням: алгоритми на основі моделі, безмодельні алгоритми.

Алгоритми на основі моделі використовують функцію переходу та винагороди для оцінки оптимальної політики. Вони використовуються в сценаріях, коли існує повна інформація про оточення та те, як воно реагує на різні дії. У RL на основі моделі агент має доступ до моделі середовища, тобто дії, які необхідно виконати для переходу з одного стану в інший, доданих ймовірностей і доданих відповідних винагород. Вони дозволяють агенту планувати наперед. Для статичних/фіксованих середовищ більше підходить навчання з підкріпленням на основі моделі.

Безмодельні алгоритми знаходять оптимальну політику з дуже обмеженими знаннями про динаміку середовища. Вони не мають жодної функції переходу/винагороди для визначення найкращої політики. Вони оцінюють оптимальну політику безпосередньо з досвіду, тобто взаємодії між агентом і середовищем, не маючи жодного натяку на функцію винагороди.

RL без моделі слід застосовувати в сценаріях, що включають неповну інформацію про середовище. У реальному світі ми не маємо фіксованого середовища. Безпілотні автомобілі мають динамічне середовище зі змінними

умовами руху, відхиленнями маршрутів тощо. У таких сценаріях безмодельні алгоритми перевершують інші методи.

2.1.2 Марковський процес прийняття рішень

Процес прийняття рішень за Марковим – це алгоритм навчання з підкріпленням, який дає спосіб формалізувати послідовне прийняття рішень.

Ця формалізація є основою для проблем, які вирішуються за допомогою навчання з підкріпленням. Компонентами марковського процесу прийняття рішень (MDP) є особа, яка приймає рішення, яка називається агентом, який взаємодіє з середовищем, у якому він знаходиться. Ці взаємодії відбуваються послідовно протягом тривалого часу.

У кожний момент часу агент отримає деяке представлення стану середовища. Враховуючи це представлення, агент вибирає дію, яку потрібно виконати. Потім середовище переходить у якийсь новий стан, і агент отримує винагороду як наслідок його попередньої дії.

Процес вибору дії із заданого стану, переходу в новий стан і отримання винагороди відбувається послідовно знову і знову. Це створює так звану траєкторію, яка показує послідовність станів, дій і винагород.

Протягом усього процесу агент навчання з підкріпленням відповідає за максимізацію загальної суми винагород, які він отримав від виконання дій у заданих станах середовища. Агент хоче не тільки максимізувати негайну винагороду, але й кумулятивну винагороду, яку він отримує за весь процес.

Важливим моментом, який слід зазначити щодо марковського MDP, є те, що він не турбується про негайну винагороду, а має на меті максимізувати загальну винагороду за всю траєкторію. Іноді він може віддати перевагу отримати невелику винагороду в наступну позначку часу, щоб згодом отримати вищу винагороду.

2.2 Генетичний алгоритм

Генетичний алгоритм – це алгоритм пошуку, натхненний біологічним процесом природного відбору, в якому особини ті, що пристосувалися до змін у їхньому середовищі, можуть вижити і дати потомство. Алгоритм належить до класу еволюційних алгоритмів. В ньому задіяні такі операції як мутація, схрещування та відбір [8].

Алгоритм заснований на аналогії з генетичною структурою та поведінкою хромосом у популяцій:

– особи у популяції конкурують за ресурси та розмноження;

– ті особини, які є успішнішими (пристосованими), потім розмножуються та дають більше потомства, ніж інші;

– гени від найпристосованого батька передаються через покоління, що іноді призводить до появи потомства більш пристосованого, ніж батьки;

– кожне наступне покоління дедалі більше пристосовується до середовища.

Кожній особі надається оцінка пристосованості, яка показує, наскільки особа добре змагається. ГА передбачає популяцію з n особин (хромосома/рішення) та їх відповідні оцінки пристосованості. Особам, які мають великі показники пристосованості, дається перевага на розмноження, гени самих успішних змішуються, особини менш успішні замінюються новими і таким чином, згодом слабкі поступово вибувають із змагання. Кожне нове покоління має в середньому більше «кращих генів», ніж особи (рішення) попередніх поколінь. Як тільки зроблене потомство досягне того, що не матиме суттєвих відмінностей від потомства, виробленого попередніми популяціями, популяція сходиться.

2.2.1 NEAT

NeuroEvolution of Augmenting Topologies (NEAT) – це генетичний алгоритм (GA) для створення штучних нейронних мереж, що розвиваються (техніка нейроеволюції), розроблений Кеннетом Стенлі та Рісто Мійкулайненом у 2002 році

під час навчання в Техаському університеті в Остіні. Він змінює як вагові параметри, так і структуру мереж, намагаючись знайти баланс між придатністю розроблених рішень та їх різноманітністю. Він заснований на застосуванні трьох ключових методів: відстеження генів за допомогою історичних маркерів, щоб дозволити кросинговер між топологіями, застосування видоутворення (еволюція видів) для збереження інновацій і поступова розробка топологій із простих початкових структур.

Традиційно топологію нейронної мережі обирає людина-експериментатор, а ефективні значення ваги з’єднання визначаються за допомогою процедури навчання. Це призводить до ситуації, коли процес спроб і помилок може знадобитися для визначення відповідної топології. NEAT є прикладом штучної нейронної мережі з розвитком топології та вагових коефіцієнтів (TWEANN), яка намагається одночасно змінювати значення ваги та відповідну топологію для нейронної мережі.

Нейроеволюція шукає в просторі поведінки мережу, яка добре справляється з поставленим завданням. Цей підхід до вирішення складних завдань управління є альтернативою статистичним методам, які намагаються оцінити корисність конкретних дій у певних станах середовища. Оскільки NE шукає поведінку, він ефективний у задачах з безперервними та багатовимірними просторами станів.

У традиційних підходах NE топологія мереж, що розвиваються, вибирається до початку експерименту. Зазвичай топологія мережі є вхідними та вихідними нейронами, де кожен вхідний нейрон підключений до кожного мережевого виходу [9].

Еволюція досліджує простір ваг зв'язків цієї повністю зв'язкової топології, дозволяючи відтворювати високопродуктивні мережі. Ваговий простір досліджується за допомогою перетину векторів ваг мереж та мутації ваг окремих мереж. Таким чином, метою NE з фіксованою топологією є оптимізація ваги зв'язків, які визначають функціональність мережі.

2.2.1.1 Генетичне кодування

Схема генетичного кодування NEAT розроблена таким чином, щоб можна було легко виставити відповідні гени у лінію при операції кросинговера двух геномів. Геноми представляють собою лінійне представлення зв'язків між. Кожен геном включає в себе гени зв'язків, кожен з яких відноситься до двох генів вузлів, між якими цей зв'язок. Гени вузлів представляють список входів НМ, прихованих вузлів та виходів, які можна з'єднати. Кожен ген з'єднання визначає вхідний і вихідний вухол цього з'єднання, вагу, enable bit (чи активований ген з'єднання) та innovation number, що дозволяє знайти відповідні гени.

Мутація в NEAT може змінити як вагу з’єднання, так і структуру нейронної мережі. Вага з’єднань змінюється як і в звичайній системі NE, при цьому кожне з’єднання або змінюється або ні в кожному поколінні. Структурні мутації можуть проходити двома способами. Кожна мутація збільшує розмір генома шляхом додавання генів. У першому способі додається один новий ген з’єднання із випадковою вагою для з’єднання двох вузлів, які раніше не були зв’язані. У другому способі (мутація додавання вузла) на місці існуючого зв’язку розміщується новий вузол. Минуле з’єднання вимикається та в геном додаються два нових з’єднання (нове з’єднання, яке веде у новий вузол, отримує вагу 1, а нове з’єднання, яке веде від нового вузла, отримує вагу, що мало попереднє з’єднання, яке було на місці нового вузла). Такий метод додавання вузлів (додавання на місці існуючого з’єднання, та встановлення ваг, які були указані раніше) використовується для того, щоб мінімізувати початковий ефект мутації. Новий вузол дещо змінює структуру але початковий ефект малий, тобто у мережі буде можливість оптимізувати та використати нову структуру, не підпадаючи під вимирання.

Завдяки мутації геноми алгоритму NEAT будуть постійно збільшуватись. В результаті будуть утворюватись геноми різного розміру, також іноді з різними з’єднаннями в одних і тих же позиціях. З великою кількістю різних топологій та

комбінацій вагових коефіцієнтів існує проблема кросинговера двох генотипів, яку вирішує відслідковування генів за допомогою історичних маркерів.

2.2.1.2 Історичні маркери

У контексті алгоритму NEAT існує інформація, яка говорить, які саме гени з якими генами співпадають любого індивіда у топологічно різноманітній популяції. Ця інформація представляє собою історичне походження кожного гена. Два гена одного й того ж самого історичного походження повинні представляти одну й ту саму структуру, хоча може і з різними вагами, так як вони обидва пішли від спільного батьківського гена в якийсь момент у минулому. Таким чином, все, що потрібно знати системі, щоб розуміти які гени з якими співпадають – це відслідковувати історичне погодження кожного гена у системі. На рис. 2.2 зображено приклад кодування геному, у якого після структурної мутації з’явилися гени з історичними маркерами 4 та 5 при додаванні нового гена вузла під номером 5, та ген зв’язку з історичним маркером 2 було вимкнено, так як новий вузол з’явився на його місці.

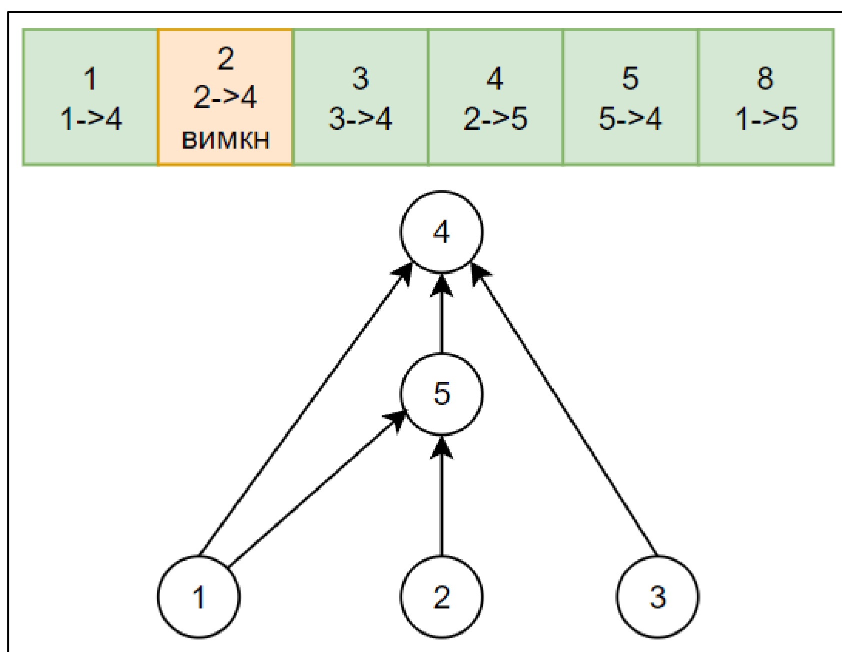

Рисунок 2.2 – приклад кодування генома 1

Відслідковування історичного походження не потребує дуже великих обчислень. Кожні, з самого початку, гени мають свій унікальній номер, та у системі існує глобальний інноваційний номер, що відповідає кількості таких генів. Кожен раз, коли з'являється новий ген (у результаті структурної мутації), глобальний інноваційний номер збільшується та присвоюється цьому гену. Таким чином, числа інновацій представляють собою хронологію появи кожного гена у системі.

Існує проблема, яка полягає в тому, що та сама структурна мутація може отримати різні номери в тому самому покоління, якщо вона випадковим чином відбулася більше одного разу. Однак, зберігаючи список інноваційних номерів, які відбулися в поточному поколінні, можна гарантувати, що коли та сама структура виникає більше одного разу через незалежні мутації в тому самому поколінні, кожній ідентичній мутації призначається однаковий інноваційний номер.

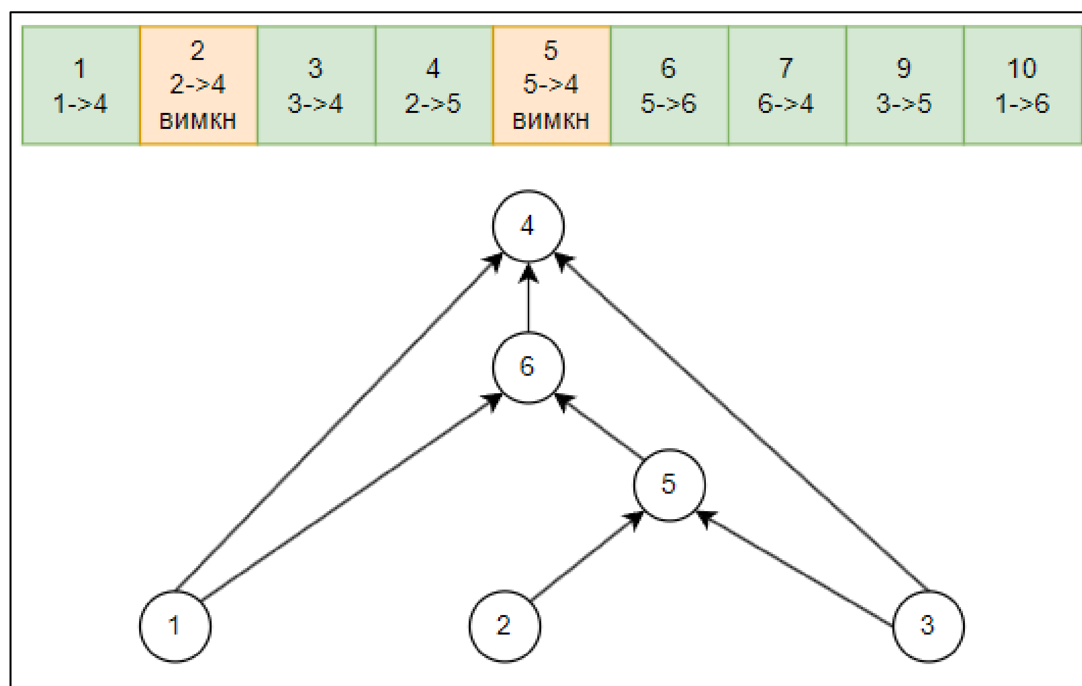

Рисунок 2.3 – приклад кодування генома 2

Історичні маркери надають алгоритму нові потужні можливості. Тепер система точно може знати, які гени з якими збігаються. При кросинговері гени в обох геномах з однаковими інноваційними номерами співставляються, ці гени

називають відповідними генами. Гени, які не збігаються, є або роз'єднаними, або надлишковими, залежно від того, чи зустрічаються вони в межах чи поза межами діапазону чисел інновації іншого батька, вони представляють структуру, якої немає в іншому геномі. У формуванні нащадків гени випадковим чином вибираються від будь-якого з батьків за відповідними генами, тоді як усі надлишкові або непересічні гени завжди включаються від більш придатного батька. Таким чином історичні маркери дозволяють алгоритму виконувати кросинговер з використанням лінійних геномів без необхідності дорогого топологічного аналізу. На рис. 2.4 зображено операцію кросинговеру двох геномів з відповідними, надлишковими та непересічними генами, які були зображені на рис. 2.2 та на рис. 2.3.

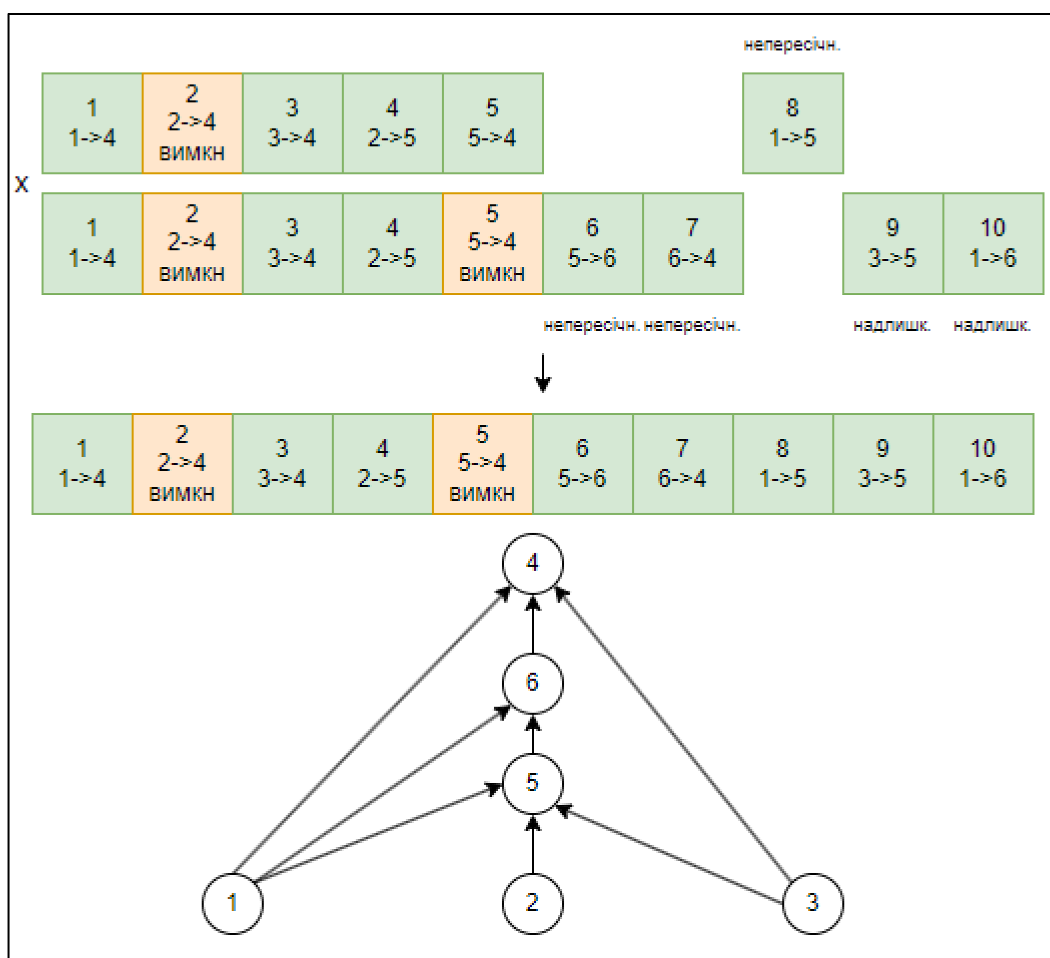

Рисунок 2.4 – операція кросинговера двох геномів

Додаючи нові гени до популяції та якісно сполучаючи геноми, які представляють різні структури, система може сформувати популяцію з різноманітною топологією. Однак виявляється, що така популяція сама по собі не може підтримувати топологічні інновації. Оскільки менші структури оптимізуються швидше, ніж великі структури, а додавання вузлів і з’єднань зазвичай спочатку знижує придатність мережі, нещодавно доповнені структури мають мало надії пережити більше одного покоління, навіть незважаючи на те, що інновації, які вони представляють, можуть мати вирішальне значення для вирішення завдання в довгостроковій перспективі. Рішення полягає в тому, щоб захистити інновації шляхом специфікації видів, тобто видоутворення.

Розділення популяції дозволяє індивідам конкурувати в основному в межах своїх власних ніш, а не з популяцією в цілому. Таким чином, топологічні інновації захищені в новій ніші, де вони мають час оптимізувати свою структуру через конкуренцію всередині ніші. Ідея полягає в тому, щоб розділити популяцію на види таким чином, щоб подібні топології були в одному виді. Історичні маркери являються ефективним параметром для визначення різниці між геномами та виявлення належності індивіда популяції до окремого виду.

Кількість надлишкових і непересічних генів між парою геномів є природною мірою відстані їх сумісності. Чим більше розрізняються два геноми, тим менше вони мають спільну еволюційну історію, а отже, менш сумісні.

2.2.1.3 Послідовність алгоритму

Як і будь-якому генетичному алгоритмі створюється початкова популяція.

Спочатку кожна особина складається лише з пронумерованих вхідних та вихідних нейронів. Кожен вихідний нейрон пов'язаний із вихідним. Тобто перед нами класичний одношаровий персептрон.

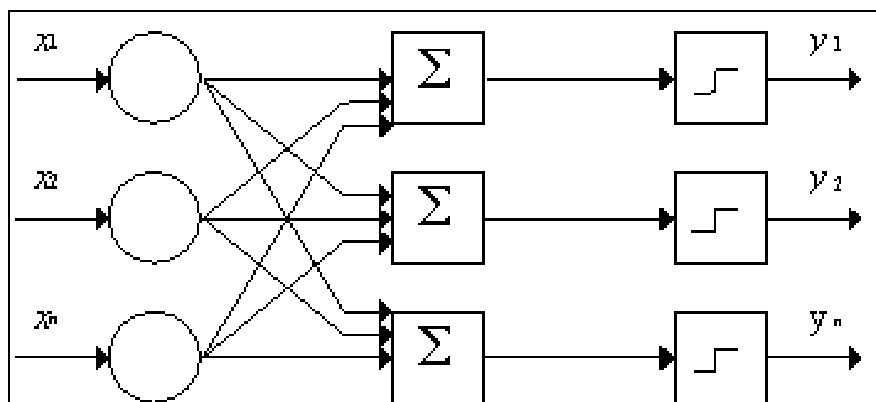

Рисунок 2.5 – Одношаровий персепрон

Далі до кожної особи застосовується фітнес-функція. Вона визначає те, наскільки ця особина підходить для виконання покладених на неї функцій. На входи подаються дані і порівнюються з очікуваними виходами. Або проводиться симуляція, за допомогою ігор, фізичних двигунів, або фізичних стимуляторів роботів.

Ця фітнес-функція ранжує всю популяцію. Ті особини, які виявилися кращими за інших, матимуть великі ймовірності на "продовження роду", але навіть гірші мають невеликий статистичний шанс. Це допомагає уникнути скочування популяції в локальний мінімум/максимум.

Далі алгоритм відбирає особин з урахуванням отриманих ймовірностей та відбувається "схрещування". Саме тут алгоритм NEAT хороший, бо має прості правила "двостатевого" схрещування генів. Найчастіше саме схрещування – найскладніша частина генетичних алгоритмів.

За допомогою схрещування генерується повністю нова популяція (у деяких версіях беруться кілька особин колишнього покоління та додаються до нового покоління). Після чого відбувається мутація – або значення ваги з певною ймовірністю змінюються, або, як в алгоритмі NEAT, випадково додаються нові зв'язки/нейрони. Після цього цикл повторюється. Таких циклів/популяцій може бути дуже багато.

Висновки до розділу 2

Було проаналізовано методи, за допомогою яких можна оптимізувати НМ для моделювання руху автомобіля у ігровому середовищі. Гра може мати складні, змінні та непередбачувані умови, в яких агент повинен приймати рішення. NEAT може адаптуватися до цих умов, еволюціонуючи або змінюючи структуру мережі, щоб вирішити нові або змінені виклики. NEAT може бути ефективним в розв'язанні складних завдань навчання з підкріпленням, одним із яких являється навчання ігрового агента, де не завжди можна знайти оптимальні рішення за допомогою традиційних методів.

3 АНАЛІЗ ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ ТА МОДЕЛЮВАННЯ СИСТЕМИ

3.1 Unity 3D

Unity – це кросплатформний ігровий рушій, розроблений компанією Unity Technologies. Він був представлений вперше у червні 2005 року. Починаючи з цього моменту, Unity постійно розширював свою функціональність, щоб підтримувати різноманітні платформи, такі як настільні комп'ютери, мобільні пристрої, ігрові консолі та платформи віртуальної реальності. Особливо відомий Unity своєю популярністю серед розробників мобільних ігор для платформ iOS і Android. Вважається простим у використанні, що робить його популярним серед початківців та розробників незалежних ігор [10]. Unity підтримує розробку як тривимірних (3D), так і двовимірних (2D) ігор, а також надає можливості для інтерактивного моделювання та іншого ігрового досвіду.

Unity дозволяє користувачам творити ігри як у форматі 2D, так і 3D. Механізм надає основний API скриптів на мові програмування C#. Перш ніж мова програмування C# стала основною для рушія, підтримувались мови Boo, яка була припинена з виходом Unity 5, і реалізація JavaScript, відома як UnityScript, яка застаріла в серпні 2017 року після релізу Unity 2017.1 на користь C#.

Доступні два окремі пайплайни візуалізації, High Definition Render Pipeline (HDRP) і Universal Render Pipeline (URP, раніше LWRP). Усі три render pipelines несумісні один з одним. Unity пропонує інструмент для оновлення шейдерів за допомогою застарілого рендерера до URP або HDRP.

Розробники чи творці можуть розробляти та продавати створені ресурси іншим розробникам ігор через Unity Asset Store. Він включає 3D- і 2D-асети та середовища, які розробники можуть купувати та продавати. Unity Asset Store було запущено в 2010 році.

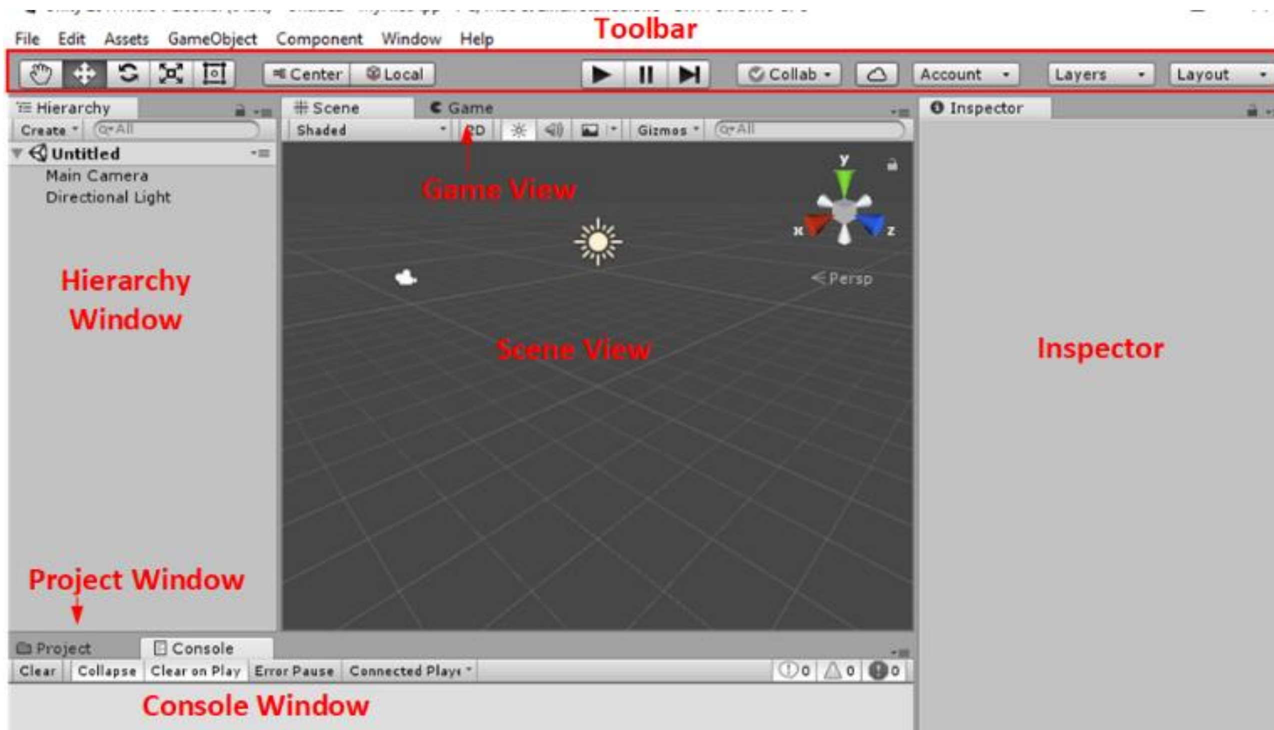

Рисунок 3.1 – Базовий інтерфейс Unity

Редактор Unity підтримується на платформах Windows, macOS і Linux, а сам механізм наразі підтримує створення ігор для понад 19 різних платформ, включаючи мобільні пристрої, настільні комп’ютери, консолі та віртуальну реальність. Unity 2022 LTS офіційно підтримує такі платформи:

– мобільні пристрої (iOS, Android, tvOS);

– персональні комп’ютери (Universal Windows Platform, Mac, Linux);

– веб-платформа WebGL;

– ігрові консолі (PlayStation 4, PlayStation 5, Xbox One, Xbox Series X/S Nintendo Switch);

– віртуальна реальність.

3.1.1 Фізика в Unity

Щоб фізична поведінка була правдоподібною, об'єкт у грі потрібно правильно прискорити та задіяти зіткнення, гравітацію та інші сили. Фізичний двигун Unity забезпечує вас компонентами для обробки симуляції фізики. За

допомогою налаштування всього кількох параметрів, можна створити об'єкти, які поводяться пасивно реалістично (тобто вони будуть переміщені в результаті зіткнень і падінь, але не почнуть рухатися самі по собі). Керуючи фізикою зі скриптів, можна надати об'єкту динаміку автомобіля, машини або навіть рухомого шматка тканини. Існують такі компоненти фізики в Unity.

1. Rigidbody (тверде тіло) – це основний компонент, що дає можливість ігровим об'єктам взаємодіяти за допомогою фізики. З прикріпленим Rigidbody об'єкт негайно почне реагувати на гравітацію. Якщо додано один або кілька компонентів Collider, то при колізіях (зіткненнях) об'єкт пересуватиметься.

2. Colliders (колайдери). Компоненти колайдера визначають форму об’єкта для фізичних зіткнень. Колайдер, який є невидимим, не обов’язково має бути точно такої ж форми, як сітка об’єкта, і насправді приблизне наближення часто є більш ефективним у ігровому процесі. Найпростіші колайдери – це так звані примітивні типи колайдерів. У 3D – це Box Collider, Sphere Collider і Capsule Collider. У 2D ви можете використовувати Box Collider 2D і Circle Collider 2D. Будь-яку їх кількість можна додати до одного об’єкта для створення складних колайдерів. Завдяки ретельному позиціонуванню та розміру компаунд-колайдери часто можуть досить добре наблизити форму об’єкта, зберігаючи при цьому низькі навантаження на процесор. Подальша гнучкість може бути отримана завдяки наявності додаткових колайдерів на дочірніх об’єктах. Однак є випадки, де навіть складні колайдери недостатньо точні. У 3D можна використовувати Mesh Colliders, щоб створити колайдер, ідентичний формою сітки об'єкта. У 2D Polygon Collider 2D згенерує форму, що приблизно збігається з графікою спрайту. Не ідеально, але ви можете змінити форму до будь-якого рівня деталізації. Ці колайдери навантажують процесор сильніше, ніж примітивні колайдери.

3. З'єднання (Joints). Існує можливість прикріпити один об’єкт твердого тіла до іншого або до фіксованої точки в просторі за допомогою компонента Joint. Unity надає різні компоненти Joint, наприклад шарнірне з’єднання чи пружинне з’єднання.

3.1.2 Симуляція руху автомобіля в Unity (Unity Wheel Collider)

Wheel Collider – це спеціальний вид колайдера для наземного транспорту. В нього вбудовані: визначення зіткнень, фізика коліс і модель тертя на основі ковзання шин. Його можна використовувати і не тільки для коліс, але він був спеціально створений для транспорту з колесами.

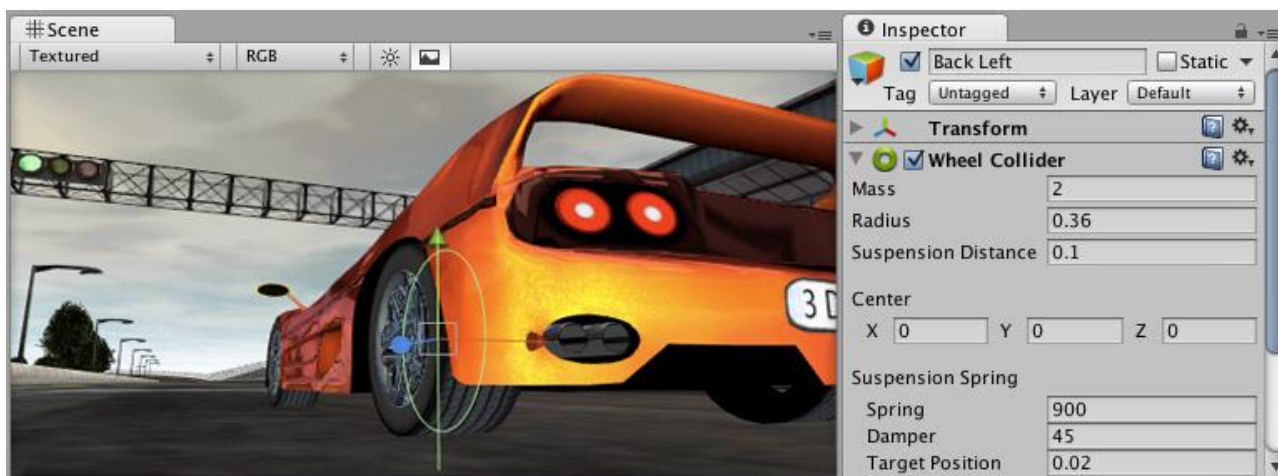

Рисунок 3.2 – Unity Wheel Collider

Основні властивості компонента:

– маса колеса;

– радіус колеса;

– значення амортизації колеса;

– suspension distance (максимальна висота підвіски колеса за віссю Y);

– центр колеса в локальному просторі об’єкта;

– suspension spring (підвіска намагається досягти значення Target Position за допомогою пружини та сили протидії):

– spring (сила пружини, яка намагається досягти Target Position, чим більше значення, тим швидше підвіска буде прагнути Target Position);

– damper (додає опір у підвіску, чим більше значення, тим повільніше рухатиметься Suspension Spring (пружина підвіски));

– target position (відстань спокою підвіски вздовж відстані підвіски. 1 відповідає повністю розширеній підвісці, а 0 відповідає повністю стиснутій підвісці. Значення за замовчуванням становить 0,5, що відповідає поведінці підвіски звичайного автомобіля);

– forward/sideways friction (властивості тертя шини, коли колесо котиться вперед та у бік).

Транспорт контролюється з коду за допомогою різних властивостей: motorTorque, brakeTorque і steerAngle.

Так як машини можуть розвивати велику швидкість, дуже важливим є правильний підхід до геометрії гоночного треку, що використовується для визначення зіткнень. Наприклад, меш-колайдер не повинен мати опуклостей або вм'ятин, які служать для прикраси видимої моделі (наприклад, стовпи огорожі). Зазвичай меш-колайдер для гоночного треку створюють окремо від видимого меша, роблячи його плавнішим.

Тертя шини у Wheel Collider можна описати кривою тертя колес, наведеною на рис. 3.3. Існують окремі криві для напряму руху колеса вперед (кочення) і напрямку вбік. В обох напрямках спочатку визначається, наскільки сильно шина ковзає (на основі різниці швидкості між гумою шини та дорогою). Потім це значення ковзання використовується для визначення сили шини, яка діє на точку контакту.

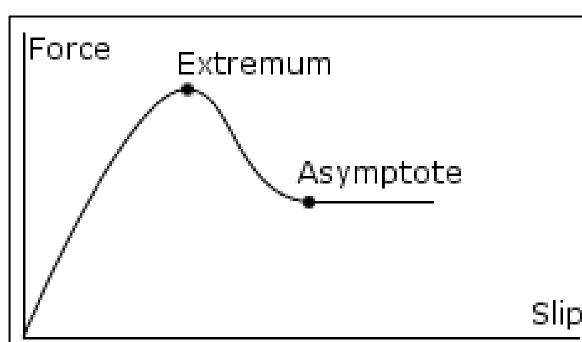

Рисунок 3.3 – Крива тертя коліс

Крива приймає міру ковзання шин як вхідну інформацію та дає силу як вихідну. Крива апроксимується двокомпонентним сплайном. Перша ділянка йде від (0, 0) до (ExtremumSlip, ExtremumValue), у якій точці тангенс кривої дорівнює нулю. Друга ділянка йде від (ExtremumSlip, ExtremumValue) до (AsymptoteSlip, AsymptoteValue), де тангенс кривої знову дорівнює нулю.

Властивість справжніх шин полягає в тому, що при низькому ковзанні вони можуть докладати високі зусилля, оскільки гума компенсує ковзання шляхом розтягування. Пізніше, коли ковзання стає дуже високим, зусилля зменшуються, оскільки шина починає ковзати або обертатися. Таким чином, криві тертя шини мають форму, як на рис. 3.3.

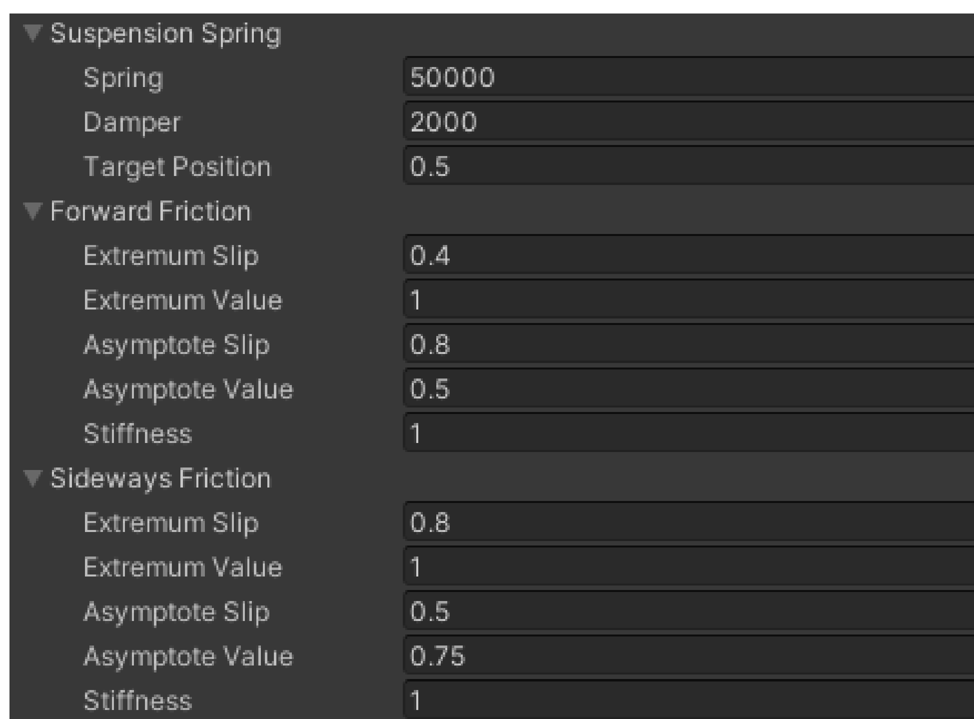

Рисунок 3.4 налаштування кривої тертя колес

Криву тертя коліс звичайно у ігровому рушії Unity можна змінити параметрами, які були наведені раніше. Параметри можна змінити у вікні налаштування (див. рис. 3.4) або ж у розробленому скрипті, змінивши відкриті для редагування параметри, що дозволяє змінювати криву динамічно.

3.1.3 Unity Cinemachine

Cinemachine, набір модулів для роботи з камерою у Unity, вирішує складні завдання з математикою та логікою відстеження об'єктів, компонування, змішування та нарізки кадрів. Його створено з метою значного спрощення процесу розробки шляхом зменшення ручних маніпуляцій і перегляду сценаріїв.

Ці модулі Cinemachine мають процедурний характер, що робить їх стійкими до помилок. Коли вносяться зміни, такі як анімація, швидкість транспортного засобу, рельєф або інші ігрові об'єкти у сцені, Cinemachine автоматично адаптується, щоб забезпечити оптимальний кадр. Немає потреби переписувати скрипти камери лише через те, що персонаж повертається у протилежному напрямку.

Cinemachine працює в реальному часі у всіх жанрах, включаючи FPS, від третьої особи, 2D, бічну прокрутку, зверху вниз і RTS. Він підтримує будь-яку кількість кадрів у вашій сцені. Його модульна система дозволяє створювати складні моделі поведінки.

Cinemachine ідеально поєднується з іншими інструментами Unity, що дозволяє використовувати його як потужний інструмент для анімації та постобробки. Також є можливість створювати власні розширення або інтегрувати їх у створені скрипти камери.

3.2 NEAT-Python

Python – це високорівнева, інтерпретована, гнучка та динамічна мова програмування. Python відомий своєю простотою та зрозумілістю синтаксису, що робить його ідеальним як для початківців так і для досвідчених програмістів. Завдяки своїй простоті та великій стандартній бібліотеці, Python дозволяє розробникам швидко створювати функціональний код і швидко розробляти програми. Python має велику кількість сторонніх бібліотек та фреймворків, які спрощують роботу з різними завданнями, що робить його ідеальним для

розв'язання різних задач. Однією із зручних для використання бібліотек являється NEAT-Python.

Бібліотека neat-python надає інструменти для створення, навчання та тестування нейромереж з використанням алгоритму NEAT. Вона дозволяє розробникам легко реалізовувати та експериментувати з нейроеволюційними алгоритмами у їх програмах на Python.

3.2.1 Файл конфігурації бібліотеки

Файл конфігурації має формат, описаний у документації модулю Python – configparser. Цей модуль надає клас ConfigParser, який реалізує базову мову конфігурації, яка забезпечує структуру, подібну до тієї, що міститься у файлах Microsoft Windows INI.

Більшість налаштувань має бути явно перераховано у файлі конфігурації. Це зменшує ймовірність того, що зміни коду бібліотеки призведуть до того, що проект мовчки використовує інші параметри NEAT. Для використання коніфгурації, бібліотека має конструктор Config, який також вимагає явного указання типів, які використовуватимуться для симуляції NEAT.

Файл конфігурації складається з кількох розділів. Немає вимог щодо впорядкування в цих розділах чи впорядкування самих розділів. Основні розділи файлу конфігурації та головні їх параметри наведено у наступних підпунктах.

3.2.1.1 NEAT

Розділ NEAT визначає параметри, що стосуються загального алгоритму NEAT або самого експерименту. Цей розділ завжди є обов’язковим і обробляється самим класом Config. Параметри розділу:

– Fitness_criterion – функція використовується для обчислення критерію завершення з набору придатності геному, допустимі значення: min, max і mean;

– Fitness_threshold – поріг, якого досягає або перевищує придатність, обчислена за допомогою Fitness_criterion, викликає припинення еволюції;

– Pop_size – кількість індивідів у кожному поколінні.

3.2.1.2 Default Stagnation

Розділ Default Stagnation визначає параметри для вбудованого класу DefaultStagnation. Цей клас відстежує, чи прогресують види, і допомагає видаляти ті, які ні. Має такі параметри:

– Species_fitness_func – функція, яка використовується для обчислення придатності виду, за замовчуванням це mean, дозволені значення: min, max та median;

– Max_stagnation – параметр максимальної кількості поколінь без видалення для індивідів, які не показали покращень (за замовчуванням це 15);

– Species_elitism – кількість видів, які будуть захищені від застою (stagnation), в основному призначений для запобігання повного вимирання (наприклад, налаштування Species_elitism, що дорівнює 3, запобіжить вилученню 3 видів із найвищою придатністю до стагнації незалежно від часу, протягом якого вони не демонстрували покращення), значення якого за замовчуванням 0.

3.2.1.3 Default Reproduction

Розділ Default Reproduction визначає параметри для вбудованого класу DefaultReproduction, який керується створенням геномів або з нуля або шляхом кросинговеру батьків. Має такі параметри для конфігурації:

– Elitism – кількість найбільш придатних особин кожного виду, які збережуться такими, як є, від одного покоління до наступного, яка за замовчуванням має значення 0;

– Survival_threshold – частка для кожного виду, яка дозволяє відтворити кожне покоління, за замовчуванням має значення 0,2;

– min_species_size – мінімальна кількість геномів на вид після розмноження, звичайно за замовчуванням встановлено значення 2.

3.2.1.4 Default Genome

Розділ Default Genome визначає параметри для вбудованого класу DefaultGenome, який має такі головні параметри:

– Activation_options – відокремлений пробілами список функцій активації (вбудовані функції активації: abs, clamped, cube, exp, gauss, hat, identity, inv, log, relu, elu, lelu, selu, sigmoid, sin, softplus, square, tanh), які можуть використовувати вузли, за замовчуванням це sigmoid (сигмоїдна);

– Activation_default – атрибут функції активації за замовчуванням, призначений новим вузлам, якщо жодного не вказано або вказано «random», один із activation_options буде вибрано випадковим чином;

– Conn_add_prob – імовірність того, що мутація зможе додати зв’язок між існуючими вузлами (дійсні значення в інтервалі [0.0, 1.0]);

– Conn_delete_prob – ймовірність того, що мутація видалить існуюче з’єднання (дійсні значення в інтервалі [0.0, 1.0]);

– Enabled_default – стандартний атрибут активності новостворених з’єднань вузлів (дійсні значення: True і False);

– Enabled_mutate_rate – ймовірність того, що мутація змінить значення атрибуту активності існуючого з’єднання (доступні значення в інтервалі [0.0, 1.0]);

– Feed_forward – параметр, який приймає True або False, що може заборонити створеним мережам мати повторювані з’єднання (вони матимуть прямий зв’язок), інакше вони можуть мати рекурентні зв’язки;

– Initial_connection – визначає зв’язок новостворених мереж геномів, за замовчуванням unconnected, може приймати значення:

– Unconnected – спочатку немає з'єднань;

– Fs_neat_nohidden – один випадково вибраний вхідний вузол має одне підключення до кожного вихідного вузла (це одна версія схеми FS-NEAT);

– Fs_neat_hidden – один довільно вибраний вхідний вузол має одне з’єднання з кожним прихованим і вихідним вузлом (це інша версія схеми FS-NEAT, якщо немає прихованих вузлів, це те саме, що fs_neat_nohidden);

– Fs_neat_hidden – один довільно вибраний вхідний вузол має одне з’єднання з кожним прихованим і вихідним вузлом (це інша версія схеми FS-NEAT, якщо немає прихованих вузлів, це те саме, що fs_neat_nohidden);

– Full_nodirect - кожен вхідний вузол підключається до всіх прихованих вузлів, якщо такі є, і кожен прихований вузол підключається до всіх вихідних вузлів, інакше кожен вхідний вузол з’єднується з усіма вихідними вузлами (геноми з feed_forward, встановленим на False, також матимуть повторювані з’єднання від кожного прихованого або вихідного вузла до самого себе;

– Full_direct – кожен вхідний вузол підключений до всіх прихованих і вихідних вузлів, а кожен прихований вузол підключений до всіх вихідних вузлів (геноми з feed_forward, встановленим на False, також матимуть повторювані з’єднання від кожного прихованого або вихідного вузла до самого себе);

– Partial_nodirect – як для full_nodirect, але кожне з’єднання має ймовірність існування, визначену числом (дійсні значення в діапазоні [0.0, 1.0]);

– Partial_direct – як для full_direct, але кожне з'єднання має ймовірність існування, визначену числом (дійсні значення в діапазоні [0.0, 1.0]);

– Node_add_prob – ймовірність того, що мутація зможе додати новий вузол (по суті, замінить існуюче з’єднання, активний статус якого буде встановлено на False), дійсні значення в діапазоні [0.0, 1.0];

– Node_delete_prob – ймовірність того, що мутація видалить існуючий вузол (і всі підключення до нього), дійсні значення в діапазоні [0.0, 1.0];

– Num_hidden – кількість прихованих вузлів, які потрібно додати до кожного геному в початковій популяції;

– Num_inputs – кількість вхідних вузлів, через які мережа отримує вхідні дані;

– Num_outputs – кількість вихідних вузлів нейронної мережі;

– Weight_init_mean – середнє значення нормального/гауссового розподілу, що використовується для вибору значень вагових атрибутів для нових з’єднань;

– Weight_init_stdev – стандартне відхилення нормального/гауссового розподілу, що використовується для вибору вагових значень для нових з’єднань;

– Weight_max_value – максимально допустиме значення ваги, вага, що перевищує це значення, буде обрізана до цього значення;

– Weight_min_value – мінімально допустиме значення ваги, вага, що нижча за це значення, буде обрізана до цього значення;

– Weight_mutate_power – стандартне відхилення нормального/гауссового розподілу з нульовим центром, з якого береться мутація значення ваги;

– Weight_mutate_rate – ймовірність того, що мутація змінить вагу з’єднання шляхом додавання випадкового значення;

– Weight_replace_rate – ймовірність того, що мутація замінить вагу зв’язку новообраним випадковим значенням.

3.2.2 Основні класи бібліотеки

3.2.2.1 Population

Клас Population бібліотеки NEAT-Python реалізує основний алгоритм еволюції (оцінка придатності всіх геномів; перевірка, чи задовольняється критерій завершення – вихід, якщо це так; створення наступного покоління з поточної популяції; розділення нового покоління на види за генетичною подібністю).

При створенні об'єкту класу, основний конструктор приймає налаштування, тобто файл конфігурації та розділи, які в ньому описані.

Клас має головну вбудовану функцію run(), яка запускає генетичний алгоритм NEAT щонайбільше для n (один із вхідних параметрів функції) поколінь. Якщо n дорівнює None, алгоритм працює, доки не буде знайдено рішення або не відбудеться повне зникнення видів. Другим із параметрів функція приймає задану

користувачем фітнес-функцію (fitness_function), яка повинна приймати тільки два параметри:

– популяцію, як список (ідентифікатор генома, геном) кортежів Python;

– поточний об'єкт конфігурації.

Задана фітнес-функція не повинна мати повертаємого значення, але повинна призначувати дійсне значення пристосованості кожному геному.

Передбачається, що фітнес-функція не змінює список геномів, самих геномів, об’єкт конфігурації, а тільки значення fitness самого генома.

Функція run() повертає об’єкт найбільш пристосованого індивіда.

3.3 Модуль Pickle

Модуль pickle в Python є вбудованим інструментом, який використовується для серіалізації та десеріалізації об'єктів Python. Серіалізація – це процес перетворення об'єкту Python у потік байтів, що може бути збережений у файлі або переданий через мережу. Десеріалізація – це процес зворотнього перетворення потоку байтів у об'єкт Python.

Основні функції модуля pickle включають:

– pickle.dump(obj, file) – ця функція серіалізує об'єкт obj та записує його у вказаний файл file;

– pickle.load(file) – ця функція зчитує серіалізований об'єкт з файлу file та десеріалізує його, повертаючи відповідний об'єкт Python.

Модуль pickle може бути корисним у випадках, коли існує необхідність зберегти стан об'єктів Python між запусками програми, або потрібно передати об'єкти через мережу.

Проте важливо зауважити, що об'єкти, серіалізовані за допомогою pickle, можуть бути вразливими до атак безпеки, оскільки pickle може виконувати деякі типи коду Python, які знаходяться у серіалізованих об'єктах.

3.4 Мережеві бібліотеки

Мережеві бібліотеки надають засоби для створення, відправлення та отримання мережевих пакетів через різні протоколи (наприклад, TCP/IP, UDP). Це дозволяє програмам взаємодіяти з іншими комп'ютерами, серверами, веб-сайтами тощо через мережу. Вони дозволяють розробникам створювати різноманітні мережеві додатки, такі як віддалений доступ до ресурсів, мережеві ігри, додатки для обміну повідомленнями чи обмін інформацією між програмами.

3.4.1 Python socket

Socket в Python – це інтерфейс програмування додатків (API), який надає можливість створення мережевих з'єднань між різними комп'ютерами. Socket дозволяє програмістам створювати, відправляти та отримувати дані через мережу з використанням різних протоколів, таких як TCP або UDP.

Основні характеристики сокетів в Python:

– створення з'єднання: сокет дозволяє створювати мережеве з'єднання між двома або більше пристроями через мережу;

– відправка та отримання даних: після створення з'єднання сокет дозволяє програмі відправляти дані на інший комп'ютер та отримувати дані від нього;

– підтримка різних протоколів: сокети в Python підтримують різні мережеві протоколи, такі як TCP (Transmission Control Protocol), UDP (User Datagram Protocol), а також протоколи для роботи з мережами IPv4 та IPv6;

– асинхронний та синхронний ввід/вивід: Python також підтримує асинхронний ввід/вивід (asyncio) з використанням сокетів, що дозволяє ефективно робити багатозадачні програми, які використовують мережеві з'єднання;

– можливості налаштування з'єднань: сокети дають можливість налаштовувати параметри з'єднань, такі як таймаути, розмір буферу тощо.

Основні функції та конструктори модуля socket в Python включають наступне:

– socket(): ця функція створює новий сокет (вона приймає два параметри: перший - це версія протоколу (наприклад, socket.AF_INET для IPv4 або socket.AF_INET6 для IPv6), а другий - це тип сокету (наприклад, socket.SOCK_STREAM для TCP або socket.SOCK_DGRAM для UDP);

– bind(): цей метод пов'язує сокет з конкретною адресою, він приймає кортеж, що містить адресу та порт;

– listen(): цей метод робить сокет приймальним, він приймає один аргумент, який вказує на максимальну кількість очікуваних з'єднань у черзі;

– accept(): цей метод очікує на з'єднання від клієнтів, він блокує виконання програми, поки не буде отримано нове з'єднання, і повертає кортеж, що містить новий сокет та адресу клієнта;

– connect(): цей метод встановлює з'єднання з іншим сокетом. Він приймає кортеж, що містить адресу та порт сервера, з яким потрібно встановити з'єднання.

Це лише кілька основних функцій та конструкторів модуля socket в Python, які найчастіше використовуються для створення мережевих додатків.

3.4.2 Класи для мережі у C#

Для простої реалізації мережевого з'єднання у мові програмування C# використовуються класи IPAddress, IPEndPoint та Socket.

IPAdress – клас, який представляє IP-адресу (IPv4 або IPv6) в мережевих додатках. Він дозволяє створювати об'єкти IP-адрес, порівнювати їх, а також перетворювати текстові рядки в об'єкти IPAddress та навпаки. IPAddress можна використовувати для встановлення з'єднання з іншими вузлами в мережі.

IPEndPoint – клас, який представляє точку з'єднання в мережевому додатку, що складається з IP-адреси та порту. Він дозволяє створювати об'єкти IPEndPoint для використання в різних мережевих операціях, таких як прослуховування з'єднань на сервері або встановлення з'єднання з клієнтом.

Socket – клас, який представляє механізм для відправлення та отримання даних через мережу. Він надає різні методи для створення, з'єднання, приймання та відправлення даних через сокет. Socket може бути використаний як на клієнтській, так і на серверній стороні мережевого додатка. Основні функції цього класу:

– Bind(EndPoint endPoint): ця функція прив'язує сокет до вказаної локальної кінцевої точки (EndPoint), такої як IP-адреса та порт;

– Listen(int backlog): ця функція робить сокет приймальним і вказує йому почекати на вхідні з'єднання в черзі, де параметр backlog вказує максимальну кількість очікуючих з'єднань;

– Accept(): ця функція блокує виконання програми, доки не буде отримано нове з'єднання від клієнта, після цього вона створює новий сокет для цього з'єднання та повертає його;

– Connect(EndPoint remoteEndPoint): ця функція встановлює з'єднання з вказаною віддаленою кінцевою точкою (EndPoint), наприклад, з сервером;

– Send(byte[] buffer, int offset, int size, SocketFlags socketFlags): ця функція відправляє дані через сокет (buffer – масив байтів, що містить дані; offset – зсув у буфері, з якого починаються дані; size – розмір даних для відправлення; socketFlags – додаткові флаги, що вказують параметри відправлення);

– Receive(byte[] buffer, int offset, int size, SocketFlags socketFlags): ця функція отримує дані через сокет (buffer – масив байтів, в який будуть зчитані дані; offset – зсув у буфері, з якого починається зчитування; size – максимальна кількість байтів, яку можна зчитати; socketFlags – додаткові флаги, що вказують параметри зчитування).

3.5 Моделювання системи

Для створення середовища моделювання руху автомобіля в заносі використовується середовище розробки Unity 3D з усіма її фізичними властивостями, створенням відповідної локації та реалізацією компонента Wheel Collider. Так, як для реалізації алгоритму NEAT було обрано гнучку за зручну у

застосуванні бібліотеку NEAT-python, постає питання взаємодії розробленого середовища та алгоритму. Для взаємодії двох окремих процесів, тобто розроблене середовище (гра) та програма, написана на Python є можливість запуску одного з процесів у вигляді підпроцесу іншого у середовищі Windows. Або ж можна використати взаємодію програм через мережеве з'єднання. Для реалізації взаємодії програм було прийнято рішення розробити мережеві інтерфейси для передачі даних між програмами, так як програми запускаються на одному пристрої, в результаті затримка буде мінімальною. Також, реалізація таким способом надає можливість у подальшому реалізувати кластерізацію для запуску середовища на окремих пристроях для паралельної симуляції дії кожного індивіда у популяції.

3.5.1 Python програма

Задача розробленої Python програми полягає у запуску алгоритму нейроеволюції, з усіма її параметрами конфігурації. Обчислення фітнес-функції полягає у створенні агента у середовищі гри та нарахуванням нагороди на кожній ітерації його дій за принципом навчання з підкріпленням. Порядок дій у фітнес-функції:

1) створення агента у середовищі та отримання спостережень, нагороди та та біту завершення симуляції;

2) перевірка: якщо біт завершення симуляції True - то перехід до пункту 5, інакше спостереження використовуються для вхідних значень НМ та фітнес-значення геному збільшується на значення нагороди;

3) вихідні значення НМ відправляються середовищу як дія;

4) отримання спостережень, нагороди та біту завершення симуляції від середовища та перехід до пункту 2;

5) завершення симуляції для поточного агента та перехід до пункту 1 для наступного індивіда.

На рис. 3.5 наведено більш детальну блок-схему раніше наведеного порядку дій обчислення фітнес-функції для окремого індивіда. UnityEnv являється

розробленим модулем в Python на основі бібліотеки для роботи з мережею, який має дві основні функції:

– reset() – надіслати середовищу сигнал про створення нового агента та отримати стан середовища;

– step(action) – надіслати середовищу параметри для дії агента та також отримати стан середовища.

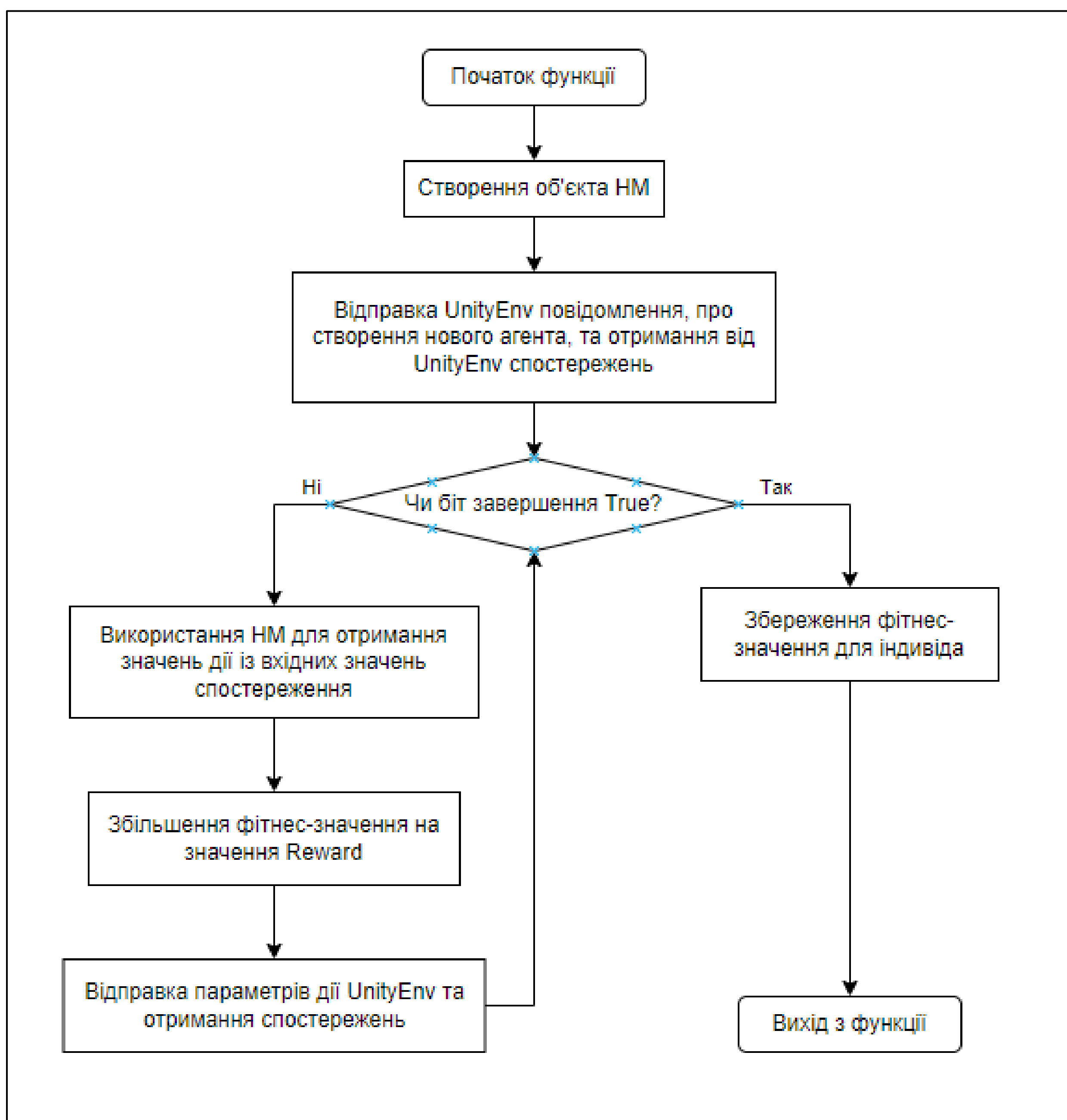

Рисунок 3.5 – блок-схема обчислення фітнес-значення для індивіда

Далі на рис. 3.6 зображено спрощену схему роботи Python програми, у якій блок “Обчислення фітнес-функції для окремого індивіда” являється алгоритмом зображеним на рис. 3.5.

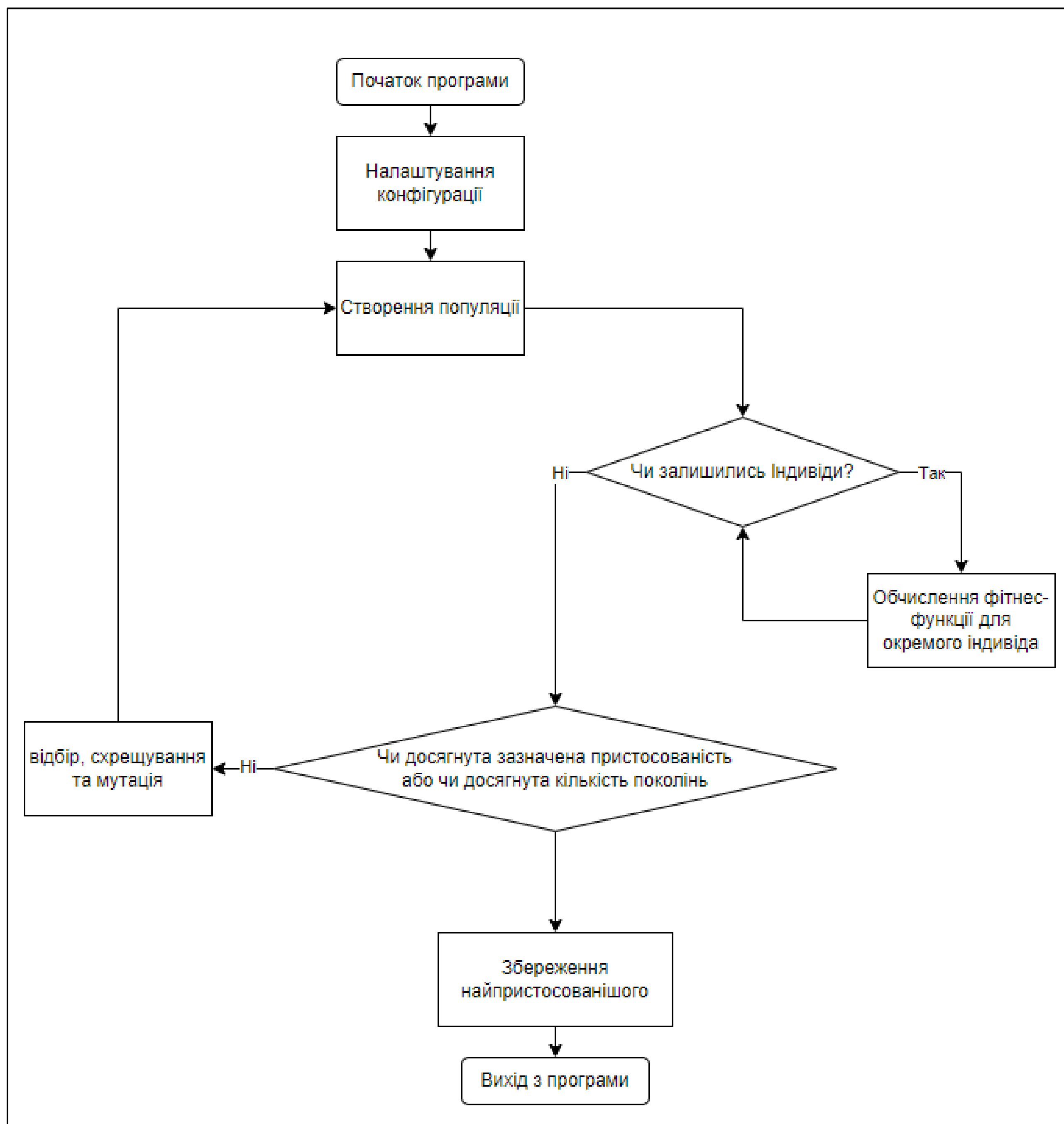

Рисунок 3.6 – блок-схема Python програми

3.5.2 Середовище дій агента у Unity

Середовище для симуляції дій агента (автомобіля, що проходить поворот) являє собою розроблену гру на Unity 3D. Основний порядок роботи середовища описаний у блок-схемі, наведеній на рис. 3.7.

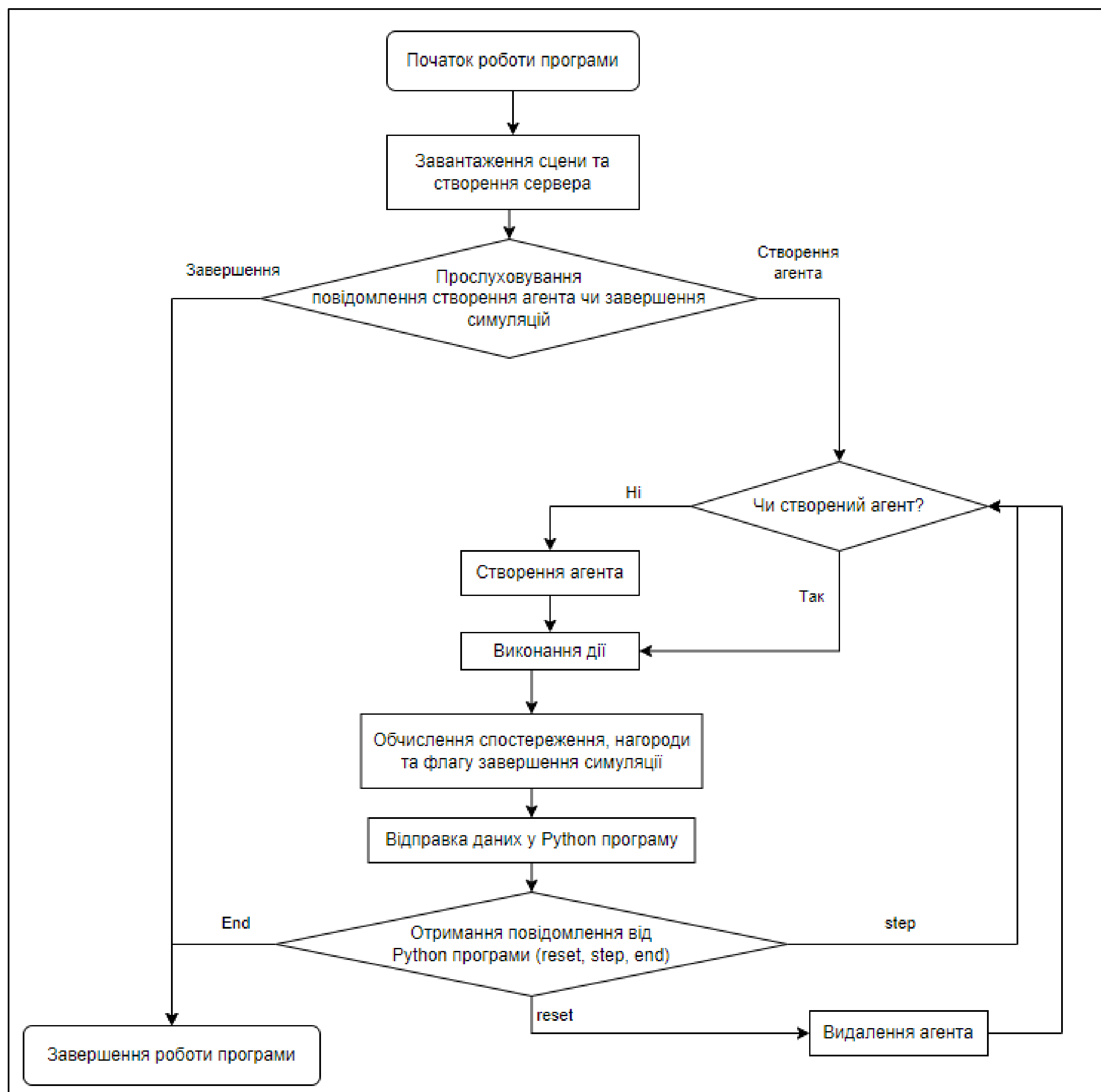

Рисунок 3.7 – блок-схема програми середовища

Розроблена програма складається з таких компонентів, як:

– розроблена сцена для взаємодії з моделлю автомобіля;

– модель автомобіля, що складається з візуальної частини та скрипта для моделювання його руху;

– скрипта для впливу заданих дій на його рух, обчислення нагороди за його кожну дію, обчислення параметрів спостереження та обчислення флагу завершення його симуляції;

– головного скрипта, що створює сервер для обробки повідомлень та керує створенням чи видаленням агентів.

Висновки до розділу 3

Було проаналізовано інструменти для розробки середовища дій агента. Рушій Unity має широкий набір інструментів для вирішення задач, таких як інструменти розробки ігрової сцени з усіма фізичними властивостями середовища для симуляції руху автомобіля, компоненти для розробки моделі самого автомобіля та інструмент для розробки кат-сцен.

Для реалізації NEAT існує популярна бібліотека neat-python, яка має зрозумілий та легкий для використання API, що дозволяє швидко реалізувати алгоритм у проєкті.

Було змодельовано систему, розроблено блок-схеми алгоритмів дій програм: як програми для симуляції середовища так і програми самого алгоритму.

4 РОЗРОБКА СИСТЕМИ ТА ТЕСТУВАННЯ

4.1 Створення сцени у Unity

Створення сцени у Unity – це процес створення візуального середовища, де будуть розміщені об'єкти, ресурси та елементи гри.

Для сцени гри було використано вже розроблений проєкт “Cartoon Race Track – Oval” з відкритим доступом для використання з Unity Asset Store.

Asset Store – це онлайн-магазин, що інтегрований в Unity, де розробники можуть придбати, продавати та обмінюватися різними ресурсами для розробки ігор та додатків. Asset Store містить широкий спектр ресурсів для використання в розробці ігор та додатків, таких як моделі 3D, текстури, анімації, звуки, плагіни, інструменти розробки, ефекти та багато іншого. Asset Store повністю інтегрований у редактор Unity, що дозволяє легко переглядати та імпортувати ресурси безпосередньо у проєкт Unity.

Cartoon Race Track – це гоночний трек у мультяшному стилі, на якому проводяться перегони або просто заїзди. Є можливість зібрати повну гоночну трасу за кілька хвилин, перетягнувши набір збірних елементів у сцену. Пакет включає такі компоненти:

– terrain object prefab (road, grass, gravel, mountains);

– tree prefab;

– garages;

– walls;

– barriers;

– spectator seats;

– banners.

Також пакет включає готову демо-сцену, яка демонструє усі додані у нього компоненти.

Початкова сцена у програмі має таку структуру (див. рис. 4.1).

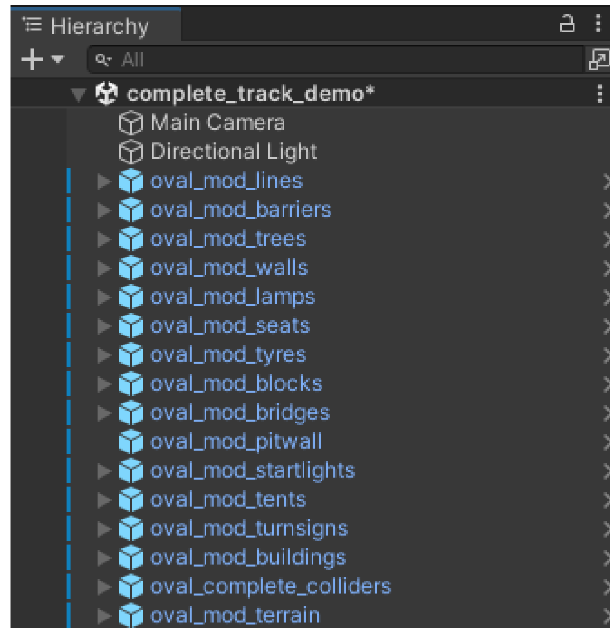

Рисунок 4.1 – структура сцени

На сцені додані усі об’єкти її стилю, головна форма ландшафту з нанесеними на неї об’єктами дороги, бар’єрами та освітленням.

На рис. 4.2 зображено вікно редактора сцени з частиною її місцевості, тобто поворотом. На цьому повороті буде виконано подальша симуляція дій агента.

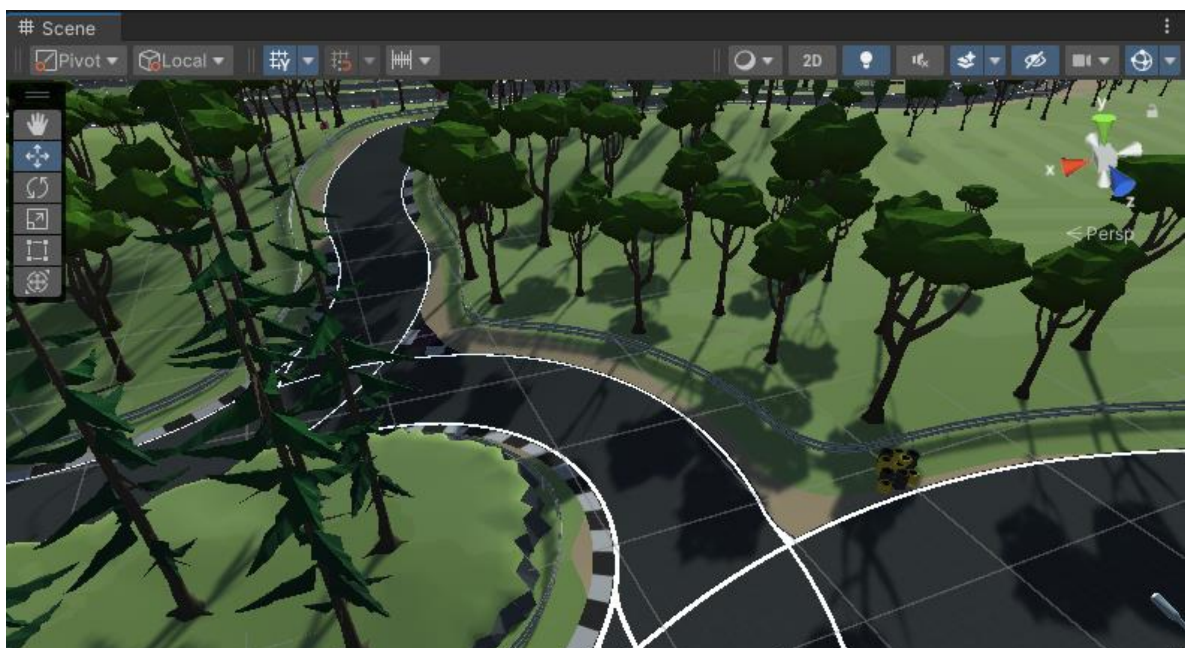

Рисунок 4.2 – вікно редагування сцени

На поточній сцені було відредаговано існуючі колайдери (див. рис. 4.3), які будуть використані для відслідковування контакту агента з бар’єрами, тобто для завершення симуляції його руху у середовищі. Також можна побачити, що на розділеннях дороги додано колайдери для обмеження обраного повороту, так як дії агента будуть обмежені тільки цим поворотом.

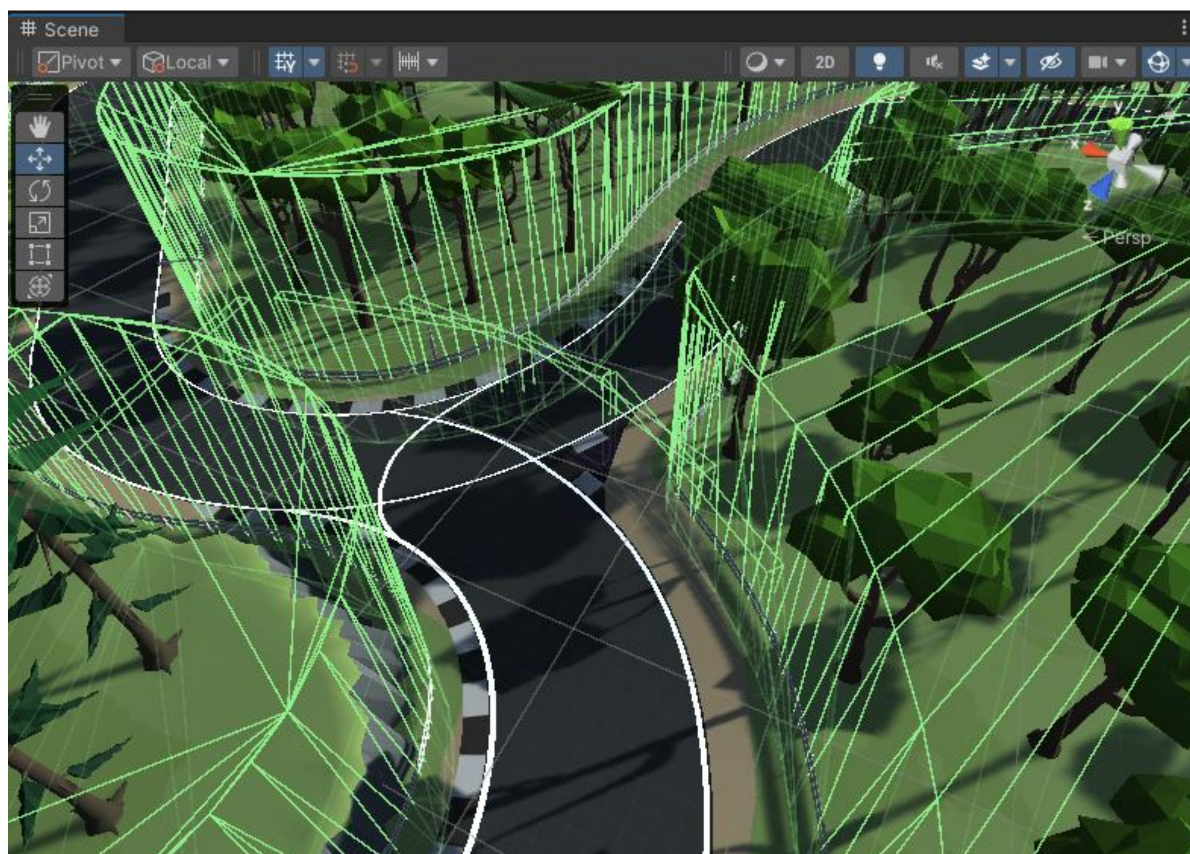

Рисунок 4.3 – колайдери сцени

4.2 Налаштування автомобіля

У якості автомобіля (агента) в середовищі буде використовуватись відповідний префаб Unity, для можливості його легкого створення та видалення на сцені, який складається з моделі автомобіля (кузову, колес) та звичайно колайдера для відслідковування зіткнень.

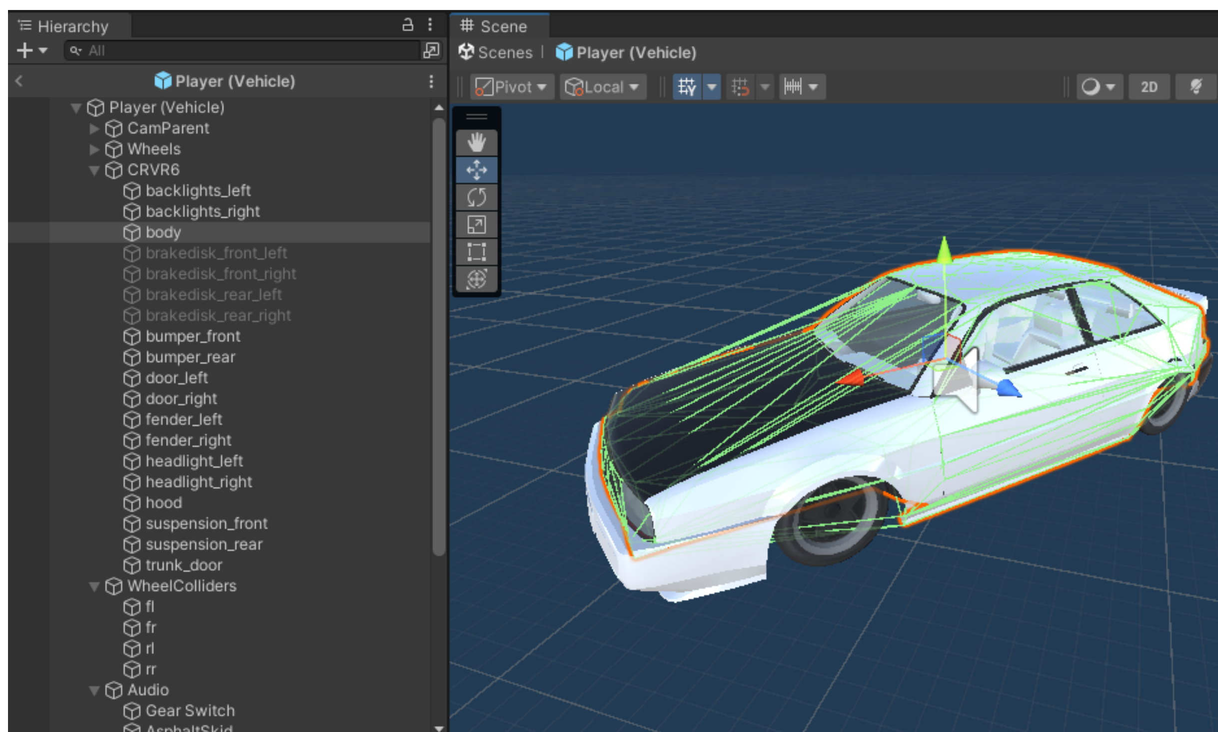

Рисунок 4.4 – префаб автомобіля

Також префаб на місцях моделей коліс, відповідно кожному має чотири об'єкти типу WheelCollider зі своїми налаштуваннями. Параметри, які були обрані для поведінки коліс та підвіски автомобіля наведені на рис. 4.5.

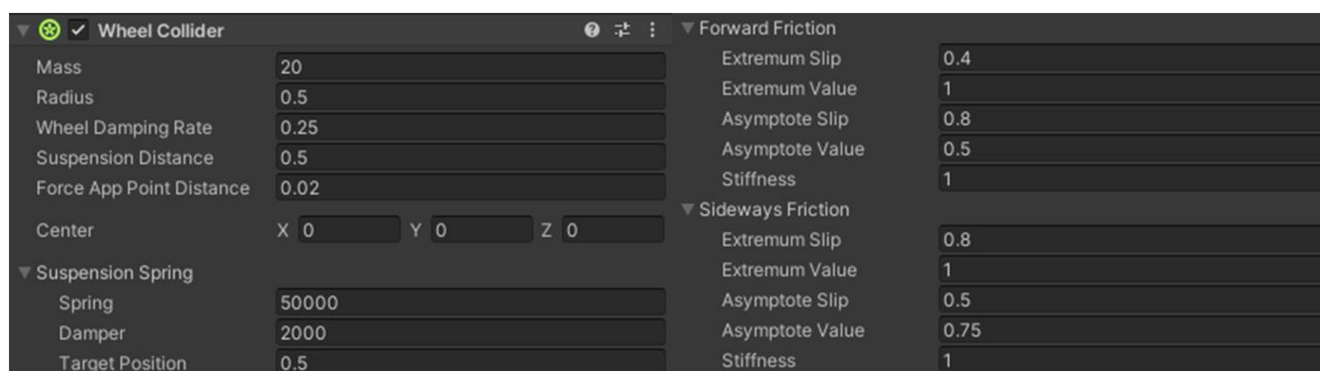

Рисунок 4.5 – параметри налаштування WheelCollider

Кожен об'єкт колеса має прикріплений скрипт для обрахування положення моделі самого колеса відносно до самого WheelCollider об’єкта.

Значення обертання коліс та їх повороту розраховується у скрипті, який використовує вхідні параметри Vertical та Horizontal, які можуть мати дійсне значення від -1 до 1. Це дає змогу як отримати значення з контролера (чи то клавіші “вгору”, ”вниз”, ”вправо” та ”вліво”, чи значення положення осі джойстика геймаду), так і надати параметри керування з виходів НМ.

4.3 Написання логіки середовища

Для створення серверу та керування створенням агентів було створено відповідний скрипт та додано його до створеного об'єкту на сцені, щоб він міг функціонувати у кожному кадрі симуляції. Також цей об'єкт було розташовано як раз на місці спавну агента, щоб його положення вже одразу можна було використати для цього.

```
[SerializeField] GameObject agentPrefab;
[SerializeField] Transform startPosition;

private Servak server;
private bool isCreatedAgent = false;

private GameObject agent;
WheelController component = null;
private bool connectionStopped = false;
```

Рисунок 4.6 – поля скрипта керування симуляцією

Головний скрипт має змінні, що наведені на рис. 4.6, де:

– agentPrefab – це об'єкт, якому буде призначенно значення префабу, що буде використовуватись для створення агента, що було описано раніше;

– startPosition – об’єкт позиції, що приймає положення (x, y та z) у 3D-просторі та параметри його повороту, який буде використано для початкової позиції агента та як раз буде мати значення положення об'єкта, до якого прикріплен цей скрипт;

– server – об'єкт системи обміну повідомленнями між програмами;

– isCreatedAgent – флаг існування агента на сцені;

– component – об’єкт компонента скрипта агента;

– agent – об’єкт самого агента на сцені.

```
if (!isCreatedAgent && !connectionStopped)
{
    agent = Instantiate(agentPrefab, startPosition.position, startPosition.rotation);
    component = agent.GetComponent<WheelController>();
    component.servak = server;

    isCreatedAgent = true;
}
if (agent == null && !connectionStopped) isCreatedAgent = false;
else if (component.connectionStopped)
{
    connectionStopped = true;
    Destroy(agent);
}
```

Рисунок 4.7 – лістинг коду основного скрипта

Основний скрипт (див. рис. 4.7) на кожному кадрі перевіряє, чи створений агент та чи не припинено з’єднання, якщо ні та ні відповідно, то створюється новий агент за вказаним префабом та заданими позиціями та поворотом від об’єкта startPosition.

Далі перевіряє чи не створений агент та не зупинено з’єднання та задає значення флагу isCreatedAgent – false, що дасть змогу створити агента у наступному кадрі. У противному випадку та якщо компонент агента указав, що з’єднання зупинено, тобто алгоритм Python програми надіслав повідомлення завершення агенту, то агент знищується та нового не буде створено.

4.4 Написання скрипта для агента

Для зміни руху автомобіля потрібно впливати на змінні Horizontal та Vertical, які були описані раніше. Було вирішено розробити функцію для зміни цих параметрів, яка на вхід буде приймати два дійсних значення відповідно, та відносно

нуля ці значення будуть впливати на збільшення чи зменшення значення відповідної осі на вказану константу. На рис зображено скрипт зміни цих параметрів, де було указано константою 0,01, так як цього достатньо для і швидкості зміни частоти обертання коліс та повороту передніх коліс, так і для точності. Можна побачити, що значення обрізаються в діапазоні від -1 до 1 та не враховується зміна обертання коліс у зворотню сторону.

```
private void setInput(float horizontal, float vertical)
{
    if (horizontal > 0) Horizontal += 0.01f;
    else Horizontal -= 0.01f;
    if (vertical > 0) Vertical += 0.01f;
    //else verticalInput -= 0.01f;
    if (Horizontal > 1.0f) Horizontal = 1.0f;
    if (Horizontal < -1.0f) Horizontal = -1.0f;

    if (Vertical > 1.0f) Vertical = 1.0f;
    if (Vertical < -1.0f) Vertical = -1.0f;

}
```

Рисунок 4.8 – функція надання дії автомобіля

Структура початкової нейронної мережі у алгоритмі буде мати вигляд повнозв’язної НМ прямого поширення із сьома входами та двома виходами. Виходи, як вже відомо будуть впливати на зміну властивостей автомобіля для зміни його руху, тобто один вихід впливає на збільшення чи зменшення частоти обертання коліс, а другий на зміну повороту коліс чи то вліво чи вправо. Входами НМ будуть параметри спостереження повернені середовищем, тобто усі поточні значення, які можуть вплинути на крок у наступному кадрі гри:

– відносна швидкість руху автомобіля на сцені;

– відстань від задньої правої точки авто до границі повороту;

– відстань від передньої правої точки автомобіля до зустріненого колайдера променем, який випущено з цієї точки під кутом 45 градусів у протилежну сторону поворота від направлення лобової частини автомобіля;

– кут, під яким рухається автомобіль відносно вектора переміщення;

– поточний кут повороту колес;

– значення Horizontal;

– значення Vertical.

Для розвідування одного повороту і у фіксовану сторону, достатньо використати один проектування одного променя вперед та під кутом у протилежну сторону від сторони напрямку повороту, враховуючи, що у заносі авто ще повернено у сторону повороту. Так, як авто має шанс зіткнутись з границею задньою частиною, то за таким же принципом проектується промінь із задньої частини автомобіля для отримання відстані.

```
private void castExplorationRay()
{
    Vector3 rayDirection;
    Ray ray;
    RaycastHit hit;

    rayDirection = Quaternion.AngleAxis(45, transform.up)
                    * -transform.forward;
    ray = new Ray(frontRightPoint.position, rayDirection);
    if (Physics.Raycast(ray, out hit) && hit.distance < 30.0f)
                    frontRightDistance = hit.distance;
    Debug.DrawRay(ray.origin, ray.direction * frontRightDistance, Color.green);
}
```

Рисунок 4.9 – проектування променя

Для проектування променя у Unity використовуються класи Ray та RaycastHit. На рис. 4.9 зображено функцію, яка обраховує довжину променя розвідки. Спочатку створюється вектор напрямку променя відносно положення автомобіля та створюється об’єкт класу Ray з положенням точки, з якої він починає проекцію та вектор його напрямку. Далі, якщо промінь стикається з колайдером, то

повертає у об’єкт класу RaycastHit усі відомі параметри зіткнення, один з яких відстань від точки проектування до точки зіткнення. Також у режимі Debug буде візуалізовано спроектований промінь.

Промінь для обчислення відстані від задньої частини створюється схожою функцією.

Для обчислення кута руху автомобіля потрібно мати вектор переміщення об’єкта та вектор його напрямку. На рис. 4.10 змінною vectorA являється вектор переміщення об’єкта, для його обрахування потрібно мати положення точки у 3D-просторі поточного кадру гри та положення тієї самої точки у минулому кадрі. Для збереження положення точки у минулому кадрі створено змінну previousFrontPoint, якій присвоюється значення положення її у поточному кадрі гри після всіх обчислень функції. Обчислення кута між векторами у Unity виконується завдяки вбудованій функції Vector3.Angle(vector1, vector2). Кадр гри може бути на стільки швидким, що кут після обчислень буде дорівнювати нулю, тому в такій ситуації виконується збереження минулого кута. Також одразу створено умову, при якій змінній кута присвоюється значення нуля, якщо швидкість недостатня, щоб кут не давав результату у нагороді, що не дасть високої пристосованості індивіда.

```
private void calculateDriftAngle()
{
    Vector3 vectorA = frontPointTransform.position - previousFrontPoint;
    Vector3 forwardVector = mainTransform.TransformDirection(Vector3.back);

    float tmpAngle = Vector3.Angle(forwardVector, vectorA);

    if (tmpAngle > 0) driftAngle = Vector3.Angle(forwardVector, vectorA);
    if (velocity < 1.0f) driftAngle = 0.0f;

    previousFrontPoint = frontPointTransform.position;
}
```

Рисунок 4.10 – функція обчислення кута дрифту

Для збереження спостережень була виділена окрема функція (див. рис. 4.11). У наведеній функції зберігається значення швидкості автомобіля у просторі (змінна velocity), викликаються раніше описані функції для обчислення відстаней променів, обчислюється поворот передніх коліс (змінна wheelAngle) у діапазоні від 0.0 до 1.0 відносно максимально можливого їх повороту. Далі присвоюється значення true флагу завершення поточної симуляції (змінна isDone) при виконанні умов:

– автомобіль зіткнувся з колайдером;

– автомобіль не показував високої швидкості деякий час;

– симуляція перевищила часовий поріг.

Для отримання часу з початку симуляції використовується об’єкт типу Stopwatch, та його функція ElapsedMilliseconds, яка повертає значення у мілісекундах від початку використання функції Stopwatch.start().

Значення швидкості, відстані задньої частини до границі повороту та кут дрифта будуть використовуватись у сумі змінної reward, яка буде відправлена для обчислення фітнес-значення. Тому ці значення потрібно нормалізувати відносно їх внеску у досягненні найкращого результату проходження повороту. Експериментальним шляхом заїздів користувача гри було визначено коефіцієнти внеску кожного з цих параметрів при більш-менш візуально позитивному заїзді. Змінні borderDistanceNormalized, velocityNormalized та driftAngleNormalized представляють значення, які вже можна використати у сумі нагороди. Також можна побачити, що відстань до границі було обернено, так як чим менша відстань задньої частини автомобіля до границі, тим більш якісним являється дрифт.

```
private void handleObservation()
{
    velocity = RB.velocity.magnitude;

    calculateDriftAngle();

    wheelAngle = steerableWheels[0].steeringAngle / wheelSteeringAngle;

    if (isCollided) isDone = true;
    if (stopwatch.ElapsedMilliseconds > 2000 && velocity < 5.0f) isDone = true;
    if (stopwatch.ElapsedMilliseconds > 15000) isDone = true;

    castExplorationRay();
    borderDistance = castRayToBorder();

    borderDistanceNormalized = velocity < 1.0f ? 0.0f : (1.0f - (borderDistance /
                                                          MAX_BORDER_DISTANCE));
    velocityNormalized = velocity / 15.0f;
    driftAngleNormalized = driftAngle / 13.0f;

    reward = velocityNormalized + borderDistanceNormalized + driftAngleNormalized;
}
```

Рисунок 4.11 – функція обрахування спостережень

Функція відправки повідомлення (див. рис. 4.12) компонує усі підготовлені дані (спостереження, значення нагороди та флаг завершення симуляції) у рядок C#, розділяючи їх символом та викликає функцію реалізації відправки повідомлення Python програмі об’єкта типу Servak.

```
private void returnMessage()
{
    string message;

    message = velocityNormalized.ToString() + "|"
            + borderDistance.ToString() + "|"
            + frontRightDistance.ToString() + "|"
            + driftAngleNormalized.ToString() + "|"
            + wheelAngle.ToString() + "|"
            + Horizontal.ToString() + "|"
            + Vertical.ToString() + "|"
            + reward.ToString() + "|"
            + (isDone ? 1.ToString() : 0.ToString());

    servak.retMessage(message);
}
```

Рисунок 4.12 – функція відправки повідомлення алгоритму

Для обробки повідомлення, що надходить від Python програми, було розроблено функцію handleMessage(), яку зображено на рис. 4.13. На початку функція отримує рядок C# з виклику функції реалізації отримання мережевого повідомлення класу Servak. Наступним кроком є перевірка умови типу повідомлення, тобто повідомлення на створення нового агента, закінчення симуляцій чи продовження виконання дії поточного агента. Якщо симуляція агента продовжується, то значить було отримано повідомлення з діями для зміни його стану, тобто два дійсних значення НМ, які далі використовуються у раніше згаданій функції setInput().

```
private void handleMessage()
{
    string message = servak.getMessage();

    if (message.Contains("R")) Destroy(gameObject);
    else if (message.Contains("Q"))
    {
        servak.retMessage("END of sim!");
        connectionStopped = true;
    }
    else
    {
        string[] splittedMsg = message.Split('|');
        float[] actions = new float[2];
        actions[0] = float.Parse(splittedMsg[0]);
        actions[1] = float.Parse(splittedMsg[1]);

        setInput(actions[0], actions[1]);
    }
}
```

Рисунок 4.13 – функція handleMessage()

Усі раніше оголошені функції використовуються у визначених функціях циклу Unity для обчислень стану об’єкта (агента) на кожному кадрі гри. В основному при розробці використовується функція Update(), але за документацією Unity відомо, що існує також функція LateUpdate(), яка викликається для кожного об’єкта на сцені вже після обчислень фізичних явищ на сцені рушієм, а Update навпаки – перед усіма обчисленнями. На рис. 4.14 можна побачити, що у Update викликається функція handleMessage(), хоча і не на першому кадрі після створення агента (можна згадати, що отримання повідомлення про створення нового агента було отримано або у початковій ініціалізації середовища або у скрипті минулого агента, тому Python програма вже чекає від середовища відповіді), та виконується обчислення нових параметрів обертання та повороту коліс. Далі, вже як відомо, після обчислення зміни фізичного середовища рушієм можна викликати у LateUpdate функцію для обчислення змінних стану середовища та повернути повідомлення, тобто існує можливість реалізувати принцип навчання з

підкріпленням структуровано лише у одному кадрі гри. Повну реалізацію програми описано у додатку А.

```
void Update ()
{

    if (!firstFrame && !connectionStopped)
    {
        handleMessage();
    }
    else firstFrame = false;
    //setInput();
    wheelControl();
}
private void LateUpdate()
{
    handleObservation();
    returnMessage();
}
```

Рисунок 4.14 – визначення функцій для обробки кадру

4.5 Написання програми NEAT алгоритму

В першу чергу для запуску алгоритму бібліотеки neat-python передбачається створення файлу конфігурації. Цей файл було створено та додано і відкориговано усі параметри для правильної роботи алгоритму.

```
# Load configuration.
config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                     neat.DefaultSpeciesSet, neat.DefaultStagnation,
                     config_file)

p = neat.Population(config)
```

Рисунок 4.15 – завантаження конфігурації

На рис. 4.15 зображено лістинг коду створення об’єкта конфігурації системи з параметрів відповідного файлу, також у параметри додано розділи конфігурації, які описано у файлі.

Як вже відомо, для роботи алгоритму передбачається створення фітнес-функції, яка в параметрах отримує кортеж з геномів поточної популяції, та повинна змінити у результаті фітнес-значення відповідно кожного генома.

```
def eval_genomes(genomes, config):
    for genome_id, genome in genomes:
        genome.fitness = 1.0
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        params = unityEnv.reset()
        reward = params[1]
        isDone = params[2]
        while not isDone:
            action = net.activate(params[0])
            params = unityEnv.step(action)
            reward = params[1]
            isDone = params[2]
            genome.fitness += reward
        print(genome.fitness)
```

Рисунок 4.16 – фітнес-функція алгоритму

Було створено функцію eval_genomes(), яку буде використано для запуску алгоритму (див. рис. 4.16). Функція буде проходити цикл із кількості геномів та у результаті змінювати фітнес-значення кожного. На початку циклу створюється об’єкт НМ на основі генома та файлу конфігурації. Далі викликається функція, яка надсилає повідомлення програмі середовища про створення нового агента та отримує початкові значення стану середовища (params). У циклі кожного генома створено цикл обробки кожної дії агента. У цьому циклі алгоритм отримує значення дії з виходів НМ за заданими параметрами стану середовища, відправляє повідомлення програмі середовища з параметрами для дій агента, збільшує фітнес-

значення поточного генома на значення отриманої нагороди за кадр та перевіряє чи не містить повідомлення від середовища позитивний флаг завершення симуляції (якщо так – перехід до наступного генома, якщо ні, то завершення роботи функції).

Сам запуск алгоритму викликається функцією run() для об'єкта популяції, яка приймає параметрами як раз раніше описану фітнес-функцію та значення максимально допустимої кількості поколінь роботи алгоритму.

```
def saveGenome(genome, file_path):
    with open(file_path, "wb") as f:
        pickle.dump(genome, f)
```

Рисунок 4.17 – функція збереження генома

За допомогою модуля для серіалізації об’єктів у python – pickle було розроблено функцію для збереження об’єкта типу Genome найкращого індивіда для подальшого запуску тестування чи використання його НМ у системі середовища з агентом. Повну реалізацію програми та файл конфігураці описано у додатку Б.

4.6 Тестування системи та аналіз результатів

Для тестування розробленої системи було спочатку запущено програму симуляції середовища (гру) та після – запущено програму роботи алгоритму, яка вже і буде керувати створенням агентів та їх управлінням у середовищі. Систему було протестовано на запуску алгоритму NEAT з розміром популяції у 20 особин та обмежено виконання алгоритму п’ятдесятьма поколіннями. За результатами роботи програми стає відомо, що максимум нагороди, який можна отримати, дорівнює приблизно 6300 (див. рис. 4.18). Також можна побачити, що вже після 24 покоління отримано найкраще значення пристосованості.

Рисунок 4.18 – залежність максимальної пристосованості від покоління

Після зазначеного покоління спостерігається коливання значення пристосованості з ростом кількості поколінь. На такий результат може впливати саме нестабільність середовища гри. Так, як зміна стану агента відбувається кожен кадр гри, а він в свою чергу може генеруватись не за однакову кількість мілісекунд, тому й відгук від середовища на одні й ті самі дії може бути з деякою похибкою. Хоча у розробленому середовищі, коли агент (автомобіль) повинен проходити поворот на мінімальній відстані від бар'єру, така похибка може коштувати колізією з бар'єром та закінченням симуляції.

Зазвичай, після роботи алгоритму найкращим рішенням являється найпристосований індивід останнього покоління, але при таких обставинах, якщо не обмежувати роботу алгоритму цільовим значенням пристосованості, рекомендовано зберегти найкращий геном індивіда кожного покоління та відсортувати за значенням пристосованості, як і було зроблено.

Хоча для створення кат-сцени руху автомобіля поворотом у заносі було б доречно провести візуальний аналіз результатів, але все ж було проведено аналіз за даними, які надає середовище.

Рисунок 4.19 – результат роботи візуалізації середовища

На рис. 4.19 зображено роботу програми середовища, де скомпоновано вид сцени для розробника та камеру, яка слідкує за автомобілем. У вікні сцени можна побачити, що автомобіль рухається під кутом та спроектовано два промені, відстані яких як раз обмежуються колайдерами.

Результат роботи програми було порівняно з експертними результатами управління агентом. Тобто було використано контролер користувачем для подання дій агенту середовища. За результати було використано середні значення показників за всю симуляцію, які були використані у розрахунку нагороди за кадр гри, тобто: швидкість у просторі, нагорода за відстань до бар'єру та кут дрифту.

Середні показники, що впливали на якість результату руху автомобіля, які були отримані у результаті тестування агента на основі генома індивіда, який показав найкращий результат у останньому поколінні: 0,84; 0,78; 0,94. Для порівняння результати експертного керування: 0,7; 0,65; 0,61.

Висновки до розділу 4

Було розроблено програму реалізації середовища для руху автомобіля за поворотом та з можливістю руху у стилі дрифту на основі Unity 3D. Також було розроблено програму реалізації алгоритму NEAT на мові програмування Python з використанням бібліотеки NEAT-python та можливістю спілкуватись із середовищем мережею. Було протестовано систему на певному розмірі популяції, з обмеженням кількості поколінь та без цільового значення нагороди. Після аналізу стало відомо, що при розмірі популяції у 20 особин алгоритму достатньо близько 25 поколінь, щоб досягти бажаний результат. Результат було порівняно як візуально, так і відносно результатів керування користувачем автомобілю. Різниця у позитивну сторону результату роботи алгоритму становить приблизно 20 відсотків на кожному показнику результату.

ВИСНОВКИ

У кваліфікаційній роботі магістра було поставлено завдання розробити систему зі змодельованим рухом автомобіля у грі на Unity 3D для полегшення створення якісних кат-сцен. Для розробки системи було проаналізовано принципи та властивості руху автомобіля у симуляціях. Проаналізувавши рушій Unity, стало відомо, що він має усі можливості для створення симуляції руху авто у заносі.

Було проаналізовано методи та алгоритми для вирішення завдання. Алгоритм NEAT може бути ефективним в розв'язанні складних завдань навчання з підкріпленням, одним із яких являється навчання ігрового агента. Еволюційні алгоритми можуть ефективно пристосовуватися до динамічного середовища та вирішувати задачі, які можуть бути важко формалізовані або мати невідому оптимальну стратегію. Загалом, NEAT може бути потужним інструментом для навчання ігрових агентів у відомому або навіть невідомому середовищі, завдяки своїм перевагам у визначенні архітектури мережі, збереженню різноманітності та підтримці динамічних змін.

Для розробки системи було вирішено використати реалізацію алгоритму у бібліотеці NEAT-python, яка має усі параметри налаштування та функції для повноцінної роботи алгоритму.

Було розроблено систему для виконання поставленого завдання. Середовищем (локацією з поворотом) дій агента (автомобіля) являється гра, розроблена на основі рушія Unity. Систему було протестовано на запуску еволюційного алгоритму з розміром популяції у 20 особин та обмеженням у 50 поколінь. Позитивними результатами виконання алгоритму було вирішено вважати тими, які більші за результат управління автомобіля користувачем, заїзд якого візуально можна вважати якісним. Результати роботи алгоритму достатньо перевищували граничні результати, тому виконання маневру автомобілем у розробленій грі за допомогою отриманої НМ можна використати у створення ігрової кат-сцени.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. King, G., Krzywinska, T. Computer Games/Cinema/Interfaces. Proceedings of the Computer Games and Digital Cultures Conference. June 6-8, 2002, Tampere, Finland. P. 141–153.

2. Abdulrahim M. On the dynamics of automobile drifting. SAE 2006 world congress & exhibition. 400 Commonwealth Drive, Warrendale, PA, United States, 2006.

3. Milliken W. F. Race car vehicle dynamics. Warrendale, PA, U.S.A : SAE International, 1995. 890 p.

4. Tire Model in Driving Simulator: веб-сайт. URL: http://code.eng.buffalo.edu/dat/sites/tire/tire.html (дата звернення: 22.12.2023).

5. Gregory J. Game engine architecture. A K Peters/CRC Press, 2017.

6. Bourg D. M., Seemann G. AI for game developers. O'Reilly Media, Inc., 2004. 390 p.

7. Lee D., Seo H., Jung M. W. Neural basis of reinforcement learning and decision making. Annual review of neuroscience. 2012. Vol. 35, no. 1. P. 287–308. https://doi.org/10.1146/annurev-neuro-062111-150512.

8. Banzhaf, Wolfgang; Nordin, Peter; Keller, Robert; Francone, Frank. Genetic Programming – An Introduction. San Francisco, Morgan Kaufmann, 1997. 496 p.

9. Stanley K. O., Miikkulainen R. Evolving neural networks through augmenting topologies. Evolutionary computation. 2002. Vol. 10, no. 2. P. 99–127. https://doi.org/10.1162/106365602320169811.

10. Dealessandri, Marie (January 16, 2020). What is the best game engine: is Unity right for you. GamesIndustry.biz. URL: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you (дата звернення: 22.12.2023).

11. Holmes, Dylan. A Mind Forever Voyaging: A History of Storytelling in Video Games. CreateSpace Independent Publishing Platform, 2012. 248 p.

12. Lebowitz J. Interactive storytelling for video games: a player-centered approach to creating memorable characters and stories. Burlington, MA : Focal Press, 2011. 319 p.

13. Dixon, Chris. DRIVING; Drifting: The Fast Art of the Controlled Slide. The New York Times, May 7, 2004. URL: https://www.nytimes.com/2004/05/07/travel/driving-drifting-the-fast-art-of-the-controlled-slide.html (дата звернення: 23.12.2023).

14. What is Drifting. DriftWorks.com: веб-сайт. URL: https://www.driftworks.com/blog/drifting/ (дата звернення: 23.12.2023).

15. Pacejka H. B. Tire and vehicle dynamics. 2nd ed. Warrendale, PA : SAE International, 2006. 642 p.

16. E. Velenis. Dynamic tyre friction models for combined longitudinal and lateral vehicle motion. Vehicle system dynamics. 2005. Vol. 43, no. 1. P. 3–29.

17. Millington, Ian. Game Physics Engine Development. CRC Press, January 2, 2010. 552 p.

18. Eberly D. H. Game physics. CRC Press, 2010. 902 p.

19. Common game development terms and definitions, Game design vocabulary. Unity: вебсайт. URL: https://unity.com/how-to/beginner/game-development-terms (дата звернення: 24.12.2023).

20. Russell S. J., Norvig P. Artificial intelligence: a modern approach. Pearson, 2020. 1136 p.

21. Steve R. AI Game Programming Wisdom. Charles River Media, April 3, 2002. 704 p.

22. На ім'я Sophy. Названий найкращий у світі водій у грі Gran Turismo, і це не людина: вебсайт. URL: https://techno.nv.ua/ukr/techno/games/shtuchniy-intelekt-dlya-gri-gran-turismo-stvorili-u-sony-50216724.html (дата звернення: 29.12.2023).

23. Sutton R. S., Barto A. G. Reinforcement learning: an introduction. A Bradford Book, 2018. 552 p.

24. Reinforcement Learning vs Genetic Algorithm – AI for Simulations: веб-сайт. URL: https://medium.com/xrpractices/reinforcement-learning-vs-genetic-algorithm-ai-for-simulations-f1f484969c56 (дата звернення: 03.01.2024).

25. Poli, R., Langdon, W. B., McPhee, N. F. A Field Guide to Genetic Programming. Lulu.com, freely available from the internet, January 2008. 233 p.

26. Eiben A. E., Smith J. E. Introduction to evolutionary computing. Springer Berlin Heidelberg, 2003. https://doi.org/10.1007/978-3-662-05094-1.

27. Goldberg David E. Genetic algorithms in search, optimization, and machine learning. Reading, Mass : Addison-Wesley Pub. Co., 1989. 412 p.

28. Stanley K. O., Miikkulainen R. Efficient Reinforcement Learning through Evolving Neural Network Topologies. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002). San Francisco, Morgan Kaufmann. P. 569–577.

29. Stanley, K. O.; Bryant, B. D.; Miikkulainen, R. Realtime neuroevolution in the nero video game. IEEE Transactions on Evolutionary Computation (Special Issue on Evolutionary Computation and Games), Vol.9, No. 6, December 2005.

30. AlMahamid F., Grolinger K. Reinforcement learning algorithms: an overview and classification. 2021 IEEE canadian conference on electrical and computer engineering (CCECE), ON, Canada, 12–17 September 2021. 2021.

31. Давидюк Ю. И. Методы нейроэволюции сетей прямого распространения. Vestnik of brest state technical university. civil engineering and architecture. 2021. № 2 – 2021. С. 49–53.

32. Jeff W. Murray. C# Game Programming Cookbook for Unity 3D (2nd edition). CRC Press, March 2021. 298 p.

33. Unity user manual. Unity documentation: веб-сайт. URL: https://docs.unity3d.com/Manual/index.html

34. NEAT-Python’s documentation. NEAT-Python: веб-сайт. URL: https://neat-python.readthedocs.io/en/latest/ (дата звернення: 08.01.2024).

ДОДАТОК А

Лістинг коду програми середовища

Файл control.cs:

```
using UnityEngine;
using myspace;

public class control : MonoBehaviour
{
    [SerializeField] GameObject agentPrefab;
    [SerializeField] Transform startPosition;

    private Servak server;
    private bool isCreatedAgent = false;

    private GameObject agent;
    WheelController component = null;
    private bool connectionStopped = false;

    void Start()
    {
        server = new Servak();
        server.getMessage();

    }
    void Update()
    {

        if (!isCreatedAgent && !connectionStopped)
        {
            agent = Instantiate(agentPrefab, startPosition.position, startPosition.rotation);
            component = agent.GetComponent<WheelController>();
            component.servak = server;

            isCreatedAgent = true;
        }
        if (agent == null && !connectionStopped)
        {
            Destroy(GameObject.Find("CamParent"));
            isCreatedAgent = false;
        }
        else if (component.connectionStopped)
        {
            connectionStopped = true;
            Destroy(agent);
            Destroy(GameObject.Find("CamParent"));
        }
    }
}
```

Файл WheelController.cs:

```
using System.Collections.Generic;
using UnityEngine;
using myspace;
using UnityEditor.Presets;

public class WheelController : MonoBehaviour {

    //MECHANIC VARIABLES
    public WheelAlignment[] steerableWheels;

    public float BreakPower;

    public float Horizontal = 0.0f;
    public float Vertical = 0.0f;
    //Steering variables
    public float wheelRotateSpeed;
    public float wheelSteeringAngle;
```

```
//Motor variables
public float wheelAcceleration;
public float wheelMaxSpeed;

//SERV PARAM
public Servak servak;

//3d points
[SerializeField] Transform mainTransform;
[SerializeField] Transform frontPointTransform;
[SerializeField] Transform backRightPoint;
[SerializeField] Transform frontRightPoint;
private Vector3 previousFrontPoint;

//reward
private float reward = 0.0f;

//OBSERVATION
private float driftAngle = 0.0f;
private float driftAngleNormalized = 0.0f;

private float wheelAngle = 0.0f;

private float velocity = 0.0f;
private float velocityNormalized = 0.0f;

private float borderDistance = 0.0f;
float borderDistanceNormalized = 0.0f;

private float frontRightDistance = 29.9f;

//CONST
private const float MAX_BORDER_DISTANCE = 21.0f;

//states
private bool isCollided = false;
private bool isDone = false;
private bool firstFrame = true;

public bool connectionStopped = false;

System.Diagnostics.Stopwatch stopwatch;

public Rigidbody RB;

private List<float> statVelocity = new List<float>();
private List<float> statDist = new List<float>();
private List<float> statAngle = new List<float>();
private float statReward = 0.0f;

private void Start()
{
    previousFrontPoint = frontPointTransform.position;
    stopwatch = new System.Diagnostics.Stopwatch();
    stopwatch.Start();
}
void Update()
{
    if (!firstFrame && !connectionStopped)
    {
        handleMessage();
    }
    else firstFrame = false;
    //setInput();
    wheelControl();
}
private void LateUpdate()
{
    handleObservation();
    returnMessage();
}
private void handleMessage()
{
```

```
        string message = servak.getMessage();

        if (message.Contains("R")) Destroy(gameObject);
        else if (message.Contains("Q"))
        {
            servak.retMessage("END of sim!");
            connectionStopped = true;
        }
        else
        {
            string[] splittedMsg = message.Split('|');
            float[] actions = new float[2];
            actions[0] = float.Parse(splittedMsg[0]);
            actions[1] = float.Parse(splittedMsg[1]);

            setInput(actions[0], actions[1]);
        }
    }
    private void returnMessage()
    {
        string message;

        message = velocityNormalized.ToString() + "|"
                + borderDistance.ToString() + "|"
                + frontRightDistance.ToString() + "|"
                + driftAngleNormalized.ToString() + "|"
                + wheelAngle.ToString() + "|"
                + Horizontal.ToString() + "|"
                + Vertical.ToString() + "|"
                + reward.ToString() + "|"
                + (isDone ? 1.ToString() : 0.ToString());

        servak.retMessage(message);
    }
    private void handleObservation()
    {
        velocity = RB.velocity.magnitude;

        calculateDriftAngle();

        wheelAngle = steerableWheels[0].steeringAngle / wheelSteeringAngle;

        if (isCollided) isDone = true;
        if (stopwatch.ElapsedMilliseconds > 2000 && velocity < 5.0f) isDone = true;
        if (stopwatch.ElapsedMilliseconds > 15000) isDone = true;

        castExplorationRay();
        borderDistance = castRayToBorder();

        borderDistanceNormalized = velocity < 1.0f ? 0.0f : (1.0f - (borderDistance / MAX_BORDER_DISTANCE));
        velocityNormalized = velocity / 15.0f;
        driftAngleNormalized = driftAngle / 13.0f;

        reward = velocityNormalized + borderDistanceNormalized + driftAngleNormalized;
        statReward += reward;

        statVelocity.Add(velocityNormalized);
        statDist.Add(borderDistanceNormalized);
        statAngle.Add(driftAngleNormalized);
    }

    private void calculateDriftAngle()
    {
        Vector3 vectorA = frontPointTransform.position - previousFrontPoint;
        Vector3 forwardVector = mainTransform.TransformDirection(Vector3.back);

        float tmpAngle = Vector3.Angle(forwardVector, vectorA);

        if (tmpAngle > 0) driftAngle = Vector3.Angle(forwardVector, vectorA);
        if (velocity < 1.0f) driftAngle = 0.0f;

        previousFrontPoint = frontPointTransform.position;
    }
    private float castRayToBorder()
```

```
{
    Vector3 rayDirection;
    Ray ray;
    RaycastHit hit;

    float tmpBorderDistance = borderDistance;
    rayDirection = Quaternion.AngleAxis(-45, transform.up) * transform.forward;
    ray = new Ray(backRightPoint.position, rayDirection);
    if (Physics.Raycast(ray, out hit)) tmpBorderDistance = hit.distance;
    Debug.DrawRay(ray.origin, ray.direction * tmpBorderDistance, Color.green);

    if (tmpBorderDistance >= MAX_BORDER_DISTANCE) return borderDistance;
    else return tmpBorderDistance;
}
private void castExplorationRay()
{
    Vector3 rayDirection;
    Ray ray;
    RaycastHit hit;

    rayDirection = Quaternion.AngleAxis(45, transform.up) * -transform.forward;
    ray = new Ray(frontRightPoint.position, rayDirection);
    if (Physics.Raycast(ray, out hit) && hit.distance < 30.0f) frontRightDistance = hit.distance;
    Debug.DrawRay(ray.origin, ray.direction * frontRightDistance, Color.green);
}

private void setInput(float horizontal, float vertical)
{
    if (horizontal > 0) Horizontal += 0.01f;
    else Horizontal -= 0.01f;
    if (vertical > 0) Vertical += 0.01f;
    //else verticalInput -= 0.01f;
    if (Horizontal > 1.0f) Horizontal = 1.0f;
    if (Horizontal < -1.0f) Horizontal = -1.0f;

    if (Vertical > 1.0f) Vertical = 1.0f;
    if (Vertical < -1.0f) Vertical = -1.0f;
}
private void setInput()
{
    Horizontal = Input.GetAxis("Horizontal");
    Vertical = Input.GetAxis("Vertical");
}
//Applies steering and motor torque
void wheelControl()
{
    for (int i = 0; i < steerableWheels.Length; i++)
    {
        //Sets default steering angle
        steerableWheels[i].steeringAngle = Mathf.LerpAngle(steerableWheels[i].steeringAngle,
                                            0, Time.deltaTime * wheelRotateSpeed);
        //Sets default motor speed
        steerableWheels[i].wheelCol.motorTorque = -Mathf.Lerp(steerableWheels[i].wheelCol.motorTorque,
                                            0, Time.deltaTime * wheelAcceleration);

        if (Vertical > 0.1) steerableWheels[i].wheelCol.motorTorque =
                -Mathf.Lerp(steerableWheels[i].wheelCol.motorTorque,
                wheelMaxSpeed, Time.deltaTime * wheelAcceleration);

        if (Vertical < -0.1)
        {
            steerableWheels[i].wheelCol.motorTorque =
                Mathf.Lerp(steerableWheels[i].wheelCol.motorTorque,
                wheelMaxSpeed, Time.deltaTime * wheelAcceleration * BreakPower);
            RB.drag = 0.3f;
        }
        else RB.drag = 0;

        if (Horizontal > 0.1) steerableWheels[i].steeringAngle =
                Mathf.LerpAngle(steerableWheels[i].steeringAngle,
                wheelSteeringAngle, Time.deltaTime * wheelRotateSpeed);

        if (Horizontal < -0.1) steerableWheels[i].steeringAngle =
                Mathf.LerpAngle(steerableWheels[i].steeringAngle,
```

```
                    -wheelSteeringAngle, Time.deltaTime * wheelRotateSpeed);
            }
        }
        private void OnCollisionEnter(Collision collision)
        {
            if (collision.collider.transform.parent.name != "oval_mod_terrain") isCollided = true;
        }
    }
```

Файл pythonAPI:

```
using System;
using System.Diagnostics;
using System.Net.Sockets;
using System.Net;
using System.Text;

namespace myspace
{
    public class Servak
    {
        IPAddress ipAddress;
        IPEndPoint endPoint;
        Socket listener;
        Socket handler;

        private Stopwatch stopwatch;
        private bool recieved;

        public Servak()
        {
            recieved = false;
            ipAddress = new IPAddress(new byte[] { 192, 168, 0, 100 });
            endPoint = new IPEndPoint(ipAddress, 12555);
            listener = new Socket(ipAddress.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
            listener.Bind(endPoint);
            listener.Listen(1);
            UnityEngine.Debug.Log("Server started");
        }

        public string getMessage()
        {
            stopwatch = new Stopwatch();
            stopwatch.Start();
            handler = listener.Accept();
            string data = null;
            byte[] bytes = new byte[1024];
            int bytesRec = handler.Receive(bytes);
            data += Encoding.UTF8.GetString(bytes, 0, bytesRec);
            Console.WriteLine(data);
            recieved = true;
            return data;
        }

        public void retMessage(string message)
        {
            if (recieved)
            {
                byte[] bytes = Encoding.UTF8.GetBytes(message);
                handler.Send(bytes);
                handler.Shutdown(SocketShutdown.Both);
                handler.Close();
                recieved = false;
                stopwatch.Stop();
            }
        }
    }
}
```

ДОДАТОК Б

Лістинг коду програми алгоритму NEAT

Файл main.py:

```
import neat
import os
import unityEnv
import pickle
import matplotlib.pyplot as plt

best_pop_fitness = 0.0
i = 0
fitneses = []
stats_max_mean = []

folder_path = ""

def eval_genomes(genomes, config):
    global best_pop_fitness, i, fitneses, stats_max_mean
    best_pop_fitness = 0.0
    fitneses = []
    for genome_id, genome in genomes:
        genome.fitness = 1.0
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        params = unityEnv.reset()
        reward = params[1]
        isDone = params[2]
        while not isDone:
            action = net.activate(params[0])
            params = unityEnv.step(action)
            reward = params[1]
            isDone = params[2]
            genome.fitness += reward
        print(genome.fitness)
        fitneses.append(genome.fitness)
        if genome.fitness > best_pop_fitness:
            best_pop_fitness = genome.fitness
            saveGenome(genome, folder_path + "best_of_gen_" + str(i) + ".pickle")
    stats_max_mean.append((max(fitneses), sum(fitneses)/len(fitneses)))
    i += 1


def run(config_file):

    # Load configuration.
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         config_file)

    p = neat.Population(config)

    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    p.add_reporter(neat.Checkpointer(5))

    # Run for up to 300 generations.
    winner = p.run(eval_genomes, 50)

    saveGenome(winner, folder_path + "bestLastGen.pickle")

    #save stats
```

```
    saveGenome(stats_max_mean, folder_path + "stats.pickle")

    # Display the winning genome.
    print('\nBest genome:\n{!s}'.format(winner))

def saveGenome(genome, file_path):
    with open(file_path, "wb") as f:
        pickle.dump(genome, f)

def test_winner(config_file, file_name):
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         config_file)

    with open(folder_path + file_name, "rb") as f:
        winner = pickle.load(f)
        net = neat.nn.FeedForwardNetwork.create(winner, config)
        params = unityEnv.reset()
        isDone = params[2]
        while not isDone:
            action = net.activate(params[0])
            params = unityEnv.step(action)
            isDone = params[2]

def test_best_of_generations(config_file, start, count):
    for j in range(start, count):
        test_winner(config_file, "best_of_gen_" + str(j) + ".pickle")

def printStats(statsParam):
    #plt.plot([x[0] for x in statsParam], color='red')
    plt.plot([x[1] for x in statsParam], color='green')
    plt.xlabel('Покоління')
    plt.ylabel('Середня пристосованість')
    plt.title('Залежність середньої пристосованості від покоління')
    plt.show()

if __name__ == '__main__':

    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config-feedforward')

    #run(config_path)
    #test_winner(config_path, "best.pickle")
    #test_winner(config_path, "bestLastGen.pickle")
    test_best_of_generations(config_path, 45, 50)

    #with open(folder_path + "stats.pickle", "rb") as f:
    #    s = pickle.load(f)
    #    printStats(s)
    print(unityEnv.sendQuit())
```

Файл unityEnv.py:

```
import socket
import numpy as np

server_ip = ''
server_port = 12555

def reset():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((server_ip, server_port))
    message = "R"
```

```
    client_socket.send(message.encode())
    data = client_socket.recv(1024)
    dataStr = data.decode()
    dataArr = dataStr.split('|')
    client_socket.close()
    return ([float(element.replace(',', '.')) for element in dataArr[0:7]],
float(dataArr[7].replace(',', '.')), bool(int(dataArr[8])))

def step(param):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((server_ip, server_port))
    message = str(param[0]) + '|' + str(param[1])
    message = message.replace('.', ',')
    client_socket.send(message.encode())
    data = client_socket.recv(1024)
    dataStr = data.decode()
    dataArr = dataStr.split('|')
    client_socket.close()
    return ([float(element.replace(',', '.')) for element in dataArr[0:7]],
float(dataArr[7].replace(',', '.')), bool(int(dataArr[8])))

def sendQuit():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((server_ip, server_port))
    message = "Q"
    client_socket.send(message.encode())
    data = client_socket.recv(1024)
    dataStr = data.decode()
    client_socket.close()
    return dataStr
```

Файл config-feedforward:

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 150000
pop_size              = 20
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.0
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1
```

```
# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 7
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 2

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.2
```