МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Західноукраїнський національний університет

Факультет комп'ютерних інформаційних технологій

Кафедра інформаційно- обчислювальних систем і управління

БЕДНАРЧУК Назар Юрійович

Інтерактивна програмна система взаємодії користувача з ігровим штучним інтелектом / Interactive software system of user interaction with game Artificial Intelligence

спеціальність: 122 - Комп'ютерні науки

освітньо-професійна програма - Комп’ютерні науки

Кваліфікаційна робота

Виконав студент групи КН-42

Н.Ю. Беднарчук

_____

Науковий керівник:

к.т.н., доцент І.В. Турченко

_____

Кваліфікаційну роботу

допущено до захисту:

"___" _____ 20___ р.

Завідувач кафедри

_____Н. В. Дзюбановська

ТЕРНОПІЛЬ – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність: 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____20__р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
БЕДНАРЧУКУ Назару Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Інтерактивна програмна система взаємодії користувача з ігровим штучним інтелектом / Interactive software system of user interaction with game Artificial Intelligence

керівник роботи: к.т.н., доцент І.В. Турченко
затверджений наказом по університету від 01 грудня 2025 року № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити

– провести аналіз предметної області та існуючих систем взаємодії користувача з ігровим штучним інтелектом;
– дослідити сучасні підходи до взаємодії користувача з неігровими персонажами;
– обґрунтувати вибір методів та алгоритмів штучного інтелекту для забезпечення адаптивної поведінки неігрових персонажів;
– здійснити постановку задачі проєктування;
– спроєктувати архітектуру, алгоритмічне та інформаційне забезпечення системи;
– розробити програмне середовище для реалізації взаємодії з ігровим ШІ та реалізувати інтеграцію розроблених моделей ШІ з ігровим рушієм Unity;
– провести тестування розробленої системи взаємодії користувача з ігровим штучним інтелектом.

5. Перелік графічного матеріалу у роботі

– Архітектура інтерактивної системи взаємодії користувача з ігровим ШІ
– Схема алгоритму роботи ігрового ШІ

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студент _____Н.Ю. Беднарчук
підпис

Керівник роботи _____к.т.н., доцент І.В. Турченко
підпис

АНОТАЦІЯ

Кваліфікаційна робота на тему «Інтерактивна програмна система взаємодії користувача з ігровим штучним інтелектом» на здобуття освітнього ступеня «бакалавр» зі спеціальності 122 «Комп'ютерні науки» освітньої програми «Комп'ютерні науки» написана обсягом у 67 сторінок і містить 11 ілюстрацій, 3 додатки та 25 використаних джерел.

Метою кваліфікаційної роботи є розробка інтерактивної програмної системи взаємодії користувача з ігровим штучним інтелектом.

Об'єктом дослідження є процеси отримання, зберігання, аналізу, обробки та використання інформації про поведінку користувача в інтерактивному ігровому середовищі.

Результатом роботи є програмна система, що поєднує механізми аналізу дій гравця, адаптивне управління неігровими персонажами, обробку станів ігрового середовища та інтеграцію алгоритмів штучного інтелекту для формування динамічної поведінки неігрових персонажів у реальному часі.

Розроблено інтерактивну програмну систему взаємодії користувача з ігровим штучним інтелектом, яку можуть використовувати розробники комп'ютерних ігор для створення адаптивних ігрових механік, динамічної поведінки неігрового персонажа та підвищення рівня залученості користувачів у процес гри. Система також може бути використана як основа для подальших досліджень у сфері ігрового штучного інтелекту та інтерактивних програмних систем.

Ключові слова: ІНТЕРАКТИВНА ПРОГРАМНА СИСТЕМА, ІГРОВИЙ ШТУЧНИЙ ІНТЕЛЕКТ, NPC, МАШИННЕ НАВЧАННЯ, UNITY, АДАПТИВНА ПОВЕДІНКА, ІГРОВІ АГЕНТИ, НАВЧАННЯ З ПІДКРІПЛЕННЯМ.

ANNOTATION

Qualification work on the topic «Interactive Software System for User Interaction with Game Artificial Intelligence» for obtaining a bachelor's degree in specialty 122 «Computer Science» of the educational program «Computer Science» is written on 67 pages and contains 11 illustrations, 3 appendices, and 25 references.

The aim of this qualification work is to develop an interactive software system for adaptive user interaction with game artificial intelligence using modern machine learning methods and decision-making algorithms.

The object of research is the processes of analysis, processing, and use of user behavior data in an interactive game environment.

The result of the work is a software system that combines player action analysis, adaptive control of non-player characters, game environment state processing, and integration of artificial intelligence algorithms to generate dynamic NPC behavior in real time.

An interactive software system for user interaction with game artificial intelligence has been developed. It can be used by game developers to create adaptive gameplay mechanics, dynamic NPC behavior, and improve user engagement in gameplay. The system can also serve as a basis for further research in the field of game artificial intelligence and interactive software systems.

Keywords: INTERACTIVE SOFTWARE SYSTEM, GAME ARTIFICIAL INTELLIGENCE, NPC, MACHINE LEARNING, UNITY, ADAPTIVE BEHAVIOR, GAME AGENTS, REINFORCEMENT LEARNING.

ЗМІСТ

ВСТУП

Індустрія комп’ютерних ігор є однією з найдинамічніших галузей інформаційних технологій, яка постійно розвивається та впроваджує нові технологічні рішення. Зростання обчислювальних можливостей комп’ютерної техніки та розвиток програмного забезпечення сприяють створенню складних інтерактивних ігрових середовищ, у яких користувач очікує високого рівня реалістичності, динамічності та непередбачуваності ігрового процесу. У зв’язку з цим особливого значення набуває проблема забезпечення якісної взаємодії користувача з ігровим середовищем та неігровими персонажами.

Одним із ключових компонентів сучасних відеоігор є ігровий штучний інтелект, який відповідає за поведінку неігрових персонажів (англ. Non-Playable Character, NPC), керування внутрішніми процесами гри, адаптацію складності та створення динамічної взаємодії між користувачем і віртуальним світом. Саме якість реалізації ігрового штучного інтелекту значною мірою визначає рівень залучення користувача, реіграбельність гри та загальне враження від ігрового процесу.

Традиційні підходи до реалізації ігрового штучного інтелекту базуються на використанні жорстко заданих алгоритмів, скінченних автоматів (англ. Finite State Machines, FSM) [1] та поведінкових дерев (англ. Behavior Trees) [2]. Такі методи дозволяють створювати базову логіку поведінки NPC, однак мають низку суттєвих обмежень. Насамперед вони характеризуються передбачуваністю поведінки, недостатньою адаптивністю до дій користувача та складністю масштабування у великих інтерактивних середовищах. У більшості випадків поведінка персонажів визначається наперед встановленими сценаріями, що знижує реалістичність взаємодії та зменшує інтерес користувача до гри після тривалого проходження.

У сучасних умовах розвитку технологій штучного інтелекту актуальним стає використання методів машинного навчання та навчання з підкріпленням (англ. Reinforcement Learning, RL) для створення адаптивних ігрових систем [3-6]. На відміну від класичних підходів, алгоритми машинного навчання дозволяють

ігровим агентам самостійно аналізувати дії користувача, накопичувати досвід та змінювати власну поведінку відповідно до ситуації в ігровому середовищі. Це відкриває можливість створення більш реалістичних NPC, які здатні навчатися, змінювати тактику поведінки та взаємодіяти з гравцем у режимі реального часу [3].

Особливо перспективним є використання навчання з підкріпленням, у межах якого ігровий агент навчається шляхом отримання винагород за виконання певних дій [7-8]. Такий підхід дозволяє формувати складні поведінкові моделі без необхідності жорсткого програмування кожної можливої ситуації. Використання сучасних фреймворків, таких як Unity ML-Agents [9], значно спрощує інтеграцію алгоритмів машинного навчання з ігровими рушіями та дозволяє реалізовувати системи адаптивного штучного інтелекту безпосередньо в середовищі розробки гри.

Актуальність даного дослідження обумовлена необхідністю створення інтерактивних програмних систем, здатних забезпечити високий рівень взаємодії між користувачем та ігровим штучним інтелектом. Сучасні користувачі очікують від відеоігор не лише якісної графіки та фізики, але й реалістичної поведінки NPC, здатності гри адаптуватися до стилю проходження, рівня навичок та індивідуальних особливостей гравця. Саме тому розробка адаптивних систем штучного інтелекту є важливим напрямом розвитку сучасної ігрової індустрії.

Метою роботи є розробка інтерактивної програмної системи взаємодії користувача з ігровим штучним інтелектом.

Для досягнення поставленої мети необхідно виконати такі завдання:

– провести аналіз предметної області та існуючих систем взаємодії користувача з ігровим штучним інтелектом;

– дослідити сучасні підходи до взаємодії користувача з неігровими персонажами;

– обґрунтувати вибір методів та алгоритмів штучного інтелекту для забезпечення адаптивної поведінки неігрових персонажів;

– здійснити постановку задачі проєктування;

– спроєктувати архітектуру, алгоритмічне та інформаційне забезпечення системи;

– розробити програмне середовище для реалізації взаємодії з ігровим ШІ та реалізувати інтеграцію розроблених моделей ШІ з ігровим рушієм Unity;

– провести тестування розробленої системи взаємодії користувача з ігровим штучним інтелектом.

Об’єктом дослідження є процеси отримання, зберігання, аналізу, обробки та використання інформації про поведінку користувача в інтерактивному ігровому середовищі.

Предметом дослідження є методи, алгоритми та програмні засоби реалізації адаптивного ігрового штучного інтелекту.

У процесі дослідження використано методи системного аналізу, об’єктно-орієнтованого програмування, машинного навчання, навчання з підкріпленням, а також сучасні технології розробки програмного забезпечення в середовищі Unity. Для реалізації системи застосовано мову програмування C#, фреймворк Unity ML-Agents та алгоритми нейронних мереж для формування поведінки NPC.

Практичне значення роботи полягає у створенні програмної системи, яка може бути використана під час розробки сучасних комп’ютерних ігор із динамічною поведінкою NPC та адаптивною взаємодією з користувачем. Запропонована система дозволяє підвищити рівень інтерактивності ігрового процесу, забезпечити персоналізацію поведінки ігрових агентів та створити більш реалістичне й захопливе ігрове середовище. Отримані результати можуть бути використані як основа для подальших досліджень у сфері ігрового штучного інтелекту та інтерактивних програмних систем.

Результати роботи опубліковано в матеріалах міжнародної науково-практичної конференції студентів і молодих вчених «ICSPM 2026: Інтелектуальні обчислювальні системи та управління проектами», м. Тернопіль, 21–22 травня 2026 р. (додаток В).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ПРОЄКТУВАННЯ

1.1 Опис предметної області ігрового штучного інтелекту

В умовах стрімкого розвитку індустрії інтерактивних розваг та відеоігор якість, глибина та природність взаємодії користувача з ігровим середовищем стали ключовими факторами комерційного успіху цифрових продуктів. Відео ігри розглядаються не лише як розважальні системи, а як складні інтерактивні симуляції, що поєднують графіку, фізичні моделі, наративні структури та інтелектуальну поведінку віртуальних агентів. У цьому контексті особливу роль відіграє ігровий штучний інтелект, який відповідає за формування поведінки неігрових персонажів, управління станом ігрового світу, балансування складності та забезпечення реактивної взаємодії з діями гравця.

Традиційні методи реалізації ігрового ШІ, зокрема кінцеві автомати дерева поведінки, системи правил та скриптові алгоритми, протягом тривалого часу залишалися основою розробки ігрової логіки. Вони характеризуються високою передбачуваністю, простотою реалізації та контрольованістю поведінки NPC, що робить їх зручними для комерційної розробки. Водночас у сучасних умовах ці підходи демонструють суттєві обмеження. Основною проблемою є детермінованість поведінки, через що досвідчені гравці можуть швидко виявляти закономірності, експлуатувати слабкі місця алгоритмів та втрачати інтерес до гри. Крім того, створення складних сценаріїв вимагає значних витрат часу на ручне програмування великої кількості логічних правил, що ускладнює масштабування системи.

Поява та активний розвиток технологій машинного навчання і навчання з підкріпленням відкривають нові можливості для еволюції ігрового штучного інтелекту. На відміну від класичних підходів, такі методи дозволяють агентам самостійно навчатися на основі взаємодії з ігровим середовищем і поведінкою гравця. Це забезпечує формування адаптивних стратегій, які можуть змінюватися

залежно від контексту ситуації, стилю гри користувача та поточного стану середовища. Використання подібних підходів дозволяє значно зменшити залежність від ручного програмування правил, підвищити варіативність поведінки NPC та створити більш природний ігровий досвід.

Актуальність дослідження також зумовлена зростаючим попитом на системи динамічного балансування складності (англ. Dynamic Difficulty Adjustment, DDA) [10], які здатні автоматично підлаштовувати рівень виклику під індивідуальні навички гравця. У сучасних іграх користувачі мають суттєво різний рівень підготовки, і використання статичних налаштувань складності часто призводить до дисбалансу ігрового процесу: або надмірної простоти, або надмірної складності. Це негативно впливає на залученість та мотивацію гравця. Тому важливим завданням є підтримка стану «потоку» (англ. Flow) [11-12], за якого складність гри оптимально відповідає рівню навичок користувача, забезпечуючи максимальне занурення в ігровий процес.

Окремої уваги потребують дослідження у сфері інтерактивних систем, які дозволяють не лише адаптуватися до гравця, але й формувати двосторонню взаємодію, коли дії користувача впливають на еволюцію поведінки ігрового штучного інтелекту. У таких системах NPC можуть змінювати тактику, стратегію та навіть поведінкові моделі на основі історії взаємодії з конкретним гравцем, що створює ефект «живого» ігрового світу та підвищує реіграбельність.

З огляду на вищезазначене, особливо актуальним є подальший розвиток інтерактивних програмних систем взаємодії користувача з ігровим штучним інтелектом, які здатні адаптуватися до змінних умов у реальному часі, навчатися на основі ігрового досвіду та ефективно функціонувати в умовах обмежених обчислювальних ресурсів клієнтських пристроїв. Реалізація таких систем дозволяє не лише покращити якість ігрового досвіду, але й розширити можливості керування ігровим наративом, створюючи більш гнучкі та інтелектуально насичені ігрові середовища.

1.2 Огляд і аналіз існуючих рішень

У галузі розробки комп’ютерних ігор значна увага приділяється створенню інтерактивних програмних систем взаємодії користувача з ігровим штучним інтелектом. Такі системи забезпечують адаптивну поведінку неігрових персонажів (англ. NPC), формують динамічну реакцію на дії користувача та підвищують рівень занурення у віртуальне середовище. Аналіз існуючих рішень дозволяє визначити основні підходи, технології та програмні засоби, що застосовуються для реалізації ігрового штучного інтелекту.

Одним із найпоширеніших підходів є використання дерев поведінки, які застосовуються для моделювання логіки прийняття рішень ігровими агентами. Даний підхід базується на ієрархічній структурі задач, що дозволяє комбінувати прості дії у складні сценарії поведінки. Дерева поведінки широко використовуються в сучасних ігрових рушіях, таких як Unity та Unreal Engine, завдяки їх гнучкості, модульності та зручності візуалізації логіки. Основною перевагою цього підходу є можливість створення складної поведінки без значного ускладнення коду, що зменшує ймовірність помилок та підвищує масштабованість системи.

Приклад структури дерева поведінки ігрового агента зображено на рисунку 1.1.

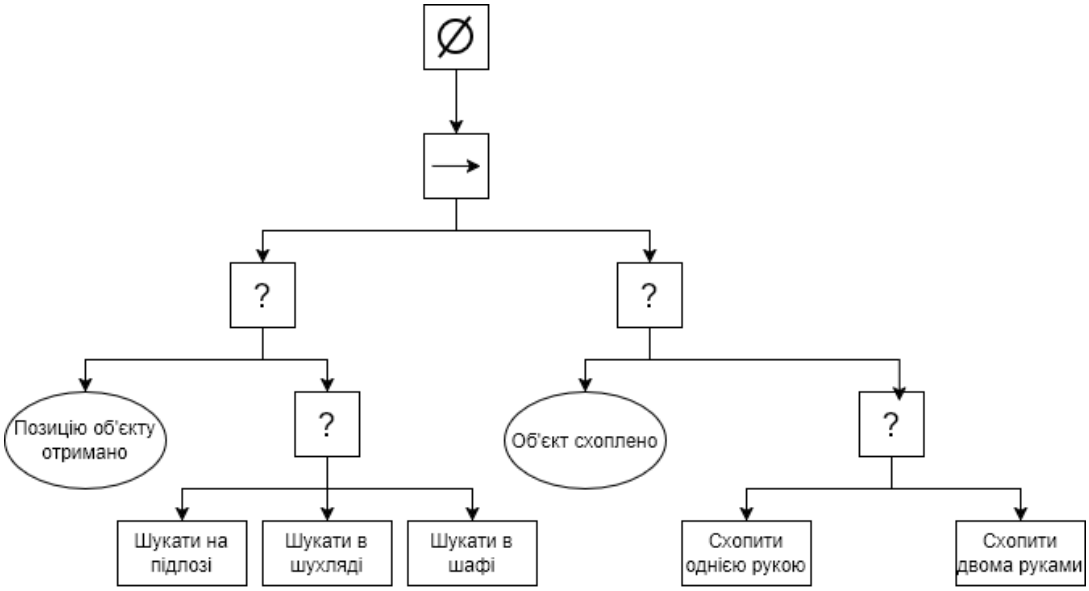

Рисунок 1.1 – Структура дерева поведінки ігрового агента

Іншим важливим підходом до реалізації ігрового штучного інтелекту є використання скінченних автоматів, які описують поведінку агента через чітко визначений набір станів та правил переходу між ними. Кожен стан відповідає певній поведінковій логіці, наприклад «патрулювання», «переслідування» або «атака», а переходи між станами ініціюються подіями або умовами ігрового середовища. FSM є відносно простими у реалізації, добре піддаються налагодженню та забезпечують передбачувану поведінку NPC, що робить їх широко використовуваними в комерційній розробці ігор.

Водночас основним недоліком FSM є швидке зростання складності при збільшенні кількості станів і переходів, що призводить до так званого «комбінаторного вибуху». У великих ігрових системах це ускладнює підтримку коду, знижує масштабованість і обмежує можливість повторного використання логіки. Саме тому на практиці FSM часто використовуються у поєднанні з іншими методами штучного інтелекту, такими як дерева поведінки або системи планування, що дозволяє підвищити гнучкість та структурованість поведінки ігрових агентів.

Суттєвим етапом розвитку ігрового штучного інтелекту стало впровадження планувальних алгоритмів, зокрема підходу Планування дій на основі цілей (англ. Goal-Oriented Action Planning, GOAP) [12-13]. Даний метод базується на концепції постановки цілей для агента та автоматичного формування послідовності дій, необхідних для їх досягнення. На відміну від FSM, де поведінка жорстко закріплена за станами, GOAP дозволяє агенту динамічно обирати дії залежно від поточного стану середовища, доступних ресурсів та пріоритетів цілей.

Основною перевагою GOAP є здатність формувати різні сценарії поведінки без необхідності явного програмування кожного з них окремо. Це забезпечує вищий рівень адаптивності та дозволяє створювати більш «інтелектуальну» поведінку NPC, яка виглядає природнішою для гравця. Алгоритм будує план дій як послідовність кроків, де кожна дія має умови виконання та результат, що змінює стан системи.

Приклад роботи GOAP зображено на рисунку 1.2, де представлено процес формування плану дій на основі поточного стану середовища та поставленої цілі, адаптована з [13].

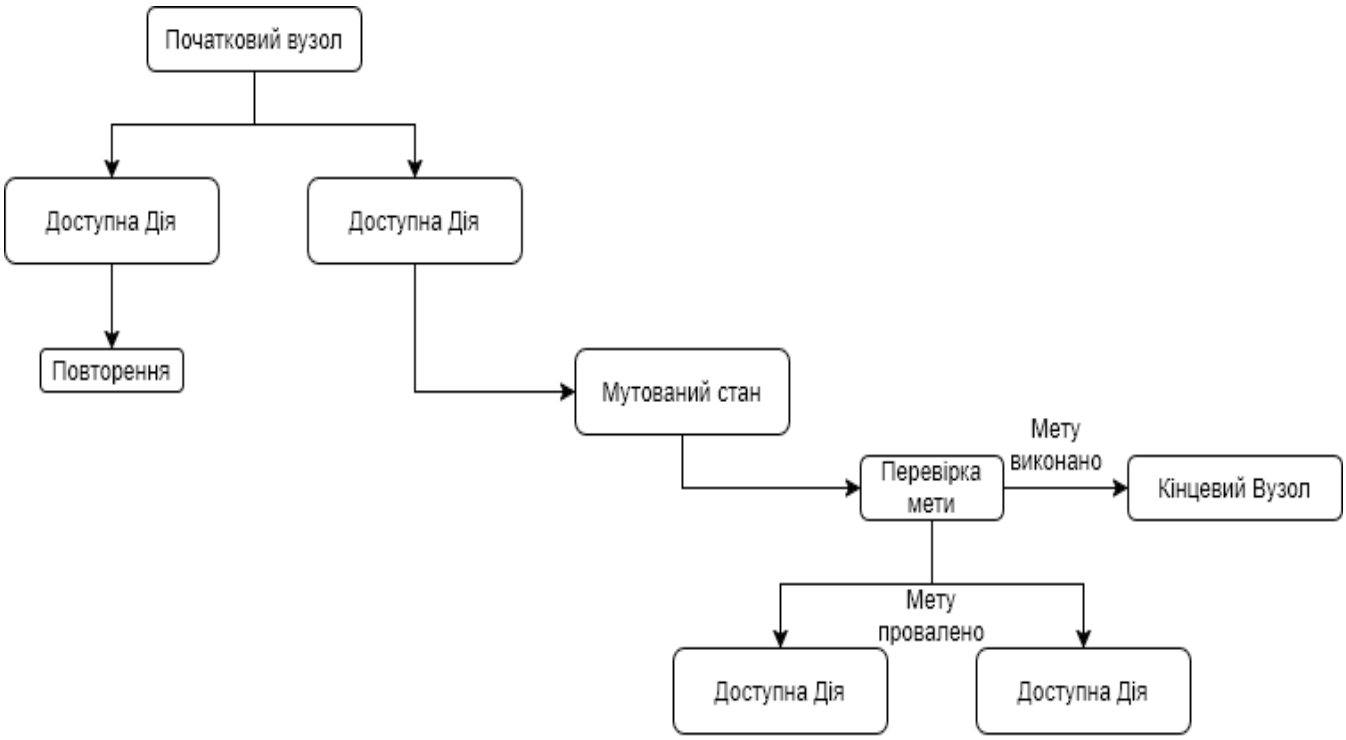

Рисунок 1.2 – Схема роботи GOAP [13]

Яскравим прикладом практичного застосування GOAP є гра F.E.A.R., у якій неігрові персонажі демонструють складну тактичну поведінку. NPC у цій грі здатні аналізувати ситуацію на полі бою, координувати дії між собою, змінювати тактику в залежності від дій гравця та самостійно обирати найбільш ефективні стратегії виживання та атаки. Це створює відчуття «розумного» супротивника та суттєво підвищує рівень занурення в ігровий процес.

Завдяки такому підходу поведінка персонажів стає значно більш варіативною та адаптивною, оскільки рішення приймаються не за фіксованими сценаріями, а на основі аналізу поточної ігрової ситуації в реальному часі.

На рисунку 1.3 наведено приклад ігрового середовища та поведінки NPC у подібних системах, що демонструє принципи роботи планувальних алгоритмів у динамічних умовах.

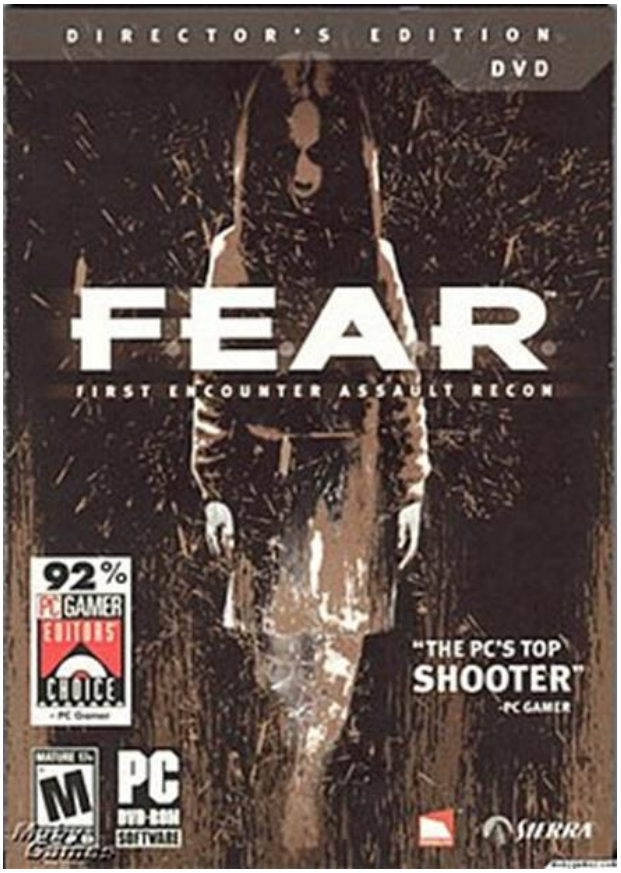

Рисунок 1.3 – Обкладинка F.E.A.R

Таким чином, використання GOAP у поєднанні з іншими методами ігрового штучного інтелекту дозволяє суттєво підвищити рівень автономності та реалістичності поведінки ігрових агентів, що є важливим кроком у розвитку сучасних інтерактивних систем.

Окрім теоретичних підходів, у сфері ігрового штучного інтелекту широко застосовуються готові програмні рішення та спеціалізовані платформи, що значно спрощують процес розробки складних поведінкових систем. Наприклад, програмний продукт xaitMap забезпечує автоматичну генерацію навігаційних сіток (англ. NavMesh) та реалізацію ефективних алгоритмів пошуку шляхів, що є критично важливим для коректного переміщення NPC у тривимірному ігровому середовищі. У свою чергу, xaitControl реалізує моделювання поведінки агентів на основі скінченних автоматів і поведінкових дерев, дозволяючи гнучко налаштовувати логіку прийняття рішень неігровими персонажами та забезпечувати контрольовану реакцію на дії гравця [14].

Використання подібних інструментів дозволяє розробникам значно скоротити час створення базової поведінки NPC, а також підвищити стабільність та передбачуваність ігрової логіки. Крім того, такі рішення легко інтегруються з популярними ігровими рушіями, зокрема Unity та Unreal Engine, що робить їх універсальними для комерційної розробки.

Яскравим прикладом складної адаптивної системи ігрового штучного інтелекту є Nemesis System, використана у серії ігор Middle-earth, розроблених студією Warner Bros. Ця система формує унікальних противників, які запам’ятовують взаємодію з гравцем, отримують індивідуальні характеристики, імена, звання та історію взаємин із користувачем. У результаті кожна зустріч з таким персонажем стає унікальною, а самі NPC можуть розвиватися, посилюватися або слабшати залежно від результатів взаємодії. Це суттєво підвищує рівень реіграбельності та емоційну залученість гравця, створюючи ефект «живого світу» [15].

У наукових дослідженнях та експериментальних розробках також активно використовуються спеціалізовані середовища для тестування алгоритмів ігрового штучного інтелекту.

Одним із таких середовищ є ViZDoom, яке дозволяє створювати агентів, що взаємодіють із тривимірним середовищем на основі візуальної інформації. Це наближує процес навчання до реальних умов сучасних відеоігор і дає можливість досліджувати ефективність алгоритмів навчання з підкріпленням у динамічних і частково спостережуваних середовищах [16].

ViZDoom активно використовується для тестування стратегій поведінки агентів, оптимізації реакцій у бойових сценаріях та дослідження здатності ШІ до довготривалого планування дій.

На рисунку 1.4 графічно представлено процес формування простору спостережень у середовищі ViZDoom, де ігрове оточення трансформується у структурований масив вхідних даних для інтелектуального агента, що поєднує послідовність візуальних кадрів та векторні змінні стану.

Рисунок 1.4 – Візуалізація процесу симуляції та збору спостережень агента у середовищі ViZDoom

Таким чином, аналіз сучасних програмних рішень і дослідницьких платформ показує, що поєднання готових інструментів (xaitMap, xaitControl), комерційних систем адаптивного ШІ (Nemesis System) та навчальних середовищ (ViZDoom) формує комплексну базу для розробки сучасних ігрових інтелектуальних систем. Використання подібних рішень у межах проєкту є доцільним, оскільки дозволяє поєднати перевірені інженерні підходи з можливостями сучасного машинного навчання для створення адаптивної та інтерактивної поведінки ігрових агентів.

На основі проведеного аналізу існуючих рішень у сфері ігрового штучного інтелекту та інтерактивних систем взаємодії користувача з ігровими агентами можна зробити висновок, що сучасні провідні компанії-розробники програмного забезпечення активно використовують гібридні підходи, які поєднують класичні алгоритми поведінки NPC з методами машинного навчання та навчання з підкріпленням. Такі рішення дозволяють досягати більшої адаптивності ігрових персонажів, підвищувати реалістичність їхньої поведінки та забезпечувати персоналізований ігровий досвід.

Досвід провідних фірм, зокрема використання поведінкових дерев, скінченних автоматів, систем планування дій, а також сучасних фреймворків для навчання агентів, таких як Unity ML-Agents, демонструє високу ефективність при

створенні складних ігрових систем. Використання подібних підходів є доцільним і при реалізації даного проєкту, оскільки вони забезпечують баланс між продуктивністю, контрольованістю поведінки NPC та можливістю їх адаптації до дій користувача.

Таким чином, можна зробити висновок, що використання досвіду провідних компаній-розробників програмних продуктів та інтеграція відомих рішень у сфері ігрового штучного інтелекту є обґрунтованим і доцільним підходом при проєктуванні та реалізації інтерактивної програмної системи взаємодії користувача з ігровим штучним інтелектом.

1.3 Постановка задачі проектування

Стрімкий розвиток індустрії комп'ютерних ігор висуває нові, підвищені вимоги до якості взаємодії між користувачем та ігровим середовищем. Сучасні гравці очікують не лише візуально привабливого контенту, а й інтелектуально насиченої, непередбачуваної та адаптивної поведінки неігрових персонажів. Традиційні підходи до реалізації ігрового штучного інтелекту, що базуються переважно на жорстко заданих правилах, скінченних автоматах або статичних деревах поведінки, часто мають обмежену гнучкість. Вони складно масштабуються при зростанні складності ігрових сценаріїв і призводять до передбачуваності дій NPC. Це знижує рівень залученості користувача та зумовлює необхідність дослідження і впровадження сучасних алгоритмів ШІ для створення більш природного й інтерактивного ігрового досвіду, здатного підтримувати баланс між складністю гри та можливостями гравця.

Мета роботи полягає у розробці інтерактивної програмної системи взаємодії користувача з ігровим штучним інтелектом.

Система повинна забезпечити адаптивну, контекстно-залежну та динамічну поведінку ігрових агентів у реальному часі на основі аналізу дій гравця, історії взаємодії та стану ігрового середовища.

Для досягнення поставленої мети було визначено такі завдання:

– провести аналіз предметної області та існуючих систем взаємодії користувача з ігровим штучним інтелектом;

– дослідити сучасні підходи до взаємодії користувача з неігровими персонажами;

– обґрунтувати вибір методів та алгоритмів штучного інтелекту для забезпечення адаптивної поведінки неігрових персонажів;

– здійснити постановку задачі проєктування;

– спроєктувати архітектуру, алгоритмічне та інформаційне забезпечення системи;

– розробити програмне середовище для реалізації взаємодії з ігровим ШІ та реалізувати інтеграцію розроблених моделей ШІ з ігровим рушієм Unity;

– провести тестування розробленої системи взаємодії користувача з ігровим штучним інтелектом.

Відповідно до мети роботи, необхідно розробити інтерактивну програмну систему взаємодії користувача з ігровим штучним інтелектом. Для реалізації цієї мети висуваються такі вимоги до виконання:

а) аналітичні вимоги:

1) провести комплексний аналіз предметної області та існуючих систем ігрового штучного інтелекту;

2) дослідити підходи до забезпечення взаємодії користувача з неігровими персонажами;

б) алгоритмічні та методичні вимоги:

1) обґрунтувати вибір методів та алгоритмів штучного інтелекту, які забезпечать адаптивну поведінку неігрових персонажів;

2) забезпечити динамічну, контекстно-залежну поведінку агентів у реальному часі на основі аналізу дій гравця, історії взаємодії та стану ігрового середовища;

в) архітектурні та технічні вимоги:

1) спроєктувати архітектуру програмної системи, включаючи її інформаційне забезпечення;

2) розробити повноцінний ігровий продукт та відповідний користувацький інтерфейс;

3) реалізувати інтеграцію розроблених моделей штучного інтелекту з ігровим рушієм Unity;

г) вимоги до тестування та оцінки:

1) провести тестування розробленої системи в ігровому середовищі;

2) оцінити ефективність запропонованих алгоритмів штучного інтелекту у забезпеченні адаптивності поведінки NPC.

2 АРХІТЕКТУРА, АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ ІНТЕРАКТИВНОЇ ПРОГРАМНОЇ СИСТЕМИ ВЗАЄМОДІЇ КОРИСТУВАЧА З ІГРОВИМ ШІ

2.1 Архітектура проєктованої системи

Метою цього підрозділу є формування концептуальних засад розробки інтерактивної програмної системи на основі глибокого навчання та навчання з підкріпленням. Запропонована система дозволяє розв’язати традиційні проблеми ігрового штучного інтелекту, такі як лінійність сценаріїв, відсутність адаптивності до стилю гравця та низька операційна ефективність класичних алгоритмів. Використання сучасних методів машинного навчання дає можливість створити ігрових агентів, здатних аналізувати поведінку користувача, змінювати власну тактику та приймати рішення в реальному часі залежно від поточного стану ігрового середовища.

На рисунку 2.1 зображено архітектуру інтерактивної системи взаємодії користувача з ігровим ШІ [24].

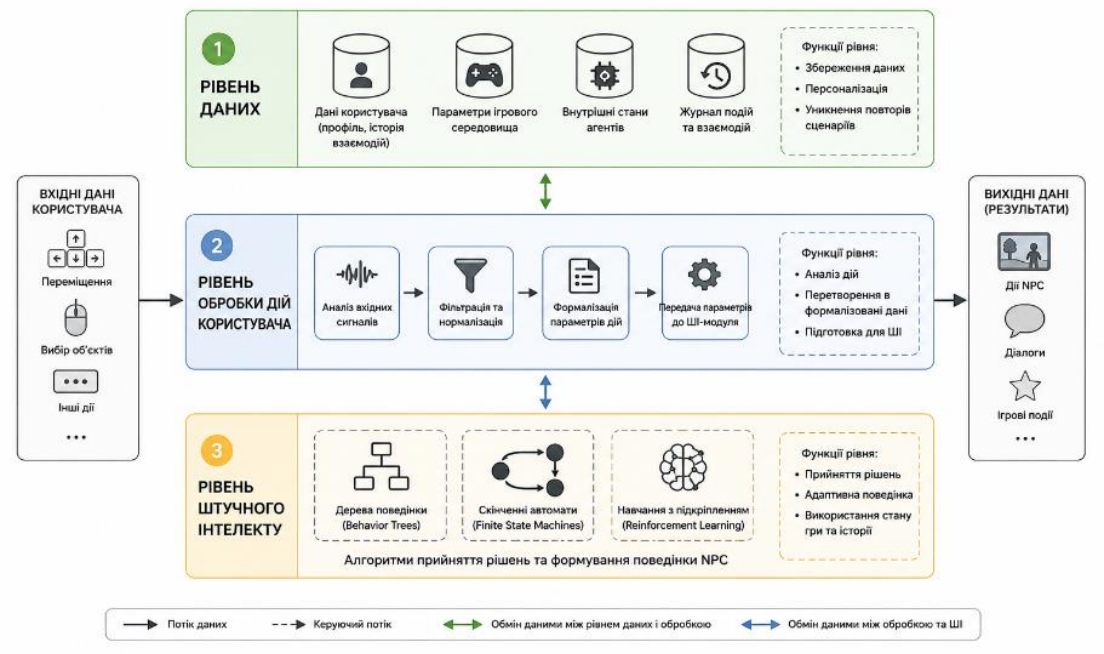

Рисунок 2.1 – Архітектура інтерактивної системи взаємодії користувача з ігровим ШІ

Для реалізації проєктних рішень обрано агентно-орієнтований підхід у поєднанні з об’єктно-орієнтованим. Агентно-орієнтований підхід використовується для проєктування ШІ як сукупності NPC, які взаємодіють із середовищем, аналізують вхідні дані та виконують самостійне прийняття рішень.

Об’єктно-орієнтований підхід застосовується для реалізації внутрішньої логіки гри в середовищі Unity, забезпечуючи інкапсуляцію станів, поведінки та взаємодії між компонентами системи. Використання даного підходу дозволяє забезпечити модульність архітектури, спрощує підтримку програмного коду та забезпечує можливість масштабування системи в майбутньому.

У межах дослідження також здійснюється розробка самої комп’ютерної гри, яка виступає тестовим середовищем для функціонування інтерактивної системи штучного інтелекту. Гра реалізована у жанрі двовимірного платформера(англ. 2D platformer), де користувач взаємодіє з ігровим середовищем через переміщення персонажа, бойову систему та дослідження рівнів. Основною особливістю гри є наявність неігрових персонажів, поведінка яких формується за допомогою алгоритмів штучного інтелекту та адаптується до стилю гри користувача.

Двовимірний формат гри обрано через його відносну простоту реалізації, що дозволяє зосередити основну увагу саме на механізмах взаємодії користувача з ігровим ШІ, а не на складності тривимірної графіки. Ігровий процес включає динамічні бойові сцени, систему пересування платформами, взаємодію з ворогами та механізми зміни поведінки NPC залежно від дій гравця.

Наприклад, у випадку агресивного стилю проходження противники можуть діяти більш координовано, атакувати швидше або змінювати дистанцію ведення бою. У свою чергу, при обережному стилі гри NPC можуть переходити до пошукової або захисної поведінки.

Для реалізації гри використовується ігровий рушій Unity, який забезпечує підтримку фізичної симуляції, анімаційної системи, навігації, роботи з двовимірною графікою та інтеграції з ML-фреймворками. Важливою перевагою Unity є можливість використання Unity ML-Agents Toolkit – спеціалізованого середовища для навчання агентів за допомогою алгоритмів Reinforcement Learning.

Це дозволяє поєднати класичну логіку гри з механізмами машинного навчання та реалізувати адаптивну поведінку NPC без необхідності повного ручного програмування сценаріїв.[17]

Логіка функціонування системи базується на багаторівневій архітектурі. Клієнтський модуль, реалізований у Unity, відповідає за фізичну симуляцію, відтворення анімацій, навігацію персонажів, обробку дій користувача та відображення інтерфейсу. Цей модуль також здійснює збір даних про поточний стан ігрового середовища, включаючи координати об'єктів, параметри здоров'я, інформацію про атаки та інші поведінкові характеристики.

Модуль збору даних (англ. Data Pipeline) виконує фіксацію параметрів ігрового стану у вигляді векторів стану (англ. state vectors), які передаються до обчислювального ядра системи. Зібрані дані використовуються як для процесу навчання агентів, так і для подальшого інференсу під час виконання гри. Це дозволяє системі аналізувати поведінку користувача в реальному часі та адаптувати логіку NPC до поточної ситуації.

Модуль інференсу та навчання (англ. AI Core) реалізує алгоритми машинного навчання та відповідає за прийняття рішень ігровими агентами. Для цього використовуються моделі глибокого навчання, які навчаються шляхом взаємодії із середовищем і отримання винагород за успішні дії. Такий підхід дозволяє агентам поступово формувати ефективні стратегії поведінки, адаптуватися до стилю гри користувача та забезпечувати більш реалістичну взаємодію.

Завдяки автоматичній ідентифікації поведінкових патернів гравця система забезпечує підтримку стану «потоку», при якому складність ігрового процесу відповідає рівню навичок користувача. Це дозволяє підвищити рівень залученості гравця, покращити реіграбельність гри та мінімізувати відчуття передбачуваності поведінки NPC.

Для забезпечення стабільності та масштабованості системи передбачено використання кластерної архітектури з механізмами резервування та ізольованими середовищами виконання. Це дозволяє забезпечити відмовостійкість системи, оптимізувати використання обчислювальних ресурсів та підтримувати можливість

подальшого розширення функціоналу інтерактивної програмної системи.

У таблиці 2.1 описана характеристика комп’ютерної системи.

Таблиця 2.1 – Характеристика комп’ютерної системи

Категорія	Деталі
Процесор (CPU)	AMD Ryzen (аналог рівня i3/i5/i7 залежно від вимог)
Відеокарта (GPU)	Сучасна AMD / NVIDIA GPU (оптимізована для тензорних операцій)
Оперативна пам'ять	16 GB DDR-4
Накопичувач	1 TB SSD NVMe
Операційна система	Windows 11
Бібліотеки/Фреймворки	PyTorch 2.1, TensorFlow 2.15, Unity ML-Agents 3.0
Мова програмування	C# (Client), Python 3.10 (AI Core)
Середовище розробки	Unity / Visual Studio Code

У таблиці 2.2 наведено порівняльний аналіз запропонованої інтерактивної системи ігрового штучного інтелекту та традиційних підходів, що базуються на скриптовій логіці й скінченних автоматах (англ. FSM).

Таблиця 2.2 – Порівняльний аналіз запропонованої інтерактивної системи ігрового штучного інтелекту та традиційних підходів

Характеристика	Запропонована система (Interactive AI)	Існуючі рішення (Scripted/FSM)
Точність адаптації	Висока завдяки динамічному навчанню	Низька, обмежена деревом рішень

Продовджження таблиці 2.2

Характеристика	Запропонована система (Interactive AI)	Існуючі рішення (Scripted/FSM)
Адаптивність	Навчається на нових стратегіях гравця	Вимагає ручного оновлення коду
Проактивність	Прогнозує дії гравця наперед	Лише реактивно відповідає на події
Реіграбельність	Створює унікальний досвід щоразу	Передбачувані патерни поведінки
Масштабованість	Легко масштабується на нових персонажів	Кожна зміна потребує переписування логіки

Проведене порівняння демонструє основні переваги використання методів машинного навчання та адаптивного штучного інтелекту у сучасних ігрових системах.

2.2 Алгоритмічне забезпечення

Алгоритмічне забезпечення системи базується на інтеграції ігрового рушія з моделями машинного навчання. Загальний алгоритм функціонування передбачає циклічний процес:

- отримання вектору стану середовища ;
- нормалізація даних ;
- інференс моделі ;
- застосування дії в грі отримання ;
- винагороди (під час навчання).

На рисунку 2.2 зображено алгоритм роботи ігрового ШІ.

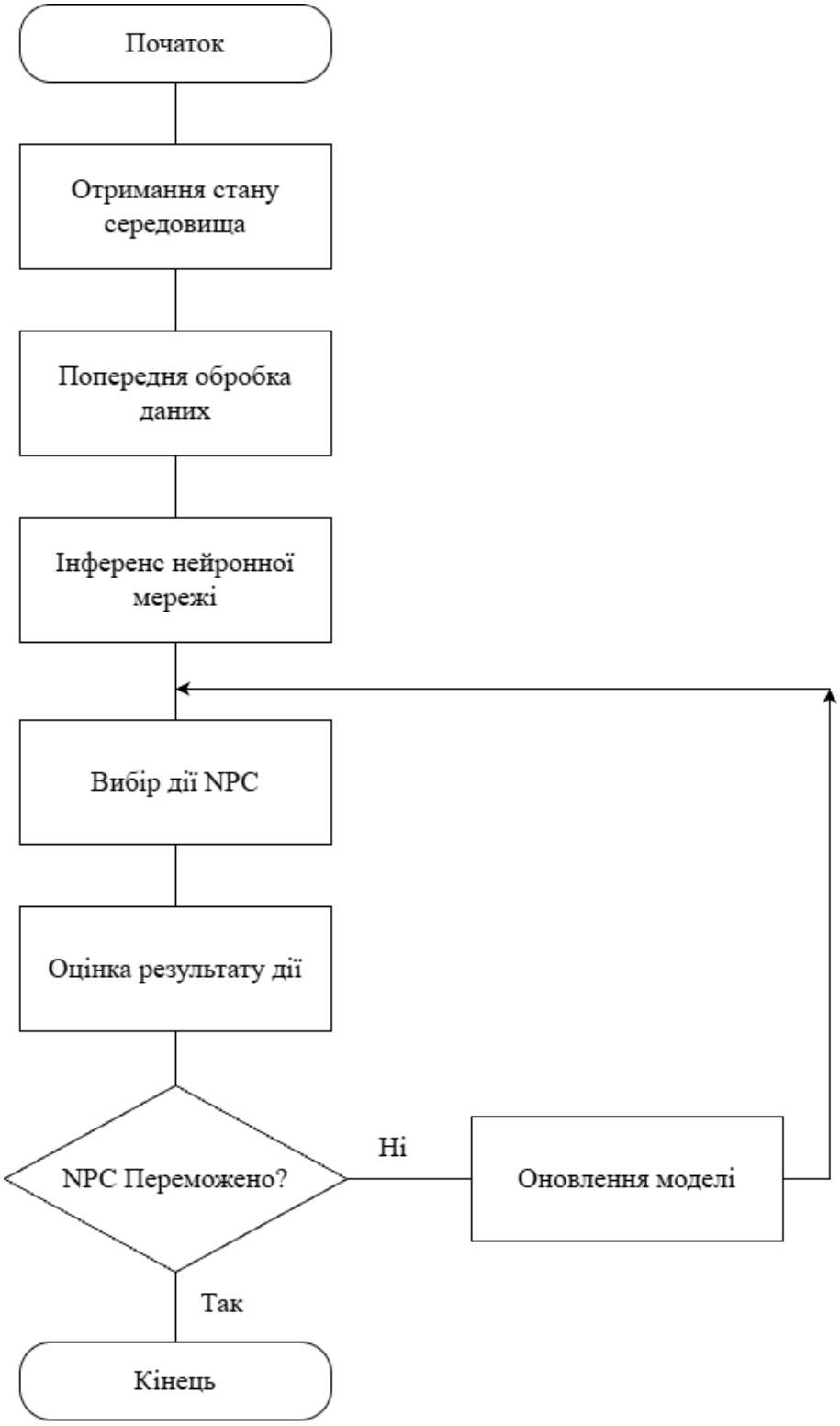

Рисунок 2.2 – Схема алгоритму роботи ігрового ШІ

Для безпосереднього вибору дій використовується алгоритм Оптимізація проксимальної політики (англ. Proximal Policy Optimization, PPO) [17-18]. PPO реалізує архітектуру «актор-критик» (англ. Actor-Critic). Мережа-актор генерує ймовірність наступного кроку, а мережа-критик оцінює його якість. Функція втрат

PPO використовує механізм обрізки (clipping), щоб уникнути деструктивних оновлень політики $L^{CLIP}(\theta)$ [18]:

$$L^{CLIP}(\theta) = \hat{E}_t\left[min(t_t(\theta)\hat{A}_t' clip(r_t(\theta)'1-\epsilon'1+\epsilon)\hat{A}_t\right], \quad (2.1)$$

де $r_t(\theta)$ – відношення нової політики до старої,

$\hat{A}_t$ – оцінка переваги (advantage),

є – гіперпараметр обрізки.

Під час прямого обчислення (інференсу) вхідна інформація (вектор стану x) проходить через шари нейронної мережі. Математично перетворення [18-19] в кожному прихованому шарі можна виразити як:

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \quad (2.2)$$

де w_i – ваги з'єднань;

b – зміщення;

f – функція активації (наприклад, ReLU або Tanh) [19-20];

Для обробки різних типів даних застосовуються різні архітектури:

1) MLP (Multi-Layer Perceptron): Обробка низькорозмірних векторних даних (координати, ресурси). Забезпечує найвищу швидкість інференсу;

2) LSTM (Long Short-Term Memory): Прогнозування довгострокових патернів поведінки гравця;

3) EfficientNet / ResNet: Використовуються як енкодери візуальних ознак (просторовий аналіз ігрової сцени). EfficientNet обрано як найперспективнішу завдяки методу складового масштабування [21], що забезпечує баланс між точністю і швидкістю обробки.

У Unity алгоритми ШІ виділяються в окремі C# компоненти. Навігаційні задачі розв'язуються за допомогою NavMeshAgent. Обмін даними між логікою ШІ

та фізичним світом відбувається асинхронно через систему подій (Event System), що знижує навантаження на процесор.

2.3 Інформаційне забезпечення системи

Оскільки розроблювана інтерактивна програмна система орієнтована на аналіз ігрового середовища та прийняття рішень у режимі реального часу, використання класичної реляційної бази даних у традиційному вигляді не є доцільним. Основний обсяг інформації формується динамічно під час виконання гри, а обробка даних здійснюється безпосередньо в оперативній пам’яті та графічній пам’яті пристрою. Такий підхід дозволяє мінімізувати затримки під час обміну даними між ігровим рушієм та модулями штучного інтелекту, що є критично важливим для забезпечення плавного та безперервного ігрового процесу.

Інформаційне забезпечення системи базується на потоковій передачі даних, використанні конфігураційних файлів та серіалізованих моделей машинного навчання. Основними джерелами інформації є поточні стани ігрового середовища, дії користувача, параметри NPC та результати роботи алгоритмів штучного інтелекту.

У процесі функціонування система постійно отримує та аналізує великий обсяг вхідної інформації, що використовується для формування поведінки ігрових агентів.

Вхідна інформація:

1) вектори стану
2) візуальні дані;
3) дані телеметрії.

До цієї категорії належать параметри, що описують поточний стан ігрового середовища:

- координати гравця та NPC;
- швидкість руху об’єктів;
- напрямок руху;

- рівень здоров’я;
- залишок ресурсів;
- стан анімацій;
- інформація про перешкоди;
- дистанція між об’єктами.

Вектори стану є основою для формування вхідних даних нейронної мережі та використовуються алгоритмами машинного навчання для прогнозування подальших дій.

Для просторового аналізу середовища система може використовувати:

- матриці пікселів;
- карти глибини;
- результати Raycast-запитів;
- інформацію про колізії;
- маски навколишніх об’єктів.

Візуальні дані особливо важливі для реалізації алгоритмів комп’ютерного бачення та моделей глибокого навчання(англ. Deep Learning)[22].

Телеметрія використовується для аналізу поведінки користувача та роботи ШІ:

- постріли;
- атаки;
- стрибки;
- використання предметів;
- ухилення;
- зміна напрямку руху;
- час проходження рівня;
- кількість смертей;
- частота взаємодій із середовищем.

Накопичення телеметричних даних дозволяє системі адаптувати складність гри та формувати більш персоналізовану поведінку NPC.

На відміну від класичних інформаційних систем, де основою є реляційна база

даних, у даній системі основний акцент зроблено на високошвидкісному обміні даними між компонентами. Інформація зберігається переважно в оперативній пам’яті, що дозволяє забезпечити мінімальний рівень затримки під час виконання інференсу моделей штучного інтелекту.

Інформаційна модель системи складається з трьох основних потоків даних:

1) потік конфігурації;
2) потік моделей;
3) потік телеметрії

Параметри роботи системи, налаштування NPC та характеристики ігрових сесій зберігаються у:

- файлах формату .json;
- конфігураціях .yaml;
- ScriptableObjects у Unity.

Такі конфігурації дозволяють:

- швидко змінювати параметри гри;
- налаштовувати поведінку агентів;
- масштабувати систему без зміни програмного коду.

Навчені нейронні мережі експортуються з Python-середовищ:

- PyTorch;
- TensorFlow.

Після навчання моделі конвертуються у:

- .onnx;
- .nn;
- формати Unity Barracuda.

Отримані файли імпортуються в Unity та використовуються для виконання інференсу безпосередньо всередині ігрового рушія.

Система збереження журналів подій забезпечує:

- запис часових рядів;
- фіксацію помилок;
- збереження поведінкових патернів;

- виявлення аномалій у роботі ШІ.

Журнали подій можуть зберігатися:

- у текстових файлах;
- у бінарних форматах;
- у форматі CSV або JSON.

Подібний підхід спрощує аналіз поведінки агентів та подальше тестування системи.

Через відсутність бази даних, проєктування даталогічної моделі реалізується на рівні архітектури класів (у нотації UML).

Динамічна модель даних формується об'єктами в пам'яті:

1) клас AgentController зберігає поточні стани (позицію, швидкість, цілі);

2) клас EnvironmentManager акумулює дані про сцену і передає сформований тензор (масив даних) до моделі інференсу;

3) структури обміну даними між Python AI Core та Unity (якщо вони працюють на різних процесах) описуються за допомогою RPC протоколів (наприклад, gRPC) із чітко типізованими повідомленнями. Це повністю замінює традиційні ERD (Entity-Relationship Diagrams) і забезпечує мінімальну затримку (latency) під час взаємодії ШІ з користувачем.

3 РЕАЛІЗАЦІЯ ІНТЕРАКТИВНОЇ ПРОГРАМНОЇ СИСТЕМИ ВЗАЄМОДІЇ КОРИСТУВАЧА З ІГРОВИМ ШТУЧНИМ ІНТЕЛЕКТОМ

3.1 Реалізація програмного забезпечення

Для програмної реалізації інформаційної системи розроблено трирівневу архітектуру, що спирається на модульний принцип та концепції об'єктно-орієнтованого програмування (ООП). Структура системи спроєктована у вигляді взаємопов'язаних шарів: представлення (англ. Presentation Layer), штучного інтелекту (англ. AI Layer) та виконання на базі ігрового рушія (англ. Execution Layer).

Зв'язки між програмними компонентами, їхнє призначення та логічну взаємодію проілюстровано на загальній структурній схемі програмної системи на рисунку 3.1.

Запропонована схема деталізує інформаційні потоки від дій користувача та ігрового рушія Unity через конвеєр збору й обробки даних до нейромережевого модуля інференсу і навчання, забезпечуючи замкнений цикл керування ігровим процесом у реальному часі:

1) шар представлення;
2) шар Штучного нтелекту;
3) шар ігрового рушія.

Шар представлення відповідає за безпосередню взаємодію з користувачем та візуалізацію ігрового світу. До його складу входять два ключові блоки:

- гравець: кінцевий користувач, який взаємодіє з грою. Він надсилає «Команди для персонажа» (введення з клавіатури, миші тощо) в ігровий рушій. Натомість гравець отримує зворотний зв'язок у вигляді обробки ігрового процесу (візуальне відображення змін на екрані);
- Unity: ігровий рушій, який є центральним хабом на цьому шарі. Він приймає команди користувача, керує станом сцени та «Відправляє дані» про поточний ігровий стан на наступний архітектурний рівень для аналізу штучним інтелектом.

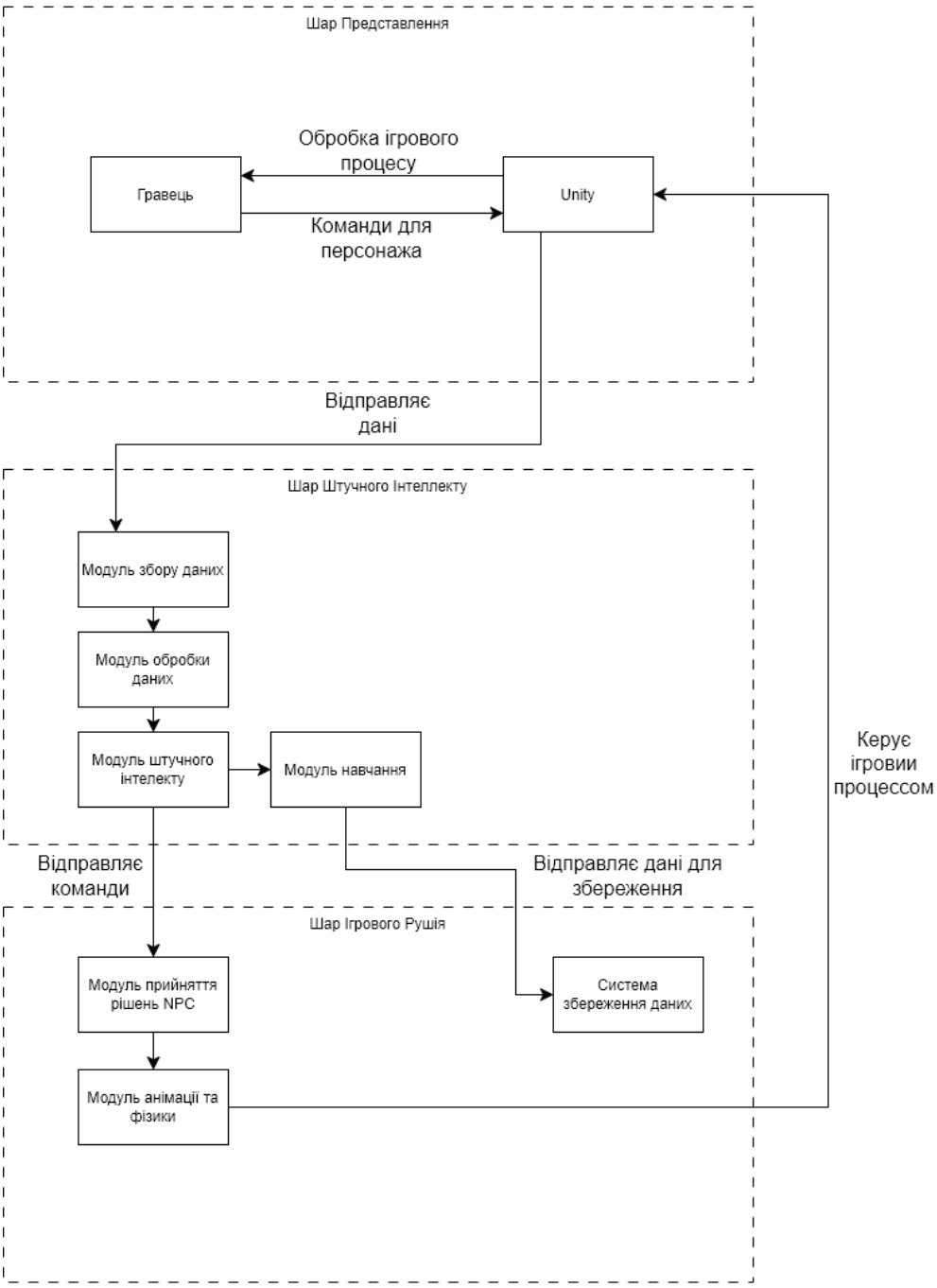

Рисунок 3.1 – Структурна схема ігрового ШІ

Шар ШІ – це ядро обчислювальної системи, де відбувається збір даних, аналіз та формування інтелектуальної поведінки. Процес обробки інформації тут організовано у вигляді послідовного конвеєра (англ. Data Pipeline):

а) модуль збору даних: Приймає первинну інформацію, надіслану з рушія Unity;

б) модуль обробки даних: Очищує, нормалізує та готує зібрані дані для нейромережевих моделей;

в) модуль штучного інтелекту: Центральний блок прийняття рішень на основі моделей глибокого навчання. Він виконує дві функції:

1) передає сформовані стани та досвід у модуль навчання для оптимізації політики агентів;

2) видає сформовані керуючі директиви (інференс) і відправляє команди на рівень ігрового рушія для виконання;

г) модуль навчання: оновлює моделі ШІ та відправляє дані для збереження у постійну пам'ять.

Шар ігрового рушія відповідає за інтерпретацію інтелектуальних команд у конкретні фізичні та логічні дії всередині гри, а також за фіксацію довгострокових даних.

Модуль прийняття рішень NPC отримує команди від модуля ШІ та транслює їх у конкретні наміри чи високорівневі завдання для неігрових персонажів.

Модуль анімації та фізики переводить рішення NPC на мову рушія – програє відповідні анімації рухів, атак чи ухилень та прораховує фізичні колізії об'єктів. Після цього цей модуль здійснює зворотний зв'язок: він безпосередньо керує ігровим процесом, повертаючи оновлений стан назад у блок Unity на шар представлення.

Система збереження даних – окремий блок, який приймає інформацію від Модуля навчання для фіксації прогресу навчання, конфігурацій або станів системи.

Система функціонує в режимі реального часу, реалізуючи циклічну логіку обробки даних із низькою затримкою:

1) дії гравця обробляються в Unity;

2) Unity транслює стан середовища на Шар Штучного Інтелекту через послідовні модулі збору, обробки та аналізу;

3) модуль ШІ формує стратегію, яка через Модуль прийняття рішень NPC та Модуль анімації/фізики змінює ігровий світ;

4) оновлений світ відображається в Unity, замикаючи коло зворотного зв'язку.

Структуру розробленого програмного забезпечення на рівні об'єктів детально відображено на UML-діаграмі класів яка зображена на рисунку 3.2.

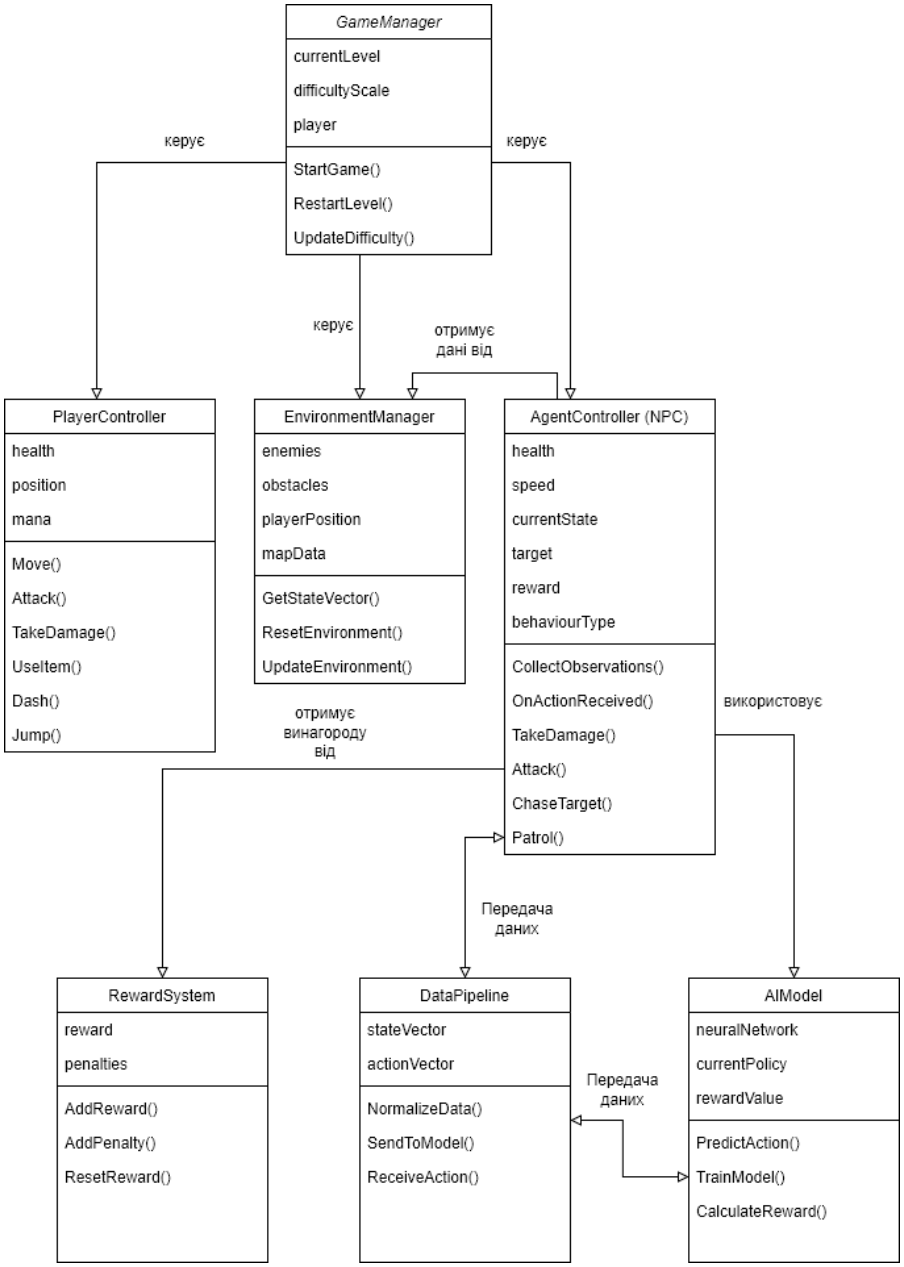

Рисунок 3.2 – UML діаграма класів

Основні класи системи:

– GameManager (Керуючий компонент): відповідає за життєвий цикл ігрової сесії та глобальну координацію між підсистемами. Поля: currentLevel

(поточний рівень), difficultyScale (коефіцієнт складності), player (посилання на контролер гравця). Методи: StartGame(), RestartLevel(), UpdateDifficulty();

– PlayerController (Контролер гравця): реалізує логіку взаємодії користувача з ігровим персонажем. Поля: health, mana, position. Методи: Move(), Attack(), TakeDamage(), UseItem(), Dash(), Jump();

– AgentController (Контролер ШІ-агента): головний інтерфейс ігрового штучного інтелекту, що забезпечує зв'язок між логікою поведінки та середовищем. Поля: health, speed, currentState (поточний стан), target (ціль), reward (сукупна винагорода), behaviorType (тип стратегії). Методи: CollectObservations(), OnActionReceived(), TakeDamage(), Attack(), ChaseTarget(), Patrol();

– AIModel (Модель інтелекту): інкапсулює логіку інференсу та навчання нейронної мережі. Поля: neuralNetwork, currentPolicy, rewardValue. Методи: PredictAction(), TrainModel(), CalculateReward();

– EnvironmentManager (Контролер середовища): виконує роль проксі-об'єкта, що акумулює стан ігрового простору. Поля: enemies (список ворогів), obstacles (перешкоди), playerPosition, mapData. Методи: GetStateVector(), ResetEnvironment(), UpdateEnvironment();

– DataPipeline (Канал обробки даних): забезпечує серіалізацію та нормалізацію потоків даних між середовищем (Unity) та архітектурою нейронної мережі. Поля: stateVector (вектор стану), actionVector (вектор дії). Методи: NormalizeData(), SendToModel(), ReceiveAction();

– RewardSystem (Система винагород): спеціалізований модуль для обчислення функцій винагороди (reward function) та штрафів (penalties) агента. Поля: reward, penalties. Методи: AddReward(), AddPenalty(), ResetReward().

Зв'язки між компонентами системи (архітектурна взаємодія):

– GameManager керує компонентами PlayerController, EnvironmentManager та AgentController;

– AgentController взаємодіє з AIModel для визначення стратегії, отримує вхідні дані (спостереження) від EnvironmentManager та аналізує ефективність своїх рішень через RewardSystem;

– DataPipeline виконує роль посередника, забезпечуючи коректний формат даних при передачі між AgentController та AIModel;

– EnvironmentManager відповідає за агрегацію об’єктів середовища, включаючи NPC та гравця.

Стани, у яких може знаходитися NPC, зображено на UML діаграмі станів на рисунку 3.3.

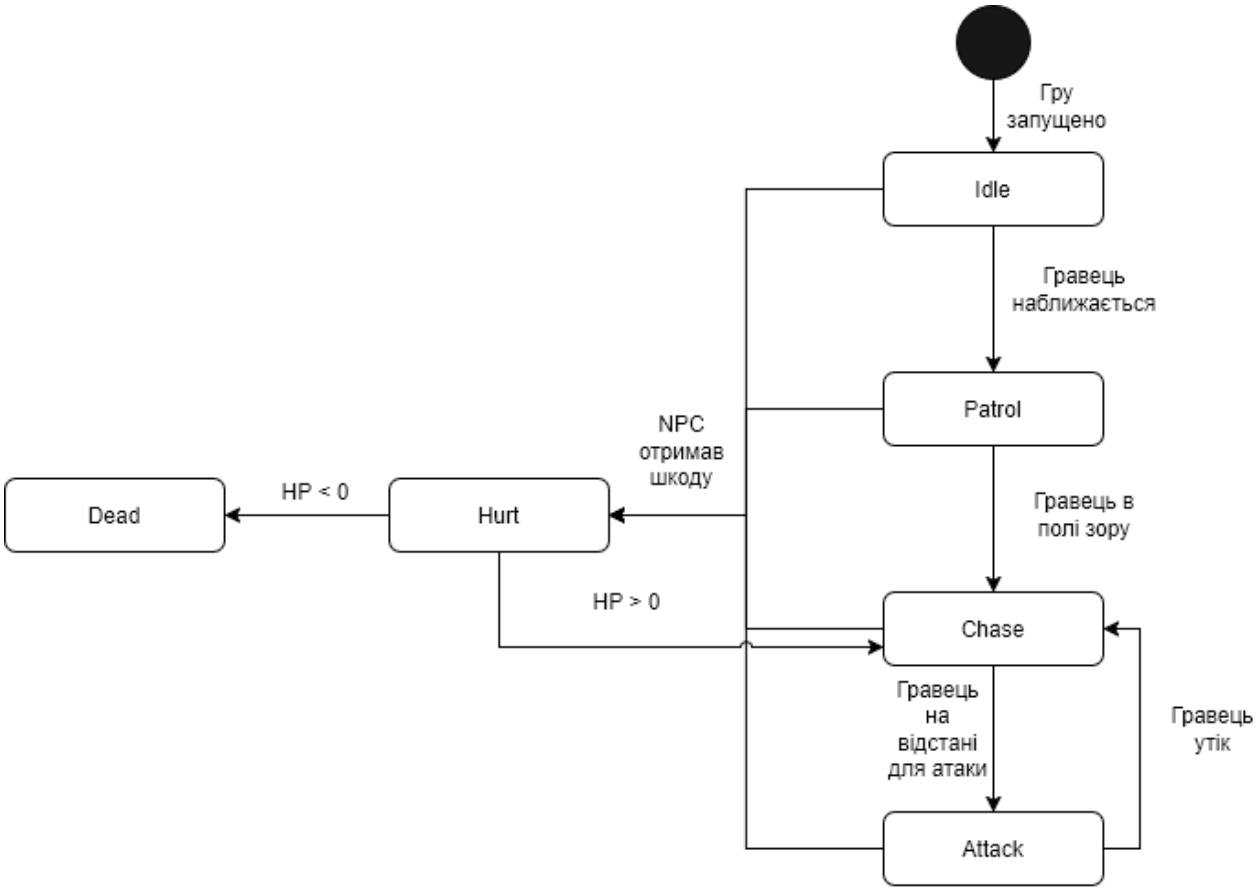

Рисунок 3.3 – UML діаграма станів

Для реалізації інтерактивної програмної системи взаємодії користувача з ігровим штучним інтелектом було обрано сучасні програмні засоби та технології, які забезпечують високу продуктивність, гнучкість розробки, підтримку алгоритмів машинного навчання та інтеграцію з ігровим середовищем у реальному часі

Основним середовищем розробки обрано ігровий рушій Unity. Даний рушій є одним із найпоширеніших інструментів для створення двовимірних і тривимірних

ігор завдяки широким можливостям інтеграції, підтримці фізики, анімацій та систем штучного інтелекту.

Вибір Unity обумовлений такими перевагами:

- підтримка мови програмування C#;
- наявність вбудованої фізичної системи;
- підтримка NavMesh та систем навігації;
- можливість інтеграції бібліотек машинного навчання;
- кросплатформеність;
- висока продуктивність роботи в реальному часі;
- велика кількість документації та готових інструментів.

Unity також забезпечує зручний механізм компонентно-орієнтованої розробки, що дозволяє легко масштабувати систему та додавати нові модулі.

Для реалізації логіки гри та поведінки ігрових агентів використовується мова програмування C#. Вона є основною мовою програмування у Unity та забезпечує високу швидкість виконання, підтримку об’єктно-орієнтованого програмування та зручні механізми роботи з подіями, фізикою та асинхронними процесами.

Перевагами використання C# є:

- простота інтеграції з Unity;
- підтримка багатопоточності;
- висока читабельність коду;
- наявність великої кількості бібліотек;
- підтримка сучасних принципів програмної інженерії.

Для реалізації алгоритмів машинного навчання та навчання з підкріпленням використовується бібліотека Unity ML-Agents Toolkit [5]. Даний фреймворк дозволяє інтегрувати моделі штучного інтелекту безпосередньо в ігрове середовище Unity та забезпечує взаємодію між Python і Unity через API.

Основними причинами вибору ML-Agents є:

- підтримка Reinforcement Learning;
- реалізація алгоритму PPO;
- підтримка навчання агентів у реальному часі;

– можливість використання нейронних мереж;
– інтеграція з TensorFlow та PyTorch;
– підтримка експорту моделей у формат .onnx.

ML-Agents дозволяє створювати адаптивних NPC, які навчаються на основі взаємодії з користувачем та ігровим середовищем.

Як основне середовище розробки використовується Visual Studio. Воно забезпечує:

– підтримку C#;
– інтеграцію з Unity;
– засоби налагодження;
– автодоповнення коду;
– аналіз помилок у режимі реального часу.

У системі використовуються такі формати:

– .json – для конфігурацій;
– .yaml – для параметрів навчання;
– .onnx – для збереження моделей нейронних мереж.

3.2 Інтерфейс користувача

Для забезпечення ефективної взаємодії користувача з ігровим штучним інтелектом у межах роботи було розроблено користувацький інтерфейс, який відповідає сучасним вимогам до зручності, інформативності та адаптивності ігрових систем.

Інтерфейс користувача є важливим елементом ігрового процесу, оскільки саме через нього користувач отримує інформацію про стан персонажа, ігрове середовище та результати взаємодії з NPC.

Під час проєктування інтерфейсу основна увага приділялася принципам мінімалізму, зрозумілості та швидкого сприйняття інформації. Інтерфейс не перевантажує екран зайвими елементами та забезпечує користувача лише

необхідною інформацією, що позитивно впливає на комфорт під час гри та дозволяє зосередитися безпосередньо на ігровому процесі.

До основних елементів інтерфейсу належать:

- індикатор здоров’я персонажа;
- панель витривалості або енергії;
- лічильники ресурсів;
- інформація про поточний стан NPC;
- меню паузи;
- елементи взаємодії з ігровим середовищем;
- візуальні ефекти отримання пошкоджень та бойових дій.

Інтерфейс реалізований засобами Unity UI Toolkit та Canvas System [23], що забезпечує гнучкість при адаптації елементів до різних роздільних здатностей екрана та платформ.

Використання компонентного підходу дозволяє легко модифікувати окремі елементи інтерфейсу без зміни загальної структури системи.

Для керування ігровим процесом та взаємодії з NPC користувач може використовувати різні типи пристроїв введення:

- клавіатуру;
- комп’ютерну мишу;
- геймпад;
- альтернативні способи вводу.

Підтримка різних типів контролерів реалізована за допомогою Unity Input System. Дана система дозволяє створювати універсальні схеми керування та автоматично адаптувати гру до різних пристроїв введення без необхідності зміни логіки програми. Це забезпечує кросплатформеність та підвищує доступність гри для користувачів .

На рисунках 3.4-3.5 представлено вікна графічого інтерфейсу реалізованого середовища для взаємодії з ігровим ШІ.

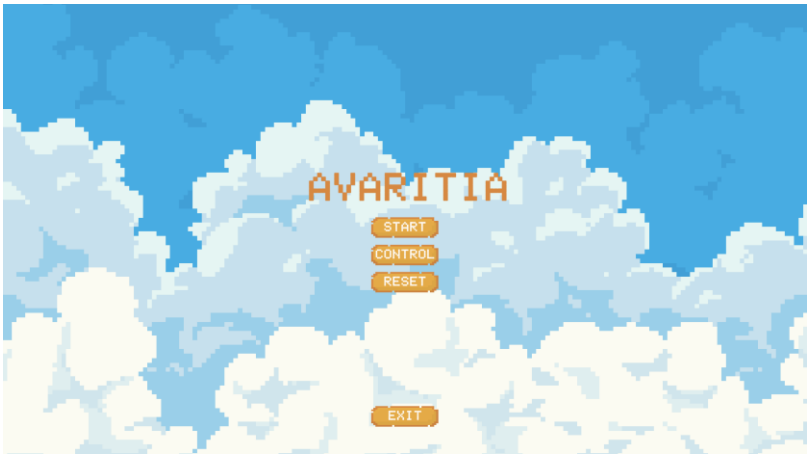

Рисунок 3.4 – Стартове вікно графічного інтерфейсу користувача середовища

Рисунок 3.5 – Графічний інтерфейс користувача в розробленому середовищі

Програмні коди NPC та керування персонажем гравця наведено відповідно у додатках А та Б.

3.3 Тестування розробленої програми

Після завершення основного етапу розробки було проведено тестування інтерактивної програмної системи взаємодії користувача з ігровим штучним інтелектом. Метою тестування була перевірка працездатності програмного забезпечення, стабільності роботи алгоритмів штучного інтелекту, коректності взаємодії між компонентами системи та оцінка загального користувацького досвіду.

Тестування проводилося у ручному режимі за участю декількох десятків користувачів, які взаємодіяли з ігровим середовищем у різних сценаріях проходження. Учасники тестування перевіряли:

- стабільність роботи гри;
- коректність функціонування NPC;
- реакцію ігрового штучного інтелекту на дії гравця;
- плавність ігрового процесу;
- зручність користувацького інтерфейсу;
- загальний рівень складності гри.

Особлива увага приділялася перевірці адаптивної поведінки NPC. Під час тестування було встановлено, що ігрові агенти здатні реагувати на дії користувача в реальному часі, змінювати власну тактику поведінки та підтримувати активну взаємодію з гравцем. Алгоритми машинного навчання забезпечували більш динамічний і непередбачуваний ігровий процес порівняно з традиційними скриптовими рішеннями.

У результаті тестування було підтверджено працездатність системи та коректність реалізації основних функціональних модулів. Більшість користувачів позитивно оцінили загальну атмосферу гри, поведінку супротивників та рівень інтерактивності. Навіть у випадках, коли користувачі не могли повністю пройти гру через складність окремих етапів, вони відзначали зацікавленість ігровим процесом та бажання продовжувати взаємодію з грою.

Також тестування дозволило виявити окремі аспекти, що потребують подальшої оптимізації, зокрема:

- балансування складності;
- покращення поведінки NPC у нестандартних ситуаціях;
- оптимізацію продуктивності окремих сцен;
- вдосконалення системи навчання агентів.

Проведене тестування підтвердило доцільність використання методів машинного навчання та адаптивного штучного інтелекту для створення сучасних інтерактивних ігрових систем. Отримані результати свідчать про ефективність

розробленої програмної системи та можливість її подальшого розвитку й масштабування.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було:

1. Проведено аналіз предметної області ігрового штучного інтелекту, визначено недоліки існуючих підходів та обґрунтовано вибір методів для реалізації адаптивної поведінки неігрових персонажів. Шляхом теоретичного дослідження та порівняння класичних алгоритмів (скінченні автомати FSM, дерева поведінки, GOAP) із сучасними методами глибокого навчання та навчання з підкріпленням. Це дозволило довести обмеженість традиційних детермінованих систем (передбачуваність, складність масштабування) та сформувати теоретико-методологічну базу для застосування алгоритму Proximal Policy Optimization на базі архітектури «актор-критик», що забезпечує здатність агентів самостійно навчатися та адаптуватися до ігрових ситуацій.

2. Спроєктовано та реалізовано багаторівневу архітектуру програмної системи, яка включає шар представлення, шар штучного інтелекту та шар ігрового рушія. З використанням об’єктно-орієнтованого та агентно-орієнтованого підходів, де логіка гри та фізика обробляються в середовищі Unity (на мові C#), а моделі нейронних мереж навчаються та виконують інференс за допомогою фреймворку Unity ML-Agents. Це забезпечило високу модульність, безперебійний обмін даними в реальному часі з мінімальними затримками та дозволило системі ефективно формувати динамічну поведінку NPC безпосередньо під час ігрового процесу.

3. Розроблено функціональний програмний продукт - комп'ютерну гру у жанрі двовимірного екшн-платформера, що виступає середовищем для функціонування створеного адаптивного ШІ. Шляхом програмування контролерів гравця та ворогів, реалізації конвеєра даних для збору телеметрії (координати, стан здоров'я, дії гравця), інтеграції системи для алгоритму RL та розробки інтуїтивно зрозумілого інтерфейсу користувача з використанням Unity UI Toolkit. Це дало змогу перенести теоретичні моделі машинного навчання у практичну площину, створивши працюючий прототип системи, де противники здатні проактивно

змінювати дистанцію, координувати атаки та реагувати на агресивний чи обережний стиль гри користувача.

4. Проведено тестування розробленої системи та оцінено ефективність алгоритмів ігрового штучного інтелекту в умовах реальної взаємодії з гравцями. Шляхом залучення тестової групи користувачів, які взаємодіяли з ігровим середовищем у різних сценаріях, із подальшим аналізом ігрового процесу, стабільності системи та реакції NPC на поведінкові патерни гравця. Отримані результати порівнювалися з традиційними скриптовими FSM-рішеннями за критеріями адаптивності та реіграбельності. Тестування підтвердило працездатність та відмовостійкість програмного забезпечення. Результати довели, що використання Reinforcement Learning перевершує жорстко задані сценарії: система забезпечує більш глибоке, вищу непередбачуваність ворогів та генерує унікальний ігровий досвід під час кожного проходження, значно підвищуючи реіграбельність.

Практична цінність одержаних результатів полягає у створенні готового до використання інструментарію та архітектурного каркаса, який розробники комп’ютерних ігор можуть інтегрувати у власні проєкти на базі рушія Unity.

Вподальшому роботу доцільно спрямувати на впровадження багатоагентного навчання з підкріпленням для складної групової взаємодії NPC.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Colledanchise, Michele; Ögren, Petter: "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees". IEEE Transactions on Robotics. 2017
2. Millington I. AI for Games. 3rd ed. CRC Press, 2019. 1026 p.
3. Sutton R. S., Barto A. G. Reinforcement Learning: An Introduction. 2nd ed. MIT Press, 2018. 552 p.
4. Lapan M. Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, and discrete optimization. 2nd ed. Packt Publishing, 2020. 794 p.
5. Silver D., Hubert T., Schrittwieser J. et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science. 2018. Vol. 362 (6419). P. 1140–1144.
6. Arulkumaran K., Deisenroth M. P., Brundage M., Bharath A. A. Deep Reinforcement Learning: A Brief Survey. IEEE Signal Processing Magazine. 2017. Vol. 34 (6). P. 26–38.
7. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 4th ed. Pearson, 2020. 1168 p.
8. Gemrot J., Burkert O., Bída M. et al. Design Patterns for Agent-Based Simulation in Games. Proceedings of the European Simulation and Modelling Conference. 2015. P. 124–131.
9. Офіційна документація Unity ML-Agents Toolkit. URL: https://github.com/Unity-Technologies/ml-agents/docs
10. Zohaib M. Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review. Advances in Human-Computer Interaction. 2018. URL: https://doi.org/10.1155/2018/5681652
11. Brockman G., Cheung V., Pettersson L. et al. OpenAI Gym. arXiv preprint arXiv:1606.01540. 2016.
12. Yannakakis G. N., Togelius J. Artificial Intelligence and Games. Springer,

2018. 357 p.

13. Young J. NPC AI planning with GOAP. URL: https://excaliburjs.com/blog/goal-oriented-action-planning/

14. Документація NavMeshAgent для Unity 6.3 LTS. 2026. URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/AI.NavMeshAgent.html

15. Taljonick, Ryan "Shadow of Mordor's Nemesis system is amazing - here's how it works. 2014. URL:https://web.archive.org/web/20150908072958/http://www.gamesradar.com/shadow-mordor-nemesis-system-amazing-how-works/

16. Офіційний сайт VizDoom. 2016. URL: https://vizdoom.cs.put.edu.pl/

17. Juliani A., Berges V. P., Teng E. et al. Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627. 2020. URL: https://arxiv.org/abs/1809.02627

18. Schulman J., Wolski F., Dhariwal P., Radford A, Klimov O. Proximal Policy Optimization Algorithms. arXiv:1707.06347v2 [cs.LG], 2016, P 12.

19. McCulloch W.S., Pitts W. A logical calculus of the ideas immanent in nervous activity. Springer Nature. 1943. P. 133

20. Штовба С. Д. Вступ до теорії нейронних мереж : навч. посіб. Вінниця : ВНТУ, 2019. 156 с.

21. Tan M., Le Q. V. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. Proceedings of the 36th International Conference on Machine Learning. 2019. Vol. 97. P. 6105–6114.

22. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016. 800 p.

23. Unity Technologies. Unity Manual: UI Toolkit. URL: https://docs.unity3d.com/Manual/UIElements.html

24. Беднарчук Н.Ю., Турченко І.В. Інтерактивна програмна система взаємодії користувача з ігровим штучним інтелектом. ICSPM 2026: Інтелектуальні обчислювальні системи та управління проєктами: збірник тез доповідей

міжнародної науково-практичної конференції студентів і молодих вчених (Тернопіль, 21-22 травня 2026 р). Тернопіль: ЗУНУ, 2026. С. 61-63

25. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп’яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

Додаток А

Програмнний код неігрового персонажа

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

public class BringerMelee : Agent
{
    [Header("Movement")]
    [SerializeField] private float speed = 5f;
    [SerializeField] private Transform player;

    [Header("Combat")]
    [SerializeField] private Transform attackPrefab;
    [SerializeField] private Transform attackPoint;
    [SerializeField] private float attackRange = 3.5f;
    [SerializeField] private float attackCooldown = 1.5f;
    [SerializeField] private float maxHealth = 100f;

    [Header("Components")]
    [SerializeField] private Rigidbody2D rb;
    [SerializeField] private Animator animator;
    [SerializeField] private Transform hitEffect;

    [Header("Debug")]
    [SerializeField] private bool debugRewards = true;

    private float currentHealth;
    private bool hurt;
    private bool isAttacking;
    private Coroutine attackCoroutine;
    private float lastAttackTime;

    // ============================================================
    // INITIALIZATION & RESET
    // ============================================================

    public override void Initialize()
    {
        if (rb == null) rb = GetComponent<Rigidbody2D>();
        if (animator == null) animator = GetComponent<Animator>();

        TryGetPlayer();

        if (player != null)
        {
            Collider2D playerCol = player.GetComponent<Collider2D>();
            Collider2D myCol = GetComponent<Collider2D>();
            if (playerCol != null && myCol != null) Physics2D.IgnoreCollision(playerCol, myCol);
```

```
    }
}

private bool TryGetPlayer()
{
    if (player == null)
    {
        GameObject p = GameObject.FindWithTag("Player");
        if (p != null) player = p.transform;
    }
    return player != null;
}

public override void OnEpisodeBegin()
{
    StopAllCoroutines();
    if (rb != null)
    {
        rb.linearVelocity = Vector2.zero;
        rb.angularVelocity = 0f;
    }

    hurt = false;
    isAttacking = false;
    currentHealth = maxHealth;
    lastAttackTime = 0f;

    transform.localPosition = new Vector3(Random.Range(-7f, 7f), 1f, 0f);

    if (animator != null)
    {
        animator.SetBool("Hurt", false);
        animator.SetFloat("Speed", 0f);
    }
}

// ============================================================
// OBSERVATIONS (Space Size: 9)
// ============================================================

public override void CollectObservations(VectorSensor sensor)
{
    float distance = 0f;
    Vector2 dir = Vector2.zero;
    Vector2 vel = Vector2.zero;
    float hp = 0f;
    float attacking = 0f;
    float cooldown = 0f;
    float inRange = 0f;

    if (TryGetPlayer() && rb != null)
    {
        distance = Vector2.Distance(transform.position, player.position);
        dir = (player.position - transform.position).normalized;
```

```
        vel = rb.linearVelocity;
        hp = currentHealth / maxHealth;
        attacking = isAttacking ? 1f : 0f;
        cooldown = Mathf.Clamp01((Time.time - lastAttackTime) / attackCooldown);
        inRange = distance <= attackRange ? 1f : 0f;
    }

    sensor.AddObservation(distance / 10f);
    sensor.AddObservation(dir.x);
    sensor.AddObservation(dir.y);
    sensor.AddObservation(vel.x / speed);
    sensor.AddObservation(vel.y / speed);
    sensor.AddObservation(hp);
    sensor.AddObservation(attacking);
    sensor.AddObservation(cooldown);
    sensor.AddObservation(inRange);
}

// ============================================================
// ACTIONS & REWARDS
// ============================================================

public override void OnActionReceived(ActionBuffers actions)
{
    // 1. Reduced passive time penalty
    AddReward(-0.0004f);

    if (!TryGetPlayer() || rb == null || hurt) return;

    float distanceToPlayer = Vector2.Distance(transform.position, player.position);
    bool inAttackRange = distanceToPlayer <= attackRange;

    int moveAction = actions.DiscreteActions[0];
    int attackAction = actions.DiscreteActions[1];

    // 2. Proactive Distance Penalty (The further away, the more it "stings")
    float distPenalty = Mathf.Clamp01(distanceToPlayer / 15f) * -0.001f;
    AddReward(distPenalty);

    // --- MOVEMENT ---
    if (!isAttacking)
    {
        Vector2 dirToPlayer = (player.position - transform.position).normalized;
        float moveDir = 0f;

        if (moveAction == 2) // Toward
        {
            moveDir = dirToPlayer.x;
            AddReward(0.001f); // Bonus for closing the gap
        }
        else if (moveAction == 1) // Away
        {
            moveDir = -dirToPlayer.x;
            AddReward(-0.002f); // Penalty for retreating
```

```
        }

        rb.linearVelocity = new Vector2(moveDir * speed, rb.linearVelocity.y);
        if (animator != null) animator.SetFloat("Speed", Mathf.Abs(rb.linearVelocity.x));
        FlipSprite();
    }

    // --- ATTACK ---
    if (inAttackRange) GiveReward(0.01f, "InAttackRange");

    bool canAttack = Time.time - lastAttackTime >= attackCooldown;
    if (attackAction == 1 && !isAttacking && canAttack)
    {
        lastAttackTime = Time.time;
        attackCoroutine = StartCoroutine(AttackRoutine());

        GiveReward(-0.005f, "AttackEffort");

        if (!inAttackRange) GiveReward(-0.05f, "WhiffPenalty");
    }
}

private IEnumerator AttackRoutine()
{
    isAttacking = true;
    if (rb != null) rb.linearVelocity = new Vector2(0f, rb.linearVelocity.y);
    if (animator != null) animator.SetTrigger("Attack");
    yield return new WaitForSeconds(0.9f);
    isAttacking = false;
    attackCoroutine = null;
}

public void CreateAttackHitbox()
{
    if (attackPrefab != null && attackPoint != null && isAttacking)
    {
        Instantiate(attackPrefab, attackPoint.position, attackPoint.rotation);
    }
}

// ========================================================
// DAMAGE & EXTERNAL REWARDS
// ========================================================

public void TakeDamage(float dmg)
{
    currentHealth -= dmg;
    hurt = true;
    isAttacking = false;
    if (attackCoroutine != null) { StopCoroutine(attackCoroutine); attackCoroutine = null; }

    GiveReward(-0.1f, "DamageTaken");

    if (currentHealth <= 0f)
```

```
    {
        GiveReward(-1.0f, "Defeat");
        EndEpisode();
    }
    else
    {
        StartCoroutine(StunRoutine(0.4f));
    }

    if (hitEffect != null) Instantiate(hitEffect, transform.position, transform.rotation);
}

private IEnumerator StunRoutine(float t)
{
    if (animator != null) animator.SetBool("Hurt", true);
    if (rb != null) rb.linearVelocity = Vector2.zero;
    yield return new WaitForSeconds(t);
    if (animator != null) animator.SetBool("Hurt", false);
    hurt = false;
}

public void RewardSuccessfulHit() => GiveReward(1.0f, "HitSuccess");

public void RewardKill()
{
    GiveReward(5.0f, "KillReward");
    EndEpisode();
}

private void GiveReward(float reward, string reason)
{
    AddReward(reward);
    if (debugRewards) Debug.Log($"[{reason}] {reward:F4} | Total: {GetCumulativeReward():F2}");
}

private void FlipSprite()
{
    if (player == null) return;
    float dx = player.position.x - transform.position.x;
    if (Mathf.Abs(dx) > 0.1f)
    {
        Vector3 s = transform.localScale;
        s.x = Mathf.Abs(s.x) * (dx > 0 ? -1f : 1f);
        transform.localScale = s;
    }
}
```

```
    public override void Heuristic(in ActionBuffers actionsOut)
    {
        var d = actionsOut.DiscreteActions;
        d[0] = Input.GetKey(KeyCode.D) ? 2 : Input.GetKey(KeyCode.A) ? 1 : 0;
        d[1] = Input.GetKey(KeyCode.Space) ? 1 : 0;
    }
}
```

Додаток Б

Програмнний код керування персонажем гравця

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

public class Character : MonoBehaviour
{
    public float phealth = 100f;
    public float pmana = 100f;
    public int charge = 0;
    public GameObject SlashEffect;
    public GameObject RangedSlashEffect;
    public GameObject DashSlashEffect;
    public GameObject SlashHitEffect;
    public GameObject PotionEffect;
    public GameObject ChargeEffect;
    public Transform AttackPoint;
    public Transform EffectPoint;
    public Slider health;
    public Slider mana;
    public int ManaPotionAmount { get; private set; }
    public Animator Animator;
    public bool isAttacking = false;
    public bool IsRangeAttacking = false;
    public static Character instance;
    public bool IsCharging = false;
    [SerializeField] public Transform hitpoint;

    public Rigidbody2D rb;
    public Transform groundCheck;
    public LayerMask groundLayer;

    private float horizontal;
    public float speed = 14f;
    public float jumpingPower = 8f;
    private bool isFacingRight = true;
    public Transform HitEffect;

    private bool canDash = true;
    public bool isDashing;
    public float dashPower = 4f;
    private float dashTime = 0.4f;
    private float dashCooldown = 1.5f;
    public TrailRenderer tr;
    public BoxCollider2D m_collider;
    public bool IsCrouching = false;
```

```
float m_ScaleX, m_ScaleY;
float m_OffsetX, m_OffsetY;
private bool hurt = false;

private void Awake()
{
   if (instance == null)
   {
      instance = this;
   }
   else
   {
      Debug.LogWarning("There are multiple instances of the Character class!");
      Destroy(this);
   }
}

private void Update()
{
   if (pmana > 100f)
   {
      pmana = 100f;
   }
   if (pmana < 0f)
   {
      pmana = 0f;
   }
   Animator.SetFloat("Speed", Mathf.Abs(rb.linearVelocity.x));

   if (isDashing)
   {
      return;
   }

   if (!isFacingRight && horizontal > 0f)
   {
      Flip();
   }
   else if (isFacingRight && horizontal < 0f)
   {
      Flip();
   }
   if (IsGrounded())
   {
      Animator.SetTrigger("IsGrounded");
      Animator.SetBool("Airborne", false);
   }
   if (!IsGrounded())
   {
      Animator.ResetTrigger("IsGrounded");
      Animator.SetBool("Airborne", true);
   }
   if (phealth <= 0 && Input.anyKey)
   {
```

```
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }
    }

    public void GainCharge()
    {
        charge = Mathf.Clamp(charge + 1, 0, 3);
        Instantiate(ChargeEffect, EffectPoint.position, EffectPoint.rotation);
        Animator.SetInteger("Charge", charge);
        Debug.Log(charge);
    }

    private void RangedSlashAttack()
    {

        StartCoroutine(shoot());
    }

    public void RangedAttack(InputAction.CallbackContext context)
    {
        if (context.performed && pmana >= 25f && IsGrounded())
        {
            charge = 0; // Reset charge at the start
            Animator.SetInteger("Charge", 0);
            speed = 0f;
            Animator.SetBool("Charging", true);
            IsCharging = true;
        }

        if (context.canceled)
        {
            IsCharging = false;
            Animator.SetBool("Charging", false);

            if (charge > 0)
            {
                // Fire immediately on release
                Animator.SetTrigger("IsRangeAttacking");
                pmana -= 25f;
                SetMana();
            }
            else
            {
                charge = 0;
                Animator.SetInteger("Charge", 0);
            }
            speed = 14f;
        }
    }

    public void SlashAttack()
    {
        Instantiate(SlashEffect, AttackPoint.position, AttackPoint.rotation);
    }
```

```
public void DashAttack()
{
   if (isDashing)
   {
      Instantiate(DashSlashEffect, AttackPoint.position, AttackPoint.rotation);
   }

}

public void Attack(InputAction.CallbackContext context)
{
   if (context.performed && IsGrounded() && !isAttacking)
   {
      Animator.SetBool("IsAttacking", true);
      isAttacking = true;
   }
}

private void SetHealth()
{
   health.value = phealth;
}
private void SetMana()
{
   mana.value = pmana;
}

public UnityEvent<Character> OnPotionCollection;

public void PotionCounter()
{
   ManaPotionAmount++;
   OnPotionCollection.Invoke(this);
}
public void PotionUse(InputAction.CallbackContext context)
{
   if (context.performed && ManaPotionAmount > 0 && pmana < 100)
   {
      ManaPotionAmount--;
      OnPotionCollection.Invoke(this);
      pmana = pmana + 25;
      Instantiate(PotionEffect, EffectPoint.position, EffectPoint.rotation);
      SetMana();
   }
}

private IEnumerator wait()
{
   speed = 0f;
   yield return new WaitForSeconds(0.5f);
   speed = 16f;
}
```

```
void Start()
{
   m_ScaleX = 0.2230458f;
   m_ScaleY = 0.3334816f;
   m_OffsetX = -0.07471263f;
   m_OffsetY = -0.05308678f;
}

private void FixedUpdate()
{
   if (isDashing) return;

   rb.linearVelocity = new Vector2(horizontal * speed, rb.linearVelocity.y);

   if (rb.linearVelocity.y < 0)
      {
      rb.AddForce(Vector2.down * 40f, ForceMode2D.Force);
      }
}

public void Dashing(InputAction.CallbackContext context)
{
   if (context.performed && canDash && speed * horizontal != 0f && IsCrouching == false)
   {
      StartCoroutine(Dash());
      m_collider.enabled = true;
   }

}
public void Jump(InputAction.CallbackContext context)
{
   if (context.performed && IsGrounded())
   {
      StartCoroutine(Jump());

   }
}

public bool IsGrounded()
{
   return Physics2D.OverlapCircle(groundCheck.position, 0.2f, groundLayer);
}

private void Flip()
{
   isFacingRight = !isFacingRight;
   transform.Rotate(0f, 180f, 0f);
}

public void Move(InputAction.CallbackContext context)
{
   horizontal = context.ReadValue<Vector2>().x;
}
```

```
public void Crouch(InputAction.CallbackContext context)
{
   if (context.performed && IsGrounded())
   {
      m_collider.size = new Vector2(0.2230458f, 0.2508897f);
      m_collider.offset = new Vector2(-0.07471263f, -0.09231792f);
      Animator.SetBool("Crouch", true);
      speed = 0f;
      IsCrouching = true;
   }
   if (context.canceled)
   {
      m_collider.size = new Vector2(m_ScaleX, m_ScaleY);
      m_collider.offset = new Vector2(m_OffsetX, m_OffsetY);
      Animator.SetBool("Crouch", false);
      speed = 16f;
      IsCrouching = false;
   }
}

public void Slide(InputAction.CallbackContext context)
{
   if (context.performed && IsCrouching == true && canDash)
   {
      Animator.SetTrigger("Slide");
      StartCoroutine(Sliding());
      m_collider.enabled = true;
   }

}
public void TakeDamage(float damage)
{
   phealth -= damage;
   SetHealth();
   StartCoroutine(Hurt());
   if (phealth <= 0)
   {
      Animator.SetTrigger("Dead");
      rb.bodyType = RigidbodyType2D.Static;
      m_collider.enabled = false;
      horizontal = 0f;
      // Tell the manager we died before we reload the scene
      if(DifficultyManager.Instance != null) {
         DifficultyManager.Instance.RecordDeath();
      }
   }
   Instantiate(HitEffect, transform.position, transform.rotation);
   Screenshake.Instance.ShakeCamera(2f, .1f);
}

private IEnumerator shoot()
{
   for (int i = 0; i < charge; i++)
   {
```

```
            Instantiate(RangedSlashEffect, AttackPoint.position, AttackPoint.rotation);
            yield return new WaitForSeconds(.25f);
        }
        charge = 0;
    }

    private IEnumerator Hurt()
    {
        hurt = true;
        Animator.SetBool("Hurt", hurt);
        speed = 0f;
        yield return new WaitForSeconds(0.4f);
        speed = 16f;
        hurt = false;
        Animator.SetBool("Hurt", hurt);
    }

    private IEnumerator Jump()
    {
        rb.linearVelocity = new Vector2(rb.linearVelocity.x, jumpingPower);
        float originalGravity = rb.gravityScale;
        Animator.SetBool("Airborne", true);
        yield return new WaitForSeconds(0.3f);
        rb.gravityScale = 3f;
        if (IsGrounded())
        {
            Animator.SetBool("Airborne", false);
            rb.gravityScale = originalGravity;
        }
    }
    private IEnumerator Sliding()
    {
        canDash = false;
        isDashing = true;
        m_collider.enabled = false;
        float originalGravity = rb.gravityScale;
        rb.gravityScale = 0f;
        rb.linearVelocity = new Vector2(horizontal * 10f * dashPower, 0f);
        yield return new WaitForSeconds(dashTime);
        rb.gravityScale = originalGravity;
        isDashing = false;
        yield return new WaitForSeconds(dashCooldown);
        canDash = true;
    }

    private IEnumerator Dash()
    {
        Animator.SetTrigger("isDashing");

        canDash = false;
        isDashing = true;
        m_collider.enabled = false;

        float originalGravity = rb.gravityScale;
```

```
        rb.gravityScale = 0f;

        float dashDirection = isFacingRight ? 1f : -1f;

        float dashSpeed = 40f;   // ← adjust this
        float dashDuration = 0.6f; // ← adjust this

        rb.linearVelocity = new Vector2(dashDirection * dashSpeed, 0f);

        yield return new WaitForSeconds(dashDuration);
        Animator.ResetTrigger("isDashing");

        rb.linearVelocity = Vector2.zero;
        rb.gravityScale = originalGravity;

        isDashing = false;

        yield return new WaitForSeconds(dashCooldown);
        canDash = true;
    }

}
```

Додаток В

Копії опублікованих результатів

Міністерство освіти і науки України
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

Матеріали

І Міжнародної науково-практичної конференції студентів і молодих вчених

«ICSPM 2026: Intelligent Computing Systems and Project Management /

Інтелектуальні обчислювальні системи та управління проєктами»

21–22 травня 2026 р.

м. Тернопіль - 2026

УДК 004.5:004.8

ІНТЕРАКТИВНА ПРОГРАМНА СИСТЕМА ВЗАЄМОДІЇ КОРИСТУВАЧА З ІГРОВИМ ШТУЧНИМ ІНТЕЛЕКТОМ

Беднарчук Назар Юрійович,
здобувач першого (бакалаврського) рівня вищої освіти,
medikass2003@gmail.com

Турченко Ірина Василівна,
к.т.н., доцент, доцент кафедри інформаційно-обчислювальних систем і управління,
itu@wunu.edu.ua
ORCID: 0000-0002-9441-6669
Західноукраїнський національний університет

Ігрові середовища виступають зручною платформою для дослідження взаємодії між людиною та штучним інтелектом, оскільки поєднують формалізовані правила, значний обсяг поведінкових даних і можливість оперативного експериментування з різними моделями взаємодії, зокрема суперництва, співпраці, наставництва, змішаної ініціативи та адаптивної взаємодії. Сукупність вимог до методів такої взаємодії, серед яких адаптивність, прозорість, збалансованість ролей, ефективність, масштабованість і орієнтація на користувацький досвід, формує практичну основу для проєктування інтелектуальних ігрових систем у вебсередовищах та багатожанрових застосуваннях, сприяючи підвищенню рівня залученості користувачів, якості взаємодії та етичної відповідальності [1].

Зростання складності сучасних комп'ютерних ігор та збільшення обсягів інтерактивного контенту призводить до підвищення вимог до якості взаємодії користувача з ігровим середовищем. Традиційні підходи до реалізації ігрового штучного інтелекту [2] часто базуються на заздалегідь визначених сценаріях, що призводить до передбачуваності поведінки неігрових персонажів та зниження рівня залученості користувача. У зв'язку з цим особливої актуальності набувають інтерактивні програмні системи, здатні адаптуватися до індивідуального стилю гравця та забезпечувати динамічну взаємодію в реальному часі [3].

Метою цього дослідження є розробка інтерактивної програмної системи, яка забезпечує інтелектуальну обробку дій користувача та формування адаптивної поведінки ігрового штучного інтелекту на основі аналізу стану ігрового середовища та використання сучасних алгоритмів прийняття рішень.

Розроблена система базується на модульному підході (рисунок 1), що забезпечує гнучкість та легкість розширення функціоналу. Основу системи складають кілька логічно пов'язаних рівнів:

– рівень даних забезпечує збереження інформації про попередні взаємодії користувача, параметри ігрового середовища та внутрішні стани агентів, що дозволяє реалізувати механізми персоналізації та уникати повторюваних сценаріїв поведінки;

– рівень обробки дій користувача виконує аналіз вхідних сигналів, таких як переміщення, вибір об'єктів або інші дії, та перетворює їх у формалізовані параметри, що використовуються для прийняття рішень ігровими агентами;

– рівень штучного інтелекту реалізує алгоритми прийняття рішень, які можуть включати дерева поведінки, скінченні автомати або методи навчання з підкріпленням. Цей рівень відповідає за формування адаптивної поведінки NPC на основі поточного стану гри та історії взаємодії.

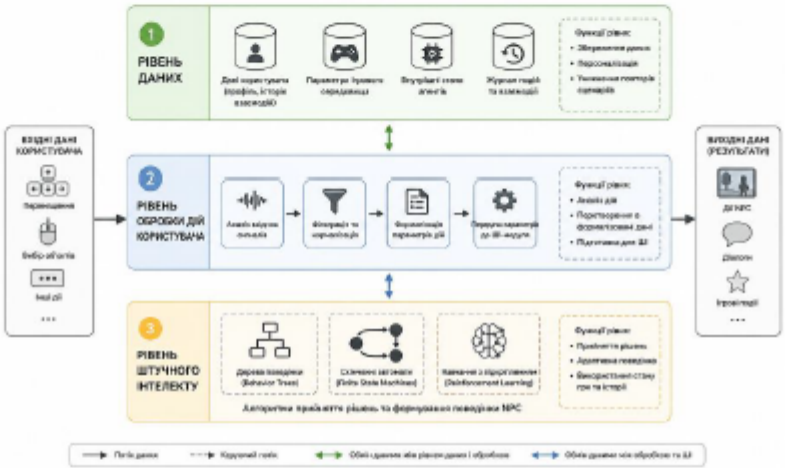

Рисунок 1 – Архітектура інтерактивної системи взаємодії користувача з ігровим штучним інтелектом

Завдання полягає у перетворенні дій користувача та станів ігрового середовища у формалізовані вхідні дані для системи штучного інтелекту. У розробленій системі реалізовано декілька підходів до обробки взаємодії:

– перший підхід полягає в адаптації поведінки агентів до стилю гри користувача. Система аналізує послідовності дій гравця та визначає характерні патерни поведінки, що дозволяє ігровому штучному інтелекту змінювати тактику у відповідь на дії користувача [4];

– другий підхід полягає у зчитуванні результатів гравця під час гри, наскільки той агрессивний із противниками, як довго проходить рівень, скільки раз він помирає між початком та перемогою, що у відповідь заставляє світ адаптуватися, роблячи неігрових персонажів сильнішими та більш згуртованими [5];

– третій підхід базується на використанні історії взаємодії, що дає змогу уникати повторюваних дій і забезпечує більш персоналізований ігровий досвід. Збереження даних про попередні стани та рішення дозволяє системі поступово вдосконалювати свою поведінку[6-7].

Програмна реалізація системи інтегрована з ігровим рушієм та підтримує обробку дій користувача в реальному часі. Використання асинхронних механізмів обробки дозволяє системі ефективно реагувати на зміни ігрового середовища без затримок, що є важливим для забезпечення плавного ігрового процесу. У таблиці 1 представлено характеристики компонентів інтерактивної програмної системи.

Таблиця 1 – Характеристики компонентів програмної системи

Компонент	Технологія	Функція
Core Language	C#	Основна мова ігрового рушія
AI Framework	Unity Machine Learning Agents Toolkit (ML-Agents	Набір інструментів для навчання ШІ у середовищі Unity[8]

Практичне значення полягає у створенні програмної системи, яка може бути використана для розробки ігор з високим рівнем інтерактивності та адаптивності. Модульна архітектура дозволяє інтегрувати додаткові алгоритми штучного інтелекту або розширювати функціональність без значних змін у структурі системи.

Висновки. Розроблена інтерактивна програмна система взаємодії користувача з ігровим штучним інтелектом забезпечує адаптивну обробку дій користувача та формування динамічної поведінки ігрових агентів. Використання модульної архітектури та сучасних підходів до реалізації штучного інтелекту дозволило створити ефективну систему, що підвищує рівень залученості користувача та покращує якість ігрового процесу. Подальші дослідження можуть бути спрямовані на використання методів машинного навчання для більш глибокого аналізу поведінки користувача та підвищення рівня адаптивності системи.

Список використаних джерел

1. Божигора Ю., Турченко І. Методи та моделі взаємодії користувача з ігровими агентами штучного інтелекту. Наукові інновації: теоретичні підходи та практич-ний вплив: матеріали наук.-практ. конф. (8-10.12.2025 р.. Неаполь. Італія). Неаполь. 2025. С. 160-163. URL: https://www.eoss-conf.com/wp-content/uploads/2025/12/Naples_Italy_08.12.25.pdf

2. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. Pearson, 2020. 1166 p.

3. Millington I., Funge J. Artificial Intelligence for Games. 2nd ed. CRC Press, 2009. 895 p.

4. Yannakakis G. N., Togelius J. Artificial Intelligence and Games. Springer, 2018. 359 p.

5. Fisher N. Dynamic Difficulty Adjustment in Games: Concepts, Techniques, and Applications. *IntechOpen*. 2026. URL: https://www.intechopen.com.

6. Freeman E., Robson E. Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software. O'Reilly Media, 2020. 694 p.

7. Zhang S., Yao L., Sun A., Tay Y. Deep Learning based Recommender System: A Survey and New Perspectives. *ACM Computing Surveys*. 2019. Vol. 52, no. 1. Art. 5. URL: https://doi.org/10.1145/3285029

8. ML-Agents Toolkit Documentation, 2024. URL: https://unity-technologies.github.io/ml-agents/ML-Agents-Toolkit-Documentation/