МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ПОЛІСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ, ОБЛІКУ ТА ФІНАНСІВ

Кафедра комп'ютерних технологій і моделювання систем

Кваліфікаційна робота
на правах рукопису

Островський Владислав Вікторович
(прізвище, ім’я, по батькові здобувача освіти)

УДК 004.85:794

КВАЛІФІКАЦІЙНА РОБОТА

Адаптивна система навчання поведінки NPC на основі Q-learning

(тема роботи)

122 Комп’ютерні науки

(шифр і назва спеціальності)

Подається на здобуття освітнього ступеня магістр

кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____

(підпис, ініціали та прізвище здобувача вищої освіти)

Керівник роботи
Ковальчук М. О.
(прізвище, ім’я, по батькові)
кандидат педагогічних наук,
доцент
(науковий ступінь, вчене звання)

Житомир – 2025

Висновок кафедри комп'ютерних технологій і моделювання систем:
за результатами попереднього захисту: _____

Протокол засідання кафедри комп'ютерних технологій і моделювання систем
№ ____ від «_____» _____ 20__ р.

Завідувач кафедри комп'ютерних технологій і моделювання систем
к.п.н., доцент _____ М. О. Ковальчук
(науковий ступінь, вчене звання) (підпис) (прізвище, ім'я, по батькові)
«_____» _____ 20___ р.

Результати захисту кваліфікаційної роботи

Здобувач вищої освіти Островський Владислав Вікторович захистив (ла)
(прізвище ,ім'я, по батькові)
кваліфікаційну роботу з оцінкою:

сума балів за 100-бальною шкалою _____
за шкалою ECTS _____
за національною шкалою _____

Секретар ЕК
лаборант кафедри _____ В. В. Корольчук
(науковий ступінь, вчене звання) (підпис) (прізвище, ім'я, по батькові)

АНОТАЦІЯ

Островський В.В. *Адаптивна система навчання поведінки NPC на основі Q-learning*. – Кваліфікаційна робота на правах рукопису.

Кваліфікаційна робота на здобуття освітнього ступеня магістр за спеціальністю 122 – Комп'ютерні науки. – Поліський національний університет, Житомир, 2025.

Обсяг кваліфікаційної роботи: 39 сторінок (19 – рисунків, 4 – додатки, 50 – джерел).

Ключові слова: навчання з підкріпленням, адаптивна поведінка, марківський процес прийняття рішень, табличний Q-learning, epsilon-greedy стратегія, штучний інтелект у іграх, системи потреб NPC.

Кваліфікаційна робота присвячена дослідженню методів створення адаптивної поведінки неігрових персонажів у відеоіграх із використанням алгоритму Q-learning. Проведено аналіз традиційних підходів та обґрунтовано перевагу методів навчання з підкріпленням. Побудовано математичну модель віртуального середовища як марківського процесу прийняття рішень, розроблено функцію винагороди для управління множинними потребами персонажа, реалізовано табличний Q-learning з epsilon-greedy стратегією.

Модель реалізовано в Unreal Engine 5 на C++. Виконано експериментальне дослідження протягом 252 поколінь, виявлено нелінійний характер збіжності з трьома фазами навчання та феномен перенавчання на пізніх етапах. Проведено порівняльний аналіз альтернативних конфігурацій, що підтвердив критичну важливість коректного підбору параметрів.

Практичне значення полягає у створенні готового рішення з відкритою архітектурою, адаптованого для широкого спектру ігрових механік.

SUMMARY

Ostrovskyi V.V. *Adaptive NPC Behavior Learning System Based on Q-learning*. – Qualification work as a manuscript.

Master's degree qualification work in specialty 122 – Computer Science. – Polissia National University, Zhytomyr, 2025.

The volume of the work: 39 pages (19 – figures, 4 – appendices, 50 – sources).

Key words: reinforcement learning, adaptive behavior, Markov decision process, table Q-learning, epsilon-greedy strategy, artificial intelligence in games, NPC needs systems.

The qualification work is dedicated to researching methods for creating adaptive behavior of non-player characters in video games using the Q-learning algorithm. An analysis of traditional approaches was conducted and the advantage of reinforcement learning methods was substantiated. A mathematical model of the virtual environment as a Markov decision process was built, a reward function for managing multiple character needs was developed, and tabular Q-learning with epsilon-greedy strategy was implemented.

The model was implemented in Unreal Engine 5 using C++. Experimental research was conducted over 252 generations, revealing a non-linear convergence pattern with three learning phases and an overfitting phenomenon in later stages. A comparative analysis of alternative configurations was performed, confirming the critical importance of correct parameter tuning.

ЗМІСТ

ВСТУП

З розвитком індустрії відеоігор значно зростають вимоги до реалістичності та інтелектуальності неігрових персонажів (NPC). Традиційні підходи до моделювання їхньої поведінки, такі як використання скриптів, скінченних автоматів чи поведінкових дерев, часто призводять до передбачуваної та шаблонної поведінки. Це негативно впливає на занурення гравця в ігровий світ та обмежує варіативність ігрового досвіду.

Використання методів машинного навчання, зокрема навчання з підкріпленням, відкриває нові можливості для створення адаптивної поведінки, що самостійно розвивається та вдосконалюється в інтерактивному середовищі. Q-learning, як один із найефективніших безмодельних алгоритмів навчання з підкріпленням, дозволяє агенту вивчати оптимальну стратегію дій на основі винагород, не вимагаючи детального програмування кожної можливої ситуації. Це робить дослідження методів застосування Q-learning для навчання поведінки NPC актуальним з наукової та практичної точок зору.

Об'єкт дослідження: процес автономного формування адаптивної поведінки неігрових персонажів через взаємодію з віртуальним середовищем на основі системи потреб та винагород.

Предмет дослідження: табличний алгоритм Q-learning для управління багатовимірною системою потреб неігрових персонажів у симуляційному середовищі Unreal Engine 5.

Мета роботи: розробка та експериментальне дослідження системи адаптивної поведінки NPC на основі Q-learning у віртуальному середовищі Unreal Engine 5.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Проаналізувати та порівняти існуючі підходи до моделювання поведінки NPC.

2. Вивчити теоретичні основи навчання з підкріпленням та алгоритму Q-learning.
3. Розробити математичну модель віртуального середовища як марківського процесу прийняття рішень та формалізувати алгоритм Q-learning з epsilon-greedy стратегією.
4. Спроектувати та реалізувати симуляційне середовище в Unreal Engine
5. Провести експериментальне дослідження протягом множини поколінь персонажів для оцінки ефективності та динаміки навчання.

У роботі використано методи системного аналізу для вивчення сучасних підходів до моделювання поведінки NPC, математичного моделювання для формалізації марківського процесу прийняття рішень та опису алгоритму Q-learning з epsilon-greedy стратегією, об'єктно-орієнтованого проектування для розробки компонентної архітектури системи на C++, а також методи комп'ютерного експерименту з прискореною симуляцією для валідації ефективності навчання.

Наукова новизна:

1. Розроблено архітектуру адаптивної системи навчання поведінки NPC з управлінням шістьма конкуруючими потребами, що інтегрує алгоритм Q-learning безпосередньо в ігровий рушій Unreal Engine 5 з використанням компонентного підходу на C++.
2. Запропоновано методику побудови функції винагороди для Q-learning, яка враховує як негайне задоволення критичних потреб, так і довгострокове підтримання балансу через бонуси за одночасне утримання множини потреб на високих рівнях.
3. Виявлено та проаналізовано феномен перенавчання в табличному Q-learning для високорозмірних просторів станів з постійною деградацією, включаючи вплив дискретизації та зниження коефіцієнта exploration на стабільність навчання.

Теоретичне значення полягає у систематизації знань про застосування навчання з підкріпленням для моделювання складної адаптивної поведінки NPC у середовищах з множиною конкуруючих цілей та постійною деградацією станів. Результати дослідження можуть бути використані для подальших наукових розробок у галузі штучного інтелекту та ігрової індустрії, зокрема для переходу до Deep Q-Learning та мультиагентного навчання.

Практичне значення полягає у створенні готового програмного рішення з відкритою архітектурою, яке може бути адаптоване для широкого спектру ігрових механік, включаючи управління ресурсами, тактичну бойову поведінку, соціальну взаємодію між агентами, що значно покращує реіграбельність через відсутність жорстко детермінованих реакцій.

РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ МОДЕЛЮВАННЯ ПОВЕДІНКИ НЕІГРОВИХ ПЕРСОНАЖІВ

1.1. Класифікація та огляд традиційних підходів

Моделювання поведінки неігрових персонажів у комп'ютерних іграх є однією з центральних задач у галузі ігрового штучного інтелекту, оскільки якість взаємодії гравця з віртуальним середовищем безпосередньо залежить від реалістичності та переконливості дій автономних агентів [1]. Протягом останніх десятиліть індустрія відеоігор зазнала стрімкого розвитку, що супроводжувалося постійним зростанням очікувань гравців щодо складності та природності NPC та бажанням розробників перевершити самих себе. У зв'язку з цим, питання вибору оптимального методу моделювання поведінки стає все більш актуальним, адже він має забезпечувати баланс між обчислювальною ефективністю, гнучкістю системи та якістю кінцевого ігрового досвіду.

Історично склалося так, що розвиток методів моделювання поведінки NPC відбувався еволюційно, від найпростіших детермінованих підходів до складних систем на основі машинного навчання. Кожен етап цієї еволюції був обумовлений як технологічними можливостями свого часу, так і змінами у вимогах до ігрового контенту. Розуміння переваг та обмежень традиційних підходів є необхідним для обґрунтованого вибору методології при розробці сучасних ігрових систем, а також для виявлення напрямків подальшого вдосконалення існуючих рішень.

Найпростішим та найбільш поширеним у ранніх іграх підходом були скриптові системи поведінки [2]. У його рамках дії персонажа визначаються жорстко запрограмованою послідовністю інструкцій, яка виконується у відповідь на специфічні тригери або події у ігровому світі. Типовим прикладом може слугувати NPC-охоронець, який патрулює певний маршрут за заданою траєкторією, а при виявленні гравця у зоні видимості переходить до стану переслідування з використанням попередньо визначеного

алгоритму руху. Скриптові системи характеризуються низькою обчислювальною складністю та високою передбачуваністю поведінки, що значно полегшує процес тестування та налагодження ігрової логіки. Крім того, такий підхід дозволяє дизайнерам створювати чітко контрольовані ігрові ситуації, де поведінка акторів служить інструментом для досягнення певних наративних або геймплейних цілей.

Наступним етапом розвитку моделювання поведінки акторів стало впровадження автоматів скінченних станів (Finite State Machines, FSM) [3]. Цей підхід представляє поведінку персонажа як набір дискретних станів, між якими відбуваються переходи на основі визначених умов або подій. Кожен стан характеризується специфічною поведінкою агента, а переходи між станами визначаються логічними умовами, які перевіряють стан ігрового середовища чи параметри в самому персонажі. Наприклад, NPC-ворог може перебувати у станах "Патрулювання", "Переслідування", "Атака" та "Відступ", причому перехід між цими станами визначається такими факторами як відстань до гравця, рівень здоров'я персонажа або наявність укриття поблизу.

Практичне застосування скінченних автоматів у ігровій індустрії широко поширене завдяки їхній простоті імплементації та ефективності виконання. Багато сучасних ігрових рушіїв надають вбудовані інструменти для візуального проектування FSM через діаграми станів, що робить цей підхід доступним навіть для непрограмістів у команді розробки.

Однак, із зростанням складності поведінки персонажів, традиційні FSM стикаються з проблемою експоненційного зростання кількості станів та переходів. Для NPC з множиною незалежних аспектів поведінки (наприклад, одночасний контроль бойових дій, емоційного стану та соціальної взаємодії) кількість можливих комбінацій станів може досягати неприйнятних значень, через що стає майже неможливо нормально

прослідкувати за працездатністю коду. Представити FSM для простого ворожого NPC можна як зображено на рисунку А.1 (див. додаток А).

Для вирішення цієї проблеми було запропоновано підхід на основі поведінкових дерев (Behavior Trees, BT) [4]. Поведінкові дерева представляють собою ієрархічну структуру, що організує логіку прийняття рішень у вигляді дерева, де внутрішні вузли визначають порядок виконання дочірніх вузлів, а листові вузли представляють конкретні дії або умовні перевірки. Цей підхід вперше здобув широке визнання після його успішного застосування у грі Halo 2 студією Bungie у 2004 році, і відтоді став де-факто стандартом для моделювання складної поведінки NPC у AAA-іграх [5][6].

Структурно поведінкове дерево складається з трьох основних типів вузлів. Композитні вузли контролюють потік виконання та включають селектори, що виконують дочірні вузли послідовно до першої невдачі, та послідовності, що виконують дочірні вузли до першого успіху. Така модульна архітектура забезпечує високий ступінь повторного використання коду та полегшує створення складної поведінки через композицію простих елементів. Приклад поведінкового дерева для NPC зображено на рисунку А.2 (див. додаток А).

Однією з ключових переваг поведінкових дерев є їхня реактивність – дерево переобчислюється на кожному кроці виконання, що дозволяє агенту швидко реагувати на зміни у середовищі без необхідності явного визначення всіх можливих переходів між станами. Це робить BT значно більш адаптивними порівняно з традиційними FSM, особливо у динамічних ігрових ситуаціях, де умови можуть змінюватися непередбачувано. Крім того, візуальна природа дерев робить їх інтуїтивно зрозумілими для дизайнерів ігор, а сучасні інструменти розробки надають графічні інтерфейси для створення та налагодження поведінки без необхідності написання коду [7].

Незважаючи на численні переваги, поведінкові дерева не позбавлені обмежень. Складні дерева можуть стати громіздкими та важкими для розуміння, особливо коли глибина вкладеності досягає значних рівнів. ВТ є суто реактивними системами і не мають вбудованих механізмів для довгострокового планування або передбачення майбутніх станів середовища. Це означає, що персонажі, керовані поведінковими деревами, приймають рішення на основі поточної ситуації без урахування можливих наслідків своїх дій у довгостроковій перспективі. В додаток, створення дійсно інтелектуальної поведінки вимагає від розробників ретельного проектування структури дерева та явного визначення всіх можливих сценаріїв, що може бути трудомістким процесом для складних ігрових систем.

1.2. Сучасні підходи на основі штучного інтелекту.

Стрімкий розвиток технологій машинного навчання та штучного інтелекту за останні два десятиліття відкрив нові можливості для створення складної поведінки неігрових персонажів, що виходять за межі традиційних детермінованих підходів. Сучасні методи на основі ШІ дозволяють створювати агентів, здатних до автономного прийняття рішень, адаптації до змінних умов середовища та навчання на основі власного досвіду. Серед різноманіття існуючих підходів особливу увагу заслуговують системи планування дій, що орієнтуються на конкретні дії (Goal-Oriented Action Planning, GOAP) та методи навчання з підкріпленням (Reinforcement Learning), кожен з яких пропонує унікальні можливості для розв'язання специфічних задач ігрового штучного інтелекту.

Концепція GOAP була вперше представлена Джеффом Ормкіном у контексті розробки штучного інтелекту для гри F.E.A.R. студією Monolith Productions у 2005 році [8][9]. Фундаментальна ідея GOAP полягає у тому, що замість явного визначення всіх можливих послідовностей дій,

розробник специфікує набір дій, а також цільовий стан, якого агент намагається досягти. Система планування автоматично будує послідовність дій, що трансформує поточний стан світу у цільовий, використовуючи алгоритми пошуку у просторі станів. Такий підхід забезпечує значно більшу гучність порівняно з жорстко закодованою логікою, оскільки один і той самий набір дій може бути скомбінований різними способами залежно від конкретної ситуації.

Формально, у системі GOAP кожна дія визначається множиною передумов, що мають бути виконані для можливості застосування дії, множиною ефектів, що описують зміни стану світу після виконання дії, та вартістю виконання певної дії. Світовий стан представляється набором пар ключ-значення, що описують релевантні властивості середовища та актора. Процес планування зводиться до пошуку оптимального шляху у графі станів від поточного стану до цільового стану, де ребра графу відповідають можливим діям. Для цього зазвичай використовується алгоритм A* з евристичною функцією, що оцінює відстань від поточного стану до цільового. Однією з важливих переваг GOAP є можливість динамічного перепланування у відповідь на зміни у середовищі, що забезпечує високу адаптивність поведінки NPC у динамічних ігрових ситуаціях. Приклад процесу планування можна побачити на рис. А.3 (див. додаток А):

Альтернативним та концептуально відмінним підходом до створення інтелектуальної поведінки NPC є застосування методів навчання з підкріпленням [10]. На відміну від GOAP, де логіка поведінки явно програмується через визначення дій та їхніх властивостей, RL дозволяє агентам самостійно виявляти оптимальні стратегії поведінки через взаємодію з середовищем та отримання винагород або покарань за свої дії. Цей підхід базується на парадигмі проб та помилок, де агент поступово покращує свою поведінку на основі накопиченого досвіду, без необхідності явного програмування правил прийняття рішень [11].

Фундаментальна відмінність між цими підходами полягає у джерелі знань про оптимальну поведінку. У випадку Goal Oriented, експертні знання розробників визначають множину можливих дій, їхні передумови та ефекти, а алгоритм планування лише комбінує ці заздалегідь визначені елементи для досягнення заданої цілі. Натомість, у навчанні з підкріпленням агент самостійно відкриває ефективні стратегії поведінки через багаторазову взаємодію з середовищем, що потенційно дозволяє виявити неочевидні або несподівані для розробників рішення. Ця властивість робить Reinforced Learning особливо привабливими для моделювання складної адаптивної поведінки у динамічних середовищах, де повне передбачення всіх можливих ситуацій є практично неможливим.

1.3. Принципи навчання з підкріпленням та особливості Q-learning.

Навчання з підкріпленням орієнтується на навчання через взаємодію агента з середовищем та отримання зворотного зв'язку у формі винагород. Ця парадигма природним чином відображає процес навчання живих організмів, що робить її привабливою для моделювання поведінки NPC у відеоіграх. Розуміння математичних основ навчання з підкріпленням, зокрема Марківських процесів прийняття рішень та алгоритму Q-learning, є критично важливим для обґрунтованого вибору методології та успішної симуляції інтелектуальної поведінки неігрових персонажів [12].

У контексті ігрового середовища стан $s \in S$ може включати інформацію про положення NPC, рівні його внутрішніх потреб, доступність об'єктів взаємодії та інші релевантні характеристики світу. Дія $a \in A$ представляє конкретне рішення агента, таке як використання певного об'єкта або переміщення до заданої локації. Функція переходу P описує динаміку середовища, визначаючи ймовірність досягнення певного стану s'

після виконання дії *a* у стані *s*. Функція винагороди *R* забезпечує зворотний зв'язок, що направляє процес навчання актора до бажаної поведінки.

Практична реалізація Q-learning використовує табличне представлення Q-функції, де рядки відповідають станам, стовпці - діям, а комірки містять Q-значення. Q-таблиця накопичує знання агента про середовище, зберігаючи для кожної пари (стан, дія) оцінку очікуваної винагороди. На початку навчання таблиця ініціалізується нульовими значеннями, а у процесі взаємодії Q-значення поступово оновлюються, збігаючись до оптимальних значень.

Алгоритм Q-learning, запропонований Крісом Уоткінсом у 1989 році, є одним з найбільш фундаментальних та широко застосовуваних методів навчання з підкріпленням для знаходження оптимальної Q-функції [13]. Ключовою особливістю Q-learning є те, що він є model-free алгоритмом, тобто не вимагає знання функції переходу або функції винагороди середовища. Замість цього, агент навчається безпосередньо через взаємодію з середовищем, спостерігаючи реальні переходи та винагороди. Крім того, Q-learning є off-policy алгоритмом, що означає можливість навчання оптимальної політики навіть при використанні субоптимальної політики для дослідження середовища під час навчання. Основою Q-learning є ітеративне оновлення оцінок Q-значень на основі спостережуваних переходів. Коли агент виконує дію a у стані s, переходить у стан s' та отримує винагороду r, Q-значення оновлюється згідно з правилом:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

де:

Q(s,a) – поточне значення дії *a* у стані *s*

α – коефіцієнт навчання (learning rate), що визначає, наскільки сильно нова інформація впливає на попереднє значення

r – винагорода, отримана за виконання дії

γ – фактор дисконтування майбутніх винагород (discount factor), що визначає, наскільки важливі віддалені результати порівняно з поточними

$max_{a'}Q(s', a')$ – максимальна оцінка серед усіх можливих дій у наступному стані

Практична реалізація Q-learning для задач з дискретними просторами станів та дій зазвичай використовує табличне представлення Q-функції у вигляді двовимірної таблиці, де рядки відповідають станам, стовпці - діям, а комірки містять Q-значення. Роль Q-таблиці у процесі навчання є центральною – вона фактично представляє накопичене знання агента про середовище, зберігаючи для кожної пари (стан, дія) оцінку очікуваної максимальної кумулятивної винагороди, яку агент може отримати, виконавши дану дію у даному стані та діючи оптимально надалі. На початку навчання Q-таблиця зазвичай ініціалізується нульовими або невеликими випадковими значеннями, що відображає повну відсутність знань агента про середовище. У процесі взаємодії з середовищем Q-значення поступово оновлюються на основі спостережуваних винагород та оцінок майбутніх станів, поступово збігаючись до оптимальних значень Q*(s,a). Після завершення навчання Q-таблиця стає повноцінною базою знань, що дозволяє агенту миттєво визначати оптимальну дію для будь-якого стану через простий пошук максимального Q-значення, без необхідності додаткових обчислень або планування. Зображення процесу Q-learning з ілюстрацією оновлення Q-таблиці на рисунку А.4 (див. додаток А).

Вибір Q-learning обґрунтовується кількома перевагами: це model-free метод, що не вимагає моделювання динаміки середовища; табличне представлення забезпечує високу інтерпретовність результатів; гарантована збіжність до оптимальної політики при достатній кількості ітерацій; обчислювальна ефективність дозволяє застосування у реальному часі після завершення навчання. Специфіка задачі управління множинними потребами NPC робить Q-learning особливо доцільним: простір станів

природно дискретний (три рівні для кожної потреби), набір дій чітко визначений (використання об'єктів), структура винагород проста (зміна рівня потреб дає прямий сигнал для навчання).

Порівняно з GOAP або поведінковими деревами, Q-learning пропонує фундаментально іншу парадигму - система самостійно виявляє оптимальні стратегії через експериментування замість явного програмування правил. Водночас, табличний Q-learning залишається достатньо простим для розуміння та інтеграції у ігрові системи.

Висновки до першого розділу

Проведено аналіз методів моделювання поведінки неігрових персонажів у комп'ютерних іграх. Розглянуто еволюцію підходів від традиційних скриптових систем та автоматів скінченних станів до сучасних методів на основі штучного інтелекту.

Встановлено, що традиційні методи характеризуються відсутністю адаптивності та вимагають повної ручної специфікації правил поведінки. Сучасні підходи, такі як GOAP, частково вирішують проблему динамічного планування, проте залишаються залежними від експертних знань розробників.

Обґрунтовано доцільність застосування навчання з підкріпленням, зокрема алгоритму Q-Learning, для автономного формування поведінки NPC через взаємодію з середовищем. Показано, що табличне представлення Q-Learning забезпечує гарантовану збіжність до оптимальної політики для задач з дискретними просторами станів та дій.

РОЗДІЛ 2. РОЗРОБКА МОДЕЛІ ТА АЛГОРИТМУ АДАПТИВНОЇ ПОВЕДІНКИ NPC НА ОСНОВІ Q-LEARNING

2.1. Формалізація MDP для віртуального середовища

Розробка нашої системи вимагає чіткої формалізації ігрового середовища як марківського процесу прийняття рішень. Така формалізація дозволяє перетворити складну задачу моделювання поведінки у математично обґрунтовану проблему оптимізації, де актор автономно навчається приймати рішення через взаємодію з віртуальним світом. У контексті розробленої системи неігровий персонаж повинен самостійно навчитися підтримувати баланс шести життєвих потреб шляхом взаємодії з об'єктами навколишнього середовища [14].

Простір станів у розробленій моделі визначається поточними рівнями всіх життєвих потреб персонажа. Система моделює шість фундаментальних потреб: Hunger (голод), Bladder (потреба у туалеті), Energy (енергія), Social (соціальна взаємодія), Hygiene (гігієна) та Fun (розваги); це взято як референс з популярної серії ігор The Sims [15]. Кожна потреба представлена числовим значенням від 0 до 100 відсотків, проте для цілей навчання ці безперервні значення дискретизуються у три якісні рівні. Рівень Low відповідає значенням від 0 до 33 відсотків і сигналізує про критичний стан потреби, Medium охоплює діапазон від 33 до 66 відсотків, а High відповідає значенням понад 66 відсотків і означає задовільний стан потреби. Загальна кількість можливих станів визначається як 729 унікальних комбінацій, де кожен стан представлений шестизначним рядком. Наприклад, стан "222222" представляє ситуацію, коли всі шість потреб перебувають на високому рівні, тоді як стан "000000" відображає зворотню.

Важливою особливістю розробленого середовища є наявність множинних екземплярів об'єктів одного типу з різними характеристиками ефективності. У віртуальному світі розміщено по 3 об’єкта двох типів на кожну потребу:

- Холодильники: 5се +15 / 15сек +40 Hunger
- Душі: 3сек +20 / 10сек +50 Hygiene
- Дивани: 5сек +10 / 12сек +45 Social
- Туалети: 3сек + 25 / 8сек +45 Bladder
- Телевізори: 8сек +20 / 20сек +50 Fun

Energy можна відновити через єдиний тип ліжка, що надає 25 пунктів за 8 секунд взаємодії.

Така варіативність об'єктів створює складне середовище для навчання, де алгоритм Q-Learning повинен автономно виявити оптимальні стратегії вибору між швидкими, але менш ефективними діями та повільними, але більш результативними альтернативами. Наявність об'єктів різної ефективності дозволяє системі навчитися контекстно-залежному прийняттю рішень. Додатковим фактором складності є випадкове розташування всіх об'єктів на ігровій карті, що змушує персонажа враховувати не лише ефективність взаємодії, але й витрати часу на переміщення до об'єкта.

Функція переходу між станами є детермінованою відносно параметрів взаємодії з об'єктами, проте враховує постійну деградацію всіх потреб. Реалізована у компоненті UNeedsComponent механіка деградації зменшує кожну потребу зі швидкістю 0.4 пункти на секунду незалежно від дій персонажа. Це означає, що під час виконання будь-якої дії, наприклад 15-секундної взаємодії з холодильником, цільова потреба Hunger збільшується на 40 пунктів, проте всі інші п'ять потреб одночасно зменшуються на 6 пункти кожна; саме такі параметри зменшення були обрані під час експерименту зі значеннями.

Розглянемо конкретний приклад процесу прийняття рішення для ілюстрації роботи формалізованої моделі. Припустимо, що неігровий персонаж перебуває у стані, де рівень Hunger становить 15 відсотків (рівень Low), а всі інші потреби знаходяться на рівні 75 відсотків (рівень High).

Система Q-Learning, маючи доступ до навченої Q-таблиці, аналізує очікувані винагороди для всіх доступних дій у цьому стані. Якщо для дії UseRefrigerator Q-значення становить значення, що є найвищим серед доступних альтернатив, персонаж обирає саме цю дію.

Після вибору дії персонаж активує систему навігації для побудови шляху до найближчого холодильника. Після досягнення холодильника розпочинається взаємодія, реалізована через компонент InteractableObject. Якщо це швидкий холодильник, взаємодія триває 5 секунд, протягом яких Hunger збільшується з 15 до 30 відсотків, переходячи з рівня Low до Medium. Водночас Energy знижується на 1 пункт, Bladder на 1 пункт, Social на 1 пункт, Hygiene на 1 пункт, та Fun на 1 пункт, проте всі ці потреби залишаються у діапазоні High, оскільки початкові значення були достатньо високими.

Після завершення взаємодії компонент QLearningComponent оновлює поточний стан, виконуючи дискретизацію нових числових значень потреб. Система обчислює винагороду на основі реалізованої у методі CalculateReward, що аналізує зміну всіх потреб. Базова винагорода 20 пунктів нараховується за будь-яку завершену дію. Додатково система обчислює покращення кожної потреби шляхом порівняння значень до та після дії, множачи сумарне покращення на коефіцієнт 8.0. У даному прикладі Hunger збільшився на 15 пунктів, що дає бонус 120 пунктів, тоді як інші потреби зменшилися на 5 пунктів сумарно, що створює невеликий негативний внесок. Оскільки п'ять з шести потреб залишилися на рівні High, система нараховує додатковий бонус 60 пунктів.

Критично важливим аспектом формалізації є визначення термінального стану та відповідного штрафу. Коли будь-яка з шести потреб досягає нульового значення, персонаж вважається "мертвим", і епізод навчання завершується. У такому випадку система нараховує штраф -500 пунктів, що значно перевищує типові винагороди за успішні дії. Цей

механізм стимулює алгоритм Q-Learning уникати станів, що можуть призвести до критичного виснаження будь-якої потреби.

Марківська властивість виконується у розробленій системі, оскільки результат виконання дії повністю визначається поточним станом персонажа та параметрами обраної взаємодії [16]. Історія попередніх дій не впливає на те, наскільки збільшиться Hunger при використанні холодильника або наскільки деградують інші потреби під час взаємодії. Така властивість є фундаментальною для застосування алгоритму Q-Learning, оскільки дозволяє системі навчатися на основі локальних переходів між станами без необхідності зберігання та аналізу повної траєкторії дій.

2.2. Побудова Q-таблиці та функції винагороди

Центральним елементом реалізованої системи навчання є Q-таблиця, що зберігає накопичені знання про ефективність кожної дії у кожному можливому стані середовища (див. додаток В) [17]. Початкова ініціалізація Q-значень відбувається автоматично при першому відвідуванні нової пари стан-дія, де значення встановлюється у 0.0. Критично важливою особливістю реалізації є механізм збереження та завантаження Q-таблиці між різними поколіннями навчання. Методи SaveQTable та LoadQTable забезпечують серіалізацію накопичених знань у JSON [18]. Це дозволяє реалізувати навчання, де кожне нове покоління персонажів успадковує досвід попередніх, продовжуючи удосконалення політики замість навчання з нуля. Компонент NPCSpawnManager координує цей процес через параметр bShareQTable, що за замовчуванням увімкнений для забезпечення спадкоємності знань. При смерті персонажа система автоматично викликає SaveQTable, гарантуючи збереження всіх оновлень Q-значень перед знищенням актора. Нове покоління завантажує збережену таблицю під час ініціалізації у методі BeginPlay.

Для наглядності можемо розібрати перші декілька записів з таблиці (див. додаток Б). Стан "110122" представляє ситуацію, де Hunger та Bladder на середньому рівні, Energy низька, а Social, Hygiene та Fun високі. Для цього стану записано чотири дії з Q-значеннями від 2.5 до 4.13, проте дія UseToilet має різко негативне значення -98.14, що вказує на невдалий досвід, коли задоволення Bladder призвело до критичного виснаження Energy через тривалість взаємодії. Стан "210102" з високим Hunger, середнім Bladder, низькими Energy та Hygiene демонструє контрастні результати: дія UseTelevision має негативне Q-значення -5, тоді як UseRefrigerator отримала позитивну оцінку 104.09, що свідчить про успішне задоволення критичної потреби голоду. Особливо показовим є стан "220012", де дві перші потреби високі, дві середні, Energy низька, а Fun висока. Обидві записані дії мають негативні Q-значення: UseRefrigerator отримала -5, а UseSofa критичні -100 пунктів. Останнє значення вказує на смерть персонажа внаслідок спроби задовольнити Social потребу замість критично низької Energy, що демонструє як система навчається розпізнавати фатальні помилки пріоритезації.

Ці приклади ілюструють ключову особливість Q-learning: Q-значення відображають не миттєву винагороду за дію, а очікувану кумулятивну винагороду з урахуванням усіх наступних станів. Оптимальний стан "222222" з усіма потребами на високому рівні має Q-значення понад 900 пунктів та понад 1000 відвідувань для кожної дії, що підтверджує успішність навченої політики у підтримці стабільного стану.

Реалізована у методі CalculateReward функція використовує багатокомпонентну структуру для оцінки якості прийнятих рішень. Базова винагорода 20 пунктів нараховується за будь-яку успішно завершену взаємодію з об'єктом, що стимулює активну поведінку. Основний компонент винагороди обчислюється як сума змін всіх шести потреб, помножена на коефіцієнт 8.0. Система аналізує різницю між значеннями потреб до та після

виконання дії, зберігаючи попередні значення у словнику PreviousNeedValues. Такий підхід дозволяє точно оцінити чистий ефект дії з урахуванням одночасного покращення цільової потреби та деградації решти.

Важливою характеристикою реалізованої функції є її фокус на відносних змінах стану замість абсолютних значень. Система не нараховує винагороду безпосередньо за високі рівні потреб, а оцінює покращення, що відбулося внаслідок дії. Це створює динамічне середовище, де оптимальна політика передбачає безперервне балансування між потребами.

2.3. Алгоритм Q-learning для навчання NPC

Q-learning для навчання неігрових персонажів ґрунтується на повторюваному процесі взаємодії актора з віртуальним середовищем, де кожна дія призводить до оновлення накопичених знань про ефективність різних стратегій поведінки. Загальний цикл навчання розпочинається зі спавну персонажа через, що ініціалізує нового актора класу ANPCCharacter з унікальним ідентифікатором та номером покоління. При ініціалізації у методі BeginPlay компонент QLearningComponent завантажує існуючу Q-таблицю з файлової системи, забезпечуючи спадкоємність знань від попередніх поколінь. Компонент NeedsComponent ініціалізує всі шість потреб випадковими значеннями у діапазоні 70-90 відсотків, створюючи різноманітні початкові умови для кожного епізоду навчання (див. рис. А.5, додаток А). Основний цикл прийняття рішень реалізовано через таймер з інтервалом два секунди. Цей інтервал забезпечує баланс між частотою оновлень та надлишковими обчисленнями, даючи персонажу достатньо часу для завершення розпочатих дій. На кожній ітерації система перевіряє поточний стан персонажа – якщо він перебуває у процесі переміщення до об'єкта або взаємодії з ним, рішення пропускається для уникнення конфліктів. У стані бездіяльності система формує список доступних дій

шляхом перевірки всіх інтерактивних об'єктів на карті, що фільтрує об'єкти за їхньою доступністю.

Вибір дії здійснюється через метод ChooseAction, що реалізує epsilon-greedy стратегію балансування між дослідженням нових варіантів та експлуатацією накопичених знань. Система генерує випадкове число у діапазоні від 0 до 1 та порівнює його з поточним значенням параметра ExplorationRate. Якщо випадкове значення менше за ExplorationRate, система обирає випадкову дію з масиву доступних альтернатив, реалізуючи дослідницьку поведінку. В іншому випадку викликається метод GetBestAction, що ітерує через всі доступні дії та обирає ту, що має найвище Q-значення для поточного стану. Після кожного оновлення Q-таблиці параметр множиться на коефіцієнт ExplorationDecay, поступово знижуючи ймовірність випадкових дій. Мінімальне значення MinExplorationRate встановлено на рівні 0.15, гарантуючи що навіть після тривалого навчання система продовжує періодично досліджувати альтернативні стратегії (див. рис. А.6, додаток А). Такі параметри були підібрані емпірично у результаті кількох невдалих спроб навчання з іншими конфігураціями, де занадто швидкий спад призводив до передчасної конвергенції до субоптимальних політик. Логіку виконання всього ігрового циклу зображено на рисунку 7 (див. додаток А).

Протягом взаємодії компонент NeedsComponent продовжує виконувати деградацію всіх потреб у методі TickComponent, що викликається кожен кадр з частотою, залежною від поточного Time Dilation. Після завершення таймера взаємодії об'єкт викликає делегат OnInteractionComplete, що тригерить метод OnInteractionComplete у персонажа. Цей метод виконує ключові кроки алгоритму навчання: оновлення поточного стану через UpdateCurrentState, обчислення винагороди через CalculateReward, та оновлення Q-значення через

UpdateQValue. Останній метод реалізує центральну формулу алгоритму Q-learning (див. додаток Г).

Метод отримує поточне Q-значення для попереднього стану та дії, обчислює максимальне Q-значення серед усіх дій у новому стані через GetMaxQValue, та застосовує формулу оновлення з коефіцієнтом навчання LearningRate зі значенням 0.3 та коефіцієнтом дисконтування DiscountFactor зі значенням 0.95. Коефіцієнт навчання 0.3 визначає швидкість адаптації до нової інформації, балансуючи між стабільністю накопичених знань та реактивністю на нові спостереження. Значення 0.95 для коефіцієнта дисконтування надає агенту здатність планувати приблизно на 20 кроків вперед, оскільки внесок винагороди на кроці n зменшується до 0.95^n від початкового значення. Ці параметри також були підібрані: занадто низький LearningRate призводив до повільної збіжності, а занадто високий створював нестабільність у навчанні.

Критичним моментом у навчання є обробка смерті персонажа. Коли будь-яка потреба досягає нульового значення, компонент NeedsComponent викликає делегат OnNPCDied, що тригерить відповідний метод у ANPCCharacter. Компонент NPCSpawnManager отримує повідомлення про смерть через делегат, та після затримки у два секунди спавниться нове покоління з інкрементованим номером Generation. Нове покоління завантажує оновлену Q-таблицю, отримуючи доступ до всього накопиченого досвіду.

Висновки до другого розділу

Розроблено математичну модель адаптивної поведінки неігрових персонажів на основі марківського процесу прийняття рішень. Формалізовано віртуальне середовище з дискретним простором станів розміром 729 комбінацій та простором дій з семи варіантів взаємодії. Визначено детерміновану функцію переходу, що враховує постійну

деградацію потреб зі швидкістю 0.4 пункти на секунду під час виконання дій.

Побудовано структуру Q-таблиці з механізмом серіалізації у JSON-формат, що забезпечує збереження накопичених знань між поколіннями персонажів. Розроблено багатокомпонентну функцію винагороди з базовою винагородою, бонусами за покращення потреб та підтримання стабільності, штрафами за критичні стани та термінальним штрафом за смерть. Показано, що система стимулює превентивне обслуговування потреб замість реактивного реагування на критичні ситуації.

Реалізовано алгоритм Q-Learning з epsilon-greedy стратегією балансування між дослідженням нових варіантів та експлуатацією накопичених знань. Підібрано параметри навчання емпірично: коефіцієнт навчання $\alpha = 0.3$ для балансу між стабільністю та реактивністю, коефіцієнт дисконтування $\gamma = 0.95$ для планування на 20 кроків вперед, початковий рівень дослідження $\varepsilon = 0.95$ з поступовим зниженням до 0.15. Забезпечено автономне навчання через безперервний цикл взаємодії без явного розділення на фази тренування та тестування.

РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ АДАПТИВНОЇ СИСТЕМИ НАВЧАННЯ NPC

3.1. Огляд середовища для розробки

Сучасний ринок ігрових рушіїв характеризується наявністю кількох домінуючих платформ, кожна з яких орієнтована на різні сегменти розробників та типи проектів. Unity займає провідну позицію у сегменті мобільних та інді-ігор завдяки низькому порозі входу та кросплатформенності. Godot набуває популярності серед незалежних розробників завдяки free open-source політиці, та легкості у вивченні [19][20]. Unreal Engine традиційно домінує у сегменті AAA-проектів та технічно складних застосувань, пропонуючи найпотужніший інструментарій для створення високоякісної графіки та складних систем геймплею [21].

Unreal Engine 5 базується на архітектурі з нативною підтримкою C++, що забезпечує максимальну продуктивність виконання та повний контроль над управлінням пам'яттю – критично важливі аспекти для реалізації алгоритмів, які потребують інтенсивних обчислень [22]. Система рефлексії Unreal Header Tool автоматично генерує метадані для класів, що дозволяє зберігати та завантажувати складні структури даних, такі як Q-таблиці, у форматі JSON без необхідності ручного написання коду серіалізації.

Вбудований AI Framework в Unreal Engine надає потужний інструментарій для створення поведінки NPC. Для задачі навчання з підкріпленням особливо цінною є можливість програмного доступу до Path Following Component, що дозволяє NPC автономно переміщуватися до цільових об'єктів взаємодії, звільняючи алгоритм Q-learning від необхідності навчатися низькорівневому управлінню рухом та дозволяючи зосередитися на високорівневому прийнятті рішень щодо вибору дій [23].

Система компонентів в Unreal Engine забезпечує високу ступінь модульності та повторного використання коду. Система делегатів та подій

дозволяє реалізувати слабко зв'язану архітектуру, де компоненти взаємодіють через оголошені інтерфейси без прямих залежностей, що критично важливо для масштабованості системи та можливості незалежного тестування окремих модулів [24][25].

Інструментарій для налагодження та профілювання в Unreal Engine 5 включає Session Frontend для аналізу продуктивності у реальному часі, Visual Logger для візуалізації поведінки AI-агентів, Gameplay Debugger для інспекції стану персонажів під час виконання, та Insights – потужну систему трасування подій з неймовірною точністю. Для нашого проекту ці інструменти є незамінними для аналізу процесу навчання, виявлення проблем у реалізації алгоритму та оптимізації продуктивності.

Критичною перевагою використання C++ в Unreal Engine для реалізації алгоритмів машинного навчання є значна різниця у продуктивності порівняно з альтернативними підходами до програмування логіки. Blueprint, незважаючи на свою доступність та зручність для швидкого прототипування, виконуються через інтерпретовану віртуальну машину, що призводить до суттєвого перенавантаження порівняно з нативним машинним кодом. Для простих геймплейних скриптів ця різниця може бути несуттєвою, однак для обчислювально інтенсивних операцій, таких як оновлення великих структур даних чи виконання складних математичних обчислень різниця може бути значною.

У контексті нашої реалізації, використання C++ забезпечує кілька специфічних переваг. Оновлення Q-таблиці, що відбувається після кожної виконаної дії кожного агента, серіалізація та десеріалізація Q-таблиці при збереженні та завантаженні навченої політики вимагає ефективної обробки великих обсягів структурованих даних, де нативні можливості C++ для роботи з пам'яттю та файловою системою забезпечують оптимальну швидкодію.

Система управління часом в Unreal Engine надає можливість програмного контролю над швидкістю симуляції через World Settings та параметр Time Dilation, що є критично важливим для прискорення процесу навчання агентів. Можливість запуску симуляції зі швидкістю 10-50x відносно реального часу дозволяє агенту накопичити еквівалент годин або навіть днів досвіду за лічені хвилини реального часу, що радикально скорочує час, необхідний для навчання прийнятної політики. При цьому, на відміну від повністю синтетичних середовищ навчання, використання реального ігрового движка гарантує, що навчена поведінка може бути безпосередньо застосовна у реальній грі без додаткового transfer learning.

Підсумовуючи, вибір Unreal Engine 5 для реалізації навчання поведінки NPC обґрунтовується поєднанням високої продуктивності C++ для обчислювально складних алгоритмів машинного навчання, потужного вбудованого AI Framework, гнучкої компонентної архітектури для модульної організації коду та інших приємних інструментів для розробників. Ці технічні характеристики роблять Unreal Engine оптимальним вибором не тільки для комерційної розробки ігор, але й для дослідницьких проектів різних варіацій.

3.2. Опис, реалізація симуляційного середовища та налаштування експерименту

Для проведення навчання було створено середовище представляє собою ізольований ігровий рівень з набором інтерактивних об'єктів, що дозволяють персонажу задовольняти різні потреби. Основною метою проектування середовища була мінімізація обчислювальних витрат при збереженні достатньої складності.

Симуляційний рівень базується на стандартному шаблоні Unreal Engine 5.6 з базовою площиною та певними підйомами. NavMesh генерується автоматично, забезпечуючи оптимальні шляхи переміщення

між об'єктами. Для оптимізації продуктивності симуляції застосовано спрощену модель фізичної взаємодії: колізії об'єктів налаштовано як прості геометричні фігури, вони реагують лише на перетин з персонажем без розрахунку складної фізики, а персонаж позбавлений анімаційної системи. Такі рішення дозволили мінімізувати навантаження на рушій, особливо в ситуаціях швидкості симуляції більше 20 [26].

У середовищі розміщено 21 інтерактивний об'єкт; розташовані фіксовано на карті у заздалегідь визначених позиціях. Це забезпечує відтворюваність експериментів, та не дає Q-learning робити неправильні висновки щодо записів в таблиці

Технічна реалізація об'єктів базується на класі AInteractableObject з параметризованою системою модифікації потреб. Кожен об'єкт визначається трьома ключовими параметрами: типом дії з переліку EActionType, тривалістю взаємодії у секундах, та масивом модифікаторів потреб, що специфікують, які саме потреби та на скільки пунктів змінюються після завершення взаємодії. Система тригерних зон на основі UBoxComponent детектує наближення персонажа та ініціює процес взаємодії. Під час взаємодії об'єкт встановлює прапорець bIsOccupied, що перешкоджає одночасному використанню кількома персонажами та забезпечує коректність симуляції; вигляд об’єкта зображено на рис. А.8, а вся сцена на рис. А.9 (див. додаток А).

Параметри алгоритму Q-Learning встановлено на основі емпіричного підбору у попередніх експериментах. Коефіцієнт навчання $\alpha = 0.3$ визначає швидкість адаптації до нової інформації, балансуючи між стабільністю накопичених знань та реактивністю на свіжі спостереження. Коефіцієнт дисконтування 0.95 надає агенту здатність планувати приблизно на двадцять кроків вперед, враховуючи довгострокові наслідки поточних рішень. Початковий рівень дослідження встановлено на 0.95, що забезпечує інтенсивне дослідження простору станів на ранніх етапах навчання.

Коефіцієнт 0.993 визначає швидкість зниження рівня дослідження. Інтервал прийняття рішень встановлено у дві секунди, щоб NPC могли завершити розпочаті дії перед вибором наступної стратегії.

Під час навчання використовувалися різні значення для прискорення часу; це дозволяє накопичити значний обсяг досвіду навчання за обмежений реальний час без компромісів щодо точності симуляції фізики та навігації. Вдала спроба мала значення часу x15 прискореного часу й одне покоління персонажа з типовою тривалістю життя 30 хвилин ігрового часу завершувалося за 2 хвилини реального часу. Система логування через компоненти UCSVLogger та UGenerationLogger фіксує детальну телеметрію кожної дії, зміни потреб, причини смерті та загальні статистики поколінь.

3.3. Аналіз результатів навчання та ефективності алгоритму

Експеримент проводився декілька разів з різними вагами, швидкістю симуляції та кількістю об’єктів і NPC на мапі. Найбільш невдала спроба нарахувала більше ніж 500 поколінь ненавчинених акторів (і це все без прискорення часу). Зараз же ми розглянемо вдалу спробу навчання, що показала прийнятний результат.

Протягом 252 поколінь загальна тривалість експерименту, з урахуванням прискорення часу у п'ятнадцять разів, становила приблизно вісім годин реального часу, протягом яких система накопичила понад 38 тисяч записів про окремі дії та зміни станів. Кожне покоління починалося з випадкової ініціалізації потреб у діапазоні 70-90 відсотків та успадковувало Q-таблицю від попереднього покоління.

Аналіз тривалості життя персонажів демонструє чітку тенденцію до покращення виживаності протягом навчального процесу (рис. А.10, додаток А). Перше покоління прожило лише 426 секунд (приблизно 7 хвилин), завершивши життя через виснаження потреби Hygiene після виконання лише 22 дій. Рання фаза навчання (покоління 1-50)

характеризувалася середньою тривалістю життя 2249 секунд (37.5 хвилин) при медіані 1681 секунда через домінування exploration-стратегії з високим рівнем epsilon (0.95 в 0.67), що призводило до випадкового вибору дій. Варіативність була високою зі стандартним відхиленням 2179 секунд. Середня кількість виконаних дій становила 156 при середньому рівні потреб 62.5 відсотка, що вказує на значну кількість передчасних смертей через неоптимальні рішення.

Проміжна фаза навчання демонструє найвищу ефективність з піком у поколіннях 60-90. У цей період epsilon знижується з 0.67 до 0.33. Середня тривалість життя у піковій зоні (60-90) досягає 4600 секунд (76.7 хвилин) при медіані 2544 секунди, що становить покращення на 104.5% порівняно з раннім етапом. Покоління 61 встановило абсолютний рекорд тривалості життя у 24553 секунди (6.8 годин), виконавши 1775 дій та підтримуючи середній рівень потреб 70.2%.

Пізня фаза навчання (покоління 151-252) характеризується деградацією продуктивності. Epsilon досягає мінімального значення 0.15-0.16, внаслідок чого 85 відсотків дій обираються на основі максимального Q-значення. Середня тривалість життя у цій фазі становить 1700 секунд (28.3 хвилини) при медіані 1231 секунда, що представляє деградацію на 63.1 відсотка порівняно з піковою фазою та на 24.4 відсотка нижче від ранньої фази. Середня кількість дій скорочується до 110 порівняно з 324 на піку), виконавши 309 дій та підтримуючи середній рівень потреб на рівні 66.6% до моменту смерті від потреби Social.

Кількість виконаних дій протягом життя служить індикатором ефективності використання доступного часу персонажем. На ранніх етапах навчання NPC виконували в середньому 156 дій за життя. У піковій фазі (60-90) персонажі виконували в середньому 324 дії, що вказує на високу активність та превентивну стратегію. Натомість пізні покоління (151-252) виконували лише 110 дій, що демонструє зниження ефективності.

Середній рівень потреб при смерті персонажа відображає якість навченої стратегії підтримання балансу між різними потребами. Ранні покоління демонстрували середній рівень 62.5 відсотка при смерті. Пікові покоління (60-90) досягали 66.1 відсотка, що представляє покращення на 5.8 відсотка. Пізні покоління підтримували середній рівень 64.1 відсотка, що є незначним покращенням відносно ранньої фази (+2.6%), але нижче піку та вказує на формування більш збалансованої політики обслуговування потреб. Це можна побачити на графіку (див. рис. А.11, додаток А).

Аналіз причин смерті протягом усіх 252 поколінь виявляє нерівномірний розподіл між різними типами потреб; це зображає рисунок 12 (див. додаток А). Потреба Social спричинила найбільшу кількість смертей, що пояснюється відносно повільною швидкістю відновлення через дивани. Така статистика демонструє, що навіть після тривалого навчання система не досягла ідеальної політики через фундаментальні обмеження середовища, де деякі потреби об'єктивно складніше підтримувати.

Аналізуючи фінальну Q-таблицю, вона містить 487 унікальних станів з 729 теоретично можливих, що становить 66.8% покриття можливих станів. Кожен досліджений стан містить від одної до шести можливих дій з відповідними Q-значеннями, що разом формує 2847 пар стан-дія. Такий рівень покриття є достатнім для ефективного функціонування системи, оскільки не відвідані стани представляють екстремальні комбінації потреб, які рідко виникають при оптимальній політиці.

Розподіл Q-значень у фінальній таблиці демонструє переважання позитивних оцінок, що підтверджує успішність процесу навчання. З 2847 пар стан-дія, 2406 (84.5%) мають позитивні Q-значення, 398 (14.0%) – негативні, та 43 (1.5%) – нульові або близькі до нуля. Середнє Q-значення становить 456.3, медіана – 523.7, що вказує на зсув розподілу в бік високих позитивних оцінок. Максимальне Q-значення досягає 1024.7 у стані з

критично низькою потребою Energy, що відображає високу цінність негайного відновлення життєво важливої потреби. Негативні Q-значення, які варіюються від -100 до -5, асоціюються переважно зі станами, які неминуче ведуть до смерті навіть за оптимальної дії. Зокрема, стан "111111" для дії UseToilet має Q-значення -100, оскільки ця комбінація зафіксована лише один раз і закінчилася смертю. Для наглядності ці дані зображено на графіку (див. рис. А.13, додаток А).

Аналіз частоти відвідувань різних станів виявляє значну нерівномірність у дослідженні простору. Стан "222222", де всі шість потреб знаходяться на високому рівні, відвіданий 6511 разів, що становить 17.0% від усіх дій. Це оптимальний цільовий стан, досягнення якого забезпечує максимальну безпеку та гнучкість подальших рішень. Наступні за частотою стани "222212", "222221", "222122" представляють конфігурації з однією потребою на середньому рівні при інших на високому, відвідані 1200-1800 разів кожен. Критичні стани типу "011222" або "201222" з однією потребою на низькому рівні відвідані значно рідше, що вказує на ефективність навченої політики у запобіганні потрапляння в небезпечні ситуації. Стани з двома або більше критичними потребами практично відсутні в історії відвідувань, оскільки такі конфігурації швидко призводять до смерті (див. рис. А.14, додаток А).

Розподіл обраних дій протягом усього експерименту демонструє наявність переваг, зумовлених як структурою винагороди, так і статистикою деградації потреб. Найбільша дія UseBed використана 6842 рази, що робить її найчастішою через швидку деградацію Energy під час тривалих взаємодій з іншими об'єктами. Відносна рівномірність розподілу з відхиленням лише 1.9% між найчастішою та найрідшою дією свідчить про збалансовану політику, де персонаж систематично обслуговує всі типи потреб замість зосередження на підмножині (див. рис. А.15, додаток А).

Кількісне порівняння трьох фаз навчання виявляє нелінійну динаміку продуктивності з піком у середині процесу (див. рис. А.16, додаток А). Медіанна тривалість життя зростає з 1681 секунди на ранньому етапі (1-50) до 2544 секунд у піковій фазі (60-90), що становить покращення на 51.3 відсотка. Проте на пізньому етапі (151-252) медіана падає до 1231 секунди, що на 51.6 відсотка нижче піку та на 26.8 відсотка нижче ранньої фази.

Boxplot-аналіз виявляє зменшення варіативності результатів на пізньому етапі, проте це зменшення супроводжується загальним зниженням продуктивності замість очікуваної стабілізації на високому рівні. Така динаміка вказує на перенавчання системи (overfitting) до специфічних патернів поведінки, що сформувалися на середніх етапах, та втрату адаптивності через критично низький рівень exploration.

Незважаючи на спостережувану деградацію продуктивності на пізніх етапах навчання, досягнуті результати демонструють високу ефективність табличного Q-Learning у контексті багатовимірного управління потребами. Абсолютний рекорд тривалості життя у 6.8 годин ігрового часу з виконанням понад півтори тисячі послідовних дій свідчить про формування складної адаптивної політики, здатної одночасно балансувати шість конкуруючих потреб з постійною деградацією. Середня тривалість життя на піковому етапі значно перевищує мінімально спланований поріг виживання, що підтверджує здатність системи не просто реагувати на критичні ситуації, а проактивно підтримувати стабільний стан протягом тривалих періодів.

Феномен перенавчання, що призвів до зниження продуктивності після покоління 150, є очікуваним наслідком фундаментальних обмежень табличного підходу в умовах високорозмірного простору станів з неперервною динамікою. Дискретизація шести потреб до трьох категорій створює простір з 729 можливих станів, проте реальна кількість релевантних траєкторій є значно більшою через залежність між послідовними станами та варіативність ефективності різних об'єктів. За

таких умов досягнення піку продуктивності на середньому етапі навчання, коли баланс exploration-exploitation є оптимальним ($\varepsilon \approx 0.50$-0.60), демонструє спроможність алгоритму виявляти ефективні стратегії навіть за обмеженої статистики відвідувань станів. Подальше зниження epsilon до мінімального рівня закономірно призводить до втрати адаптивності, проте сам факт досягнення високої продуктивності на проміжному етапі підтверджує коректність архітектури системи та вибору параметрів навчання.

Для повної оцінки ефективності розробленого підходу необхідно розглянути результати в контексті альтернативних конфігурацій експерименту та порівняти з іншими спробами налаштування системи, які демонструють принципово відмінні патерни навчання або повну відсутність збіжності.

Порівняльний аналіз п'яти альтернативних конфігурацій експерименту демонструє критичну важливість коректного налаштування параметрів навчання. Спроби 1, 3 та 4 характеризуються стабільно низькою тривалістю життя на рівні 70-90 секунд протягом усіх 100 поколінь без жодних ознак покращення, що вказує на повну відсутність збіжності алгоритму. Спроба 2 демонструє дещо вищі показники у діапазоні 150-180 секунд, проте також без тенденції до зростання. Найбільш показовою є Спроба 5, де спостерігаються екстремальні коливання тривалості життя від 0 до 1050 секунд, що свідчить про нестабільність навчального процесу через неоптимальні значення коефіцієнтів α або γ (див. рис. А.17, додаток А).

Аналіз кількості виконаних дій підтверджує неефективність конфігурацій, що були використані. Спроби 1-4 демонструють стабільно низьку активність на рівні 3-10 дій за життя, тоді як Спроба 5, незважаючи на періодичні сплески до 55 дій, не формує консистентної стратегії. Для порівняння, успішна конфігурація на піковій фазі досягала середнього

показника 324 дії за життя, що у 30-50 разів перевищує результати невдалих спроб (див. рис. А.18, додаток А).

Динаміка середнього рівня потреб при смерті виявляє фундаментальну відмінність між успішною та невдалими конфігураціями. Спроби 4 та 5 стабільно підтримують потреби на рівні 70-75 відсотків, що парадоксально вказує на передчасну смерть через неефективну політику, а не виснаження ресурсів. Спроби 1-3 демонструють значно нижчі показники 15-35 відсотків, що свідчить про неспроможність системи навчитися базовому обслуговуванню потреб (див. рис. А.19, додаток А). Ці результати підтверджують, що ми, в крайній ітерації навчання, обрали правильні та більш підходящі значення для навчання.

Висновок до третього розділу

Розроблено та реалізовано повноцінне експериментальне середовище для дослідження ефективності алгоритму Q-Learning у задачі управління багатовимірною системою потреб неігрових персонажів. Створено симуляційне середовище в Unreal Engine 5.6 з 21 інтерактивним об'єктом, що забезпечує варіативність параметрів ефективності та дозволяє відтворювати експерименти з ідентичними початковими умовами. Налаштовано прискорення симуляції з коефіцієнтом 15x для скорочення загального часу навчання до восьми годин реального часу при збереженні коректності збору статистичних даних.

Проведено комплексне експериментальне дослідження неігрових персонажів з накопиченням понад 38 тисяч записів про окремі дії та зміни станів. Виявлено нелінійну динаміку навчального процесу з трьома чітко вираженими фазами: ранньою фазою з середньою тривалістю життя 2249 секунд, піковою фазою (покоління 60-90) з досягненням 4600 секунд, та пізньою фазою зі зниженням до 1700 секунд. Встановлено абсолютний рекорд тривалості життя у 24,553 секунди (6.8 годин) з виконанням 1775

послідовних дій, що підтверджує здатність системи формувати складні довгострокові стратегії виживання.

Проаналізовано причини деградації, включаючи катастрофічне забування раніше накопичених знань, втрату інформації через дискретизацію потреб до трьох рівнів, та відсутність механізмів диференціації всередині дискретних категорій станів. Проведено порівняльний аналіз п'яти альтернативних конфігурацій експерименту, що демонструють повну відсутність збіжності або екстремальну нестабільність навчального процесу.

ВИСНОВКИ

У результаті роботи розроблено та експериментально перевірено систему адаптивної поведінки неігрових персонажів на основі алгоритму Q-Learning. Робота охоплює повний цикл від теоретичного обґрунтування та математичної формалізації до практичної реалізації та комплексного експериментального дослідження ефективності розробленого підходу.

Проведено системний аналіз існуючих підходів до моделювання поведінки NPC, що виявив фундаментальні обмеження традиційних методів. Встановлено, що скриптові підходи вимагають детального програмування кожної можливої ситуації розробником, що призводить до передбачуваної та шаблонної поведінки персонажів. Це обґрунтовує необхідність переходу до методів машинного навчання, які дозволяють агентам автономно формувати оптимальні стратегії через взаємодію з середовищем без явного програмування правил для кожного можливого стану.

Спроектовано та реалізовано симуляційне середовище в UE 5 з використанням компонентної архітектури на мові C++. Розроблено систему автоматичного управління життєвим циклом персонажів з підтримкою прискореної симуляції та комплексного логування всіх дій для подальшого статистичного аналізу. Порівняльний аналіз альтернативних конфігурацій підтвердив критичну важливість систематичного підбору параметрів навчання для специфіки конкретного середовища.

Результати дослідження демонструють, що Q-Learning є прогресивним та перспективним підходом для створення адаптивної поведінки NPC у сучасних відеоіграх, незважаючи на об'єктивно більшу складність порівняно з традиційними скриптовими методами. Розроблена система вимагає значно більших обчислювальних ресурсів на етапі навчання, потребує систематичного моніторингу метрик збіжності для виявлення проблем перенавчання, та вимагає глибокого розуміння як

теоретичних основ навчання з підкріпленням, так і специфіки предметної області для коректного налаштування параметрів. Проте з розвитком обчислювальних потужностей сучасного обладнання, зростанням кількості спеціалістів з машинного навчання в ігровій індустрії, та накопиченням best practices щодо налаштування алгоритмів для різних типів ігрових механік, Q-Learning та інші методи навчання з підкріпленням стають все більш практично застосовними для комерційних проєктів.

Ключова перевага підходу полягає у фундаментально іншій парадигмі розробки поведінки: замість явного програмування правил для сотень можливих ситуацій, розробник визначає лише цільову функцію через систему винагород, а система автономно формує оптимальну стратегію для досягнення цих цілей. Це відкриває можливості для створення NPC, які демонструють природну адаптивність до непередбачуваних ситуацій, здатність до довгострокового планування на основі очікуваної корисності майбутніх станів, та варіативну поведінку, що значно підвищує реіграбельність через відсутність жорстко детермінованих реакцій.

Перспективи подальших досліджень включають перехід до Deep Q-Learning для роботи з неперервними просторами станів, впровадження мультиагентного навчання для моделювання кооперативної взаємодії, застосування ієрархічного Q-Learning для розбиття складних завдань на підзавдання, та інтеграцію механізмів transfer learning для перенесення накопичених знань між різними ігровими сценаріями. З огляду на швидкий розвиток методів навчання з підкріпленням та зростання їхньої доступності через open-source бібліотеки, найближче десятиліття може стати переломним для масового впровадження адаптивних AI-систем у комерційних відеоіграх.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. NPC Definition. Cambridge Dictionary : веб-сайт. URL: https://dictionary.cambridge.org/uk/dictionary/english/npc (дата звернення: 10.12.2025).
2. Game AI Pro : веб-сайт. URL: https://www.gameaipro.com/ (дата звернення: 10.12.2025).
3. Finite State Machines. Brilliant.org: веб-сайт. URL: https://brilliant.org/wiki/finite-state-machines/ (дата звернення: 10.12.2025).
4. The Craft of Behavior Trees. Medium: веб-сайт. URL: https://medium.com/@sion_denis/the-craft-of-behavior-trees-ab6b181ce21a (дата звернення: 10.12.2025).
5. Halo Franchise. Halo Fandom Wiki: веб-сайт. URL: https://halo.fandom.com/wiki/Halo_Franchise (дата звернення: 10.12.2025).
6. Handling Complexity in the Halo 2 AI. Game Developer Conference: веб-сайт. URL: https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai (дата звернення: 10.12.2025).
7. Behavior Trees or Finite State Machines. Opsive Documentation: веб-сайт. URL: https://opsive.com/support/documentation/behavior-designer/behavior-trees-or-finite-state-machines/ (дата звернення: 10.12.2025).
8. Goal-Oriented Action Planning. Web Archive: веб-сайт. URL: https://web.archive.org/web/20131109111207/http://web.media.mit.edu/~jorkin/goap.html (дата звернення: 10.12.2025).
9. F.E.A.R. First Encounter Assault Recon. FEAR Fandom Wiki: веб-сайт. URL: https://fear.fandom.com/wiki/F.E.A.R._First_Encounter_Assault_Recon (дата звернення: 10.12.2025).
10. Playing Atari with Deep Reinforcement Learning. ArXiv: веб-сайт. URL: https://arxiv.org/abs/1312.5602 (дата звернення: 10.12.2025).
11. Reinforcement Learning: A Survey. ArXiv: веб-сайт. URL: https://arxiv.org/abs/cs/9605103 (дата звернення: 10.12.2025).
12. A Markovian Decision Process. JSTOR: веб-сайт. URL: https://www.jstor.org/stable/24900506 (дата звернення: 10.12.2025).

13. Q-learning. Springer Link: веб-сайт. URL: https://link.springer.com/article/10.1007/BF00992698 (дата звернення: 10.12.2025).

14. Markov Decision Processes. Wiley Online Library: веб-сайт. URL: https://onlinelibrary.wiley.com/doi/10.1002/9780470400531.eorms0499 (дата звернення: 10.12.2025).

15. Needs System. The Sims FreePlay Wiki: веб-сайт. URL: https://simsfreeplay.fandom.com/wiki/Needs (дата звернення: 10.12.2025).

16. Markov Chains. Brilliant.org: веб-сайт. URL: https://brilliant.org/wiki/markov-chains/ (дата звернення: 10.12.2025).

17. Q-Table Representation. Stack Overflow: веб-сайт. URL: https://stackoverflow.com/questions/42547787/q-table-representation (дата звернення: 10.12.2025).

18. JSON in Unreal Engine. Unreal Engine Forums: веб-сайт. URL: https://forums.unrealengine.com/t/lets-talk-json/100749/10 (дата звернення: 10.12.2025).

19. Unity Documentation. Unity Technologies: веб-сайт. URL: https://docs.unity.com/en-us (дата звернення: 10.12.2025).

20. Godot Engine Documentation. Godot: веб-сайт. URL: https://docs.godotengine.org/en/stable/ (дата звернення: 10.12.2025).

21. Unreal Engine 5.7 Documentation. Epic Games: веб-сайт. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-7-documentation (дата звернення: 10.12.2025).

22. C++ Memory Management. W3Schools: веб-сайт. URL: https://www.w3schools.com/cpp/cpp_memory_management.asp (дата звернення: 10.12.2025).

23. Artificial Intelligence in Unreal Engine 5. Packt Publishing GitHub: веб-сайт. URL: https://github.com/PacktPublishing/Artificial-Intelligence-in-Unreal-Engine-5 (дата звернення: 10.12.2025).

24. Components in Unreal Engine. Epic Games Documentation: веб-сайт. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/components-in-unreal-engine (дата звернення: 10.12.2025).

25. Delegates and Lambda Functions in Unreal Engine. Epic Games Documentation: веб-сайт. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/delegates-and-lambda-functions-in-unreal-engine (дата звернення: 10.12.2025).

26. NavMesh Implementation. Unreal Engine Forums: веб-сайт. URL: https://forums.unrealengine.com/t/navmesh-a-comprehensive-series-on-generation-and-implementation/1244090 (дата звернення: 10.12.2025).

27. Sutton R.S., Barto A.G. Reinforcement Learning: An Introduction. 2nd ed. MIT Press, 2018. 552 с.

28. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 4th ed. Pearson, 2020. 1136 с.

29. Millington I., Funge J. Artificial Intelligence for Games. 3rd ed. CRC Press, 2018. 892 с.

30. Yannakakis G.N., Togelius J. Artificial Intelligence and Games. Springer, 2018. 348 с.

31. Gregory J. Game Engine Architecture. 3rd ed. CRC Press, 2018. 1200 с.

32. Rabin S. Game AI Pro: Collected Wisdom of Game AI Professionals. CRC Press, 2013. 520 с.

33. Buckland M. Programming Game AI by Example. Jones & Bartlett Learning, 2004. 520 с.

34. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016. 800 с.

35. Lapan M. Deep Reinforcement Learning Hands-On. 2nd ed. Packt Publishing, 2020. 826 с.

36. Francois-Lavet V., Henderson P., Islam R., Bellemare M.G., Pineau J. An Introduction to Deep Reinforcement Learning. Now Publishers, 2018. 150 с.

37. Graesser L., Keng W.L. Foundations of Deep Reinforcement Learning. Addison-Wesley, 2019. 320 с.

38. Plaat A. Learning to Play: Reinforcement Learning and Games. Springer, 2020. 350 с.

39. Sewak M. Deep Reinforcement Learning: Frontiers of Artificial Intelligence. Springer, 2019. 200 с.

40. Zai A., Brown B. Deep Reinforcement Learning in Action. Manning Publications, 2020. 450 с.

41. Stroustrup B. The C++ Programming Language. 4th ed. Addison-Wesley, 2013. 1376 с.

42. Meyers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media, 2014. 336 с.

43. Lippman S.B., Lajoie J., Moo B.E. C++ Primer. 5th ed. Addison-Wesley, 2012. 976 c.

44. Josuttis N.M. The C++ Standard Library: A Tutorial and Reference. 2nd ed. Addison-Wesley, 2012. 1100 c.

45. Macklin W., Sherif W., Plowman S. Unreal Engine 5 Game Programming with C++. Packt Publishing, 2023. 384 c.

46. Sewell B. Unreal Engine 4 Scripting with C++ Cookbook. Packt Publishing, 2016. 550 c.

47. Romero S., Sewell B. Unreal Engine 4.x Scripting with C++ Cookbook. 2nd ed. Packt Publishing, 2019. 624 c.

48. Sherif W. Learning C++ by Building Games with Unreal Engine 4. 2nd ed. Packt Publishing, 2019. 422 c.

49. Plowman S. Game Development Projects with Unreal Engine. Packt Publishing, 2020. 500 p.Raschka S., Mirjalili V. Python Machine Learning. 3rd ed. Packt Publishing, 2019. 772 c.

50. Cookson H. Elevating Game Experiences with Unreal Engine 5. 2nd ed. Packt Publishing, 2022. 480 c.

ДОДАТКИ

Додаток А – Схеми алгоритмів та архітектура інтелектуальних агентів моделі

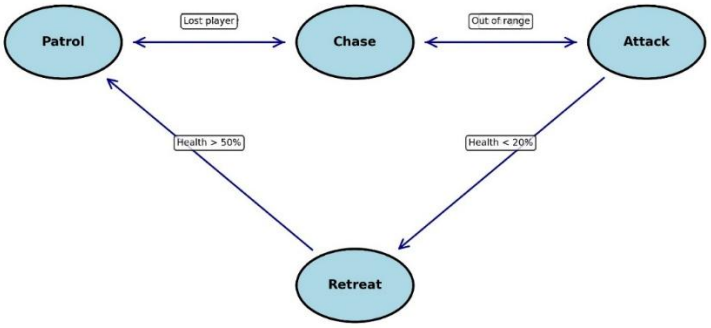

Рис. А.1 – приклад FSM

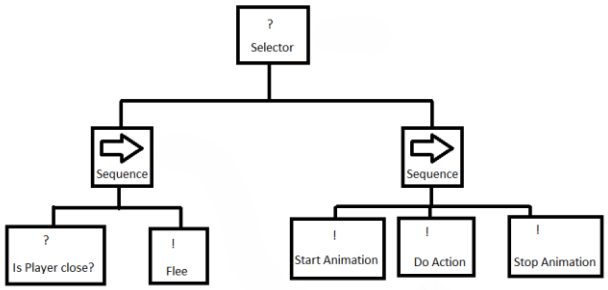

Рис. А.2 – ВТ з подвійною дією

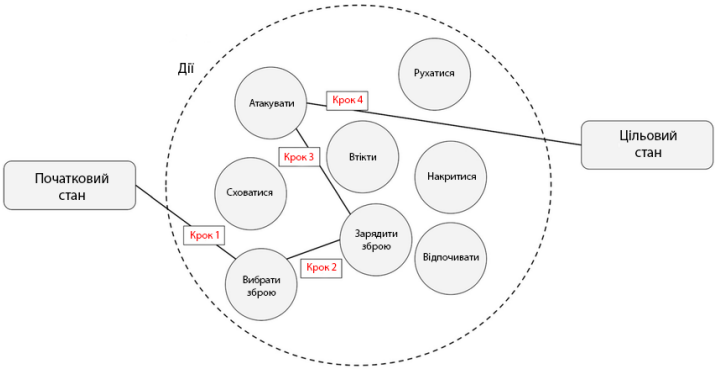

Рис. А.3 – Зображення алгоритму

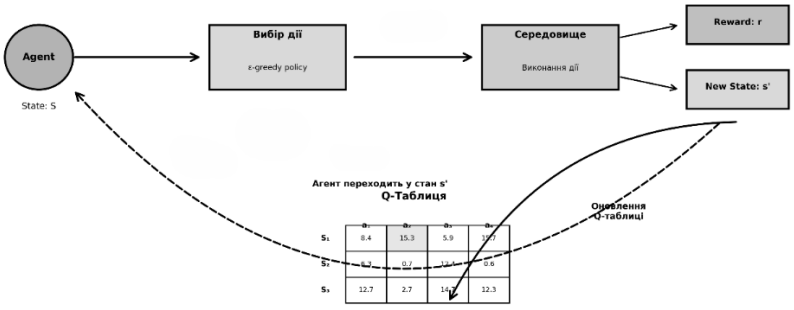

Рис. А.4 – Процес Q-learning

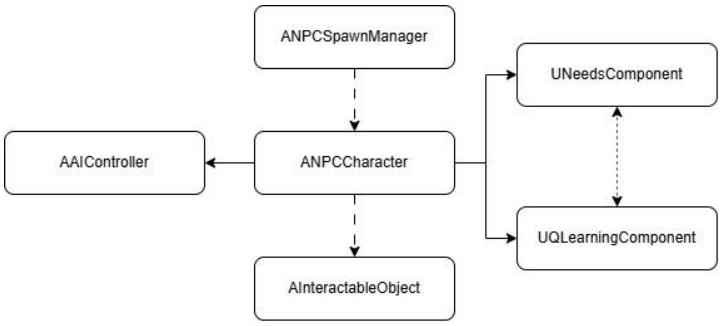

Рис. А.5 – взаємодія класів

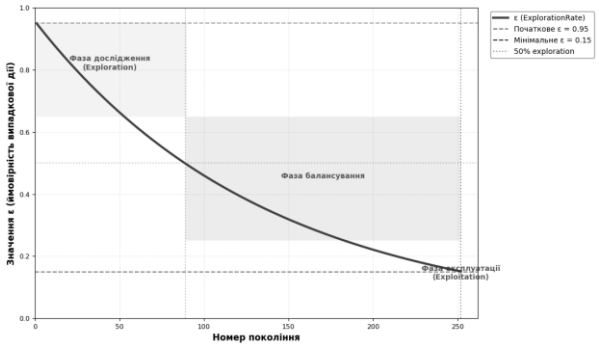

Рис. А.6 – зміна коефіцієнту дослідження

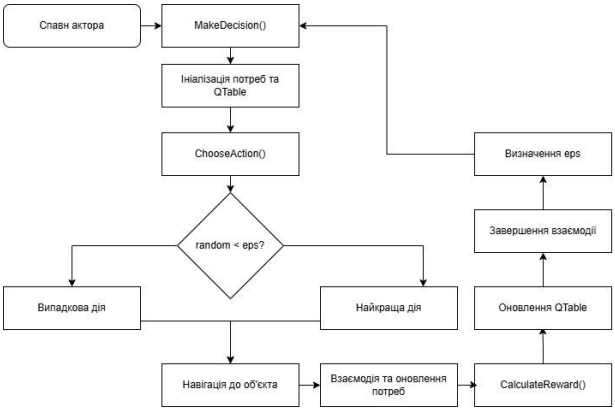

Рис. А.7– цикл навчання NPC

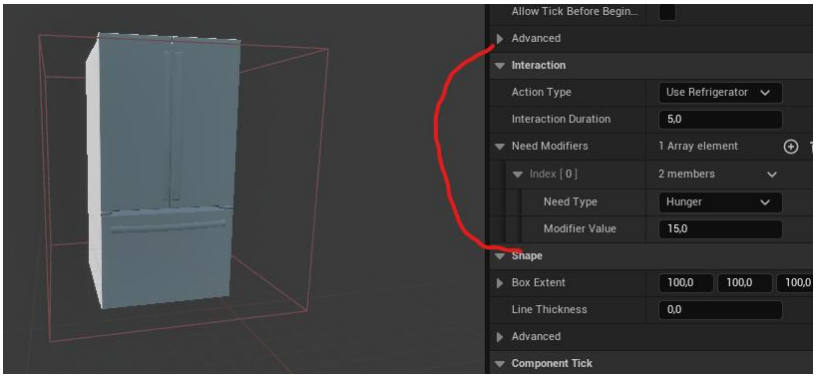

Рис. А.8– налаштування об'єкта в BP редакторі

Рис. А.9 – Загальний вигляд ігрового середовища

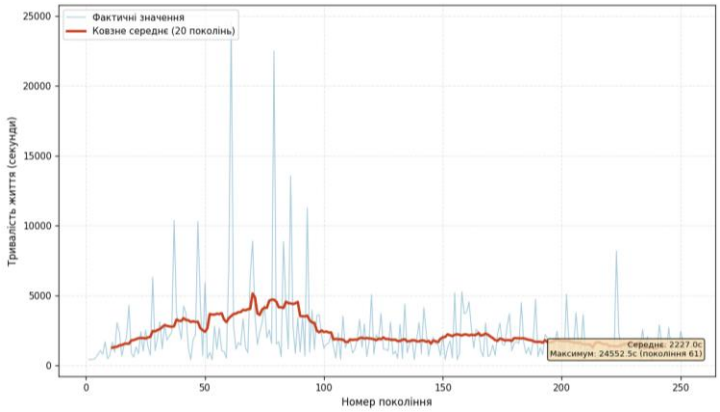

Рис. А.10 – тривалість життя NPC

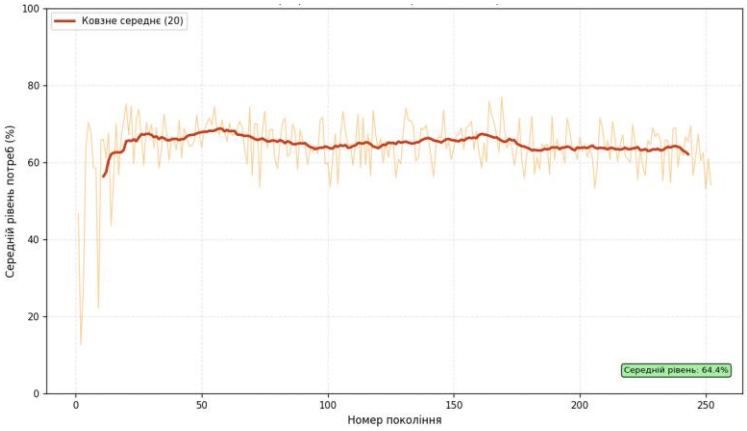

Рис. А.11 – якість підтримки потреб

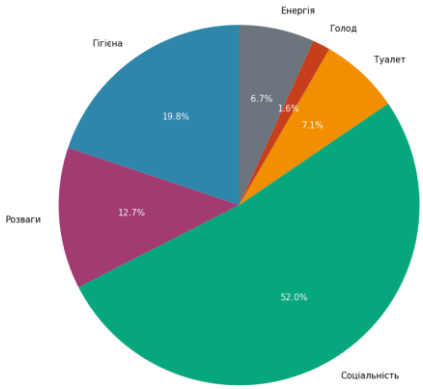

Рис. А.12 – зображення смертностей

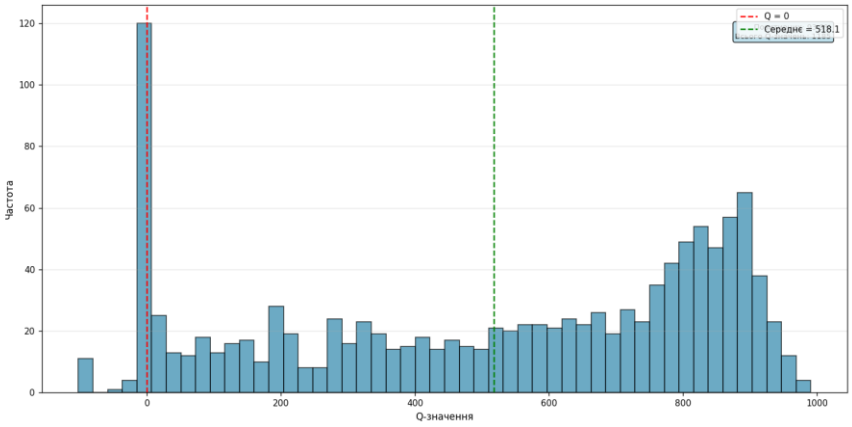

Рис. А.13 – розподіл Q-значень

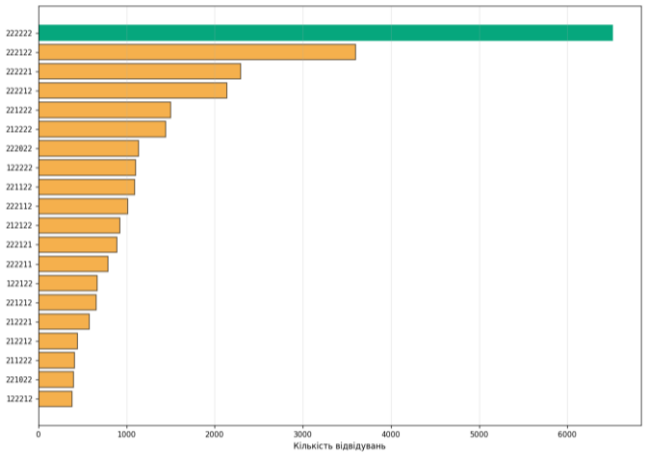

Рис. А.14 – 20 найбільш частих значень

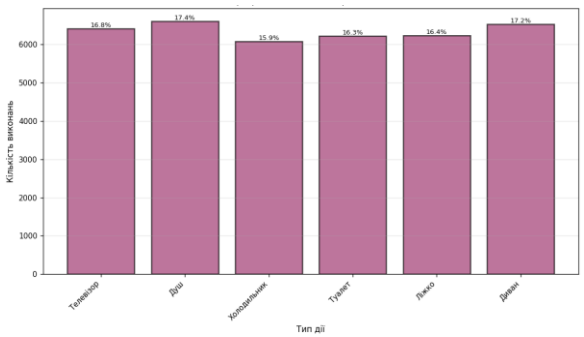

Рис. А.15 – розподіл обраних дій

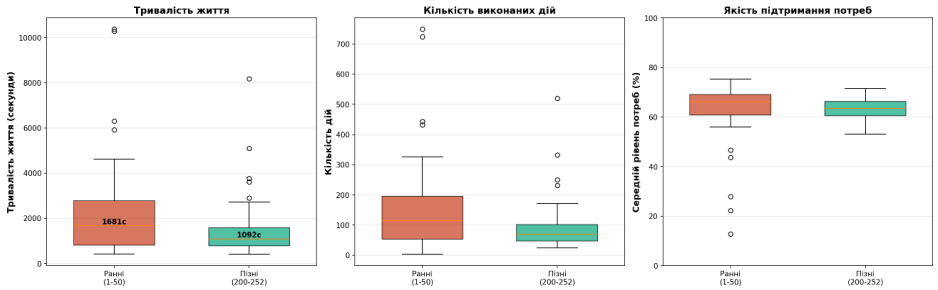

Рис. А.16 – порівняння між початковими та фінальними поколіннями

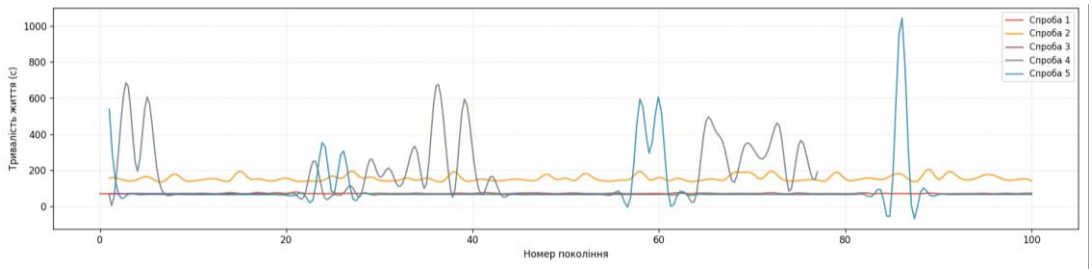

Рис. А.17 – Порівняння тривалості життя у невдалих спробах

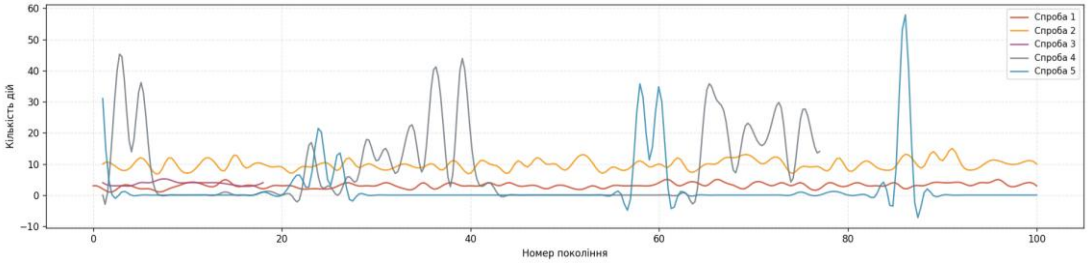

Рис. А.18 – Порівняння кількості дій у невдалих спробах

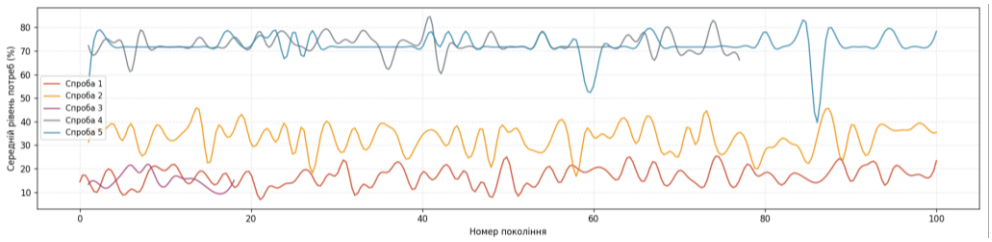

Рис. А.19 – Порівняння середнього рівня потреб у невдалих спробах

Додаток Б – Приклад QTable


```
"110122":
        {
            "1":
            {
                "Value": 2.5,
                "TimesVisited": 1
            },
            "2":
            {
```

```
            "Value": 3.625,
            "TimesVisited": 1
        },
        "6":
        {
            "Value": 4.1312499046325684,
            "TimesVisited": 1
        },
        "4":
        {
            "Value": -98.140937805175781,
            "TimesVisited": 1
        }
    },
    "210102":
    {
        "1":
        {
            "Value": -5,
            "TimesVisited": 1
        },
        "2":
        {
            "Value": 104.09530639648438,
            "TimesVisited": 1
        }
    },
    "220012":
    {
        "3":
        {
            "Value": -5,
            "TimesVisited": 1
        },
        "6":
        {
            "Value": -100,
            "TimesVisited": 1
        }
    },
    "112102":
    {
        "6":
        {
            "Value": 2.5,
            "TimesVisited": 1
        },
        "1":
        {
            "Value": 2.5,
            "TimesVisited": 1
        }
    }
```


Додаток В – Базові структури для Q-Learning

```
#pragma once

#include "CoreMinimal.h"
#include "NeedType.h"
#include "QLearningTypes.generated.h"

UENUM(BlueprintType)
enum class ENeedLevel : uint8
{
    Low         UMETA(DisplayName = "Low (0-33%)"),
    Medium      UMETA(DisplayName = "Medium (33-66%)"),
    High        UMETA(DisplayName = "High (66-100%)"),

    MAX         UMETA(Hidden)
```

```
};

UENUM(BlueprintType)
enum class EActionType : uint8
{
    Idle            UMETA(DisplayName = "Idle"),
    UseTelevision   UMETA(DisplayName = "Use Television"),
    UseShower       UMETA(DisplayName = "Use Shower"),
    UseRefrigerator UMETA(DisplayName = "Use Refrigerator"),
    UseToilet       UMETA(DisplayName = "Use Toilet"),
    UseBed          UMETA(DisplayName = "Use Bed"),
    UseSofa         UMETA(DisplayName = "Use Sofa"),

    MAX             UMETA(Hidden)
};

USTRUCT(BlueprintType)
struct FNPCState
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite)
    TMap<ENeedType, ENeedLevel> NeedLevels;

    FNPCState()
    {
        for (int32 i = 0; i < (int32)ENeedType::MAX; i++)
        {
            NeedLevels.Add((ENeedType)i, ENeedLevel::Medium);
        }
    }

    FString GetStateKey() const
    {
        FString Key = "";

        for (int32 i = 0; i < (int32)ENeedType::MAX; i++)
        {
            ENeedType NeedType = (ENeedType)i;
            ENeedLevel Level = NeedLevels.Contains(NeedType) ?
                               NeedLevels[NeedType] : ENeedLevel::Medium;
            Key += FString::FromInt((int32)Level);
        }

        return Key;
    }

    static ENeedLevel ValueToLevel(float Value)
    {
        if (Value <= 33.0f) return ENeedLevel::Low;
        if (Value <= 66.0f) return ENeedLevel::Medium;
        return ENeedLevel::High;
    }

    bool operator==(const FNPCState& Other) const
    {
        return GetStateKey() == Other.GetStateKey();
    }
};

USTRUCT(BlueprintType)
struct FQValue
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite)
    float Value;

    UPROPERTY(BlueprintReadWrite)
    int32 TimesVisited;

    FQValue() : Value(0.0f), TimesVisited(0) {}
    FQValue(float InValue) : Value(InValue), TimesVisited(1) {}
};
```

```
USTRUCT(BlueprintType)
struct FActionQValues
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite)
    TMap<EActionType, FQValue> ActionValues;

    FActionQValues() {}
};

USTRUCT(BlueprintType)
struct FLogEntry
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite)
    float Timestamp;

    UPROPERTY(BlueprintReadWrite)
    int32 NPCID;

    UPROPERTY(BlueprintReadWrite)
    int32 Generation;

    UPROPERTY(BlueprintReadWrite)
    EActionType Action;

    UPROPERTY(BlueprintReadWrite)
    FString StateKey;

    UPROPERTY(BlueprintReadWrite)
    float Reward;

    UPROPERTY(BlueprintReadWrite)
    float Lifetime;

    UPROPERTY(BlueprintReadWrite)
    TMap<ENeedType, float> CurrentNeeds;

    FLogEntry() : Timestamp(0.0f), NPCID(0), Generation(0), Action(EActionType::Idle),
                  StateKey(""), Reward(0.0f), Lifetime(0.0f) {}
};

USTRUCT(BlueprintType)
struct FQLearningParams
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Q-Learning")
    float LearningRate = 0.5f;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Q-Learning")
    float DiscountFactor = 0.9f;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Q-Learning")
    float ExplorationRate = 0.95f;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Q-Learning")
    float ExplorationDecay = 0.9998f;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Q-Learning")
    float MinExplorationRate = 0.15f;
};

USTRUCT(BlueprintType)
struct FNeedModifier
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Needs")
    ENeedType NeedType;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Needs")
    float ModifierValue;
```

```
    FNeedModifier() : NeedType(ENeedType::Hunger), ModifierValue(0.0f) {}
    FNeedModifier(ENeedType Type, float Value) : NeedType(Type), ModifierValue(Value) {}
};
```

Додаток Г – Код оновлення Q-значень

```
void UQLearningComponent::UpdateQValue(float Reward)
{
    if (!NeedsComponent)
    {
        UE_LOG(LogTemp, Error, TEXT("UpdateQValue: No NeedsComponent!"));
        return;
    }

    float CurrentQ = GetQValue(PreviousState, PreviousAction);
    float MaxNextQ = GetMaxQValue(CurrentState, GetAllActions());

    float NewQ = CurrentQ + Params.LearningRate *
                 (Reward + Params.DiscountFactor * MaxNextQ - CurrentQ);

    SetQValue(PreviousState, PreviousAction, NewQ);

    if (Params.ExplorationRate > Params.MinExplorationRate)
    {
        Params.ExplorationRate *= Params.ExplorationDecay;
        Params.ExplorationRate = FMath::Max(Params.ExplorationRate,
                                            Params.MinExplorationRate);
    }

    UE_LOG(LogTemp, Warning, TEXT("Q-Update: State=%s, Action=%d, Reward=%.2f,
OldQ=%.2f, NewQ=%.2f, MaxNextQ=%.2f, ε=%.3f"),
           *PreviousState.GetStateKey(), (int32)PreviousAction, Reward,
           CurrentQ, NewQ, MaxNextQ, Params.ExplorationRate);
}
```