Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів машинного навчання для оптимізації процедурної генерації рівнів у розробці ігор
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-3

Микита КОВАЛЬОВ
(Власне ім’я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник проф. Ігор ШУБІН
(посада, Власне ім’я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри ____________ Кирило СМЕЛЯКОВ
(підпис) (Власне ім’я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________
(підпис)
«___» ________ 2025 р.

**ЗАВДАННЯ**
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Ковальову Микиті Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методів машинного навчання для оптимізації процедурної генерації рівнів у розробці ігор»

Затверджена наказом по університету від 15.04. 2025р. № 290 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20.06.2025

3. Вихідні дані до роботи дослідження методів машинного навчання для оптимізації процедурної генерації рівнів, зокрема Q-learning у середовищі Unity. Аналіз існуючих підходів до процедурної генерації, застосування алгоритмів навчання з підкріпленням для оптимізації генерації рівнів, програмна реалізація на C#.

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз методів машинного навчання для процедурної генерації рівнів, дослідження алгоритму Q-learning та його застосування у цій сфері, розробка моделі процедурної генерації рівнів на основі Q-learning, програмна реалізація та тестування моделі у Unity, аналіз отриманих результатів і оцінка роботи системи.*

# КАЛЕНДАРНИЙ ПЛАН

| № | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання | 16.04.25 | *виконано* |
| 2 | Аналіз предметної галузі і постановка задачі | 17.04.25-21.04.25 | *виконано* |
| 3 | Аналіз методів машинного навчання для процедурної генерації рівнів | 22.04.25-28.04.25 | *виконано* |
| 4 | Розробка моделі процедурної генерації рівнів на основі Q-learning | 29.04.25-15.05.25 | *виконано* |
| 5 | Аналіз результатів і оцінка роботи системи | 16.05.25-20.05.25 | *виконано* |
| 6 | Підготовка до апробації результатів дослідження. Публікація матеріалів | 20.05.25-27.05.25 | *виконано* |
| 7 | Підготовка пояснювальної записки | 28.05.25-06.06.25 | *виконано* |
| 8 | Підготовка презентації та доповіді | 07.06.25-11.06.25 | *виконано* |
| 9 | Перевірка на плагіат | 12.06.25 | *виконано* |
| 10 | Нормоконтроль | 13.06.25 | *виконано* |
| 11 | Рецензування | 13.06.25 | *виконано* |
| 12 | Попередній захист | 16.06.25 | *виконано* |
| 13 | Занесення диплома в електронний архів | 17.06.25 | *виконано* |
| 14 | Допуск до захисту у зав. кафедри | 17.06.25 | *виконано* |

Дата видачі завдання 16 квітня 2025р.

Студент ________________________ Микита КОВАЛЬОВ
(підпис)

Керівник роботи ________________ проф. Ігор ШУБІН
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

**РЕФЕРАТ / ABSTRACT**

Звіт: 65 с., 16 рис., 5 дод., 20 джерел.

МАШИННЕ НАВЧАННЯ, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, Q-LEARNING, ОПТИМІЗАЦІЯ, UNITY, C#, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, АЛГОРИТМ, МОДЕЛЮВАННЯ, АНАЛІЗ, ТЕСТУВАННЯ, AI, ГЕНЕРАЦІЯ РІВНІВ.

Об’єктом дослідження є методи машинного навчання для оптимізації процедурної генерації рівнів у розробці ігор.

Метою роботи є підвищення ефективності процедурної генерації рівнів шляхом застосування алгоритмів навчання з підкріпленням, зокрема Q-learning.

Методи розробки базуються на таких технологіях, як Unity, C#, та алгоритми навчання з підкріпленням.

У результаті роботи було досліджено існуючі методи машинного навчання, проведено аналіз підходів до процедурної генерації рівнів, розроблено модель генерації на основі Q-learning, програмно реалізовано та протестовано її в Unity, а також здійснено аналіз отриманих результатів і оцінку ефективності системи.

MACHINE LEARNING, PROCEDURAL GENERATION, Q-LEARNING, OPTIMIZATION, UNITY, C#, REINFORCEMENT LEARNING, ALGORITHM, MODELING, ANALYSIS, TESTING, AI, LEVEL GENERATION.

The object of the study is machine learning methods for optimizing procedural level generation in game development.

The aim of the study is to increase the efficiency of procedural level generation by applying reinforcement learning algorithms, in particular Q-learning.

The development methods are based on technologies such as Unity, C#, and reinforcement learning algorithms.

As a result of the work, existing machine learning methods were investigated, approaches to procedural level generation were analyzed, a generation model based on Q-learning was developed, implemented and tested in Unity, and the results obtained were analyzed and the system efficiency was evaluated.

Завідувачу кафедри
ПІ_______________
(скорочена назва кафедри)
проф. Кирилу СМЕЛЯКОВУ
(вчене звання, власне ім'я, прізвище)

Заява щодо самостійного виконання кваліфікаційної роботи та можливості її публікації в електронному архіві відкритого доступу ElArKhNURE.

Я, Ковальов Микита Віталійович, студент гр. ІПЗм-23-3, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів машинного навчання для оптимізації процедурної генерації рівнів у розробці ігор», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата　　　　　　　　　　　　　　　　　Підпис

# ЗМІСТ

Вступ……………………………..…………………………………………...7
1 Аналіз предметної галузі і постановка задачі…………………………………8
1.1 Аналіз предметної галузі……………………………………………………...8
1.2 Актуальність роботи…………………………………………………………..8
1.3 Постановка задачі……………………………………………………………..9
2 Аналіз методів машинного навчання для процедурної генерації рівнів………...11
2.1 Традиційні методи процедурної генерації……………………………………11
2.2 Застосування машинного навчання у процедурній генерації………………14
2.3 Обґрунтування вибору підходу………………………………………………..18
3 Розробка моделі процедурної генерації рівнів на основі Q-learning……………22
3.1 Формалізація задачі навчання з підкріпленням………………………………22
3.2 Архітектура та реалізація моделі……………………………………………...23
4 Аналіз результатів і оцінка роботи системи……………………………………34
4.1 Результати роботи алгоритму…………………………………………………34
4.2 Оцінка роботи системи………………………………………………………...36
4.3 Обмеження та недоліки моделі………………………………………………...37
Висновки……………………………..…………………………………………...39
Перелік джерел посилання…………………………..…………………………41
Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії……………………………………………………42
Додаток А Звіт результатів перевірки на унікальність тексту в базі хнуре………44
Додаток Б Програмний код генерації карти за допомогою алгоритму клітинного автомата…………………………………………………………………………46
Додаток В Апробація результатів роботи……………………………………...56
Додаток Г Слайди презентації…………………………………………………58
Додаток Ґ Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008:2015…………………………...66

# ВСТУП

Процедурна генерація рівнів (PCG) є одним із ключових напрямків у сучасній розробці ігор, оскільки дозволяє створювати унікальні сценарії та локації без необхідності ручного проектування кожного елемента. Цей підхід сприяє не лише зниженню трудомісткості виробництва, але й підвищує реіграбельність продукту, відкриваючи нові можливості для творчості розробників.

За останні роки значну увагу отримали методи машинного навчання, що дозволяють адаптивно генерувати контент із врахуванням специфічних вимог до складності, балансу та логічної послідовності гри [1]. До того ж, інтеграція таких методів у процес розробки дає змогу оптимізувати роботу команд, заощаджуючи ресурси та скорочуючи час на тестування кінцевого продукту.

У цій кваліфікаційній роботі розглядаються сучасні підходи до застосування машинного навчання для генерації ігрових рівнів, із детальним аналізом переваг і недоліків кожного з них. Особлива увага приділяється алгоритмам навчання з підкріпленням, адже результати теоретичних досліджень свідчать про їхню високу ефективність для оптимізації процедурної генерації. Водночас, необхідно врахувати, що практична реалізація завжди включає опис теоретичних основ обраного підходу, що дозволить краще обґрунтувати прийняті рішення.

Метою даної роботи є розробка та апробація моделі генерації ігрових рівнів з використанням методів машинного навчання, де основним алгоритмом виступатиме Q-Learning. Проведення дослідження дозволить не лише порівняти ефективність, а й встановити потенціал для їх подальшого вдосконалення в контексті інтерактивних розважальних продуктів.

# 1 АНАЛІЗ ПРЕДМЕТОЇ ГАЛУЗІ І ПОСТАНОВКА ЗАДАЧІ

## 1.1 Аналіз предметної галузі

Розробка комп’ютерних ігор на сьогодні є однією з найбільш динамічних та інноваційних сфер цифрових технологій. Вона охоплює широкий спектр технічних, дизайнерських і когнітивних завдань, серед яких важливе місце посідає формування ігрового середовища, зокрема генерація рівнів. У сучасних проєктах створення контенту ручними засобами стає дедалі менш ефективним через високі вимоги до масштабу, варіативності та персоналізації ігрового досвіду.

Відтак, індустрія все активніше звертається до процедурних підходів, які дозволяють автоматизувати створення ігрового контенту. Це, у свою чергу, ставить нові виклики перед розробниками: як забезпечити якість ігрового процесу, адаптивність рівнів під різні стилі гри, уникнути повторюваності та надати гравцеві унікальний досвід.

Однією з перспективних відповідей на ці виклики стає інтеграція інструментів штучного інтелекту, зокрема машинного навчання. Вони дозволяють не лише автоматизувати процес створення рівнів, а й враховувати взаємодію користувача, аналізувати ігрові сценарії, адаптувати складність та структуру гри в реальному часі. У цьому контексті поєднання процедурної генерації з методами машинного навчання відкриває нові можливості для персоналізованого та гнучкого дизайну ігрових середовищ.

## 1.2 Актуальність роботи

З кожним роком вимоги до якості, масштабованості та персоналізації комп’ютерних ігор зростають. Користувачі очікують не лише цікавого сюжету та візуальної привабливості, а й високого рівня варіативності ігрового досвіду, який би залишався новим і захопливим при кожному проходженні. У відповідь на ці виклики розробники все частіше використовують процедурну генерацію, що дозволяє створювати контент автоматично, з мінімальним ручним втручанням.

Проте традиційні алгоритми генерації часто обмежуються статичними правилами, що не дає змоги гнучко адаптувати структуру рівня під конкретні сценарії гри або поведінку гравця. Це знижує потенціал процедурної генерації як інструменту для побудови по-справжньому динамічного та інтерактивного середовища [11].

У цьому контексті особливої актуальності набуває застосування методів машинного навчання, зокрема навчання з підкріпленням. Такі підходи дозволяють системі самостійно вивчати ефективні стратегії побудови рівнів на основі взаємодії з ігровим середовищем та цільовими метриками. Це відкриває можливість створення гнучких, адаптивних та більш реалістичних генераторів ігрового контенту.

Актуальність даної роботи полягає у дослідженні можливостей поєднання процедурної генерації з машинним навчанням для підвищення якості, варіативності та ігрової цінності згенерованих рівнів. Такий підхід відповідає сучасним тенденціям розвитку ігрової індустрії та має значний потенціал для практичного застосування в реальних проєктах [14].

## 1.3 Постановка задачі

Попри активне використання процедурної генерації в ігровій індустрії, більшість реалізованих рішень залишаються статичними та не враховують контексту поведінки гравця, динаміки складності чи логічної послідовності рівня. Зазвичай структура рівня визначається заздалегідь заданими правилами або випадковим розміщенням елементів, що обмежує гнучкість і призводить до повторюваності контенту. При цьому зростає попит на генеративні системи, здатні враховувати змінні умови, цільову складність та взаємозв’язки між компонентами.

Основна проблема полягає в тому, як змоделювати процес генерації так, щоб система самостійно навчалась створювати рівні, адаптовані до заданих умов, не потребуючи ручного коригування на кожному етапі. У контексті машинного навчання ця задача не є тривіальною: необхідно визначити, як представити

структуру рівня у вигляді станів і дій, сформулювати функцію винагороди, яка б відповідала очікуваній якості, і забезпечити ефективне навчання агента в умовах великої кількості можливих конфігурацій.

Проблематика дослідження зосереджується на тому, чи здатен агент, який використовує Q-Learning, ефективно будувати рівень з нуля без жорстко закладених шаблонів, лише на основі підкріплення. Також постає питання, які критерії слід закладати у функцію винагороди, щоб результат був не просто технічно коректним, а й ігрово доцільним. Не менш важливо як оцінювати якість згенерованого рівня: формально, через метрики, чи емпірично, через взаємодію з ним.

Таким чином, суть задачі полягає не лише у технічній реалізації алгоритму, а в побудові такого навчального середовища, яке дозволяє оцінити потенціал Q-Learning для генерації осмисленого, структурованого ігрового контенту у реалістичних умовах.

# 2 АНАЛІЗ МЕТОДІВ МАШИННОГО НАВЧАННЯ ДЛЯ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ РІВНІВ

## 2.1 Традиційні методи процедурної генерації

Процедурна генерація рівнів на основі плиток (Tile-based generation) є одним із найпоширеніших методів створення ігрового середовища. Цей підхід передбачає розбиття простору на дискретні комірки, які розміщуються відповідно до певних алгоритмічних правил. Основними алгоритмами, що застосовуються в межах даного підходу, є Wave Function Collapse (WFC), Cellular Automata та алгоритм Marching Squares [17].

Одним із найбільш сучасних і ефективних методів генерації рівнів є алгоритм Wave Function Collapse, який базується на принципах ентропійного скорочення. У процесі роботи алгоритму кожна клітинка рівня спочатку містить множину можливих станів (тайлів), після чого набір варіантів поступово скорочується відповідно до встановлених правил сусідства.

Основні етапи роботи алгоритму:

– ініціалізація, коли кожна клітинка приймає всі допустимі стани;

– вибір комірки, визначається клітинка з мінімальною ентропією, тобто з найменшою кількістю можливих варіантів;

– фіксація тайла, випадковим чином або на основі ймовірнісного розподілу вибирається конкретний тайл;

– оновлення сусідніх клітинок, коригується набір можливих станів для прилеглих комірок відповідно до правил сумісності;

– повторення процесу, кроки повторюються до повного заповнення рівня.

Рисунок 2.1 демонструє етапи роботи алгоритму Wave Function Collapse. Даний метод забезпечує генерацію складних, узгоджених структур із дотриманням топологічних та композиційних вимог, що робить його придатним для створення рівнів із чіткими архітектурними обмеженнями.

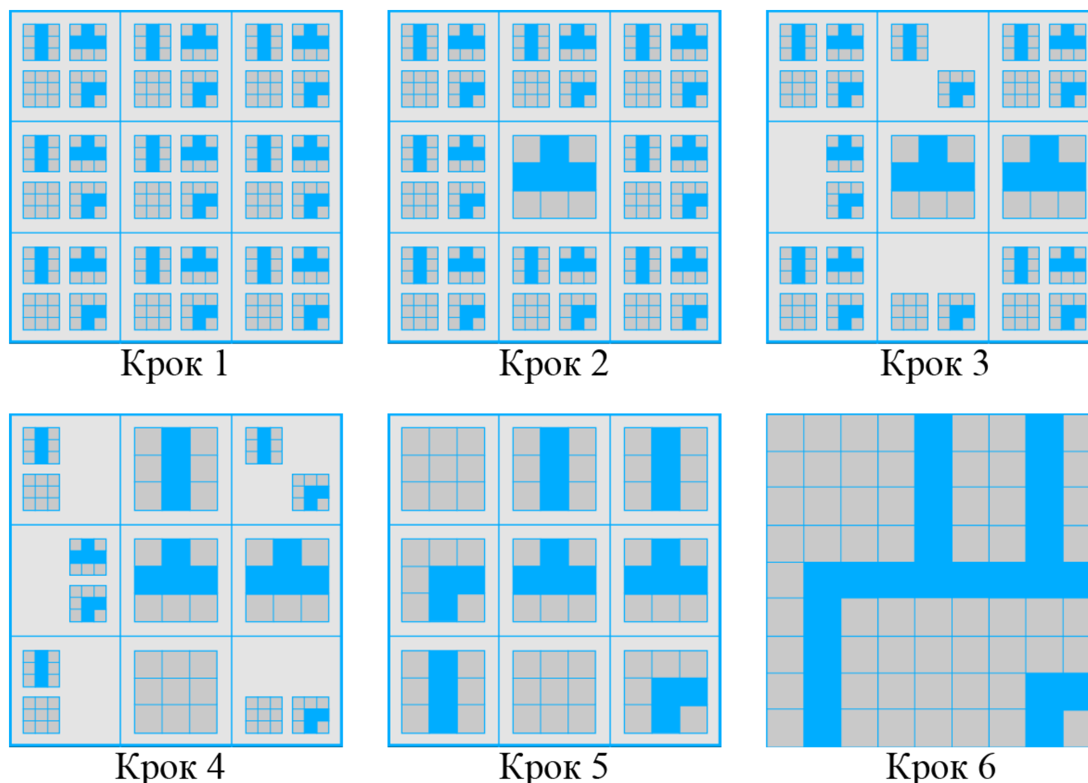


Рисунок 2.1 – Етапи роботи алгоритму Wave Function Collapse [2]

Однак алгоритм є досить ресурсомістким і може викликати труднощі під час роботи з великими картами.

Метод Cellular Automata використовується переважно для створення рівнів із природними, нерегулярними формами, такими як печери та підземні локації. Основний принцип роботи даного алгоритму полягає у застосуванні локальних правил для зміни станів клітинок на основі їхнього оточення.

Процес генерації рівня за допомогою клітинних автоматів зазвичай включає такі етапи:

– ініціалізація, випадкове заповнення рівня стінами та відкритими просторами (зазвичай у межах 40–55% для стін);

– застосування правил еволюції, коли для кожної клітинки підраховується кількість сусідніх «стін». Наприклад якщо навколо клітинки перебуває понад 4–5 стін, вона також перетворюється на стіну, або якщо сусідніх стін менше 3, клітинка стає прохідною;

– ітераційна обробка, алгоритм виконується кілька разів для досягнення більшої узгодженості рівня;

– остаточна оптимізація, видаляються ізольовані області та додаються ключові елементи рівня (входи, виходи).

Рисунок 2.2 демонструє згенерований рівень методом Cellular Automata.

Рисунок 2.2 – Приклад згенерованого рівня методом Cellular Automata (зроблено самостійно)

Метод є простим у реалізації та ефективним для створення нерегулярних середовищ. Проте він не завжди забезпечує контрольовану структуру рівня, що може бути небажаним у випадках, коли потрібні чітко визначені архітектурні об'єкти, такі як коридори та кімнати.

Метод Marching Squares є модифікацією підходу Cellular Automata, що використовується для побудови плавних контурів на основі дискретної сітки. На відміну від класичних методів, де кожна клітинка має чітко визначений стан, цей алгоритм працює з проміжними значеннями та використовує апроксимацію меж.

Основні етапи роботи алгоритму:

– формування сітки значень, карта рівня поділяється на регулярну сітку, де кожна клітинка містить числове значення (наприклад, висоту або щільність матеріалу);

– аналіз локальних патернів, для кожної клітинки визначаються значення чотирьох її кутових вершин. Залежно від того, які з них перевищують заданий поріг, формується один із 16 можливих шаблонів;

– генерація контурів, на основі знайдених патернів визначаються сегменти меж, які з'єднують відповідні точки триангуляції;

– згладжування меж, застосовується лінійна інтерполяція або інші методи апроксимації для коригування форми контуру.

Рисунок 2.3 ілюструє згенерований рівень методом Marching Squares.

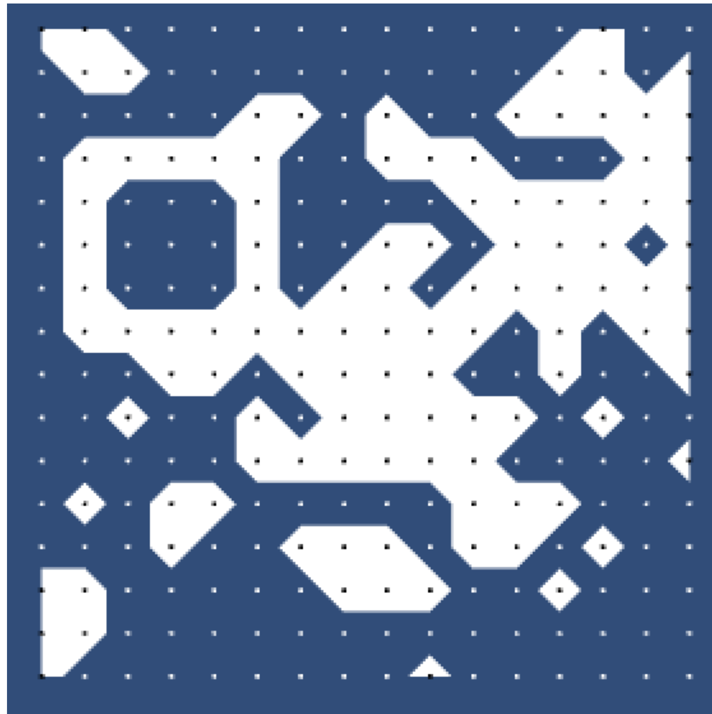

Рисунок 2.3 – Згенерований рівень методом Marching Squares [3]

Завдяки використанню інтерполяції та триангуляції, алгоритм Marching Squares дозволяє створювати плавніші та природніші контури місцевості порівняно з іншими традиційними підходами. Однак він менш ефективний у випадках, коли необхідно зберегти чітку структуру рівня з прямокутними кімнатами та коридорами.

## 2.2 Застосування машинного навчання у процедурній генерації

Використання методів машинного навчання у процедурній генерації рівнів дозволяє значно підвищити якість, різноманітність та адаптивність створюваних ігрових просторів. На відміну від класичних алгоритмічних підходів, що базуються на жорстко визначених правилах, машинне навчання дає змогу формувати складні

та динамічні структури, навчаючись на прикладах реальних рівнів або адаптуючи генерацію під гравця в режимі реального часу.

Основними напрямами використання машинного навчання у процедурній генерації є застосування генеративно-змагальних мереж, використання нейронних мереж та глибокого навчання для моделювання складних патернів, а також навчання з підкріпленням, що дозволяє адаптувати ігровий світ під дії користувача.

Генеративно-змагальні мережі (Generative Adversarial Networks, GAN) є одним із найефективніших підходів до процедурної генерації рівнів [4]. Вони дозволяють нейромережі навчатися на існуючих даних та створювати нові рівні, що стилістично відповідають вихідному матеріалу. Архітектура GAN складається з двох моделей: генератора, який створює рівні, та дискримінатора, що оцінює їхню автентичність. Завдяки цій взаємодії мережі поступово вдосконалюються, підвищуючи якість згенерованого контенту.

Особливістю використання GAN у розробці ігор є можливість створювати унікальні рівні, зберігаючи при цьому внутрішню логіку та стиль гри. Наприклад, модель, навчена на рівнях платформерів, зможе генерувати схожі структури, але без прямого копіювання вихідного матеріалу. Також GAN може адаптувати рівні під стиль гри користувача, змінюючи їхню складність залежно від успіхів гравця [12].

На рисунку 2.4 представлено приклад алгоритму генерації рівня за допомогою GAN.

Попри значні переваги, метод має обмеження. Навчання моделей потребує значних обчислювальних ресурсів, а результати не завжди гарантують іграбельність рівнів без додаткової перевірки. Водночас розвиток GAN відкриває нові можливості для автоматизації створення контенту, роблячи процес розробки ефективнішим.

Використання нейронних мереж у процедурній генерації рівнів базується на здатності моделей знаходити закономірності у великих наборах даних та створювати новий контент, що відповідає визначеним критеріям. [13]

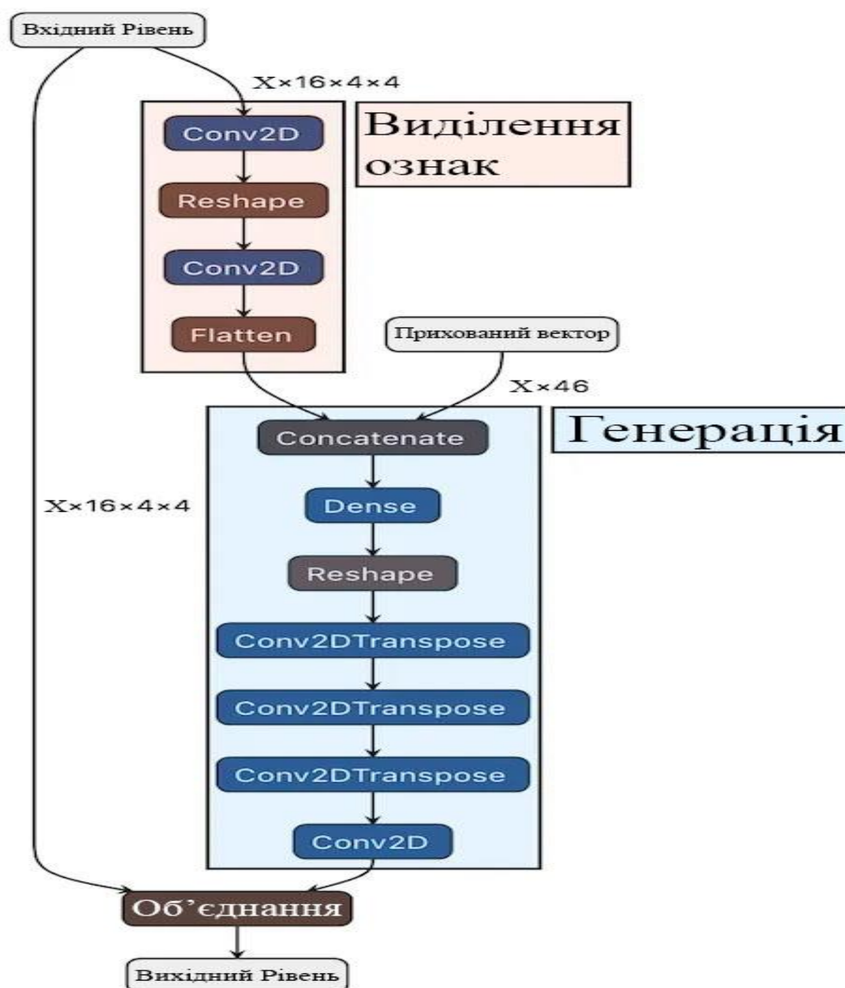


Рисунок 2.4 – Генерація рівня за допомогою GAN [5]

Глибоке навчання дозволяє алгоритмам не лише копіювати існуючі патерни, а і комбінувати їх у нових, несподіваних варіаціях, що робить рівні більш різноманітними та адаптивними [6].

Одним із ефективних підходів у цьому напрямі є використання згорткових нейронних мереж (Convolutional Neural Networks, CNN). CNN добре працюють із двовимірними структурами, тому їх широко застосовують для аналізу тайлових карт рівнів. Вони можуть не лише відтворювати архітектурні особливості рівнів, але й прогнозувати складність ігрового процесу на основі їхньої структури [7]. Наприклад, модель може аналізувати співвідношення платформ, перешкод і ворогів, визначаючи рівень виклику, який відчуватиме гравець. На рисунку 2.5 відображено прогнозування складності за допомогою CNN.

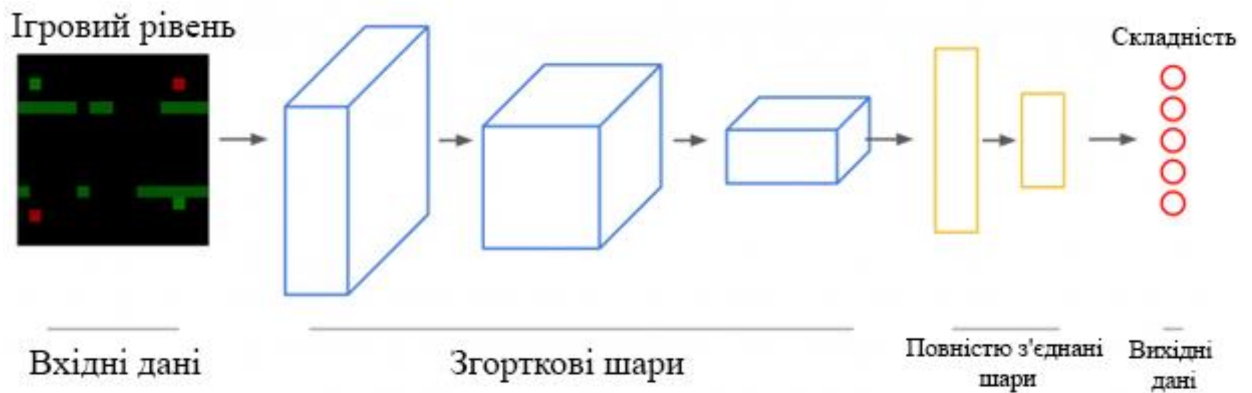


Рисунок 2.5 – Прогнозування складності використовуючи CNN [8]

Використання згорткових нейронних мереж у цьому контексті дозволяє не тільки аналізувати рівні, але й генерувати нові карти зі збереженням стилістичних особливостей гри. Наприклад, модель може навчитися відтворювати характерні патерни розташування платформ та ворогів, створюючи збалансовані рівні, які відповідають заданому рівню складності.

Навчання з підкріпленням (Reinforcement Learning, RL) є потужним інструментом для створення процедурно згенерованих рівнів, оскільки воно дозволяє агенту самостійно вивчати оптимальні структури рівнів через взаємодію з середовищем. На відміну від інших підходів, RL не вимагає великого набору навчальних даних, а натомість використовує систему винагород і штрафів для поступового вдосконалення створюваних рівнів [9].

Одним із найпопулярніших алгоритмів у цій сфері є Q-learning, який дозволяє агенту знаходити оптимальні рішення шляхом проб і помилок. Застосування даної техніки дозволяє навчити модель автоматично генерувати рівні, які відповідають заданим критеріям складності, прохідності або навіть динамічного пристосування до стилю гри користувача. Наприклад, агент може отримувати винагороду за створення рівнів, які забезпечують баланс між викликом і можливістю проходження, що дозволяє автоматично генерувати якісний контент без втручання людини [16]. На рисунку 2.6 відображено, що агент взаємодіє із середовищем, отримуючи від нього інформацію про стан та винагороду та дозволяє алгоритму адаптуватися до гравця.

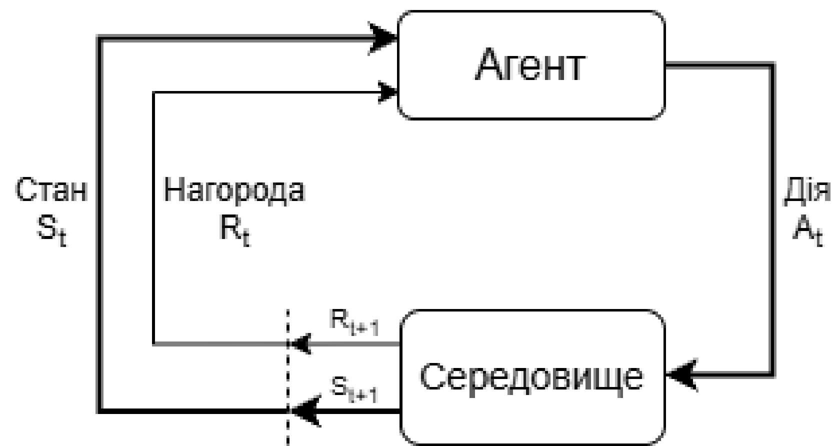


Рисунок 2.6 – Схема взаємодії агента з середовищем у навчанні з підкріпленням (зроблено самостійно)

Окрім Q-Learning, значного поширення набули глибокі методи навчання з підкріпленням, такі як Deep Q-Networks (DQN) та Proximal Policy Optimization (PPO). Вони використовують нейромережі для аналізу ігрового середовища та прийняття складних рішень у процесі генерації рівнів. Наприклад, DQN дозволяє агенту знаходити довгострокові закономірності в рівнях, тоді як PPO ефективніше адаптується до змінних умов гри. Завдяки цим підходам можливо створювати не лише статичні рівні, а й адаптивні структури, що змінюються в режимі реального часу залежно від поведінки гравця.

Застосування RL у генерації рівнів відкриває широкі перспективи, особливо для адаптивного геймдизайну. Алгоритми можуть створювати рівні в реальному часі, реагуючи на дії гравця, що робить ігровий досвід більш індивідуальним. Однак практична реалізація цього підходу потребує ретельного налаштування середовища навчання та значних обчислювальних ресурсів, що є ключовими викликами при його використанні.

## 2.3 Обґрунтування вибору підходу

Розробка процедурно згенерованих рівнів вимагає ретельного вибору підходу, оскільки кожен метод має свої переваги та обмеження. Оптимальний підхід повинен забезпечувати не лише якісну генерацію рівнів, а й відповідати

технічним вимогам, таким як адаптивність, ефективність та керованість процесу генерації.

Одним із традиційних підходів є генерація на основі правил. Вона передбачає створення рівнів за наперед визначеними шаблонами та логікою, що гарантує контрольований результат. Однак такий метод обмежує варіативність рівнів і погано адаптується до змінних умов гри.

Генеративно-змагальні мережі дозволяють отримувати нові рівні на основі аналізу існуючих зразків. Вони добре відтворюють стилістику та структуру навчальних даних, проте можуть створювати нерівномірні або непрохідні рівні, оскільки не мають механізму оцінки їхньої іграбельності.

Методи глибокого навчання, зокрема згорткові нейронні мережі, можуть оцінювати складність рівня та адаптувати генерацію під певні параметри. Проте самостійно вони не здійснюють процес генерації, а лише допомагають аналізувати та відбирати відповідні результати.

Еволюційні алгоритми базуються на поступовій оптимізації рівнів через мутації та відбір найкращих варіантів. Вони здатні знаходити ефективні рішення, проте їхня продуктивність сильно залежить від правильної побудови функції пристосованості, що робить їхнє налаштування досить складним процесом.

На цьому фоні навчання з підкріпленням виглядає перспективною альтернативою, оскільки воно не потребує великих наборів даних, може адаптувати рівні в процесі навчання та дає змогу керувати параметрами генерації.

Методи навчання з підкріпленням дають змогу агенту самостійно досліджувати різні варіанти побудови рівнів і вдосконалювати їх на основі отриманих винагород. Це відрізняє RL від інших підходів, які або покладаються на заздалегідь підготовлені набори даних, або працюють за жорстко визначеними правилами. Запропонований підхід відкриває широкі можливості для створення рівнів, що не тільки відповідають загальним критеріям якості, а й динамічно адаптуються під стиль гри конкретного користувача.

Один із ключових аспектів RL – його здатність до експериментального пошуку. Наприклад, алгоритми Q-Learning та Deep Q-Networks можуть ітеративно

покращувати результати, аналізуючи наслідки своїх дій у процесі навчання. У результаті вони можуть навчитися створювати збалансовані рівні, які з одного боку мають певний рівень виклику для гравця, а з іншого – залишаються прохідними та цікавими [5].

Ще однією перевагою є можливість адаптації рівнів у режимі реального часу. Уявімо ситуацію, коли гравець швидко проходить рівні без труднощів або, навпаки, застряє на певних ділянках. RL може автоматично коригувати структуру рівня, змінюючи розташування ворогів, ресурсів або перешкод. Це створює більш динамічний і персоналізований досвід, який значно відрізняється від традиційної процедурної генерації.

Крім того, RL ефективно поєднується з іншими методами. Наприклад, інтеграція RL із еволюційними алгоритмами може покращити процес пошуку оптимальних рішень, оскільки еволюційні механізми допомагають швидше знаходити ефективні конфігурації рівнів. Аналогічно, використання RL разом із алгоритмами планування шляху дозволяє створювати не тільки випадкові, а й структуровані рівні з логічними маршрутами для гравця.

Незважаючи на численні переваги, використання методів навчання з підкріпленням у процедурній генерації рівнів стикається з низкою значних викликів. Однією з основних проблем є велика кількість обчислювальних ресурсів, необхідних для навчання агентів. Процес навчання агентів за допомогою алгоритмів, таких як Q-Learning або DQN, вимагає численних ітерацій і значних обчислювальних потужностей для обробки величезних масивів даних, що може стати суттєвим бар'єром для застосування RL в реальних проектах, особливо в умовах обмежених ресурсів.

Ще одним викликом є налаштування функції винагороди. Для того, щоб агент правильно навчався і створював якісні рівні, необхідно розробити ефективну функцію винагороди, яка коректно відображатиме бажані характеристики рівнів (наприклад, складність, цікавість, прохідність). Невірно налаштована функція винагороди може призвести до створення нецікавих або непрохідних рівнів,

оскільки агент може намагатися максимізувати винагороду, не враховуючи важливих аспектів ігрового процесу.

Також важливим є питання збалансованості між експлорацією і експлуатацією. У навчанні з підкріпленням агент повинен знаходити компроміс між пошуком нових стратегій (експлорація) та використанням уже набутих знань (експлуатація). Надмірна експлорація може призвести до надмірного часу навчання, тоді як надмірна експлуатація може обмежити варіативність і генерацію нових, цікавих рівнів.

Іншою проблемою є необхідність інтеграції RL з іншими методами, такими як планування шляху або еволюційні алгоритми. У таких випадках потрібно забезпечити синхронізацію між різними підходами, що може ускладнити розробку і інтеграцію системи. Крім того, необхідність коригування рівнів в реальному часі вимагає від системи високої швидкості обробки, що може бути проблемою при складних рівнях або великих обсягах даних.

Важливо також враховувати складність у розробці і тестуванні таких систем. Агент, який працює за принципом RL, проходить через велику кількість тестувань і може мати різні варіанти поведінки залежно від умов навчання. Це ускладнює процес верифікації і забезпечення стабільності роботи генератора рівнів у фінальній версії гри.

Таким чином, хоча навчання з підкріпленням є потужним інструментом для генерації процедурних рівнів, його реалізація пов'язана з певними труднощами, що вимагають ретельного підходу до налаштування параметрів, вибору обчислювальних ресурсів і інтеграції з іншими методами.

# 3 РОЗРОБКА МОДЕЛІ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ РІВНІВ НА ОСНОВІ Q-LEARNING

## 3.1 Формалізація задачі навчання з підкріпленням

У контексті процедурної генерації рівнів задача навчання з підкріпленням може бути сформульована як задача пошуку оптимальної конфігурації ігрового простору, що відповідає заданим критеріям складності, прохідності та ігрового балансу.

Середовищем у цій задачі виступає сіткова структура рівня, де кожна комірка може містити різні ігрові об'єкти, такі як стіни, відкриті проходи, вороги чи корисні предмети. Агент взаємодіє з середовищем, змінюючи розташування цих об'єктів у рамках певного набору дозволених дій.

Стан середовища S визначається як матриця значень, де кожен елемент відповідає певному типу об’єкта. Множина можливих дій A включає операції зміни структури рівня, наприклад:

– додавання або видалення елементів, переміщення ворогів, зміна проходів;

– модифікація структури, розширення вузьких коридорів, регулювання розміщення об’єктів.

Для коректного навчання агент повинен отримувати винагороду за дії, які покращують якість рівня. Функція винагороди може враховувати такі фактори:

– прохідність рівня, гарантований шлях до виходу;

– баланс складності, розумне розташування ворогів і ресурсів;

– різноманіття шляхів і варіативність проходження.

Метою алгоритму є знаходження такої послідовності дій, яка призведе до створення рівня, що відповідає заданим критеріям складності та прохідності. В результаті генерації очікується отримати рівні, які не лише є технічно прохідними, а й забезпечують цікавий ігровий процес.

Таким чином, задача навчання з підкріпленням у процедурній генерації рівнів зводиться до оптимізації структури рівня на основі послідовних взаємодій агента з середовищем, спрямованих на покращення загальної якості рівня.

## 3.2 Архітектура та реалізація моделі

Основна мета реалізації, це створити систему, яка здатна оптимізувати згенеровані рівні, використовуючи навчання з підкріпленням. Для цього рівень проходить через кілька стадій модифікації, у яких агент Q-Learning аналізує поточний стан карти та вибирає одну з можливих дій для його покращення.

Процес відбувається наступним чином:

– початковий рівень створюється за допомогою алгоритму клітинного автомата, що забезпечує базову структуру рівня (стіни та прохідні області);

– система оцінює рівень, визначаючи його основні характеристики: наявність шляху між точками старту та виходу, кількість ворогів і скарбів, щільність прохідних областей тощо;

– на основі аналізу агент обирає одну з можливих дій, таких як згладжування карти, видалення стін, додавання ворогів або скарбів тощо;

– обрана дія застосовується до карти. Після цього процес повторюється, доки агент не досягне заданої кількості ітерацій;

– коли навчання завершується, система застосовує постобробку, видаляючи недоступні зони, ворогів у них тощо.

Рівень створюється за допомогою алгоритму клітинного автомата, який змінює карту на основі певних правил. Це дозволяє отримати природний розподіл стін і прохідних зон, уникаючи шаблонності.

Алгоритм працює так:

– створюється випадкова карта, де кожен осередок із певною ймовірністю є стіною;

– для кожної клітинки підраховується кількість сусідніх стін, після чого клітинка змінюється відповідно до порогових значень;

– процес повторюється кілька ітерацій для досягнення стабільної структури.

Фрагмент коду, що реалізує цей підхід, наведено на рисунку 3.1. Варто зазначити, що на рисунку представлено лише частину реалізації алгоритму. Повний вихідний код класу наведено в додатку А.

```
public class CellularAutomataLevel : MonoBehaviour
    Frequently called   1 usage   Hukuta K.
    void GenerateMap()
    {
        map = new int[width, height];
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                if (x == 0 || y == 0 || x == width - 1 || y == height - 1)
                    map[x, y] = 0;
                else
                    map[x, y] = (rand.Next(0, 100) < fillPercent) ? 1 : 0;
            }
        }
    }

    Frequently called   2 usages   Hukuta K.
    void SmoothMap()
    {
        int[,] newMap = new int[width, height];
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                int floorCount = CountFloorNeighbors(x, y);
                if (floorCount > 4)
                    newMap[x, y] = 1;
                else if (floorCount < 4)
                    newMap[x, y] = 0;
                else
                    newMap[x, y] = map[x, y];
            }
        }
        map = newMap;
    }
```

Рисунок 3.1 – Генерація карти алгоритмом клітинного автомата (зроблено самостійно)

Після ініціалізації середовища, проходить процес навчання, де на кожному кроку виконується:

– аналіз поточного стану рівня;

– вибір оптимальної дії на основі Q-таблиці (з певною ймовірністю випадковий вибір для дослідження);

– виконання обраної дії та отримання нового стану;

– обчислення винагороди та оновлення Q-таблиці згідно з наступною формулою:

$$Q(s,a) \leftarrow Q(s,a) + \alpha * \left(r * \gamma * \max_{a'} Q(s',a') - Q(s,a)\right) \quad (1)$$

де $Q(s,a)$ - поточне значення функції корисності для пари стан-дія. Воно визначає очікувану суму винагород, яку агент може отримати, виконуючи дію $a$ у стані $s$.

$\alpha$ – швидкість навчання, яка визначає ступінь, з якою нова інформація замінює стару. Значення $\alpha$ зазвичай знаходиться в діапазоні [0,1].

$r$ – негайна винагорода, отримана після виконання дії $a$ у стані $s$.

$\gamma$ – коефіцієнт дисконтування, який визначає вагу майбутніх винагород. Значення $\gamma$ також знаходиться в діапазоні [0,1]; чим ближче $\gamma$ до 1, тим більший вплив мають віддалені нагороди.

$\max_{a'} Q(s',a')$ – максимальне очікуване значення корисності серед усіх можливих дій $a'$ у новому стані $s'$. Іншими словами, агент визначає максимальне очікуване значення винагороди, яке може бути досягнуте за умови застосування оптимальної політики від моменту переходу у стан $s'$.

Метод ComputeStateMetrics відіграє ключову роль у визначенні стану рівня для агента Q-learning, конвертуючи структурну інформацію карти у компактне числове представлення. Основне завдання цього методу – проаналізувати геометрію рівня та ігрові об'єкти, що впливають на процес навчання агента.

Насамперед, метод отримує карту рівня з об'єкта CellularAutomataLevel у вигляді двовимірного масиву, де кожен осередок позначає певний тип поверхні. Виконується підрахунок кількості прохідних клітин, що дає загальну характеристику про розмір доступного для переміщення простору.

Критично важливим аспектом є перевірка існування прохідного шляху між початковою та кінцевою точками рівня, що здійснюється за допомогою функції PathExists(). Цей параметр є бінарним (true/false) і безпосередньо впливає на можливість проходження рівня агентом.

Наступним кроком метод визначає кількість ворогів на карті, включаючи як звичайних супротивників (об'єкти з тегом "Enemy"), так і можливого боса ("Boss"). Загальна кількість ворогів обчислюється шляхом сумування знайдених об'єктів відповідних категорій.

Окрему увагу приділено аналізу ігрових ресурсів: підраховується загальна кількість скарбів, розташованих на рівні, що дозволяє оцінити потенційні винагороди для агента. Результати обчислень упаковуються в об'єкт StateMetrics, який містить такі ключові метріки рівня:

– FloorCount, це кількість прохідних клітин;
– PathValid, логічний прапор існування зв’язного шляху;
– EnemyCount, загальна кількість ворогів (включаючи босів);
– TreasureCount, кількість скарбів.

На рисунку 3.2 наведено код реалізації методу ComputeStateMetrics.

```
public class QLearningRunner : MonoBehaviour
    StateMetrics ComputeStateMetrics()
    {
        int[,] mapData = level.GetMap();

        int floorCount = 0;

        for (int x = 0; x < mapData.GetLength(0); x++)
        {
            for (int y = 0; y < mapData.GetLength(1); y++)
            {
                if (mapData[x, y] == 1)
                    floorCount++;
            }
        }

        bool pathValid = level.PathExists(level.start, level.end);

        var enemies = GameObject.FindGameObjectsWithTag("Enemy");
        var boss = GameObject.FindGameObjectWithTag("Boss");
        int totalEnemies = enemies.Length + (boss ? 1 : 0);

        int treasureCount = CountTreasuresInTilemap();

        return new StateMetrics
        {
            FloorCount = floorCount,
            PathValid = pathValid,
            EnemyCount = totalEnemies,
            TreasureCount = treasureCount
        };
    }
```

Рисунок 3.2 – Метод визначення стану рівня (зроблено самостійно)

Метод GetPossibleActions виконує критичну функцію у процесі адаптації рівня відповідно до поточного стану карти, визначеного метриками StateMetrics.

Основне його завдання – сформувати список можливих дій для модифікації рівня, що сприятимуть оптимізації ігрового середовища для навчання агента Q-learning. На рисунку 3.3 наведено код реалізації методу GetPossibleActions.

```
public class QLearningRunner : MonoBehaviour

    private List<int> GetPossibleActions(StateMetrics metrics)
    {
        List<int> actions = new List<int>();

        float floorRatio = (float)metrics.FloorCount / (level.width * level.height);

        actions.Add(floorRatio < 0.5f
            ? (int)LevelActions.RandomWallRemove
            : (int)LevelActions.SmoothMap);

        if (!metrics.PathValid) actions.Add((int)LevelActions.ForcePath);
        if (metrics.EnemyCount < 5) actions.Add((int)LevelActions.RespawnEnemies);
        if (metrics.TreasureCount < 3) actions.Add((int)LevelActions.RespawnTreasures);

        if (actions.Count == 0)
        {
            actions.AddRange(new[]
            {
                (int)LevelActions.SmoothMap,
                (int)LevelActions.RandomWallRemove,
                (int)LevelActions.ForcePath
            });
        }

        return actions;
    }
```

Рисунок 3.3 – Отримання можливих дій для агента (зроблено самостійно)

На початковому етапі обчислюється співвідношення прохідних клітин до загальної площі рівня. Якщо воно менше 50%, це вказує на надмірну закритість рівня, тому до списку можливих дій додається RandomWallRemove, яка випадковим чином видаляє частину стін. В іншому випадку використовується SmoothMap, що згладжує карту, роблячи її більш збалансованою.

Після цього перевіряється, чи існує зв’язний шлях між початковою та кінцевою точками. Якщо його немає, включається дія ForcePath, що примусово створює прохід.

Аналізується також кількість ворогів і скарбів. Якщо кількість ворогів менша за п’ять, додається дія RespawnEnemies, яка збільшує їхню присутність на рівні.

Якщо ж скарбів менше трьох, активується RespawnTreasures, що розміщує додаткові об’єкти нагороди.

У випадку, якщо жодна з умов не була виконана і список можливих дій залишився порожнім, до нього додається базовий набір операцій, який включає згладжування рівня, випадкове видалення стін та примусове створення шляху. Це гарантує, що завжди буде доступний хоча б один варіант модифікації рівня.

Наступний метод ApplyAction виконує безпосереднє застосування вибраної дії до рівня, змінюючи його структуру відповідно до поточних потреб системи. Він отримує вхідний параметр action, який є цілим числом, і приводить його до відповідного значення переліку LevelActions. Наступним кроком за допомогою конструкції switch виконується вибір і виклик відповідної функції для внесення змін до рівня. На рисунку 3.4 наведено код реалізації методу ApplyAction:

```
public class QLearningRunner : MonoBehaviour
    Frequently called  1 usage  Hukuta K. *
    void ApplyAction(int action)
    {
        var levelAction = (LevelActions)action;

        switch (levelAction)
        {
            case LevelActions.SmoothMap:
                level.SmoothMapExternally();
                break;
            case LevelActions.RandomWallRemove:
                level.RandomWallRemoveExternally();
                break;
            case LevelActions.ForcePath:
                level.ForcePath();
                break;
            case LevelActions.RespawnEnemies when enemySpawner != null:
                enemySpawner.RespawnEnemies();
                break;
            case LevelActions.RespawnTreasures when treasureSpawner != null:
                treasureSpawner.SpawnTreasures();
                break;
        }
    }
```

Рисунок 3.4 – Метод обрання дії до рівня (зроблено самостійно)

Якщо обраною дією є SmoothMap, викликається метод SmoothMapExternally, що виконує згладжування карти. У випадку RandomWallRemove активується

RandomWallRemoveExternally, що видаляє випадкові стіни, збільшуючи прохідність рівня.

Коли дією є ForcePath, застосовується метод ForcePath, що забезпечує наявність зв'язного шляху між стартовою та фінальною точками рівня.

Якщо вибрана дія RespawnEnemies, і при цьому існує відповідний генератор ворогів, відбувається виклик RespawnEnemies, що відновлює або додає ворогів на рівні. Аналогічно, якщо дією є RespawnTreasures, і є доступний генератор скарбів, активується метод SpawnTreasures, що розміщує скарби на карті.

Таким чином, метод ApplyAction є ключовим механізмом виконання змін у середовищі, дозволяючи агенту Q-learning безпосередньо впливати на структуру рівня та адаптувати його відповідно до вибраних стратегій навчання.

Після цього іде метод CalculateReward, який відповідає за формування числової оцінки (нагороди), що визначає якість поточного стану рівня з точки зору навчання агента Q-learning. Ця оцінка використовується для коригування стратегії агента та спрямування його до оптимальних конфігурацій карти. На рисунку 3.5 наведено код методу CalculateReward.

```
Frequently called   1 usage   Hukuta K.
float CalculateReward(StateMetrics metrics)
{
    bool pathValid = metrics.PathValid;

    float floorRatio = (float)metrics.FloorCount / (level.width * level.height);
    float floorScore = floorRatio is >= 0.4f and <= 0.6f ? 1.0f : 0.0f;

    float enemyScore = (metrics.EnemyCount > 0) ? 0.5f : 0.0f;
    float treasureScore = (metrics.TreasureCount > 0) ? 1.0f : 0.0f;

    float reward = (pathValid ? 1.0f : 0.0f) * (floorScore + enemyScore + treasureScore);
    return reward;
}
```

Рисунок 3.5 – Метод формування числової винагороди (зроблено самостійно)

На початковому етапі метод отримує об'єкт StateMetrics, який містить основні характеристики рівня, та визначає, чи існує зв'язний шлях між початковою

та фінальною точками карти. Цей параметр (pathValid) відіграє ключову роль у розрахунку винагороди.

Наступним кроком обчислюється частка прохідних клітин (floorRatio), яка визначається як відношення кількості прохідних клітин до загальної площі рівня. Якщо значення цього показника знаходиться в межах від 0.4 до 0.6, рівень вважається збалансованим, і відповідний коефіцієнт floorScore отримує значення 1.0. В інших випадках цей параметр дорівнює 0.0, оскільки надмірно закриті або занадто відкриті рівні можуть ускладнювати навчання агента.

Оцінка кількості ворогів враховується через параметр enemyScore. Якщо на рівні присутні вороги (metrics.EnemyCount > 0), агент отримує 0.5 бала, що стимулює створення загроз у навколишньому середовищі.

Аналогічно, кількість скарбів впливає на treasureScore. Якщо на рівні є хоча б один скарб (metrics.TreasureCount > 0), агент отримує 1.0 бала, що заохочує створення винагород у просторі рівня.

Фінальне значення винагороди розраховується шляхом множення булевого значення pathValid (1.0 або 0.0) на суму всіх отриманих балів (floorScore + enemyScore + treasureScore). Запропонований підхід забезпечує, що рівень буде отримувати позитивну винагороду лише в тому випадку, якщо існує шлях між стартовою та фінальною точкою.

Метод повертає значення reward, яке агент використовує для оцінки ефективності своїх дій у рамках навчання.

Важливим методом при реалізації Q-learning є метод GetAction, який реалізує вибір дії для агента в середовищі навчання. Його основне завдання – визначити, яку саме дію слід виконати в поточному стані рівня на основі стратегії агенту.

Процес вибору дії базується на ε-жадібній стратегії (epsilon-greedy), що дозволяє балансувати між дослідженням нових варіантів (exploration) та експлуатацією вже відомих оптимальних рішень (exploitation).

На першому етапі метод генерує випадкове число в діапазоні [0,1] та порівнює його з параметром epsilon. Якщо значення менше за epsilon, агент

здійснює випадковий вибір дії з доступного списку possibleActions, що стимулює дослідження середовища.

Якщо ж випадкове число перевищує epsilon, агент обирає найкращу відому дію на основі значень Q-функції. Для цього здійснюється перебір всіх можливих дій із набору possibleActions, і для кожної з них отримується значення Q-функції за допомогою GetQValue. Дія, яка має найвище значення Q, визначається як оптимальна та повертається методом.

Запропонований підхід дозволяє агенту одночасно накопичувати знання про середовище та використовувати вже здобутий досвід для ефективного навчання. На рисунку 3.6 зображено код реалізації метода GetAction.

```
public class QLearningAgent

    Frequently called  1 usage  Hukuta K.
    public int GetAction(int state, List<int> possibleActions, System.Random rand)
    {
        if (rand.NextDouble() < epsilon)
        {
            return possibleActions[rand.Next(possibleActions.Count)];
        }

        float maxQ = float.MinValue;
        int bestAction = possibleActions[0];
        foreach (int action in possibleActions)
        {
            float q = GetQValue(state, action);
            if (q > maxQ)
            {
                maxQ = q;
                bestAction = action;
            }
        }
        return bestAction;
    }
```

Рисунок 3.6 – Метод вибору дії агента (зроблено самостійно)

Для того щоб об’єднати усі описані вище методи, був реалізован метод RunEpisode. Він виконує основний цикл навчання агента Q-learning, забезпечуючи послідовне оновлення його стратегії на основі отриманих метрик стану рівня. Головна функція цього методу - організація одного епізоду взаємодії агента з

ігровим середовищем, що включає вибір дії, модифікацію рівня та коригування Q-таблиці. На рисунку 3.7 наведена реалізація методу RunEpisode.

```
public class QLearningRunner : MonoBehaviour
    Frequently called   1 usage   Hukuta K.
    void RunEpisode()
    {
        var currentMetrics = ComputeStateMetrics();
        int currentStateKey = ComputeStateKey(currentMetrics);

        List<int> actions = GetPossibleActions(currentMetrics);

        int chosenAction = agent.GetAction(currentStateKey, actions, rand);

        ApplyAction(chosenAction);

        var newMetrics = ComputeStateMetrics();
        int newStateKey = ComputeStateKey(newMetrics);

        float reward = CalculateReward(newMetrics);

        agent.UpdateQ(currentStateKey, chosenAction, newStateKey, reward, GetPossibleActions(newMetrics));
    }
```

Рисунок 3.7 – Метод основного циклу навчання (зроблено самостійно)

На першому етапі метод отримує поточний стан рівня, викликаючи ComputeStateMetrics для аналізу його структурних характеристик. Потім цей стан кодується в компактне числове представлення за допомогою ComputeStateKey, що дозволяє агенту ефективно працювати з дискретними станами середовища.

Наступний крок це визначення доступних дій для модифікації рівня, які повертає функція GetPossibleActions. Використовуючи цей набір дій, агент обирає оптимальну дію chosenAction через метод GetAction. Вибір здійснюється з урахуванням поточного стану рівня, можливих альтернатив і стратегічних налаштувань агента.

Після визначення дії вона застосовується до рівня методом ApplyAction, що змінює його структуру відповідно до вибраного механізму корекції.

Наступним кроком проводиться повторний аналіз оновленого стану рівня через ComputeStateMetrics, а також перерахунок нового ключа стану (ComputeStateKey). Це дозволяє агенту оцінити наслідки здійсненої дії.

Ключовим етапом є розрахунок винагороди (reward) за новий стан, що виконується функцією CalculateReward.

Отримане значення використовується для оновлення Q-таблиці агента методом UpdateQ, який виконує оновлення значень Q-функції відповідно до алгоритму Q-learning. Його основна мета – коригування оцінки корисності дій агента, що дозволяє покращити прийняття рішень у наступних епізодах навчання.

На першому етапі метод отримує значення Q-функції для попереднього стану oldState і вибраної дії action за допомогою GetQValue [15].

Після цього визначається максимальне Q-значення для наступного стану newState. Для цього виконується перебір усіх можливих наступних дій possibleNextActions, після чого обирається максимальне значення функції GetQValue.

Оновлене значення Q-функції розраховується відповідно до формули, яка була наведена вище. Отримане значення зберігається у qTable, яка виступає як основне сховище знань агента. На рисунку 3.8 відображена реалізація методів UpdateQ та GetQValue.

```
Frequently called   1 usage   Hukuta K.
public void UpdateQ(int oldState, int action, int newState, float reward, List<int> possibleNextActions)
{
    float oldQ = GetQValue(oldState, action);
    float maxNextQ = float.MinValue;
    foreach (int nextAction in possibleNextActions)
    {
        float q = GetQValue(newState, nextAction);
        if (q > maxNextQ) maxNextQ = q;
    }

    float updatedQ = oldQ + learningRate * (reward + discountFactor * maxNextQ - oldQ);
    qTable[(oldState, action)] = updatedQ;
}

Frequently called   3 usages   Hukuta K.
private float GetQValue(int state, int action)
{
    if (!qTable.ContainsKey((state, action)))
    {
        qTable[(state, action)] = 0f;
    }
    return qTable[(state, action)];
}
```

Рисунок 3.8 – Реалізація методів для оновлення Q-значення (зроблено самостійно)

Таким чином, метод UpdateQ є критичним компонентом Q-learning, забезпечуючи поступове вдосконалення прийняття рішень агентом у процесі навчання [18].

# 4 АНАЛІЗ РЕЗУЛЬТАТІВ І ОЦІНКА РОБОТИ СИСТЕМИ

## 4.1 Результати роботи алгоритму

Під час експериментів із застосуванням Q-Learning було проведено декілька запусків, щоб оцінити, як налаштування (ε, швидкість навчання та коефіцієнт дисконтування) впливають на якість та швидкість генерації рівнів. У ході навчання алгоритм опрацював певну кількість епізодів (зазвичай від 50 до 300, залежно від експерименту). При виконанні 60–70 епізодів стали виявлятися початкові закономірності в оновленні Q-таблиці. При підвищеній швидкості навчання (наприклад, 0.1–0.5) агент швидко змінював значення Q, але іноді це призводило до менш стабільної збіжності, оскільки великі корегування підсилювали випадкові фактори. Навпаки, з меншою швидкістю навчання (близько 0.01–0.05) алгоритм концентрувався на послідовнішому вдосконаленні, хоч і потребував більшої кількості ітерацій.

Важливо було також налаштувати ε (ймовірність випадкової дії), оскільки на початку експериментів велике ε (0.3–0.5) допомагало дослідити різні варіанти дій і знаходити кращі стратегії генерації. Однак у фінальній стадії навчання занадто високе ε перешкоджає експлуатації накопичених знань і може збивати агент із уже знайдених хороших рішень. Емпірично було підібрано ε близько 0.3, що дозволило зберегти баланс між пошуком нових стратегій і стабільним впровадженням поточного знання. Коефіцієнт дисконтування γ (0.9) виявився достатнім, аби агент враховував цінність дій із певною перевагою майбутніх винагород, але не ігнорував важливість поточних результатів.

За період навчання відповідні значення функції винагороди поступово зростали, що свідчило про засвоєння агентом закономірностей якіснішої генерації поверхонь, зв’язних шляхів та розміщення ворогів із скарбами. У ряді експериментів агент досягав стабільного приросту винагороди після 100-120 епізодів, в інших потребував до 170-190. При встановленні бажаного рівня зв’язності карти та збалансованості розподілу ворогів і скарбів допустимо припинення тренувального процесу або коригування параметрів.

Щоб максимально розкрити вплив впровадження Q-learning, рисунок 4.1 ілюструє початкову конфігурацію рівня, де випадковий розподіл елементів призводить до нерегулярного формування стін із як хаотичними, так і надмірно щільними структурами, а жоден алгоритм ще не виконував оптимізаційних корекцій.

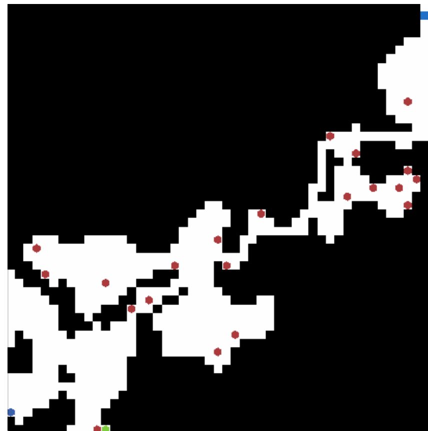

Рисунок 4.1 – Згенерований рівень клітинним автоматом (зроблено самостійно)

На противагу цьому, рисунок 4.2 дасть змогу оцінити, як унаслідок навчання агент поступово сформував зв’язні шляхи, відкоригував щільність стін, а також відрегулював кількість і місця розташування ворожих істот та скарбів на тому же самому рівні що і на прошлому рисунку.

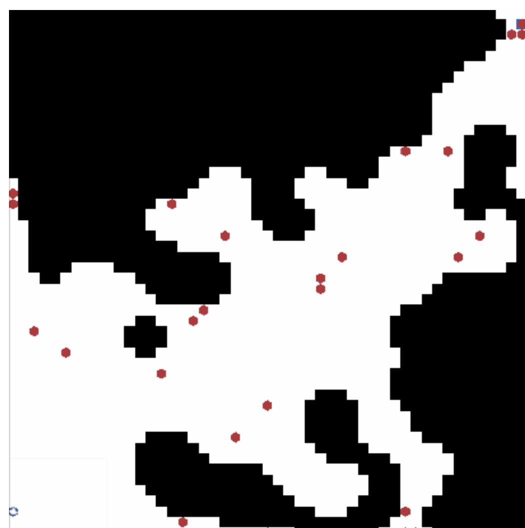

Рисунок 4.2 – Рівень на який було застосовано Q-learning (зроблено самостійно)

Така візуалізація наочно відображає поступове вдосконалення структури рівнів та збалансованість ігрових об’єктів, підтверджуючи доцільність і ефективність алгоритму Q-Learning у задачах процедурної генерації.

## 4.2 Оцінка роботи системи

У середовищі Unity згенеровані рівні виявилися достатньо зв'язними завдяки використанню процедури перевірки шляху [10]. Крім того, система змогла автоматично підлаштовувати складність рівня за рахунок операцій SmoothMap і RandomWallRemove, у ситуаціях з недостатнім простором для переміщення алгоритм розширював прохідність, тоді як при надмірній кількості вільного простору застосовувалося згладжування. Ворогів та скарби розташовувалося пропорційно до розміру карти й рівня складності, що підтвердило збалансованість навченої моделі. Система також демонструвала адаптивність: у разі зміни початкових налаштувань чи умов рівня (наприклад, коли відбувається сильне ущільнення стін) вона поступово змінювала політику дій та знаходила нові варіанти генерації і розташування об'єктів.

Серед сильних сторін даного підходу слід відзначити його гнучкість, що забезпечується можливістю регулювання параметрів, таких як швидкість навчання, $\varepsilon$ і $\gamma$, здатність до адаптації (агент вчиться з урахуванням поточної структури карти), а також потенційну автоматизацію оптимізації рівнів без ручного налаштування. Серед обмежень – відносно високі обчислювальні витрати (при великій карті й складних діях алгоритм може потребувати чимало симуляцій), а також необхідність коректно налаштувати гіперпараметри, щоб уникнути надто тривалого навчання або залипання на локальних стратегіях. Загалом, результати експериментів свідчать, що Q-learning є ефективним інструментом для процедурної генерації рівнів, дозволяючи створювати ігрові локації зі збалансованою складністю та різноманітною структурою. Подальше вдосконалення підходу може бути досягнуто шляхом інтеграції Q-learning з іншими методами, наприклад, із використанням нейронних мереж для більш гнучкого представлення станів, а також шляхом врахування додаткових критеріїв якості, таких як ігрові метрики або поведінкові патерни гравця.

Крім того, було виявлено, що зміна нагородної функції суттєво впливає на характер згенерованих рівнів: більш жорстке покарання за тупикові зони

стимулювало створення просторіших і більш прохідних локацій, тоді як заохочення за збереження структурованості призводило до формування рівнів з чітко вираженими маршрутами та зонами ризику. Це підтверджує, що навіть прості зміни у формулюванні цілей агента дозволяють гнучко контролювати архітектурні риси майбутньої карти, не змінюючи саму структуру алгоритму.

### 4.3 Обмеження та недоліки моделі

Незважаючи на продемонстровану ефективність, запропонована модель має низку обмежень. По-перше, класичний Q-learning погано масштабується на великі простори станів: зі зростанням розміру карти або кількості можливих дій обсяг таблиці Q значно збільшується, що призводить до зростання обчислювальних витрат і часу навчання. У складних середовищах це може зробити процес генерації надмірно повільним або нестабільним. По-друге, представлення станів у вигляді дискретних комірок не завжди дозволяє моделі уловлювати глобальні закономірності чи просторові шаблони, які важливі для цілісної структури рівня.

Ще одним недоліком є чутливість до налаштувань гіперпараметрів. Невдале значення коефіцієнтів $\varepsilon$, $\gamma$ або швидкості навчання $\alpha$ може призвести до повільної або неефективної генерації, переобучення, або ж до формування тривіальних шаблонів рівнів. Крім того, відсутність пам’яті про попередні епізоди (на відміну від, наприклад, Deep Q-Networks) унеможливлює узагальнення отриманого досвіду на нові ситуації без повторного навчання.

Також слід зазначити, що модель оцінює рівні з огляду лише на задану функцію винагороди, яка формалізує критерії якості з технічної точки зору, але не враховує суб’єктивні аспекти, такі як цікавість геймплею або задоволення користувача. Це обмежує придатність моделі для створення рівнів у комерційних або креативно орієнтованих проєктах без подальшого втручання дизайнера.

Окремо варто згадати, що класичний підхід Q-learning не враховує просторову топологію рівня як графа. У випадках, коли важливими є зв’язки між кімнатами, вузлами або об’єктами, доцільним є використання Graph Q-learning або

поєднання Q-learning з графовими нейронними мережами (GNN). Такі методи дозволяють агенту оперувати не лише координатами або сітками, а й структурованими представленнями карти, що підвищує якість генерації при складній логіці зв’язків. Однак реалізація таких моделей є значно складнішою та потребує суттєвих обчислювальних ресурсів і обґрунтованого проєктування графової структури станів.

# ВИСНОВКИ

Проведене дослідження методів машинного навчання для оптимізації процедурної генерації рівнів у розробці ігор дозволило глибоко проаналізувати як традиційні алгоритмічні підходи, так і сучасні адаптивні технології, зокрема методи навчання з підкріпленням, серед яких особливу увагу було приділено алгоритму Q-learning. Розроблена модель успішно продемонструвала здатність адаптувати структуру ігрового середовища, забезпечуючи оптимізацію таких параметрів, як прохідність, баланс між елементами рівня, а також розташування ворогів і скарбів, що сприяє підвищенню якості кінцевого продукту. Практична реалізація моделі засвідчує ефективність застосування Q-learning для автоматизації процесу генерації рівнів, проте також виявляє певні виклики, пов’язані з високими обчислювальними витратами та необхідністю ретельного налаштування гіперпараметрів. Отримані результати свідчать про значний потенціал подальшого вдосконалення методики, зокрема шляхом інтеграції з іншими сучасними підходами, що може сприяти більш гнучкому представлення станів середовища та підвищенню адаптивності системи до динамічних умов ігрового процесу. Таким чином, робота не лише розкриває теоретичні аспекти проблематики, а й пропонує практичне рішення, яке може стати базою для подальших досліджень у сфері автоматизованої генерації контенту для інтерактивних розважальних продуктів.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Advancements in Generative AI: A Comprehensive Review of GANs, GPT, Autoencoders, Diffusion Model, and Transformers / S. Bengesi та ін. arXiv.org. [Електронний ресурс] URL: https://arxiv.org/abs/2311.10242 (дата звернення: 16.04.2025).

2. Wave Function Collapse for procedural generation in Unity. *PVS-Studio*. URL: https://pvs-studio.com/en/blog/posts/csharp/1027/ (дата звернення: 20.04.2025).

3. Marching Squares, a Unity C# Tutorial. *Catlike Coding*. URL: https://catlikecoding.com/unity/tutorials/marching-squares/ (дата звернення: 20.04.2025).

4. Whiteway E. Using a Generative Adversarial Network to author playable Super Mario Bros. levels. *Medium*. URL: https://medium.com/@eway/using-a-generative-adversarial-network-to-author-playable-super-mario-bros-levels-223736bd4ce (дата звернення: 22.04.2025).

5. Yannakakis G. Deep learning for procedural content generation. Georgios N. Yannakakis. [Електронний ресурс] URL: https://yannakakis.net/wpcontent/uploads/2021/03/liu2021deep.pdf (дата звернення: 24.04.2025).

6. An Introduction to Convolutional Neural Networks (CNNs). datacamp. [Електронний ресурс] URL: https://www.datacamp.com/tutorial/introduction-toconvolutional-neural-networks-cnns (дата звернення: 25.04.2025).

7. Shin S. Game Level Generation Using Neural Networks. *Game Developer | Game Industry News, Deep Dives, and Developer Blogs*. URL: https://www.gamedeveloper.com/programming/game-level-generation-using-neural-networks (дата звернення: 27.04.2025).

8. Franceschelli G., Musolesi M. Reinforcement Learning for Generative AI: State of the Art, Opportunities and Open Research Challenges. arXiv.org. [Електронний ресурс] URL: https://arxiv.org/abs/2308.00031 (дата звернення: 30.04.2025).

9. Qiu Y. Research on Path Planning Based on Q-learning. ResearchGate. [Електронний ресурс] URL: https://www.researchgate.net/publication/387710642_Research_on_Path_Planning_Based_on_Q-learning. (дата звернення: 04.05.2025).

10. Unity - Scripting API: UnityEngine.AIModule. [Електронний ресурс] URL: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/UnityEngine.AIModule.html (дата звернення: 08.05.2025).

11. Sutton R. S., Barto A. G., Bach F. Reinforcement Learning: An Introduction. MIT Press, 2018. 552 c.

12. Mennen V. Machine Learning Basics: a Comprehensive Guide about Machine Learning: Machine Learning. Independently Published, 2021.

13. Bhasin H. Hands-on Deep Learning. Berkeley, CA : Apress, 2024. URL: https://doi.org/10.1007/979-8-8688-1035-0 (дата звернення: 13.05.2025).

14. Russell S. J., Norvig P. Artificial Intelligence: A Modern Approach. Pearson Education, Limited, 2010. 1151 c.

15. Thorn A. Mastering Unity Scripting. Packt Publishing, 2015. 380 c.

16. Nandy A., Biswas M. Neural Networks in Unity: C# Programming for Windows 10. Apress, 2018. 172 c.

17. Smith M., Queiroz C. Unity 5.x Cookbook. Packt Publishing - ebooks Account, 2015. 493 c.

18. Github MykytaKovalov-V URL: https://github.com/MykytaKovalov-V/2025_M_PI_IPZm-23-3_Kovalov_M_V (дата звернення 15.05.2025)

19. A. Khovrat and V. Kobziev, "Using Recurrent and Convulation Neural Networks to Indentify the Fake Audio Messages," 2023 IEEE 7th International Conference on Methods and Systems of Navigation and Motion Control (MSNMC), Kyiv, Ukraine, 2023, pp. 174-177, doi: 10.1109/MSNMC61017.2023.10329236.

20. H. Falatiuk, M. Shirokopetleva and Z. Dudar, "Investigation of Architecture and Technology Stack for e-Archive System," 2019 IEEE International Scientific Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), Kyiv, Ukraine, 2019, pp. 229-235, doi: 10.1109/PICST47496.2019.9061407.

## ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

19. A. Khovrat and V. Kobziev, "Using Recurrent and Convulation Neural Networks to Indentify the Fake Audio Messages," 2023 IEEE 7th International Conference on Methods and Systems of Navigation and Motion Control (MSNMC), Kyiv, Ukraine, 2023, pp. 174-177, doi: 10.1109/MSNMC61017.2023.10329236.

20. H. Falatiuk, M. Shirokopetleva and Z. Dudar, "Investigation of Architecture and Technology Stack for e-Archive System," 2019 IEEE International Scientific Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), Kyiv, Ukraine, 2019, pp. 229-235, doi: 10.1109/PICST47496.2019.9061407.