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

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук, проф.
_____Ю. П. Кондратенко
«____»_____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

ІНТЕЛЕКТУАЛЬНА СИСТЕМА АДАПТАЦІЇ ПАРАМЕТРІВ ІГРОВОГО ПРОСТОРУ ДЛЯ КОМП'ЮТЕРНИХ ІГОР

Спеціальність 122 «Комп'ютерні науки»

122 – КРМ – 601.21810325

Виконав студент 6-го курсу, групи 601
_____І. І. Чернов
«19» лютого 2024 р.

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

Миколаїв – 2024

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

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

ЗАТВЕРДЖУЮ
Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____ Ю. П. Кондратенко
«____» _____20___ р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Чернову Іллі Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи магістра «Інтелектуальна система адаптації параметрів ігрового простору для комп'ютерних ігор».

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

Затв. наказом Ректора ЧНУ ім. Петра Могили від «19» жовтня 2023 р. № 201

2. Строк подання студентом роботи 19 лютого 2024 р.

3. Вхідні (початкові) дані до роботи: набір даних для навчання системи адаптації, середовище Unity з розробленою системою адаптації. Очікуваний результат: система адаптації параметрів ігрового простору.

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

- дослідження історії ігрового дизайну, аналіз рівнів складності в ігрових застосунках;
- аналіз використання методів штучного інтелекту в іграх;
- опис систем адаптації параметрів ігрового простору.

5. Перелік графічного матеріалу: презентація.

6. Завдання до спеціальної частини: Аналіз робочих умов та санітарно – гігієнічних вимог в офісному приміщенні.

7. Консультанти:

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

Керівник роботи __д-р техн. наук, проф. Гожий О. П._____
(наук. ступінь, вчене звання, прізвище та ініціали)

_____
(підпис)

Завдання прийнято до виконання ___Чернов І. І._____
(прізвище та ініціали)

_____
(підпис)

Дата видачі завдання «__31__» _____жовтня____ 2023 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи магістра

Тема: Інтелектуальна система адаптації параметрів ігрового простору для комп'ютерних ігор

№	Найменування роботи	Початок	Закінчення	Примітки
1	Визначення керівника і теми КРМ. Подання заяви на затвердження теми КРМ	01.09.2023	10.10.2023	Виконано
2	Отримання завдання на виконання КРМ	11.10.2023	01.11.2023	Виконано
3	Складання календарного плану на період виконання КРМ	02.11.2023	10.11.2023	Виконано
4	Огляд літератури за темою дослідження	11.11.2023	26.11.2023	Виконано
5	Проходження передатестаційної практики, збір та аналіз матеріалів до КРМ	27.11.2023	23.12.2023	Виконано
6	Аналіз предметної області та розробка технічного завдання. Моделювання результатів	25.12.2023	12.01.2024	Виконано
7	Опис фахової частини КРМ, зокрема дослідження систем динамічного налаштування складності, реалізація обраних технологій з аналізом отриманих результатів	13.01.2024	25.01.2024	Виконано
8	Розробка спеціальної частини з охорони праці та методичної частини	26.01.2024	02.02.2024	Виконано
9	Перший попередній захист КРМ на засіданні комісії кафедри	29.01.2024	29.01.2024	Виконано
10	Корегування роботи за результатами попереднього захисту	30.01.2024	05.02.2024	Виконано
11	Доробка та остаточне оформлення КРМ	06.02.2024	11.02.2024	Виконано
12	Другий попередній захист КРМ на засіданні комісії кафедри	12.02.2024	12.02.2024	Виконано
13	Подання КРМ, її електронної копії та інших документів (відгуку, рецензії) до захисту	19.02.2024	20.02.2024	Виконано
14	Захист КРМ перед екзаменаційною комісією (ЕК)	26.02.2024	27.02.2024	Виконано

Розробив студент Чернов І. І. _____
(прізвище та ініціали) (*підпис*)

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

«09» листопада 2023 р.

АНОТАЦІЯ

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

Чернова Іллі Ігоровича

на тему: **“ІНТЕЛЕКТУАЛЬНА СИСТЕМА АДАПТАЦІЇ ПАРАМЕТРІВ ІГРОВОГО ПРОСТОРУ ДЛЯ КОМП’ЮТЕРНИХ ІГОР”**

Актуальність даної системи полягає у потребі вирішити одну із задач ігрового дизайну для забезпечення кращого ігрового досвіду для користувачів. Задача полягає у підборі правильного рівня складності під кожного гравця окремо. Вирішення даної проблеми сприятиме розширенню цільової аудиторії розробників, що відкриє нові можливості для просування їхніх продуктів і призведе до збільшення фінансових надходжень.

Об’єктом дослідження є процес класифікації рівня навичок гравців та зміна параметрів ігрового простору.

Предметом дослідження є методи та алгоритми класифікації рівня навичок гравців.

Метою дослідження є створення інтелектуальної системи адаптації ігрового простору.

В результаті виконання роботи було проведено дослідження підходів розробників до проблеми складності в відео-іграх, проаналізовано деякі із жанрів ігрових застосунків у тому числі і rogue-подібні ігри, на прикладі яких демонструвалась робота системи. Було проведено аналіз наявних аналогів, наведені шляхи вирішення проблеми та розроблено відповідну систему для адаптації параметрів ігрового простору.

Дана робота складається із трьох розділів. У першому розділі йде опис та аналіз предметної сфери, у другому – опис використаних методів та інструментів для розробки системи, у третьому – описаний процес розробки та тестування системи, наведені результати її роботи. Загальний обсяг роботи – 118 сторінок. Кваліфікаційна робота магістра складається із одного додатку, 30 рисунків, 1 таблиці і посилання на 32 літературних джерел.

Ключові слова: штучний інтелект, нейронні мережі, багатошаровий перцептрон, ігрові застосунки, ігровий дизайн.

ABSTRACT

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

Chernov Illia

“INTELLIGENT GAME SPACE PARAMETER ADAPTATION SYSTEM FOR COMPUTER GAMES”

The relevance of this system lies in the need to solve one of the problems of game design to ensure a better gaming experience for users. The task is to select the right level of difficulty for each player individually. Solving this problem will contribute to the expansion of the target audience of developers, which will open up new opportunities for the promotion of their products and will lead to an increase in financial income.

The object of the study is the process of classifying the skill level of players and changing the parameters of the game space.

The subject of research are methods and algorithms for classifying the skill level of players.

The purpose of the research is to create an intelligent system for adapting the game space.

As a result of the work, a study of developers' approaches to the problem of complexity in video games was conducted, some of the genres of game applications were analyzed, including roguelike games, on the example of which the work of the system was demonstrated. An analysis of existing analogues was carried out, ways to solve the problem were given, and a suitable system was developed for adapting the parameters of the game space.

This work consists of three sections. The first section contains a description and analysis of the subject area, the second section describes the methods and tools used for system development, and the third section describes the process of system development and testing, as well as the results of its work. The total volume of work is 118 pages. The master's qualification work consists of one appendix, 30 figures, 1 table and references to 32 literary sources.

Key words: artificial intelligence, neural networks, multilayer perceptron, game applications, game design.

ЗМІСТ

ВСТУП

В наш час все більше людей починають цікавитись відеоіграми. І причина цьому цілком зрозуміла. З розвитком комп’ютерних технологій розробники відеоігор навчились створювати проєкти, які по графічному стилю не відстають від великих кіно-блокбастерів. Разом із цим відеогра являє собою мультимедійний продукт з яким користувач активно взаємодіє не як спостерігач, як це відбувається в фільмах, а як безпосередній учасник подій.

Але відеоігри мають не тільки розважальну цінність. Так створюються різноманітні симулятори. Це можуть бути симулятори для водіння різноманітного автомобільного транспорту, управління потягами, літаками, морськими судами. Окрім засобів пересування та транспортування існують симулятори, які навчають створенню певних систем, наприклад, створення транспортної системи для перевезень вантажу чи людей, системи керування польотами і т. п.

Окрім симуляторів створюються також розвиваючі проєкти. Ігри-головоломки позитивно впливають на когнитивні здібності людського мозку.

Але загалом всі відеоігрові застосунки поєднує одне – в них цікаво грати та вони можуть затягнути гравця на безліч годин. Це досягається за допомогою підтримання певного рівня складності гри. Навіть в так званих казуальних іграх є деякі системи, які створюють певний виклик для користувача.

Є безліч проєктів з різною складністю для будь-якого користувача, але немає тих, які б динамічно пристосовувались під нього. Зазвичай можна обрати якусь одну складність, або вона взагалі тільки одна, та почати гру, з часом крива складності постійно зростає до певного максимуму, який визначили розробники. Це може призводити до ситуацій, де гравець потрапляє у ситуацію з якої деякий час не може вийти, так званий софтлок. Цю проблему можна було вирішити динамічним налаштуванням ігрового середовища таким чином, щоб гравець зміг просунутись далі. Ігрове середовище означає ворогів, мапу, їх складність, предмети, які знаходить користувач.

Актуальність даної системи полягає у потребі вирішити одну із задач ігрового дизайну для забезпечення кращого ігрового досвіду для користувачів. Задача полягає у підборі правильного рівня складності під кожного гравця окремо. Вирішення даної проблеми сприятиме розширенню цільової аудиторії розробників, що відкриє нові можливості для просування їхніх продуктів і призведе до збільшення фінансових надходжень.

Об’єктом дослідження є процес класифікації рівня навичок гравців та зміна параметрів ігрового простору.

Предметом дослідження є методи та алгоритми класифікації рівня навичок гравців.

Метою дослідження є створення інтелектуальної системи адаптації ігрового простору.

1 АНАЛІЗ СКЛАДНОСТІ В ВІДЕОІГРАХ ТА СИСТЕМ ЇЇ ДИНАМІЧНОГО НАЛАШТУВАННЯ. ШЛЯХИ ВИРІШЕННЯ ПРОБЛЕМИ

1.1 Підходи розробників до складності в відеоіграх

В наш час ігри мають декілька рівнів складності, які завчасно визначаються розробниками. В деяких проєктах складність можна змінити в будь який момент гри, в деяких – ні. Але раніше, ще в часи аркадних автоматів складність була тільки одна і до того ж найважча. Це було обумовлено тодішньою організацією клубів з ігровими автоматами.

Аркадні ігри часів ігрових автоматів були популярні в 70-80-х роках. Ці ігрові автомати були поширені у багатьох місцях, таких як аркади, торгові центри, кінотеатри та крамниці [1].

Для того щоб почати гру користувачеві необхідно було закинути монетку до ігрового автомата. Саме через цю особливість тогочасних ігрових машин розробники навмисно робили ігри складними, щоб гравець частіше програвав та продовжував витрачати свої монетки на нові спроби.

Але попри це розробники витримували баланс і не робили ігри занадто складними, якщо гравець витрачав деякий час на вивчання рівнів, ворогів та всіх здібностей свого ігрового персонажа то гра ставала набагато простіша але все одно кидала виклик користувачеві [2].

Основи теперішнього дизайну рівнів складності були покладені також у часи аркадних автоматів [3].

У 1978 році гра Space Invaders від розробника Томохіро Нісікадо стала світовою сенсацією. Це була перша гра з новою на той час структурою складності. Вона полягала у тому, що при зменшенні кількості ворогів вони ставали швидшими. Ця структура забезпечила постійний приріст складності із проходженням гри далі [3].

На початку розробки Нісікадо не планував такої системи але виявивши, що з нею гра стала набагато цікавіша, він розробив рівні таким чином, що кожен наступний був трохи складнішим ніж попередній.

Якщо зобразити ріст складності у Space Invaders у вигляді графіка, то вона буде виглядати наступним чином:

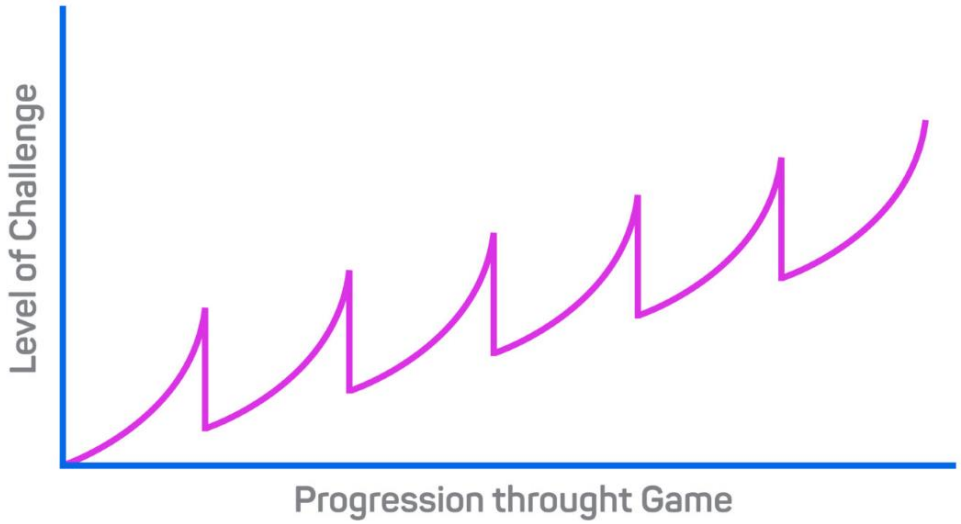

Рисунок 1.1 – Крива складності

Ця концепція використовується розробниками, а точніше геймдизайнерами і в наш час, оскільки вона є основою.

Використання цієї практики із збільшенням рівню складності гри з часом, можна було зробити гру цікавішою без додавання нових елементів до неї. Наприклад, можна було збільшити якісь параметри для ворожих юнітів таких як швидкість пересування чи швидкість атаки, збільшити кількість хітів, які втрачав гравець якщо отримував пошкодження від ворога або пастки.

Наступний крок у геймдизайні було здійснено у 1980 році, з появою гри Pac-Man. Розробники Pac-Man`а додали нову змінну до формули складності – здібності. На мапі рівня розкидані бонуси, які при підбиранні гравцем дають йому певний ефект. Так, наприклад, бонус енерджайзер робить гравця здатним переслідувати та

атакувати привидів, які раніше його переслідували. Навіть якщо не брати до уваги інші бонуси присутні в грі, наявність єдиного енерджайзера повністю змінює поведінку гравця, забезпечуючи новий ігровий досвід [1, 2].

Для балансу здібностей гравця, та підтримання постійного приросту складності розробники зробили так, що бонуси мають менший ефект на гравця зі збільшенням пройдених рівнів. Тобто за допомогою бонусів розробники не спростили гру, а навпаки ускладнили її. Так, на перших декількох рівнях ефект бонусів сильний та спрощує їх проходження, але зі зменшенням їхньої ефективності на пізніх рівнях з'являється тактичний елемент. Гравець має думати чи підбирати йому посилення зараз чи пізніше, він має краще аналізувати рівень, його структуру та місцезнаходження ворогів на ньому.

Згодом розробники зрозуміли, що додавання бонусів до гри можуть слугувати не тільки для збільшення їхньої складності, а й для розширення ігрових можливостей, використовуючи елементи з інших жанрів, та підвищення розважальної цінності відеоігор.

Сігер Міямото був першим розробником, який у своїй грі Donkey Kong, яка в своїй основі є платформером, додав бонус молоток. Цей бонус повністю змінював поведінку гравця, змінюючи його навички [4]. Гравець втрачав можливість стрибати, але натомість отримував здібність вдарити молотком, саме ця особливість змінювала жанр гри з платформеру на екшн.

Найкраще цю ідею імплементувала гра Super Mario Bros. Але ця гра пішла далі, розробники не тільки змішували декілька жанрів, але й створювали такі сценарії, де використовуючи інструменти одного жанру гравець мав вирішити проблему іншого. Наприклад, використання вогняних куль для зачистки платформи на яку необхідно перестрибнути для продовження проходження рівня.

Змішування жанрів дає розробникам можливість ускладнювати гру для підтримання інтересу гравця та разом із цим урізноманітнити ігровий процес створюючи нові сценарії.

Перехід від одного набору навичок до іншого в момент, коли гравцю стає трохи нудно, призводить до стану потоку [5]. Це психологічний стан, який викликає ейфорію у гравців, коли вони абсолютно занурені у гру. Більшість гравців згадують моменти, коли вони були настільки втягнуті у гру, що втратили відчуття часу. Такі відчуття характерні не лише для геймерів. Бігуни, ремісники, музиканти та багато інших можуть відчути це саме, коли занурюються у свою улюблену "медитативну" діяльність. Змішані ігри можуть занурити гравців у психологічний стан, який можна назвати змішаним потоком. Гравець тимчасово фокусується на одному завданні, а потім переходить до іншого. Якщо гра добре пророблена, вона перемикається між наборами навичок, що дозволяє тривалий час зберігати ейфорію потоку, навіть коли гравець починає втомлюватися.

З появою 3D ігор та збільшення кількості зацікавлених в ігровій індустрії розробникам довелось дещо змінити підхід до складності в своїх проєктах. Довелось відмовитись від звичного єдиного рівня складності та створити так звані рівні, які всім нам відомі – легкий, середній, складний [6]. Звичайно рівнів складності може бути і більше, в залежності від гри, але базові три рівня завжди будуть присутні. Це було зумовлено збільшенням кількості користувачів і не всі вони хотіли витрачати велику кількість часу для вивчання гри для отримання насолоди від неї.

Для прикладу можна навести серію ігор Devil May Cry. Це серія ігор в жанрі слешер з елементами платформера. У головного героя є доступ як до зброї дальнього так і ближнього бою. Тобто до двох жанрів перечислених раніше, розробники додали ще й елементи шутеру. В кожній із частин перед початком гри можна було обрати складність. Вибір складності впливав не тільки на цифрові характеристики головного героя та ворогів, а й на систему збереження прогресу та доступу до деяких предметів. Так, наприклад, вибираючи складний рівень, гравець при програшу мав починати рівень з самого початку, вороги мали більше хітів та наносили більше пошкоджень, також гравець мав обмежений доступ до предметів які відновлювали хіти головного героя. Якщо гравець проходив гру для нього

відкривались нові рівні складності. Найважчий з яких називався Dante Must Die, де у гравця був лише один хіт, тобто він вмирав з одного удару, в той час як вороги мали звичне значення хітів. Додавання даного рівня складності надавало бажаний виклик для гравців, які в досконалості вивчили всі механіки гри.

Ця схема присутня в іграх і наш час, єдине що змінилось це те, що ігри загалом стали простішими. Це знову ж зумовлене збільшенням кількості користувачів. Особливо під час світового карантину кількість людей зацікавлених в комп’ютерних іграх почала різко зростати. Розробники, не бажаючи втрачати таку кількість потенційних клієнтів, почали зменшувати складність ігор [2].

Але це мало й зворотній ефект, маючи мету вгодити всім, вийшло так, що розробники не вгодили нікому. Результатом цього є ігри, особливо від великих ігровий компаній, граючи в які гравець може взагалі не відчути ніякого виклику. Що зменшує інтерес гравця до проєкту. Іноді доходить то ситуацій коли частини гри з геймплеєм стають нудною перешкодою, яку необхідно подолати для того щоб дізнатись історію гри.

1.1.1 Souls – подібні ігри

Піджанр, який стрімко почав набирати популярність в масах у 2023 році це souls-подібні ігри або soulslike.

Soulslike піджанр, який був вигаданий гравцями для того щоб якось узагальнити ігри, які мали спільні риси, а саме:

- велику складність від самого початку;
- сетінг темного фентезі;
- специфічну бойову систему з механікой витривалості;
- специфічний дизайн мап, які зв’язані між собою.

Широкої популярності набула гра Dark Souls від компанії FromSoftware, на честь якої було названо піджанр.

Студія випускала і інші ігри до Dark Souls, але популярність отримала саме ця. Інші ігри які виходили після були побудовані за тими самими схемами, що і

попередня але розробники намагались з кожною новою частиною покращити формулу.

В даній грі розробники не давали гравцеві обрати рівень складності, він був лише один – складний. Але це було плюсом та стало особливою рисою гри. Геймдизайнери підійшли з особливою увагою до створення світу та ворогів для досягнення балансу складності у проєкті. На перший погляд може скластись враження, що сама гра вороже налаштована проти гравця. Складні комбінації ворогів проти яких потрібно битись, підступні пастки, які можуть вбити з одного удару, та заплутана структура рівнів. Наприклад, вузький довгий коридор з лучником в кінці, та декількома ворогами ближнього бою, вороги, які чекають в засідках і т. д. Все це, звичайно, відсіяло деяку частину гравців, але ті хто продовжив проходження отримали незабутні враження від проєкту. Оскільки на перший погляд ворожий світ, та складні зіткнення з ворогами які здаються занадто складними, після декількох спроб та аналізу мувсетів супротивників гра стає легшою. Гравці, які повністю освоїли гру навіть кидали собі додатковий виклик, проходячи гру зі стартовим зломаним мечем та без обладунків.

Для підтримання постійного виклику для гравця розробники використовували декілька прийомів описаних раніше.

Перше – використання самого рівня з пастками. Кожен наступний рівень в який потрапляв гравець змушував його аналізувати оточення та бути уважним, оскільки не знаєш, що на тебе чекає. Наприклад, рівень, який присутній у кожній грі FromSoftware – це ядовите болото, більша частина якого це отруйна вода.

Друге – самі вороги та їхні комбінації в сутичках. У кожного ворога є певний мувсет, в когось він доволі простий в когось ні. Кожен з них потребує індивідуального підходу. В кожній сутичці невелика кількість супротивників. Кожен бій – це своєрідна головоломка, де гравцеві необхідно швидко вирішити якого ворога подолати першим, чи може він якось використати оточення на свою користь і т. д.

Третє – прокачка головного героя. За подолання ворога гравець отримує валюту – душі, за допомогою яких він може покращити характеристики свого персонажа. Але і тут не достатньо бездумно розкидати очки. Оскільки на прокачку впливає бажання гравця, а саме, в яких обладунках він хоче ходити, важких, середніх чи легких, яку зброю він хоче використовувати, чи потрібна йому магія. Все це додає додатковий елемент до складності гри.

1.1.2 Rogue-подібні ігри

Піджанр, який з'явився в 1980 році з появою гри Rouge, на честь якої і був названий піджанр roguelike, тобто ті, що схожі на Rouge.

Rogue виявився настільки популярним, що надихнув ціле покоління дизайнерів ігор на його наслідування, що призвело до хвилі ігор, які спільно називаються «roguelike» [8, 9].

Основними елементами жанру є процедурна генерація рівнів, ворогів на них, та предметів, які знаходить гравець. Ігровий цикл даних ігор полягає у тому, щоб гравець зібравши предмети на рівнях, які якимось чином його посилюють пройшов гру, у випадку смерті гравець втрачає весь прогрес, всі предмети та повертається на початок. В наш час є деякі модифікації цієї формули. Одна з них – метапрогресія. Тобто розробники дають гравцеві можливість посилювати свого персонажа на протязі всієї гри. Наприклад, гравець граючи отримує певну валюту, потім після поразки він повертається у стартову зону, де може обрати постійні посилення такі як приріст певних характеристик, збільшення шансу випадання більш цінних предметів і так далі.

1.2 Визначення основних рис rogue-подібних ігор

Для визначення основних рис rogue-подібних ігор було проаналізовано два проєкти в цьому жанрі: Hades та Dead Cells, які є одними з найпопулярніших його представників.

Hades – це бойовик у стилі roguelike, розроблений Supergiant Games, відомий своїм захоплюючим сюжетом і швидкими боями (див. рис. 1.1).

Рисунок 1.2 – Гра Hades

Незважаючи на те, що він відрізняється від деяких традиційних roguelike проєктів, він містить декілька спільних для жанру механік:

– процедурно згенеровані рівні. Кожна спроба втечі через підземний світ передбачає процедурно згенеровані кімнати та шляхи. Хоча певні області та вороги можуть бути постійними, макет змінюється, забезпечуючи унікальність кожного забігу;

– постійна смерть. Якщо персонаж гравця, Загрей, падає в битві, він повертається до дому Аїда, втрачаючи всі блага та покращення, отримані під час спроби втечі. Однак прогрес підтримується завдяки розвитку історії, розблокуванню нової зброї, здібностей і взаємодії з іншими персонажами;

– управління ресурсами. Такі ресурси, як темрява, ключі, самоцвіти та нектар, збираються під час пробігів. Їх можна використовувати, щоб покращити здібності Загрея, розблокувати нову зброю, покращити центр підземного світу

(Будинок Аїда) або налагодити стосунки з іншими персонажами, впливаючи на наступні пробіжки;

– рандомізовані бонуси. Олімпійські боги пропонують Загрею різні блага під час його спроб втечі. Ці переваги покращують здібності, змінюють атаки або надають нові здібності, створюючи різноманітні стилі гри. Бонуси пропонуються випадковим чином, заохочуючи адаптивність стратегії;

– різноманітність зброї та здібностей. Загрей може володіти різною зброєю, кожна зі своїм унікальним стилем атаки та аспектами, які можна вдосконалювати та змінювати. Крім того, здібності та покращення можна отримати через Дзеркало ночі, що дозволяє налаштувати та спеціалізувати стиль гри;

– битви та виклики з босами. Кульмінацією кожного рівня є зіткнення з босами, яке представляє складні битви, які перевіряють навички та здатність гравця пристосовуватися. Ці боси мають чіткі схеми атаки та механіки, які гравці повинні навчитися долати;

– каральна складність. Hades підтримує складну криву складності. У міру прогресу гравців вороги стають сильнішими, і для виживання потрібні стратегічне мислення та вмілий геймплей;

– rogue-lite прогресія. Незважаючи на постійну механіку смерті, певні прогресії зберігаються під час пробігів. Коли гравці продовжують грати та розблоковують різні оновлення, здібності та елементи історії, наступні спроби стають більш керованими;

– інтеграція оповіді. На відміну від деяких традиційних rogue-подібних ігор, «Hades» переплітає свій наратив зі структурою roguelike. З кожною спробою втечі гравці дізнаються більше про персонажів, їхні стосунки та історію, надаючи мотивацію для наступних пробіжок.

Ці елементи roguelike ідеально поєднуються із захоплюючою історією, високоякісною озвучкою та приголомшливим оформленням, що робить Hades видатною грою в цьому жанрі.

Іншим яскравим прикладом є гра Dead Cells (див. рис. 1.2). Dead Cells – це ігровий екшн-платформер, натхненний жанром Metroidvania, розроблений Motion Twin і Evil Empire.

Рисунок 1.3 – Гра Dead Cells

Механіки жанру у даній грі представлені наступним чином:

– процедурна генерація. Кожного разу, коли ви починаєте забіг, рівні генеруються процедурно. Розташування взаємопов’язаних областей, розташування ворогів і розподіл здобичі різняться, забезпечуючи свіжий досвід з кожним проходженням;

– постійна смерть. Якщо персонаж гравця помирає, він втрачає всі зібрані клітини (які використовуються для покращень) і креслення (для розблокування нової зброї/предметів). Гравець відроджується на початку гри, йому доведеться починати подорож знову, хоча й з розблокованими певними постійними оновленнями;

– управління ресурсами. Клітини, основна валюта для покращень, отримують, перемагаючи ворогів і босів. Ці кошти можна інвестувати в постійні

оновлення, які зберігаються протягом пробігів, наприклад, покращення здоров’я, збільшення шкоди, розблокування нової зброї або вдосконалення навичок;

– випадкова зброя та предмети. Гра пропонує широкий вибір зброї, кожна з якої має унікальні риси, типи шкоди та стилі гри. Гравці знаходять і розблоковують креслення під час пробіжок, що дозволяє отримати доступ до різної зброї та предметів у наступних спробах;

– складні вороги та боси. Складні вороги з відмінними моделями атак і поведінкою. Бої з босами є особливо складними, випробовуючи навички та адаптивність гравця;

– вирівнювання та здібності. Гравці отримують клітини, перемагаючи ворогів, які можна інвестувати в постійні оновлення, щоб збільшити здоров’я, розблокувати нові здібності та покращити бойові навички. Крім того, Scrolls of Power, які можна знайти на всіх рівнях, дозволяють гравцям вибирати покращення характеристик (здоров’я, шкода або навички) відповідно до свого стилю гри;

– можливості, що відкриваються, і прогрес. У міру того, як гравці просуваються та збирають креслення, вони відкривають нову зброю, здібності та предмети, які можуть з’явитися під час наступних пробігів. Це відкриває додаткові стратегії та збірки, додаючи глибини та різноманітності ігровому процесу;

– rogue-lite прогресія. Хоча смерть карає, певні постійні покращення зберігаються протягом пробігів, створюючи відчуття прогресу та полегшуючи наступні спроби;

– швидкий ігровий процес. Гра побудована таким чином, що змушує гравця діяти швидко, це стосується всього, пересування по рівню, вибору екіпірування для забігу та битв;

– відтворюваність у Roguelike. Поєднання процедурної генерації, різноманітної зброї та складного ігрового процесу гарантує, що кожен захід буде унікальним, спонукаючи до кількох проходжень, щоб відкрити нові стратегії та секрети.

Загалом, Dead Cells поєднують елементи roguelike з дослідницькою та бойовою механікою ігор Metroidvania, пропонуючи складну та багаторазову гру.

Отже, провівши аналіз rogue-подібних ігор, можна сказати, що головними рисами жанру roguelike є:

– процедурна генерація рівнів;

– процедурна генерація зіткнень з ворогами;

– процедурна генерація предметів та бонусів, які знаходить гравець під час забігів;

– достатня кількість предметів та зброї, щоб різноманітнити кожен новий забіг;

– наявність лише одного рівня складності – складного, з подальшою можливістю відкриття ще важчих рівнів;

– велика увага ігровому процесу, на відміну від історії.

1.3 Аналіз аналогів

Системи адаптації параметрів ігрового простору не широко використовуються в відеоігровій індустрії. Але є декілька аналогів, але вони не відносяться до rogue-подібних ігор, що дещо зменшує їх можливості до зміни ігрового середовища.

Першим прикладом гри з такою системою є AI War. Гра у жанрі стратегії, де гравця змушують обирати між деякими цілями, які впливають на рівень складності та стратегічні варіанти, якими можна користуватись. Якщо гравець виправдовує очікування системи або перевищує їх вона змінює налаштування ігрового середовища з ціллю ускладнення гри.

Іншим прикладом є серія ігор Left 4 Dead від розробника Valve. Гра у жанрі кооперативного шутеру. Команді з 4 гравців необхідно пройти рівень та виконати певні задачі в процесі. Під час цього система слідкує за успішністю гравців, на скільки добре в них виходить долати перешкоди виставлені нею. Ця система називається The Director [10]. До параметрів, які може контролювати система

належать: кількість ворогів, частота появи елітних ворогів, частота появи зброї та амуніції до неї, разом з цим контролювалась кількість аптечок, які регенерують хіти героїв. Окрім цього у системи був контроль над самими задачами, які було необхідно виконати гравцям. Також The Director мав контроль над драматизмом. Це досягалось за допомогою візуальних ефектів, динамічної музики та спілкування персонажів В залежності від того на скільки добре було пройдено рівень система вирішувала складність іншого, налаштовуючи відповідні параметри.

The Director мав повний доступ до мапи, що дає йому можливість над вибором місць, це з’являться вороги. Система постійно слідкує за гравцем виставляючи йому певну оцінку напруженості. На рівень напруженості впливають ситуації, у яких знаходиться гравець, наприклад, коли гравець долає ворога у певній близькості від нього його рівень напруженості виростає, якщо він був атакований елітним ворогом рівень напруженості стає максимальним. На вибір цих місць впливають самі гравці, а саме їх місце знаходження, в якій вони ситуації, борються чи досліджують місцевість, їхній статус та навички. Окрім генерації рівнів під час їх завантаження система також може впливати на кількість ворогів, та предметів у реальному часі. Хоча у системи немає повного контролю над мапою вона все таки може змінювати шляхи проходження рівнів за допомогою маніпуляції подій, кожного разу направляючи гравців різними шляхами, які різняться від рівня до рівня. Також є частини рівнів де система вимикається.

Нажаль провести більш детальний аналіз систем неможливо, оскільки в міру високої конкуренції на ринку відео ігор розробники не розголошують деталей створення таких систем та використаних методів розробки для них.

1.4 Нейронні мережі

Штучний інтелект (ШІ) – це галузь комп'ютерних наук, яка займається створенням систем, що можуть виконувати завдання, які зазвичай потребують людського інтелекту. Основна мета штучного інтелекту полягає в розробці

алгоритмів та програм, які наділені здатністю мислити, розуміти, вирішувати проблеми, навчатися та адаптуватися до нових ситуацій [11].

ШІ включає в себе різноманітні підгалузі, такі як машинне навчання, обробка природної мови, комп'ютерний зір, робототехніка, експертні системи та інші. Одна з основних цілей ШІ полягає в створенні імітації людського мислення та розв'язання задач, які раніше вважалися виключно людськими.

Штучний інтелект знаходить широке застосування в різних сферах життя, таких як медицина, фінанси, технології, транспорт, маркетинг та інші. Він дозволяє автоматизувати процеси, робити прогнози, аналізувати дані та розробляти системи, які можуть приймати рішення на основі обчислювальної логіки [11, 12].

Нейронні мережі – моделі, які імітують структуру людського мозку, будучи його спрощеною версією [13]. Так само як і мозок людини нейронні мережі складаються з:

– нейронів, між якими присутні зв’язки з певними вагами;

– вхідних, вихідних та прихованих шарів для отримання інформації, її обробки та генерації відповіді;

– функція активації, яка є математичною операцією, яка визначає вихід нейрона на основі ваг та вхідних даних.

Нейронні мережі – це потужний інструмент для розв'язання різноманітних задач у багатьох галузях, і вони стають все більш важливим елементом в сфері штучного інтелекту та обробки даних [13, 14].

Для того щоб отримати працездатну модель нейронної мережі її необхідно спочатку навчити. Тренування нейронних мереж – це процес налаштування параметрів (ваг) моделі так, щоб здійснювати правильні передбачення або класифікації для задачі, яка розв'язується. Для отримання навченої моделі необхідно підготувати відповідні дані, поділити їх на набори для тестування та тренування відповідно. Потім проводиться вибір кількості шарів, нейронів у кожному з них, функції активації та швидкість навчання. Далі відбувається саме тренування моделі, її оптимізація та перевірка її роботи на тестових даних.

Перцептрони – це основна будівельна одиниця багатьох моделей штучних нейронних мереж [15]. Вони базуються на моделі роботи людського мозку та здатність до навчання через алгоритми оптимізації. Перцептрони складаються зі штучних нейронів, які приймають вхідні сигнали, обробляють їх та генерують вихідний сигнал.

Одношаровий перцептрон складається з одного шару нейронів, який приймає вхідні дані, обробляє їх та видає вихід. Використовується для бінарної класифікації задач, де можна провести лінійне розділення даних [16].

Багатошаровий перцептрон складається з кількох шарів: вхідний, прихований та вихідний. Інформація подається через вхідний шар, обробляється прихованими шарами з багатьма нейронами, і виходить через вихідний шар. Багатошарові перцептрони можуть вирішувати більш складні проблеми, оскільки вони здатні навчатися складним залежностям між вхідними та вихідними даними. Вони є універсальними та можуть навчатись складним залежностям між вхідними та вихідними даними [17].

Однак, багатошарові перцептрони можуть бути схильні до перенавчання при недостатній регуляризації, а також вони можуть вимагати багато даних для ефективного навчання.

Порівнюючи одношарові та багатошарові мережі можна прийти до наступних висновків наведених у табл. 1.

Порівнявши мережі, видно, що кожен їх вид підходить то певного роду задач. У нашому випадку кращим вибором буде багатошарова мережа через її більший потенціал та особливості ігрового середовища. Простір гри має багато непередбачуваних сценаріїв оскільки це середовище, яке постійно змінюється саме по собі, в додачу гравець проводить додаткові маніпуляції з середовищем змінюючи його поведінку ще сильніше. Також впровадження багатошарової мережі може надати можливість покращити систему у майбутньому, додаючи до неї нові вхідні дані.

Таблиця 1.1 – Порівняння особливостей одношарових та багатошарових мереж

Характеристики	**Одношарові мережі**	**Багатошарові мережі**
Кількість шарів нейронів	Має один шар нейронів, що складається з вхідних та вихідних вузлів	Містить кілька прихованих шарів нейронів, включаючи вхідний та вихідний шар
Використання	Використовується для простих задач класифікації або регресії	Здатний моделювати складні залежності та вирішувати складніші завдання
Можливості	Має обмежену здатність вирішувати складні проблеми через лінійну функцію активації	Використовує нелінійні функції активації для моделювання складних залежностей
Обсяг даних та ресурси	Низька складність обчислень та навчання	Вимагає більше даних та обчислювальних ресурсів для навчання та підтримки
Результативність на складних завданнях	Може показувати гірші результати на складних завданнях	Здатний досягати кращих результатів на складних завданнях з більшим обсягом даних

1.5 Ігрові рушії

Ігровий рушій є ядром будь-якої гри, відповідаючи за її технічну частину. Використання рушіїв спрощує процес розробки завдяки наявності графічного інтерфейсу та вбудованих бібліотек. Основна перевага полягає у можливості

створювати мультиплатформові застосунки для різних пристроїв, таких як ПК, Xbox та PlayStation.

Ця складова відповідає за майже всю основну функціональність гри, включаючи візуальну частину, анімації, звук, фізичні розрахунки, скрипти, штучний інтелект та управління ресурсами. Зазвичай один рушій можна використовувати для кількох різних проєктів.

Термін "ігровий рушій" з'явився в 1990-х роках під час популяризації 3D ігор у жанрі шутерів. Розробники стали оптимізувати ресурси, ліцензуючи певні компоненти ПЗ (програмного застосунку) зі своїх проектів для використання у нових іграх. Це дозволило економити час і кошти.

Деякі рушії дозволяють розробникам отримувати додатковий дохід від кожного проекту, розробленого на їхній основі. Наприклад, компанія Epic Games отримує 5% від продажу проектів, розроблених з використанням Unreal Engine.

Ігрові рушії часто мають компонентну архітектуру, що дозволяє змінювати або розширювати наявні системи новими. Їх використовують не лише в ігровій індустрії, а й у рекламі, фільмах, архітектурній візуалізації тощо.

Ці системи базуються на графічних API (application programming interface), таких як Direct3D або OpenGL, а також використовують низькорівневі бібліотеки для апаратно незалежного доступу до комп'ютера. До найпопулярніших рушіїв належать GameMaker Studio, Unreal Engine та Unity.

Ігровий рушій – потужний інструмент у руках розробника, який набагато спрощує процес створення застосунків. До основних причин вибору саме готового ігрового рушія, а не створення власного, належать:

– ефективність розробки. Рушії надають готові інструменти та функціонал, які значно полегшують роботу розробників. Замість написання коду з нуля для кожної складової гри (від візуальних ефектів до фізичних розрахунків), розробники можуть використовувати вбудовані функції рушія, що заощаджує час та зусилля;

– мультиплатформеність. Ігрові рушії дозволяють створювати ігри для різних платформ, таких як ПК, консолі, мобільні пристрої та інші. Це дозволяє розробникам максимально розширити аудиторію своїх продуктів;

– готові ресурси та активи. Багато рушіїв мають свої власні магазини активів, де розробники можуть знаходити готові моделі, текстури, анімації, звуки тощо. Це дозволяє швидше створювати готовий контент для гри;

– спільнота та підтримка. Популярні ігрові рушії мають великі спільноти користувачів, форуми, документацію та підтримку. Це означає, що розробники можуть легше отримати допомогу, поради та рішення проблем під час розробки;

– інтегрованість інструментів. Рушії зазвичай постачаються з широким спектром інструментів, таких як візуальні редактори, системи фізики, редактори анімації, інструменти для роботи зі звуком та багато іншого. Це дозволяє розробникам працювати всередині одного інтегрованого середовища;

– оновлення та підтримка. Рушії часто оновлюються та підтримуються розробниками, що забезпечує регулярне вдосконалення функцій та виправлення помилок, забезпечуючи актуальність рішення для розробки.

1.6 Особливості ігрових застосунків

Ігрові застосунки є мультимедійним продуктом, системи якого працюють в реальному часі. Тобто для їх роботи потрібно виділяти багато обчислювальних ресурсів [18].

Усі системи, які були запрограмовані розробниками, графічна частина та звук все це потрібно враховувати при додаванні нових систем. А саме як вплине додавання нових систем на продуктивність гри. Якщо в застосунку вже працюють доволі ресурсомісткі частини, то просто стає неможливим додати до нього ще одну складну систему.

Враховуючи особливості певних застосунків, інколи не є можливим додати складну систему до гри без впливу на продуктивність. Отже, при створенні системи адаптації ігрового середовища потрібно також враховувати як обраний тип мережі

може вплинути на роботу гри, особливо якщо це rogue-подібні ігри в основі яких лежить хаос, який досягається випадковою генерацією. Наприклад граючи в Risk of rain 2 гравець може досягти такого моменту в грі, коли на екрані присутні великі кількості ворогів, що вже саме по собі призводить до падіння кількості кадрів в секунду (fps). Якщо разом із цим враховувати що при нанесенні атаки проводиться велика кількість додаткових розрахунків для модифікації її поведінки в залежності від предметів, які має гравець, то під час атаки гра взагалі може зависнути на декілька секунд.

Рисунок 1.4 – Ігровий простір гри Risk of rain 2 при великому скупченні ворогів

Тобто при додаванні нейромереж до застосунків, які працюють у реальному часі, таких як ігри, важливо враховувати кілька ключових факторів:

– витрати обчислювальних ресурсів. Нейромережі можуть бути обчислювально витратними. Додавання їх до реального часу може вимагати значних обчислювальних ресурсів, що може вплинути на продуктивність системи, особливо в ігрових застосунках;

– затримки. Нейромережі, зазвичай, потребують часу для виконання прогнозів або обробки даних. Забезпечення низької затримки є критичним у реальному часі, тому потрібно забезпечити ефективні методи оптимізації для зменшення затримок;

– обмеження ресурсів пристрою. На різних пристроях можуть бути обмежені ресурси, такі як енергопотреба, обсяг пам'яті, обчислювальна потужність. Нейромережі повинні бути оптимізовані для роботи в обмежених умовах;

– адаптація до змінних умов. В іграх можуть виникати різноманітні сценарії та умови гри. Нейромережі повинні бути здатні адаптуватися до змін у реальному часі і зберігати ефективність;

– збереження дизайну гри. Нейромережі повинні інтегруватися так, щоб підтримувати основні дизайнерські принципи гри і взаємодіяти із гравцем, як це очікується від гри в реальному часі.

1.7 Шляхи створення інтелектуальної системи адаптації середовища

Дану систему можна створити декількома способами. Можна вирішувати задачу класифікації або прогнозування.

Бажаний результат можна отримати вирішуючи задачу прогнозування. Для цього необхідно збирати в реальному часі дані про гравця для створення відповідного часового ряду, що саме по собі може вплинути на продуктивність гри. Постійний збір даних та їх запис до файлу може бути доволі ресурсомістким, особливо якщо проводиться часто. Можна зменшити навантаження на систему зменшивши частоту записів але це може вплинути на точність прогнозу. Даний спосіб може спрацювати але він не є універсальним.

Дану систему можна створити вирішивши задачу класифікації. Тобто присвоїти гравцеві певні рівні успішності та використовуючи зібрані параметри під час гри віднести його до певного класу. Цей клас далі буде використовуватись системою для створення відповідних рівнів.

Для вирішення даної задачі підійшли б багатошарові перцептрони або мережі підсиленого навчання.

Unity має інструменти для підсиленого навчання, але цей процес є довшим ніж навчання перцептрону. Разом із цим MLP має менше гіперпараметрів та його навчання можна провести поза середовищем ігрового застосунку, що спрощує процес навчання та тестування.

MLP так само як і звичайні перцептрони здатні розв’язувати дані їм задачі на достатньо малій кількості прикладів, що означає що отримана система буде здатна правильно розпізнати різноманітні залежності, які не використовувались при навчанні.

Це видно на прикладі наступної задачі. Візьмемо монохромне зображення розміром 256 × 256 пікселів. Для вхідних даних перцептрона використовуватимемо координати точки (16 S-елементів усього, представлених 8 бітами кожен). На виході ми будемо очікувати отримати інформацію про кольори цієї точки. дана задача є дуже простою для перцептрона, і йому достатньо всього 1,500 A-елементів для успішного розв'язання, де A-елемент – це асоціативний елемент. Також при збільшенні кількості A-елементів зменшується кількість необхідних циклів навчання.

Разом із цим це дасть змогу зменшити кількість необхідних записів потрібних для навчання нейронної мережі у тому випадку, якщо розробниками буде прийнято рішення про навчання мережі під час роботи гри. Це в свою чергу дало б змогу системі налаштуватись під окремого користувача.

1.8 Постановка задачі

Складність в відео ігрових застосунках є однією з найважливіших їхніх частин, які змушують гравця продовжувати грати та підтримувати його інтерес до проєкту. Але підібрати складність для гри, яка б підійшла широкому колу людей доволі складна задача для ігрових дизайнерів, навіть беручи до уваги доволі поширену схему з трьох базових рівнів таких як простий, нормальний та складний.

Цю проблему можливо вирішити маючи ігровий простір, який може підлаштовуватись під користувача. Ігровий простір – це мапа гри, вороги, присутні на ній, їхня складність, предмети які знаходить гравець та частота їх появи на мапі.

Дана система цілком може бути реалізована в усіх ігрових жанрах, проблема буде полягати в їхніх особливостях. Наприклад, проєкти, які доволі сильно спираються на лінійну прогресію гравця та історію не зможуть в повній мірі змінювати параметри власного простору, оскільки деякі із них по дизайну мають бути незмінними. Через такі особливості деяких жанрів студії не створюють такі системи оскільки їх розробка зайняла б багато часу та ресурсів. Але їх додавання навіть до таких жанрів могли б вирішити головну проблему таких ігор – відтворюваність (replayability). Тобто такі ігри було б цікавіше проходити заново.

Але не всі відео ігри мають подібні обмеження. Rogue-подібні ігри, мають повний доступ до всіх параметрів ігрового простору. Оскільки основою жанру є процедурна генерація рівнів, ворогів та предметів то система адаптації параметрів ігрового простору може бути реалізована у повній мірі.

Такі системи створюються за допомогою методів штучного інтелекту [19] та можуть мати доступ до різної інформації про гравця, та доступ до певних параметрів простору гри. Використання методів штучного інтелекту надає системі бажаної гнучкості та здатність адаптуватись під різноманітні сценарії, приймаючи відповідні рішення. Загальна схема такої системи виглядала б наступним чином:

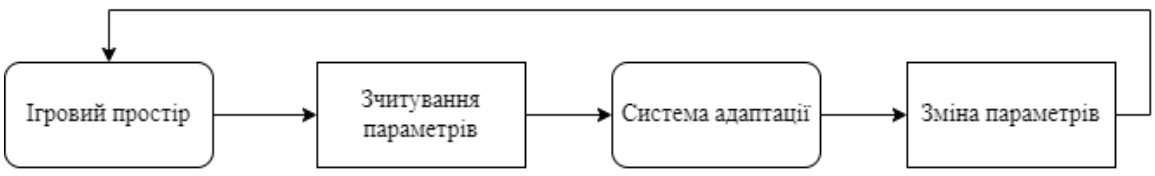

Рисунок 1.5 – Приклад схеми системи адаптації ігрового простору

Висновки до розділу 1

Провівши дослідження відповідних публікацій та підручників було проведено опис поставленого завдання, описані особливості ігрового середовища,

його параметри, маніпулюючи якими можна впливати на ігровий досвід користувачів. Було наведено причини та наслідки, які можна отримати реалізувавши таку систему у відео ігрових проєктах.

Було проведено аналіз ігрового дизайну та його розвиток з часом в ігор різних епох, від часів аркадних автоматів до нашого часу. Зроблено більш детальний опис ігрових проєктів, які вважаються прикладами гарного ігрового дизайну. Було проведено аналіз прикладів ігор в жанрі roguelike для кращого розуміння особливостей жанру та зроблено відповідні висновки.

Було проведено аналіз наявних аналогів на двох прикладах AI War та серії ігор Left 4 Dead. У даних відео ігрових застосунках присутні системи для адаптації під дії гравця.

Були описані нейронні мережі та перцептрони. Наведено інформацію про навчання нейронних мереж. Було проведено порівняння багатошарових та одношарових мереж.

Було наведено загальну інформацію про ігрові рушії та наведені причини їх вибору. Було проведено опис ігрового рушія Unity, його особливості та процес створення застосунків із його використанням.

2 МЕТОДИ ТА ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ДЛЯ СТВОРЕННЯ СИСТЕМИ АДАПТАЦІЇ ПАРАМЕТРІВ ІГРОВОГО ПРОСТОРУ

2.1 Глибоке навчання. Багатошаровий перцептрон

Глибоке навчання – це галузь машинного навчання, спрямована на використання багатошарових нейронних мереж для вирішення завдань, які включають в себе розпізнавання шаблонів, класифікацію, регресію, обробку природних мов, генерацію зображень та інші завдання в області обробки інформації [20, 21].

Глибока нейронна мережа (ГНМ або DNN) представляє собою розширену версію поверхневої нейронної мережі (ШНМ), що включає в себе багато шарів для обробки даних [21]. Ця мережа перетворює вхідні дані в вихідні, ієрархічно виділяючи та агрегуючи ознаки, що призводить до поступового підвищення рівня абстракції даних від входів до виходів.

У порівнянні з невеликими нейронними мережами, глибокі нейронні мережі завдяки збільшенню кількості нейроелементів та зв'язків набувають значної обчислювальної потужності і здатності моделювати більш складні залежності. За допомогою спеціалізації шарів та високої ієрархічності обробки даних, вони також стають більш зручними для сприйняття та аналізу людьми. Крім того, спеціалізація шарів у глибоких нейронних мережах робить їх більш придатними для інтеграції в мережеві моделі апріорної інформації про предметну область.

Ігровий простір є досить складним та непередбачуваним середовищем. Його стан змінюється кожного кадру, а в середньому – це 60 кадрів на секунду. Ігри можуть мати велику кількість різноманітних факторів, таких як гравець, інші персонажі, об'єкти, території тощо. Поведінка гравців і динаміка ігрового світу може бути непередбачуваною, що унеможливлює врахування усіх ситуацій які можуть відбутись. Системи повинні бути гнучкими і адаптовуватися до змін у грі, таких як зміна стратегії гравця або виникнення нових сценаріїв. Враховуючи всі ці

особливості, було прийнято рішення використовувати штучний інтелект при розробці системи адаптації ігрового простору.

Оскільки ігрове середовище є доволі динамічним було вирішено використовувати багатошарові мережі для створення системи [19]. Архітектура таких мереж надає можливість створювати більш складні, нелінійні зв'язки між даними, які поступають на вхід системи та результатами на виході.

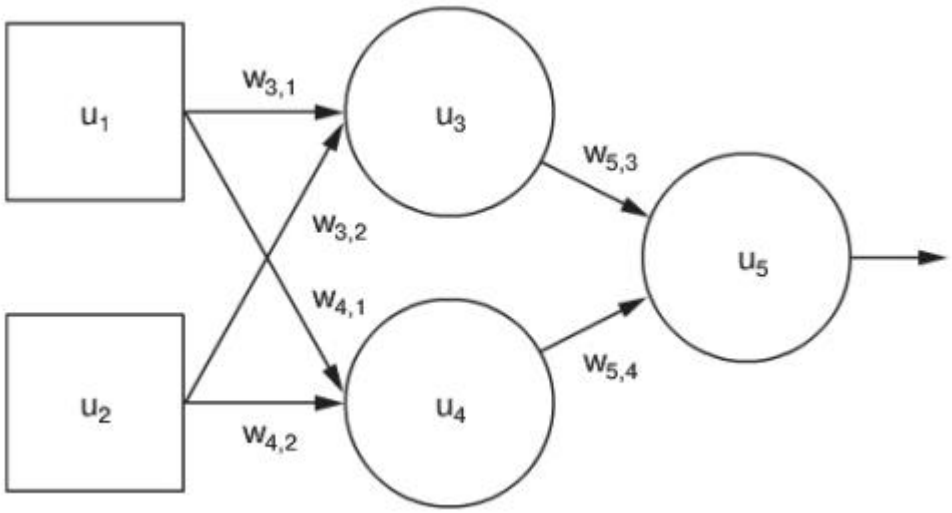

Рисунок 2.1 – Приклад багатошарового перцептрону

Багатошарові мережі дають можливість створити більш складні системи, що не є можливим при використанні одношарових мереж. В таких мереж вхідний шар задають вхідне значення, а приховані та вихідні шари є функціями. Результат сумування отримується за допомогою певної функції, частіше всього використовують функцію сигмоїди [22]. Дана функція та її похідна використовувалась і при створенні агенту складності.

$$f(x) = \frac{1}{(1 + e^{-x})}. \quad (2.1)$$

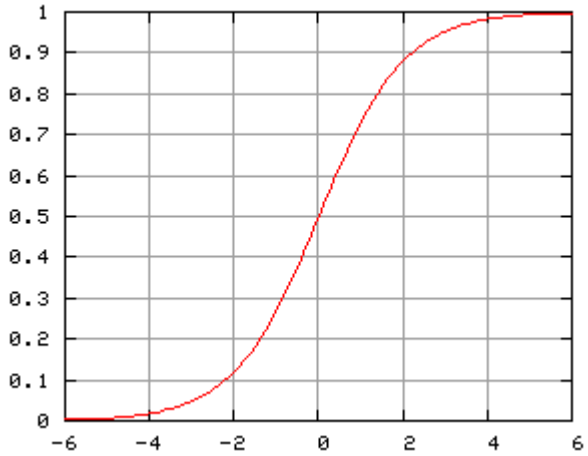

Рисунок 2.2 – Графік функції сигмоїди

Для навчання агенту використовувався алгоритм зворотного розповсюдження (backpropagation algorithm). Даний алгоритм є одним з найпопулярніших алгоритмів для навчання. Його робота полягає в зміні ваг. Із назви алгоритму видно, що помилка буде розповсюджуватись у зворотному порядку, тобто від вихідного шару до вхідного, що відрізняється від проходу сигналу при звичайній роботі мережі.

В області теорії штучного інтелекту часто розглядають агентів, які можуть вчитися та адаптуватися до свого оточення. В умовах невизначеності важливо не лише аналізувати поточну інформацію, але й враховувати загальний контекст ситуації, в яку агент потрапив [24]. У цьому випадку використовують перцептрони зі зворотнім зв'язком. Крім того, в деяких випадках важливо збільшити швидкість навчання перцептрона, наприклад, за допомогою моделювання рефрактерності.

2.1.1 Алгоритм зворотного розповсюдження

Алгоритм зворотного розповсюдження (backpropagation) є ключовим етапом в навчанні штучних нейронних мереж, особливо в нейронних мережах глибокого навчання (Deep Learning). Процес зворотного розповсюдження включає кілька етапів.

1. Прямий прохід (forward pass). На початку, вхідні дані подаються на вхід нейронної мережі. Дані проходять через кожен шар мережі, де здійснюються операції лінійного комбінування і застосування активаційних функцій. Результати передаються через мережу до виходу.

2. Обчислення втрат (loss computation). Порівнюються вихідні значення мережі з очікуваними значеннями (цільовими величинами). Обчислюється функція втрати, яка вказує на те, наскільки великі або малі відмінності між прогнозованими і справжніми значеннями.

3. Зворотний прохід (backward pass). Розпочинається з виходу і розповсюджується назад через мережу. Обчислюються градієнти функції втрати відносно параметрів мережі за допомогою правила ланцюгового виводу (chain rule). Отримані градієнти вказують на те, як великою має бути зміна кожного параметра для зменшення втрат.

Функція для вихідного шару:

$$\delta_0 = (C_i - u_0)u_0(1 - u_0), \tag{2.2}$$

де i – номер нейрона вихідного шару;

C_i – очікуване значення сигналу нейрона вихідного шару;

u_0 – реальне значення сигналу нейрона вихідного шару.

Функція для прихованого шару:

$$\delta_i = \omega_{ij}\delta_0 u_i(1 - u_i), \tag{2.3}$$

де i – номер нейрона прихованого шару;

j – номер нейрона вихідного шару;

ω_{ij} – ваговий коефіцієнт;

u_i – значення сигналу нейрона прихованого шару.

4. Оновлення параметрів (parameter update). Використовуючи градієнти, мережа оновлює свої параметри (ваги і зсуви) з метою мінімізації функції втрат. Цей процес може використовувати різні оптимізаційні алгоритми, такі як стохастичний градієнтний спуск (SGD) або його модифікації, для ефективного навчання.

Функція для ваг між прихованим та вихідним шаром:

$$\omega_{ij} = \omega_{ij} + \rho\delta_0 u_i, \tag{2.4}$$

де ρ – це коефіцієнт навчання (або крок навчання).

Функція між прихованим та вхідним шаром:

$$\omega_{ij} = \omega_{ij} + \rho\delta_i u_i. \tag{2.5}$$

Вибір значення для кроку навчання процес ітеративний. Необхідно підібрати таке його значення при якому нейронна мережа досягне бажаного значення та не "перескочить його". Якщо значення кроку буде занадто великим нейронна мережа може пропустити бажані значення ваг і вже не досягне очікуваного користувачем результату. При занадто малих значеннях коефіцієнту навчання процес може досить сильно затягнутись у часі.

Даний процес повторюється до досягнення максимальної кількості ітерацій, або до досягнення бажаного значення помилки.

2.1.2 Цикл навчання мережі

Розглянемо навчання мережі більш детально.

На початку алгоритму здійснюється прямий прохід сигналу від вхідного шару до вихідного.

Розрахунок значення сигналу для прихованого шару проводиться за наступною формулою:

$$u_j = f\left(\sum_{i=1}^{n} \omega_{ij} x_i + b_j\right), \tag{2.6}$$

де i – номер нейрона у вхідному шарі;

j – номер нейрона у прихованому шарі;

n – кількість нейронів у шарі;

x_i – вхідний сигнал від нейрона i;

b_j – зсув для нейрона j;

$f(x)$ – активаційна функція (сигмоїда).

Зсув b_j в даному в випадку використовується для надання моделі можливості більш гнучко налаштовувати наскільки активно нейрон реагує на вхід, навіть коли всі вхідні сигнали рівні нулю. Такий зсув може бути корисним для забезпечення адаптивності нейронної мережі та розраховується за наступною формулою:

$$b_j = w_b * \text{зсув}, \tag{2.7}$$

де w_b – вага зсуву;

зсув – базове значення зсуву.

Далі за цією ж формулою відбувається розрахунок для значення сигналу в вихідному шарі.

Після цього йде підрахунок середньоквадратичної помилки для корекції вагових коефіцієнтів.

$$mse = 0{,}5 * (t - y)^2, \tag{2.8}$$

де t – очікуване значення;

y – реальне значення.

Далі йде застосування зворотного розповсюдження. Проводиться коригування ваг для кожного шару за формулами 2.4, 2.5. Тут використовується похідна активаційної функції, в нашому випадку це похідна сигмоїди, похідна якої виглядає наступним чином

$$f`(x) = x * (1 - x). \quad (2.9)$$

Спершу оновлюються ваги між вихідним та прихованим шаром після чого оновлюється зміщення для вихідного шару за формулою 2.10. Потім процес повторюється для ваг між прихованим та вхідним шаром та значенням зсуву за формулою 2.11.

$$w_{ib} = w_{ib} + (\rho * \delta_0 * \text{зсув}_i). \quad (2.10)$$

$$w_{ib} = w_{ib} + (\rho * \delta_i * \text{зсув}_i). \quad (2.11)$$

На цьому зворотній прохід завершується. Даний процес повторюється допоки не буде досягнуто бажаного значення помилки або не досягнуто певної кількості ітерацій циклу навчання.

Алгоритм зворотного розповсюдження використовується для створення нейроконтролерів для персонажів комп'ютерних ігор, але їх можна використовувати і для систем. Перевагами нейроконтролера є реакція на непередбачувані ситуації та те, що він не є строго заданою функцією.

Здійснюючи навчання нейроконтролера на обмеженому числі прикладів на виході отримується система, яка може приймати певні рішення в нових непердбачуваних ситуаціях. Будь-які незначні зміни в середовищі можуть викликати різні відгуки у системи, що робить її поведінку більш природньою.

Нейроконтролер створювався поза ігровим середовищем для пришвидшення його навчання, та полегшення процесу розробки. Середовищем для його створення було обрано Visual Studio.

2.2 Інтегроване середовище розробки Visual Studio

Visual Studio – це інтегроване середовище розробки (IDE) від Microsoft, призначене для створення різноманітних програмних продуктів [25]. Воно підтримує багато мов програмування, таких як C#, C++, F#, Visual Basic і інші. Користувач може використовувати Visual Studio для розробки різноманітних застосунків, включаючи веб-сайти, десктопні програми, мобільні додатки та ігри.

Дане інтегроване середовище розробки було обрано по ряду деяких причин:

– інтеграція з різними мовами програмування. Visual Studio підтримує багато мов програмування, забезпечуючи інтеграцію для розробки різноманітних типів програм;

– структуроване середовище розробки. Інтерфейс Visual Studio розділений на різні панелі, які спрощують роботу з проектами та визначенням різних елементів програми;

– редактор коду. Включає зручний редактор коду з підсвічуванням синтаксису, автоматичним завершенням інструкцій та іншими функціями для покращення продуктивност;

– відлагодження та профілювання. Visual Studio надає потужні засоби відлагодження, включаючи крок-за-кроком виконання коду, аналіз стеку викликів, панелі перегляду змін змінних тощо;

– підтримка для мобільних платформ. Розробка мобільних додатків для платформ, таких як Android і iOS, з використанням Visual Studio та відповідних інструментів;

– підтримка технологій веб-розробки. Вбудована підтримка для розробки веб-застосунків з використанням технологій, таких як ASP.NET, HTML, CSS і JavaScript;

– інтеграція з системами контролю версій. Підтримка роботи з різними системами контролю версій, такими як Git, для відстеження змін у коді;

– модульність та розширення. Visual Studio дозволяє розширювати свою функціональність за допомогою різноманітних плагінів і розширень, що дозволяє користувачам налаштовувати своє робоче середовище;

– інтеграція з хмаровими сервісами. Можливість взаємодії з хмаровими платформами Microsoft, такими як Azure, для розгортання та керування додатками;

– підтримка ігрової розробки. Вбудовані інструменти для розробки ігор, включаючи підтримку для популярних двигунів, таких як Unity.

Але нейроконтролер не є єдиною частиною системи. Необхідно створити систему, яка не тільки робить певні висновки, класифікуючи гравця, але й виконує певні дії. Для цього є необхідність створення додаткових підсистем, які отримуючи інформацію від нейроконтролера, будуть створювати відповідні рівні. Ці підсистеми створювались на базі ігрового рушія Unity.

2.3 Генератор рівнів

Генерація рівнів сама по собі є доволі складним завданням. Оскільки це генерація для комп'ютерних ігор, необхідно враховувати такий фактор як дизайн рівнів та ігровий дизайн [9]. Тобто не можна просто створити все повністю за допомогою випадкової генерації, у цього процесу мають бути певні правила, які відповідають баченню розробника. Але попри всі складнощі даного підходу сам по собі, якщо все зроблено вірно, він забезпечує безліч сценаріїв для користувачів, затримуючи їх довше у грі, та спрощує розробку знімаючи частину навантаження із дизайнерів рівнів.

Існує декілька підходів для випадкової генерації рівнів. Можна використовувати шум Перліна, можна розбити рівень на безліч частин та випадковим чином їх об'єднувати, або можна створити випадкову мапу кімнат.

Було вирішено генерувати рівні через мапу кімнат. Даний підхід використовується у популярних іграх нашого часу. Наприклад, гра Binding of Isaac має таку структуру рівнів.

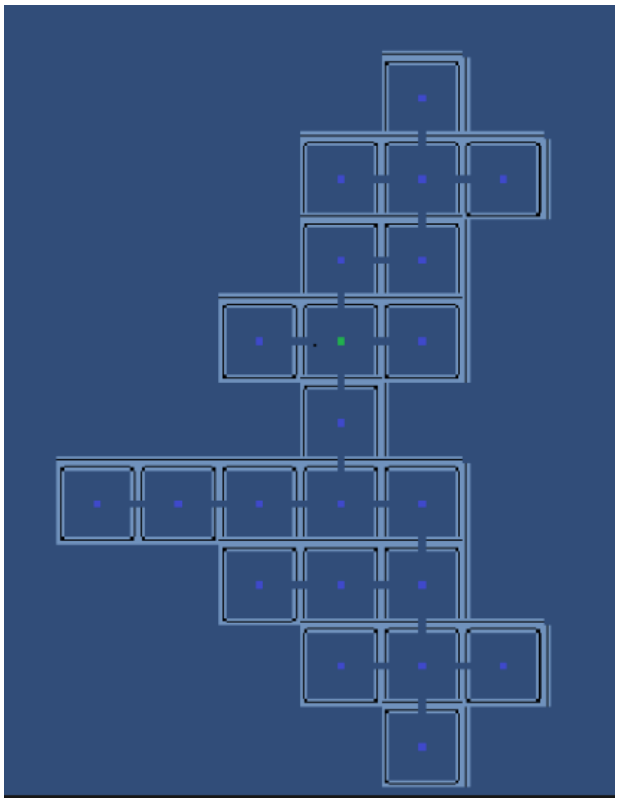

Рисунок 2.3 – Приклад згенерованого рівня з гри Binding of Isaac

Створення рівнів проходить в 2 етапи. Перший – створення розмітки рівня, другий – створення самого рівня по якому гравець зможе переміщуватись.

Генерація розмітки виглядає наступним чином:

1) створення колекції, яка буде відображати уявну сітку мапи;
2) створити першу, стартову кімнату в центрі сітки;
3) оскільки направлень всього 4, то випадково обрати одне з 4 та одночасно перевірити чи обрана клітинка не зайнята;
4) створити кімнату.

Далі кроки 3 – 4 повторюються до моменту досягнення бажаної кількості кімнат. Після створення кімнат необхідно створити між ними з'єднання. Тож необхідно пройтись по створеній мапі та з'єднати кімнати, які є сусідніми одна до одної. Після з'єднання необхідно призначити кімнатам відповідний тип (стартова, бойова, кімната зі скарбами). Цей процес залежить від обраного рівня складності, а саме відношення між бойовими кімнатами та кімнатами зі скарбами – чим

більший рівень складності був обраний, тим менші шанси у гравця натрапити на кімнату зі скарбами.

Після завершення створення розмітки починається процес створення самого рівня на ігровій сцені. Для цього передається відповідна інформація про обраний рівень складності до контролеру сутичок, який визначає кількість ворогів та їх хвиль для кожної бойової кімнати. Потім йде створення самих кімнат.

2.4 Інтегроване середовище розробки Unity

Unity – це мультиплатформений ігровий рушій для створення інтерактивних 2D або 3D-додатків [26]. Даний рушій існує вже досить давно і є одним із найпопулярніших інструментів для створення ігор. Його використовують як розробники-любителі так і повноцінні ігрові студії. Окрім ігрової індустрії даний двигун використовується і в інших сферах. Оскільки рушій має багато вбудованого функціоналу такого наявність таймлайну для створення анімацій, VFX-графіки, які дозволяють створювати різноманітні ефекти, система освітлення та розрахунку фізичних явищ та багато іншого його використовують для різноманітних симуляцій, у фільмах чи мультфільмах, при розробці архітектурних проектів, тощо. Також даний рушій має дуже добре описану документацію, де можна знайти відповіді майже на всі питання. Існує безліч навчального матеріалу на різноманітні теми, від створення простих контролерів персонажів, до процедурних анімацій чи використання штучного інтелекту в комп’ютерних іграх. Даний навчальний матеріал присутній у різних формах, у текстовій на форумах, у відео від незалежних користувачів або від самих Unity.

Основна мета Unity – дозволити розробникам створювати високоякісні ігри та візуально привабливі додатки для різних платформ, таких як комп'ютери, мобільні пристрої, консолі, віртуальна реальність та інші. Для цього Unity надає:

– графічний інтерфейс. Unity має інтуїтивно зрозумілий інтерфейс, що дозволяє візуально розміщувати об'єкти, налаштовувати їх властивості та створювати складні анімації без програмування;

– скрипти. Розробники можуть використовувати C# або JavaScript для програмування логіки гри. Це дозволяє створювати різноманітні ігрові механіки та взаємодію об'єктів;

– ресурси. Unity підтримує різноманітні формати медіафайлів, такі як зображення, звуки, моделі 3D, анімації, що дозволяє легко інтегрувати різноманітний контент у гру чи додаток;

– кросплатформеність. Розроблені в Unity проекти можна експортувати на різні платформи, такі як iOS, Android, Windows, macOS, консолі та інші;

– спільна робота. Unity надає зручні інструменти для спільної роботи команд розробників, що дозволяє спростити процес колективної роботи над проектами;

– розвиток інтерактивних додатків. Unity можна використовувати не лише для створення ігор, але й для створення інтерактивних візуальних додатків, симуляцій, тренажерів та інших проектів.

Unity використовує мову програмування C# для розробки ігор та інтерактивних додатків. Це забезпечує розробникам потужний та ефективний інструментарій для створення різноманітних проектів.

Оскільки гра є складним мультиплатформеним застосунком то використання мови програмування C# та об’єктно-орієнтованого підходу значно спрощує процес розробки таких великих складних систем.

Об'єктно-орієнтоване програмування (ООП) – це парадигма програмування, що базується на концепції "об'єктів". Використовуючи ООП, програмісти моделюють об'єкти, які представляють реальні або віртуальні сутності, та визначають взаємодію між цими об'єктами.

Unity повністю підтримує об’єктно-орієнтований підхід та його принципи.

В Unity, класи визначають об'єкти, які існують в грі. Наприклад, може бути клас "Player", який має характеристики (наприклад, здоров'я, швидкість) та методи (наприклад, "продвинутися" чи "атакувати"). Об'єкти, створені з цього класу, будуть представляти конкретних гравців в грі.

Unity сприяє інкапсуляції через використання модифікаторів доступу (public, private, protected) та властивостей. Це допомагає обмежувати доступ до даних та методів, забезпечуючи правильне взаємодію між об'єктами.

Наслідування використовується для створення нових класів, які успадковують властивості та методи інших класів. У відеоігровій розробці це може включати створення базового класу "Enemy" і дочірніх класів, таких як "Zombie" або "Robot", які успадковують основні характеристики ворогів, але можуть мати унікальні особливості.

В Unity поліморфізм може виявлятися використанням віртуальних методів та інтерфейсів. Наприклад, у випадку персонажів гри, всі вони можуть мати віртуальний метод "Attack", але реалізувати його по-різному.

Робота з інтерфейсом рушія теж має певні особливості. Так вище описані класи, як ті що присутні в самому рушії так і ті, що були створені користувачем розглядаються рушієм як компоненти. Створюючи будь-який об’єкт на сцені користувач має назначити йому певні необхідні компоненти для його роботи. Так, наприклад, створюючи пустий об’єкт користувач отримує майже пустий об’єкт без жодного візуального відображення. Єдиний компонент, що присутній за замовчуванням та який неможливо видалити це Transform, який відповідає за переміщення об’єкту по сцені – простір гри. Якщо ж користувач додасть SpriteRenderer – компонент для відображення спрайтів, які є звичайним зображенням, він зможе задати певне зображення для свого об’єкта і тим самим побачити його. Якщо ще додати RigidBody компонент на об’єкт почне діяти гравітація, у нього з’явиться маса та можна буде створювати певні фізичні розрахунки.

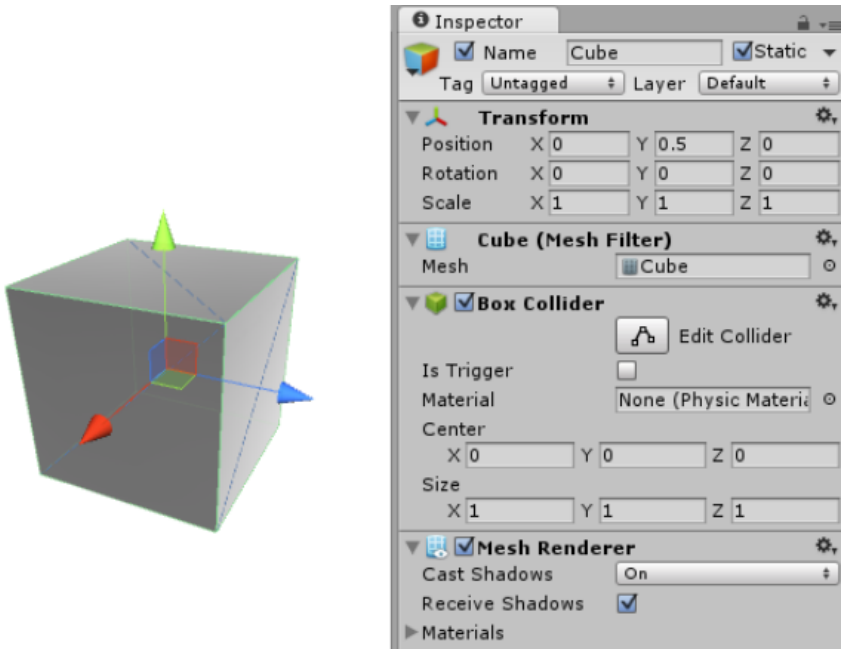

Рисунок 2.4 – Компоненти об'єкта Cube

Як видно на рис. 2.4, у об'єкта під назвою "Cube" є наступні компоненти:

– Transform – відповідає за позицію об'єкта в простору сцени;

– Cube (Mesh Filter) – посилання на 3D-модель кубу;

– BoxCollider – колайдер, компонент, який надає можливість об'єктам взаємодіяти один з одним. Він описує форму фізичного тіла;

– Mesh Renderer – відповідає за відображання моделі в просторі гри.

Процес створення ігрового застосунку на базі Unity виглядає наступним чином:

– створення сцен;

– створення логіки поведінки об'єктів;

– тестування та налагодження;

– експорт та публікація.

2.5 Програмне забезпечення Aseprite

Для перевірки та демонстрації роботи системи необхідно було створити графічну частину застосунку. Було вирішено створювати 2D-графіку оскільки це набагато швидше та простіше. Також використання 2D спрощує процес оптимізації застосунку. Для цього використовувалось програмне забезпечення під назвою Aseprite.

Aseprite – це програмне забезпечення для створення та редагування растрової графіки і анімацій. Aseprite є популярним інструментом серед художників та розробників відеоігор, які займаються розробкою піксельного мистецтва та анімацій. Це легкий у використанні та потужний редактор, який дозволяє створювати вражаючу графіку для відеоігор, мультфільмів та інших проектів.

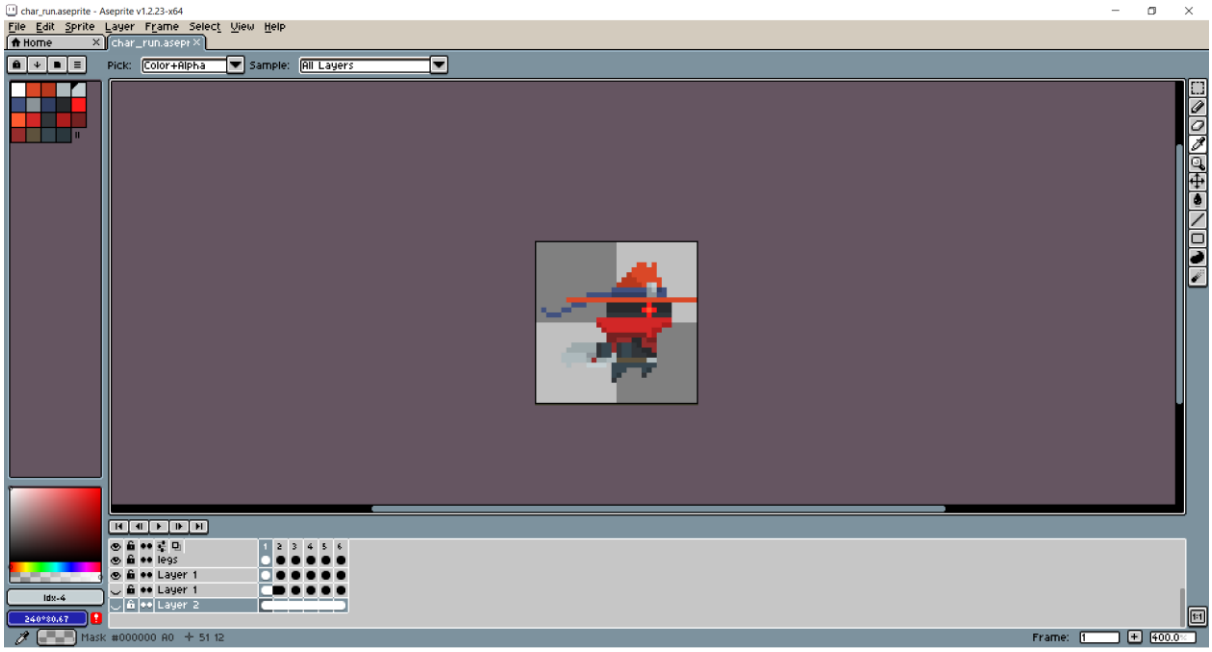

Рисунок 2.5 – Робоче вікно Aseprite

2.6 Програмне забезпечення Photoshop

Для фінального редагування створених анімацій та спрайтів використовувався Photoshop. Adobe Photoshop – це відомий графічний редактор,

розроблений компанією Adobe Inc. Це програмне забезпечення представляє собою потужний інструмент для роботи з растровою графікою, фотографіями, та творчими проектами. Хоча частіше дане забезпечення використовується для роботи із зображеннями високого розміру він також підходить для роботи із піксельною графікою. Провівши деякі налаштування можна отримати доволі зручний інструмент для редагування зображень малих розмірів. Дане програмне забезпечення було обране через більш зручний інструментарій ніж у попередньому Aseprite. Aseprite використовувався в основному для створення анімацій та відповідних spritesheets, що є розкадровкою створеної анімації, яка в подальшому додається до рушія.

Висновки до розділу 2

Було описано методи створення системи адаптації параметрів ігрового простору. Було наведено відповідні формули, проведений детальний опис алгоритму зворотного розповсюдження.

Описані інформаційні технології, які використовувались для створення системи.

Наведено приклади та проведений опис алгоритму генерації рівнів.

3 РОЗРОБКА ТА ДОСЛІДЖЕННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ РОБОТИ СИСТЕМИ АДАПТАЦІЇ ІГРОВОГО ПРОСТОРУ

Дана система складається із декількох підсистем які працюють разом. Першою є агент, який працює за допомогою нейронної мережі. Саме він, отримуючи інформацію від ігрового середовища, здійснює вибір наступного рівня складності. Другою є система генерації рівнів. Вона працює отримуючи дані від агенту складності. Генерація рівнів залежить від отриманих даних, так чим більший рівень складності буде обрано тим більша кількість кімнат буде згенерована. Самі кімнати поділяються на 3 типи – стартова, бойова та кімната зі скарбами. Рівень складності впливає на відсоткове співвідношення бойових кімнат до кімнат зі скарбами. Чим більший рівень складності тим менші шанси у гравця натрапити на кімнату зі скарбами. Третьою є система, яка контролює самі сутички, а саме кількість хвиль в бойових кімнатах та кількість ворогів на одну хвилю.

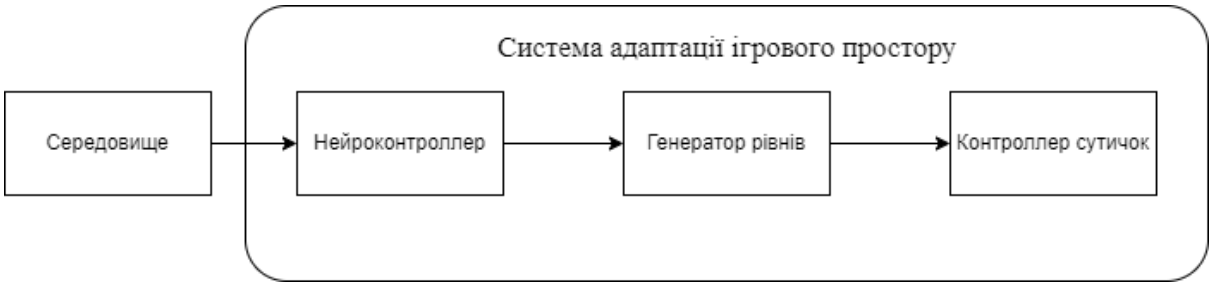

Рисунок 3.1 – Принцип роботи системи

Цикл роботи системи виглядає наступним чином:

1) гравець проходить рівень;
2) запис показників гравця;
3) перевірка показників;
4) визначення наступного рівня складності;
5) генерація наступного рівня.

3.1 Створення агенту складності

Для пришвидшення розробки та навчання агенту його створення проводилось поза середовищем ігрового рушія.

Навчання нейронних мереж поза середовища Unity може мати декілька переваг порівняно з навчанням у Unity, особливо якщо йдеться про використання моделей у широкому спектрі застосувань:

– універсальність. Навчальні задачі, що не вимагають використання конкретного віртуального середовища, можуть бути розв'язані в більш універсальний спосіб. Моделі можна навчати на реальних або синтетичних даних, що відкриває можливість застосування їх у різних областях. Оскільки дана система не прив’язана до конкретного проєкту то доцільно навчати її поза ігровим середовищем, що надасть змогу використовувати її для різних ігор;

– швидкість тренування. Використання відокремленого середовища для навчання може бути ефективніше з точки зору обчислювальних ресурсів і часу. Деякі завдання можуть бути вирішені швидше, коли не потрібно взаємодіяти з графічним інтерфейсом Unity або обмежуватися фізичним моделюванням у середовищі Unity;

– незалежність від графіки. У випадку, коли навчання не пов'язане з обробкою графічних об'єктів або фізичних властивостей, можна обійтися без використання графічних функцій Unity, спрощуючи тим самим процес тренування;

– можливість використання спеціалізованих бібліотек. Зовнішні інструменти та бібліотеки для навчання глибоких нейронних мереж можуть бути легше і ефективніше інтегровані в навчальний процес поза середовищем Unity, спрощуючи взаємодію зі специфічними інструментами та бібліотеками.

Середовищем розробки було обрано Visual Studio. Для створення агенту, який є нейронною мережею створився звичайний консольний застосунок в якому було описано логіку навчання та тестування.

3.2 Вхідні дані для мережі

Для правильної роботи агента необхідно вирішити які саме дані будуть подаватись на його входи та відповідні правила. Тобто спочатку необхідно було вирішити які саме дані можуть відображати рівень навичок гравця. Звичайно в залежності від гри ці дані будуть різні та їх кількість може бути більшою або меншою. До можливих статистик можна віднести:

– час проходження рівня. Час за який гравець долає рівень або якусь частину мапи, якщо гра у відкритому світі, може напряму свідчити про обізнаність гравця;

– точність стрільби. Якщо це гра у жанрі шутеру точність стрільби це доволі добрий показник рівня навичок гравця, оскільки це свідчить про його ознайомленість із конкретною грою та із жанром вцілому;

– кількість пошкоджень. Очки здоров'я теж можуть бути гарним показником рівня навичок користувача. Чим менше він їх втрачає тим кращим гравцем його можна назвати;

– середній час гри. Тобто скільки саме часу користувач зможе прожити у світі без смертей;

– відсоток перемог та поразок. Дана статистика використовується в онлайн іграх із змагальною складовою;

– час реакції. Вимірюється час між подією в грі та реакцією гравця. Це може бути важливим показником швидкості та точності;

– ефективність стратегії. Якщо гра базується на стратегічних рішеннях, можна вивчати, наскільки гравець успішно використовує стратегії та тактики.

Для системи адаптації параметрів ігрового простору було вирішено використовувати кількість очків здоров'я, попередньо обраний рівень складності, кількість пошкоджень які гравець отримав піл час проходження рівня та кількість пошкоджень які було нанесено гравцем. Час не є добрим показником прогресу у даному випадку, оскільки rogue-подібні ігри доволі хаотичні через велику кількість

систем які працюють за допомогою випадкової генерації, тому передбачити час не є можливим.

3.3 Структура створюваної мережі

Процес навчання проводився на різних структурах мережі.

Перші ітерації проводились на мережах із 4 вхідними нейронами, 3 нейронами прихованого шару та 4 нейронами вихідного.

При тестуванні таких мереж виявилось, що дана структура не призводить до бажаного результату.

Після деяких ітерацій тренування було вирішено зупинитись на наступній структурі нейронної мережі.

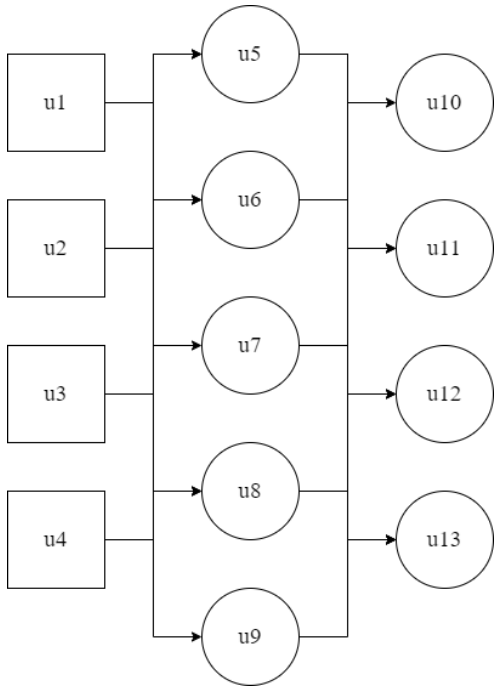

Рисунок 3.2 – Архітектура мережі

Як видно на рис. 3.2 дана мережа складається із 4 нейронів вхідного шару, 5 нейронів прихованого шару, та 4 нейронів вихідного шару.

3.4 Розробка нейронної мережі

Далі було проведено розробку нейронної мережі. Для початку було створено допоміжні функції такі як функції для заповнення відповідних колекцій, які репрезентували шари нейронної мережі, випадковими значеннями ваг та функції для підрахунку сигмоїди та її похідної.

Після чого було створено функції для опису прямого та зворотного сигналів проходження по мережі.

```
static void FeedForward()
{
    double sum;

    for(int i = 0; i < HIDDEN_NEURONS; i++)
    {
        sum = 0.0;

        for(int j = 0; j < INPUT_NEURONS; j++)
        {
            sum += inputs[j] * wih[j, i];
        }

        sum += wih[INPUT_NEURONS, i];

        hidden[i] = Sigmoid(sum);
    }

    for(int i = 0; i < OUTPUT_NEURONS; i++)
    {
        sum = 0.0;

        for(int j = 0; j < HIDDEN_NEURONS; j++)
        {
            sum += hidden[j] * who[j, i];
        }

        sum += who[HIDDEN_NEURONS, i];

        actual[i] = Sigmoid(sum);
    }
}
```

```
static void BackPropagate()
{
    for(int i = 0; i < OUTPUT_NEURONS; i++)
    {
        erro[i] = (target[i] - actual[i]) * SigmoidDerivative(actual[i]);
    }

    for(int i = 0; i < HIDDEN_NEURONS; i++)
    {
        errh[i] = 0.0;

        for(int j = 0; j < OUTPUT_NEURONS; j++)
        {
            errh[i] += erro[j] * who[i, j];
        }

        errh[i] *= SigmoidDerivative(hidden[i]);
    }

    for(int i = 0; i < OUTPUT_NEURONS; i++)
    {
        for(int j = 0; j < HIDDEN_NEURONS; j++)
        {
            who[j, i] += LEARN_RATE * erro[i] * hidden[j];
        }

        who[HIDDEN_NEURONS, i] += LEARN_RATE * erro[i];
    }

    for(int i = 0; i < HIDDEN_NEURONS; i++)
    {
        for(int j = 0; j < INPUT_NEURONS; j++)
        {
            wih[j, i] += LEARN_RATE * errh[i] * inputs[j];
        }

        wih[INPUT_NEURONS, i] += LEARN_RATE * errh[i];
    }
}
```

Рисунок 3.3 – Функції для прямого та зворотного проходження сигналу

Далі було створено відповідну структуру для створення навчального набору.

```
struct DataSample
{
    public double Health;
    public double DifficultyLevel;
    public double HitCount;
    public double HitPerSecond;
    public double[] output;
}
```

Рисунок 3.4 – Структура DataSample

Як видно на рис. 3.4 структура DataSample відображає дані для мережі описані раніше та додатково вона містить масив output який є бажаним результатом роботи мережі.

```
new DataSample
{
    Health = 10.0,
    DifficultyLevel = 1.0,
    HitCount = 20.0,
    HitPerSecond = 0.5,
    output = new double[] {1.0, 0.0, 0.0, 0.0}
},
```

Рисунок 3.5 – Приклад вхідного запису

Як видно на рис. 3.5 в структурі є 5 полів:

– Health. Дане поле відображає очки здоров'я користувача, значення у даного поля лужать у межах від 0 до 100;

– DifficultyLevel. Поле, яке відображає обрану раніше складність. Значення цього поля лежать в межах від 1 до 4. 1 відображає найпростіший рівень складності, 4 – найскладніший;

– HitCount. Поле, яке відображає кількість пошкоджень отриманих гравцем. Дане поле може мати різні порогові значення в залежності від гри, але в цьому випадку максимальне значення є 45;

– HitPerSecond. Кількість пошкоджень які наніс гравець ворожим персонажам. Максимальне значення може різнитись в залежності від проєкту, в цьому випадку максимальним є значення 3;

– Output. Поле яке є колекцією. Можливими значеннями можуть бути {1, 0, 0, 0}, що значить що обрано перший рівень складності, або {0, 0, 1, 0} – третій рівень складності. Дане поле відображає бажаний результат роботи системи.

Всього навчальний набір складається із 27 записів, де описані різноманітні комбінації. Даний набір не має великої кількості записів, оскільки мережі необхідно знайти певні відношення між даними та навчитись їх визнавати на реальних даних. Як видно із описаного датасету, діапазон значень є доволі великим, саме тому і було вирішено використовувати нейронні мережі.

3.4 Навчання та тестування нейронної мережі

Навчання проводилось поза середовищем ігрового рушія. Зупинка циклу навчання проводилась після досягнення певної кількості ітерацій.

Перший цикл проводився в 100000 ітерацій з кроком 0,1, кількість нейронів прихованого шару – 3. Як видно на рис. 3.6 точність мережі становить 58% та не є задовільною. Є помилки як в навчальному так і в тестових наборах.

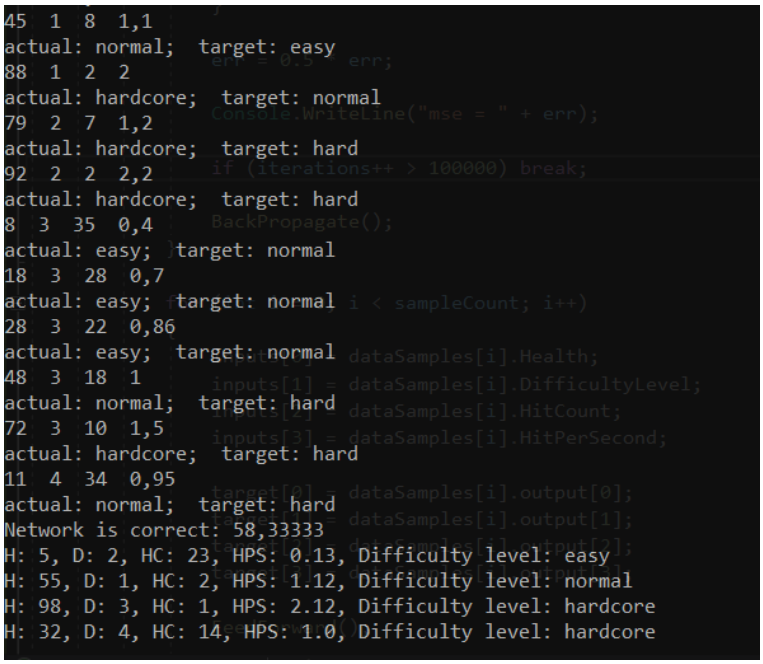

Рисунок 3.6 – Результати першого циклу навчання

Другий цикл – 100000 ітерацій, крок навчання – 0,01, кількість нейронів прихованого шару – 3.

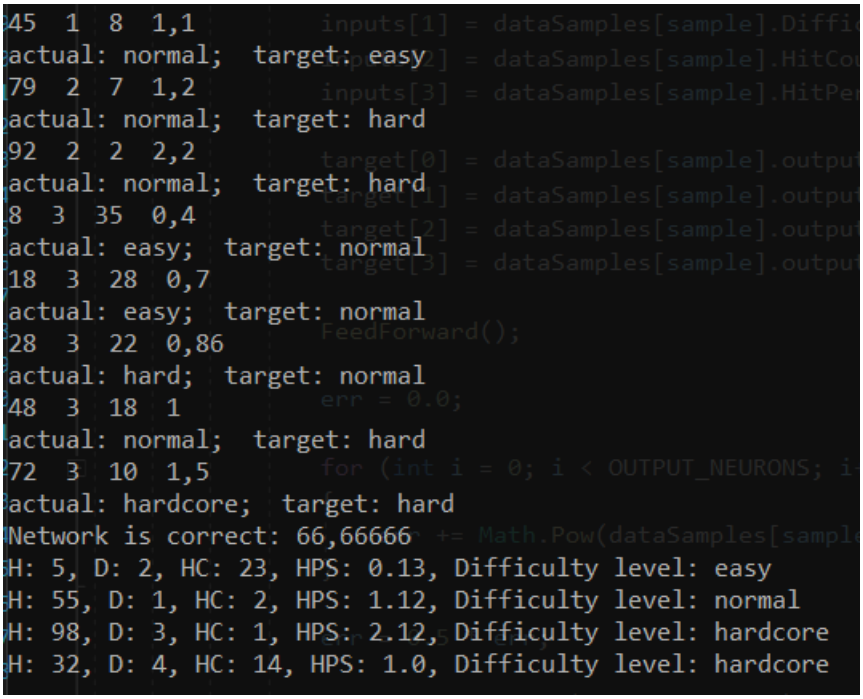

Рисунок 3.7 – Результати другого циклу навчання

Після зміни кроку навчання результат покращився та став рівним 66%, але все ще не є задовільним. Присутні помилки в обох наборах.

Після чого було вирішено не стартувати кожного разу з випадкових ваг, а після отримання певного значення точності мережі зберегти ваги та починати нові цикли навчання з них. Даний підхід до навчання називається методом контрольних точок.

Термін "контрольні точки" (checkpoint) в контексті навчання нейронних мереж застосовується для опису збережених станів моделі під час тренування. Контрольні точки використовуються для збереження параметрів мережі та інших важливих даних, щоб можна було пізніше відновити навчання або використовувати навчену модель для інференсу без повторного тренування.

Важливо відзначити, що термін "контрольні точки" застосовується в широкому контексті до різних областей, де можуть використовуватися збережені стани для подальшого використання або відновлення.

У контексті навчання нейронних мереж контрольні точки можуть включати в себе:

– ваги та параметри мережі. Збереження ваг і параметрів, які були навчені протягом певного періоду тренування;

– оптимізатор та стан оптимізації. Інформація про оптимізатор та його стан в даний момент тренування, що дозволяє відновити процес оптимізації;

– метрики тренування. Збереження значень метрик, таких як точність чи функція втрат, щоб мати інформацію про ефективність мережі;

– інші параметри. Залежно від конкретного застосування, можуть також зберігатися інші параметри чи дані, які важливі для подальшого тренування або використання моделі;

– використання контрольних точок дозволяє зменшити ризик втрати результатів тренування в разі непередбачених обставин або зберегти проміжні результати для подальших експериментів.

В даному випадку мінімальним допустимим значенням було обрано 60%. Далі якщо точність мережі на наступному циклі навчання буде більшим за попередні то ваги цього циклу будуть записані до файлу та використанні при новому навчанні. Даний підхід надав змогу більш точно налаштовувати процес навчання. Оскільки можна було змінити ваги між циклами. Після чого було проведено третій цикл, результат якого був 68%.

Четвертий цикл – 100000 ітерацій, крок 0,001, кількість нейронів прихованого шару – 3.

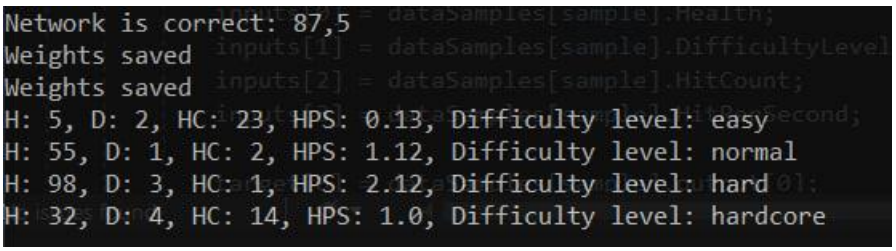

Рисунок 3.8 – Результати четвертого циклу навчання

Як видно на рис. 3.8 точність мережі зросла – 87%, але все ще не є задовільною, також присутні помилки при тестовому наборі. Після цього проводилось ще декілька циклів навчання але значення точності мережі не змінилось. Було вирішено спочатку перевірити та розширити навчальну вибірку, оскільки на початку її розмір був меншим від описаного раніше. Після перевірки було усунено декілька помилок в уже наявних записах та додані нові для кращої репрезентації залежностей.

Оскільки було змінено навчальну вибірку, продовжувати навчання із попередньої контрольної точки не було можливим. Якщо змінюється навчальна вибірка, наприклад, за рахунок додавання нових класів чи прикладів, то розподіл даних може змінитися. Ваги, навчені на попередніх даних, можуть бути менше ефективними для нових даних. Це один з факторів який необхідно враховувати при навчанні мережі за допомогою контрольних точок.

Для продовження процесу навчання було скинуто всі записи попередніх ваг та знову проводилось навчання з випадкових значень. Далі все робилось як було описано раніше – якщо точність перевищувала межу, ваги зберігались. Але даний підхід все ще не дав бажаних результатів. Мережа повинна перевищувати 90%, натомість було отримано 82%. І знову був досягнутий ліміт. Зміна кроку навчання не приводила до жодних змін. До того ж точність мережі була гіршою за попередні спроби. Тому було вирішено змінити архітектуру мережі – збільшити кількість нейронів прихованого шару з 3 до 4.

І знову зміна архітектури мережі призвела до необхідності починати процес навчання з самого початку. Якщо змінюється архітектура мережі (наприклад, зміна кількості шарів чи нейронів), то параметри, збережені в контрольній точці, можуть втратити свою актуальність або не відповідати новій структурі мережі. При зміні архітектури можуть виникнути проблеми згасання або вибуху градієнта, особливо якщо нова архітектура відрізняється за розміром чи структурою. Це може призвести до труднощів у навчанні.

Тому було скинуто попередні ваги та процес навчання було почато з початку. Через кілька циклів навчання було отримано значення 88%, що є кращим за попередні спроби але все ще не є задовільним результатом. Наступні цикли не покращили результати моделі. Тому було вирішено збільшити кількість нейронів прихованого шару з 4 до 5.

Останній цикл – 100000 ітерацій, крок навчання 0,001, кількість нейронів прихованого шару – 5.

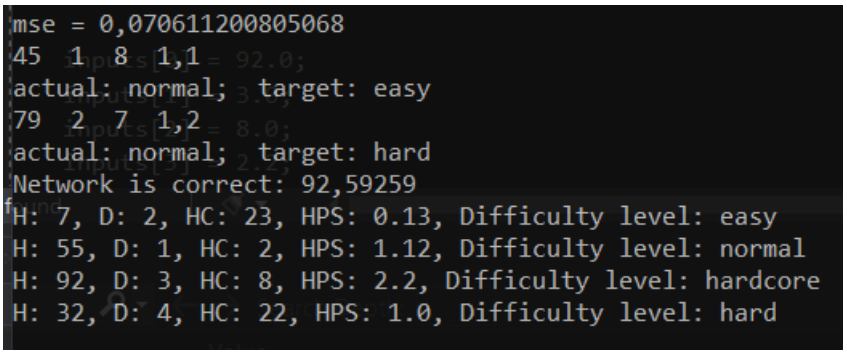

Рисунок 3.9 – Результати останнього циклу навчання

Як видно на рис. 3.9 точність моделі зросла до 92%, що є задовільним. Тестова вибірка була класифікована вірно. Середньоквадратична похибка склала 0,07.

3.5 Інтеграція нейронної мережі в ігрове середовище Unity

Після закінчення навчання необхідно інтегрувати отриману модель в ігрове середовище Unity. Для цього необхідно враховувати певні особливості роботи рушія, а саме життєвий цикл застосунку (unity`s lifecycle).

Unity надає ряд подій та методів, які можна перевизначити у скриптах для керування життєвим циклом об'єктів у грі. Основні етапи цього циклу включають:

– Awake. Метод Awake викликається один раз при створенні об'єкта на сцені. Використовується для ініціалізації ресурсів, які будуть використовуватися під час життєвого циклу об'єкта;

– OnEnable. Метод OnEnable викликається при кожному разі, коли об'єкт стає активним. Використовується для відновлення після виклику OnDisable;

– Start. Метод Start викликається перед першим оновленням. Зазвичай використовується для ініціалізації, яка потрібна після Awake, але перед початком гри;

– Update. Метод Update викликається кожен кадр і використовується для обробки основної логіки гри;

– FixedUpdate. Метод FixedUpdate викликається з фіксованою частотою (зазвичай використовується для фізичних операцій);

– LateUpdate. Метод LateUpdate викликається після всіх Update. Використовується для гарантованого виклику після оновлення інших об'єктів;

– OnDisable. Метод OnDisable викликається, коли об'єкт стає неактивним. Використовується для прибирання ресурсів та виключення змін;

– OnDestroy. Метод OnDestroy викликається перед тим, як об'єкт буде знищений. Використовується для прибирання залишкових ресурсів і відписки від подій.

Цей цикл дозволяє точно налаштовувати та контролювати роботу коду під час життєвого циклу об'єктів в Unity. Насправді життєвий цикл Unity набагато складніший за описаний вище, але оскільки робота системи здійснюється при старті гри нас цікавить тільки перша група методів: Awake(), Start().

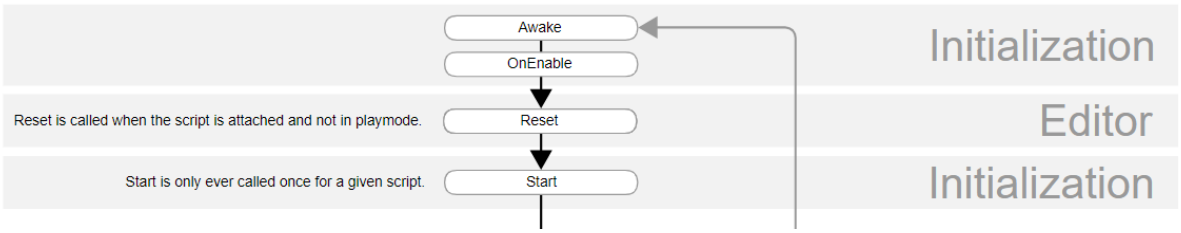

Рисунок 3.10 – Частина життєвого циклу застосунку

Як видно із рис. 3.10 даний набір методів та подій виконується перед самим початком гри та те, що Awake() виконується раніше за Start(). Це необхідно розуміти для вірної ініціалізації всіх систем. Тому що деякі з них мають ініціалізуватись в Awake() інші ж в Start(). Це необхідно, бо системам необхідно отримати відповідні посилання для роботи із необхідними їм даними.

3.6 Створення генератора рівнів

Генератор рівнів – це система, яка розроблялась в середовищі Unity.

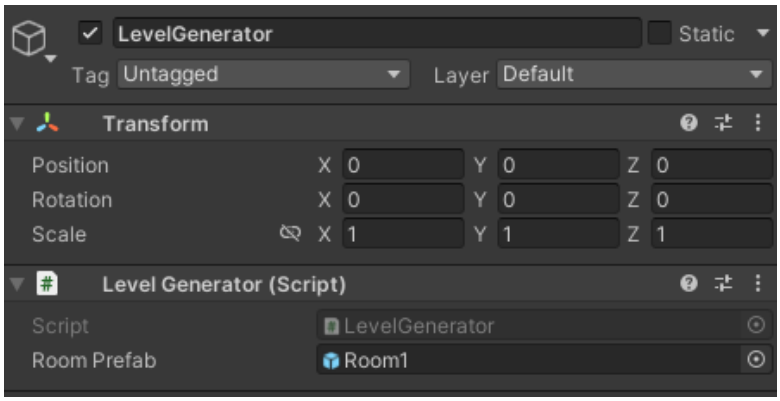

Рисунок 3.11 – Компоненти об'єкта Level Generator

Оскільки це система, а не фізичний об’єкт єдиними компонентами є Transform та Level Generator, який є скриптом із логікою роботи системи. Room Prefab – це посилання на префаб кімнати, яку буде створювати генератор.

Префаб – це особливий тип ассетів, що дозволяє зберігати весь GameObject з усіма компонентами та значеннями властивостей. Префаб виступає в ролі шаблону для створення екземплярів об'єкта, що зберігається в сцені. Будь-які зміни в префабі негайно відображаються і на всіх його екземплярах, при цьому є можливість перевизначати компоненти та налаштування для кожного екземпляра окремо.

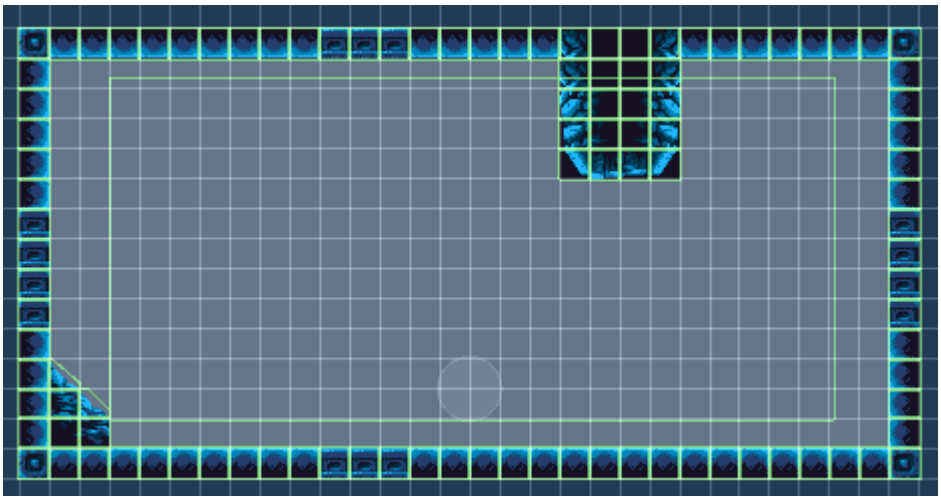

Рисунок 3.12 – Вид префабу Room

Префаб Room створювався з використанням тайлсетів.

Тайлсет, або набір тайлів, представляє собою комплект фрагментів, які використовуються у грі. Кожен окремий тайл у цьому наборі представляє графічний об'єкт, такий як ділянка місцевості, об'єкт або персонаж. Високоякісні набори часто включають перехідні тайли, що дозволяють безшовно з'єднувати їх і створювати різноманітне та візуально згуртоване ігрове оточення. Наприклад, між тайлами різних ландшафтів використовується той, який забезпечує плавний перехід від одного до іншого. Кожен тайл може мати фіксовану орієнтацію або обертатися під певним кутом, надаючи можливість створення різноманітних зображень. Тайлсети можуть бути об'єднані в один файл або розділені на кілька файлів.

Мапа тайлів, або tilemap, це схема, яка визначає розташування тайлів і визначає дизайн та візуальний вигляд ігрового світу. Розміщуючи тайли на сітку, можна створювати складні ландшафти, лабіринти або будь-яке інше ігрове середовище. Деякі редактори надають можливість накладати карти тайлів у декілька шарів, що допомагає створювати складні та унікальні зображення.

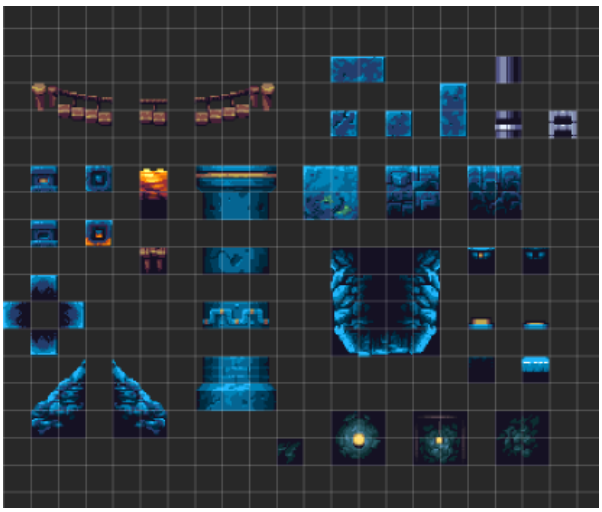

Рисунок 3.13 – Використаний тайлсет

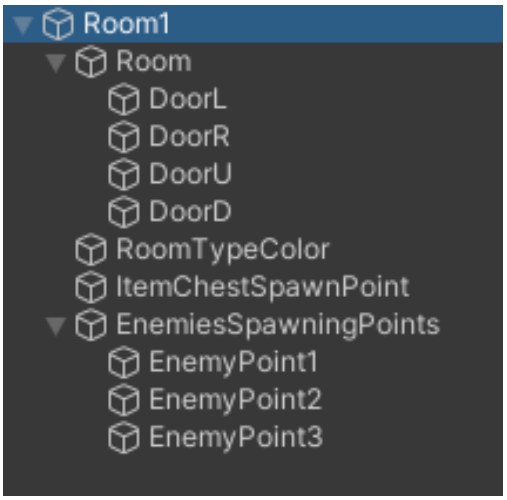

Рисунок 3.14 – Структура префабу Room

Структура префабу Room1 наведена на рис. 3.14. Тут присутні двері на всі 4 сторони, які відчиняються чи зачиняються при певних умовах. До цих умов належать умови знаходження гравця в кімнаті та наявність сусідніх кімнат. Перевірка знаходження гравця в кімнаті проводиться із використанням трігерного колайдеру. Це колайдер, який реєструє не фізичні зіткнення, а пересічення його границь та знаходження в них.

```
private void Awake()
{
    DifficultyAgent difficultyAgent = new DifficultyAgent();

    difficultyAgent.ReadWeights("Assets/Data/wih.txt", difficultyAgent.wih);
    difficultyAgent.ReadWeights("Assets/Data/who.txt", difficultyAgent.who);
    difficultyAgent.ReadData("Assets/Data/PlayerSkill.txt", difficultyAgent.inputs);

    _difficulty = difficultyAgent.DetermineDifficulty();
    Debug.Log("Difficulty: " + _difficulty);

    _maxRooms += _difficulty;
    CreateLayout();
    CheckNeighbors();
    AssignRoomType();

    foreach (Room room in _rooms)...

    SpawnRooms();
}
```

Рисунок 3.15 – Алгоритм роботи генератора рівнів

На рис. 3.15 наведено роботу генератора рівнів. Видно, що спочатку йде створення екземпляру агенту складності та зчитування ваг, які були отримані після навчання. Далі йде зчитування параметрів користувача та починається робота агенту складності.

```
public int DetermineDifficulty()
{
    FeedForward();

    return Difficulty(actual);
}
```

Рисунок 3.16 – Функція для визначення рівня складності

Метод DetermineDifficulty(), який викликається із скрипту генератора рівня є функцією прямого проходження сигналу по мережі, далі йде повернення отриманого рівня складності – число в діапазоні від 0 до 3.

Після визначення рівня складності починається генерація мапи та створення кімнат в ігровому просторі. При створенні кімнаті призначається відповідний тип та кількість ворожих хвиль.

3.7 Перевірка роботи системи

Перевірка роботи системи проводилась на декількох прикладах. Результати наведені на рис. 3.17 – 3.20.

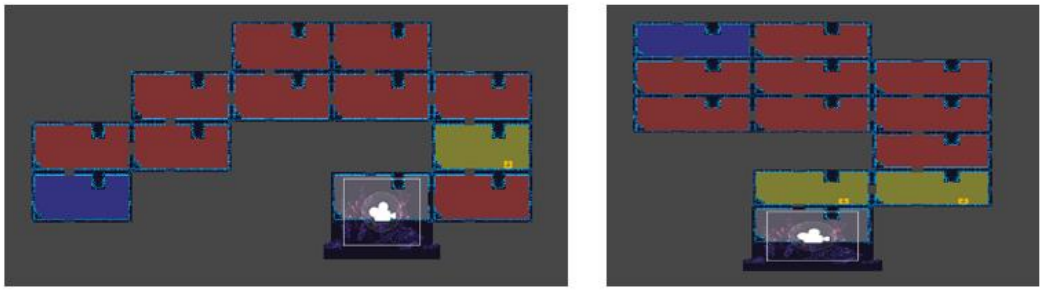

Рисунок 3.17 – Перша перевірка роботи системи

Для перевірки роботи системи використовувались записані параметри успішності гравця на попередніх рівнях (див. рис. 3.18, 3.20). Поле Chosen difficulty level відповідає за обраний системою рівень складності.

Було проведено декілька запусків на одній і тій самій складності для перевірки роботи нейроконтролера, який класифікує гравця та для перевірки роботи додаткових систем, таких як генератор рівнів та контролеру кімнат. Які, в даному випадку, відповідають за процес зміни параметрів відповідно обраного рівня складності.

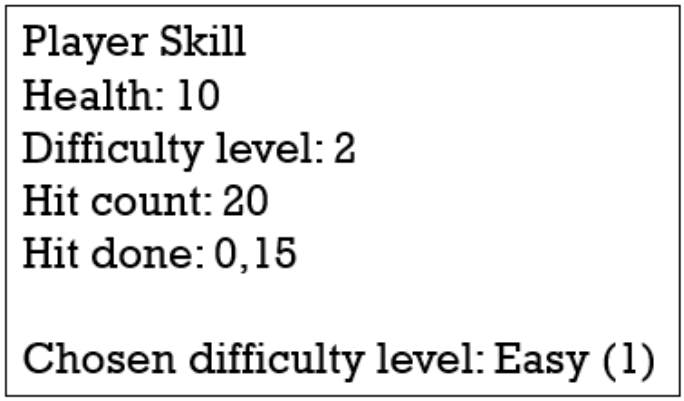

Рисунок 3.18 – Результати роботи системи при першій перевірці

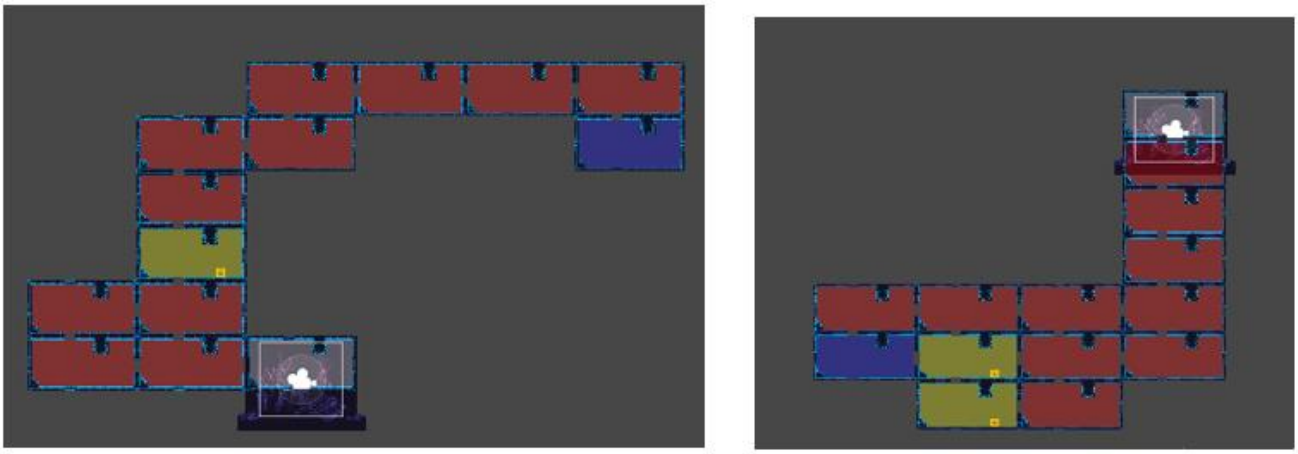

Рисунок 3.19 – Друга перевірка роботи системи

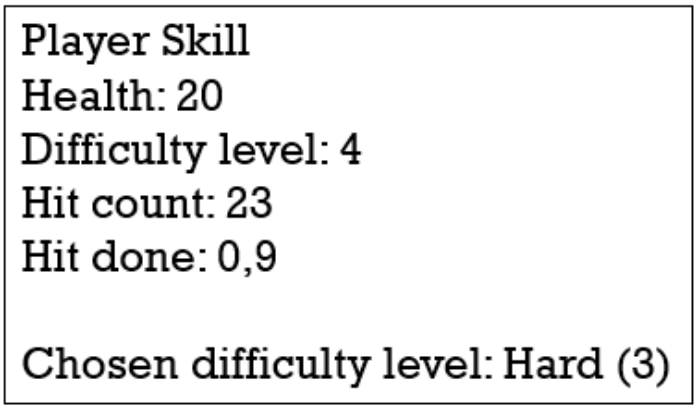

Рисунок 3.20 – Результати роботи системи при другій перевірці

Як видно із наведених рисунків система вірно класифікувала гравця та обрала відповідний рівень складності, після чого було згенеровано відповідний рівень. В наведених прикладах кожна кімната зафарбована відповідним кольором для демонстраційних цілей. Червоний колір відображає кімнати з ворогами, жовтий – кімнати зі скарбами, синій – перехід на наступний рівень, білий – стартова кімната.

Структура рівнів відрізняється оскільки їхня генерація випадкова.

Хоча створена система добре відпрацювала її можна дещо покращити. Першим кроком можна назвати додавання більшої кількості параметрів, які відповідають за успішність гравця. Це дало б змогу системі точніше працювати з точки зору експерта. Другим кроком є додавання більшої кількості параметрів ігрового простору, на які здатна вплинути система. Можливо це буде не тільки структура мапи рівня, а й самі його складові, тобто кімнати. Різноманітні типи та структури кімнат могли б покращити досвід користувача. Окрім кімнат можна також впливати на поведінку ворожих персонажів. Робити її відповідно складнішою чи простішою. Також можна було б впливати на типи ворогів, які б з’являлись в залежності від складності. Тобто якщо рівень складності був обраний високим, то гравець зіткнеться із раніше небаченим ворогом.

Висновки до розділу 3

Було описано структуру системи та її цикл, проілюстровано принцип її роботи. Було описано процес створення нейроконтролера, його вхідні дані. Було проведено опис структури створюваного нейроконтролера та процес його навчання. Був описаний метод контрольних точок за допомогою якого проводилось навчання.

Було описано життєвий цикл ігрового рушія Unity, який необхідно було враховувати при інтеграції нейроконтролера до ігрового середовища.

Було описано розробку системи та проведено її тестування.

Результати роботи системи демонструють, що багатошаровий перцептрон є підходящим інструментом для вирішення складних задач ігрового дизайну.

ВИСНОВКИ

Виконавши кваліфікаційну роботу магістра було проведено аналіз предметної сфери, а саме складності в відеоіграх, різних підходів розробників до складності та оглянуто наявні аналоги. Було проаналізовано жанр rouge-подібних ігор для кращого його розуміння, це дало змогу розробити систему, яка впливає на більшу кількість параметрів ігрового простору. Було проведено дослідження наявних аналогів. Було проведено огляд інструментів та підходів для вирішення поставленої задачі та описано особливості відео-ігрових застосунків, які потрібно враховувати при розробці системи.

Було описано методи глибокого навчання, багатошаровий перцептрон, його цикл навчання та наведено відповідні формули. Були описані інструменти, які були використані при розробці системи, а саме:

- інтегроване середовище розробки Unity;
- інтегроване середовище розробки Visual Studio;
- програмне забезпечення Aseprite;
- програмне забезпечення Photoshop.

Був описаний процес створення системи адаптації параметрів ігрового простору, а саме процес навчання та тестування, створення додаткових підсистем. Описаний процес інтеграції розробленої нейронної мережі до середовища Unity.

В результаті виконання роботи було створено інтелектуальну систему адаптації параметрів ігрового простору для комп’ютерних ігор. Багатошаровий перцептрон являється інтелектуальною складовою системи. Архітектура мережі виглядає наступним чином:

- 4 нейрони вхідного шару;
- 5 нейронів прихованого шару;
- 4 нейрони вихідного шару.

Навчання інтелектуальної складової проводилось в декілька етапів, до моменту досягнення бажаного результату. Точність системи – 92%, середньоквадратична помилка – 0,07.

Було створено додаткові компоненти системи, а саме генератор рівнів та контролер сутичок. Разом усі підсистеми склали інтелектуальну систему адаптації параметрів ігрового простору для комп’ютерних ігор.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Donovan Tristan. Replay: The History of Video Games. East Sussex: Yellow Ant, 2010. 516 с.
2. Wolf M. J. P. Encyclopedia of Video Games: The Culture, Technology, and Art of Gaming / ed. by Mark J. P. Wolf. Santa Barbara, — CA: Greenwood, 2012. 788 с.
3. Вступ до історії геймдизайну : вебсайт. URL: https://vokigames.com/ua/vstup-do-istoriyi-gejmdyzajnu-chastyna-1-arkadni-igry/ (дата звернення 15.11.2023).
4. Kohler Chris. Power-Up: How Japanese Video Games Gave the World an Extra Life. Brady Games, 2005. 336 с.
5. Burgun Keith. Game Design Theory: A New Philosophy for Understanding Games. A K Peters / CRC Press, 2012. 188 с.
6. Elias George Skaff. Characteristics of Games. MIT Press, 2012. 336 с.
7. Bates Bob. Game Design (2nd ed.). Thomson Course Technology, 2004. 450 с.
8. Going Rogue: A Brief History of the Computerized Dungeon Crawl : вебсайт. URL: https://web.archive.org/web/20160919020229/http://insight.ieeeusa.org/insight/content/views/371703.
9. Procedural vs. Randomly Generated Content in Game Design : вебсайт. URL: https://www.gamedeveloper.com/design/procedural-vs-randomly-generated-content-in-game-design#close-modal (дата звернення: 05.01.2024)
10. Left 4 Dead Hands-on Preview : вебсайт. URL: https://webcitation.org/66UDpyT8d?url=http://www.left4dead411.com/left-4-dead-preview-pg2 (дата звернення: 01.12.2023).
11. Russell Stuart J., Norvig Peter. Artificial Intelligence: A Modern Approach. Upper Saddle River, New Jersey: Prentice Hall, 2003. 1136 с.

12. Domingos Pedro. The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World. Basic Books, 2015. 352 с.

13. Hopfield J. J. Neural networks and physical systems with emergent collective computational abilities. / Proc. Natl. Acad. Sci., 1984. V. 9.

14. Gallant S. L. Neutral Network Learning and Expert Systems. Cambridge Mass, MIT Press, 1994. 382 с.

15. Розенблатт Ф. Принципи нейродинаміки: Перцептрони та теорія механізмів мозку, 1965. 480 с.

16. Minsky M., Papert S. Perceptrons: An Introduction to Computational Geometry. Cambridge, Mass.: MIT Press, 1969. 308 с.

17. Hornik Kurt. Approximation Capabilities of Multilayer Feedforward Networks, 1991.

18. Intel. Game Optimization Methodology : вебсайт. URL: https://www.intel.com/content/www/us/en/docs/gpa/user-guide/2022-4/game-optimization-methodology.html (дата звернення: 20.12.2023).

19. Matt Buckland. Programming Game AI By Example, 2004. 495 с.

20. On Definition of Deep Learning. / Zhang W. J., Yang G., Ji C., Gupta M. M. World Automation Congress (WAC), 2018.

21. Calin Ovidiu L. Deep learning architectures: a mathematical approach. Springer series in the data sciences. Cham, Switzerland: Springer, 2020. 790 с.

22. G. Cybenko. Approximations by superpositions of sigmoidal functions. Mathematics of Control, Signals, and Systems, 1989.

23. Morgan Kaufman. Artificial Intelligence for Games, 2005. 896 с.

24. M. Tim Jones. AI Application Programming, 2004. 496 с.

25. Офіційний сайт Visual Studio : вебсайт. URL: https://www.visualstudio.com (дата звернення 10.11.2023).

26. Офіційний сайт Unity3D : вебсайт. URL: https://unity3d.com (дата звернення 10.11.2023).

27. MacKay D. Information Theory, Inference and Learning Algorithms. Cambridge University Press, 2003. 640 с.

28. Poole David, Mackworth Alan, Goebel Randy. Computational Intelligence: A Logical Approach. New York: Oxford University Press, 1998. 576 с.

29. Christopher Bishop. Neural Networks for Pattern Recognition. Oxford University Press, 1995. 502 с.

30. Philip D. Wasserman. Neural Computing: Theory and Practice. Coriolis Group, 1989. 102 с.

31. Simon S. Haykin. Neural Networks: A Comprehensive Foundation. Prentice Hall, 1999. 842 с.

32. Rosenblatt Frank. The Perceptron: A Probabilistic Model For Information Storage And Organization in the Brain, 1958. 408 с.

ДОДАТОК А
ЛІСТИНГ КОДУ РОЗРОБЛЕНОЇ СИСТЕМИ

Файл MLP.cs

```
using System;
using System.IO;
using System.Globalization;

namespace difficultyAgent
{
   class Program
   {
      const int INPUT_NEURONS = 4;
      const int HIDDEN_NEURONS = 5;
      const int OUTPUT_NEURONS = 4;

      static double[,] wih = new double[INPUT_NEURONS + 1, HIDDEN_NEURONS];
      static double[,] who = new double[HIDDEN_NEURONS + 1, OUTPUT_NEURONS];

      static double[] inputs = new double[INPUT_NEURONS];
      static double[] hidden = new double[HIDDEN_NEURONS];
      static double[] target = new double[OUTPUT_NEURONS];
      static double[] actual = new double[OUTPUT_NEURONS];

      static double[] erro = new double[OUTPUT_NEURONS];
      static double[] errh = new double[HIDDEN_NEURONS];

      const double LEARN_RATE = 0.001;

      static string[] difficultyLevels = {"easy", "normal", "hard", "hardcore"};

      static DataSample[] dataSamples = {
         new DataSample
         {
            Health = 10.0,
```

```
    DifficultyLevel = 1.0,
    HitCount = 20.0,
    HitPerSecond = 0.5,
    output = new double[] {1.0, 0.0, 0.0, 0.0}
},
new DataSample
{
    Health = 22.0,
    DifficultyLevel = 1.0,
    HitCount = 16.0,
    HitPerSecond = 0.8,
    output = new double[] {1.0, 0.0, 0.0, 0.0}
},
new DataSample
{
    Health = 45.0,
    DifficultyLevel = 1.0,
    HitCount = 8.0,
    HitPerSecond = 1.1,
    output = new double[] {1.0, 0.0, 0.0, 0.0}
},
new DataSample
{
    Health = 73.0,
    DifficultyLevel = 1.0,
    HitCount = 4.0,
    HitPerSecond = 1.6,
    output = new double[] {0.0, 1.0, 0.0, 0.0}
},
new DataSample
{
    Health = 88.0,
    DifficultyLevel = 1.0,
    HitCount = 2.0,
    HitPerSecond = 2.0,
    output = new double[] {0.0, 1.0, 0.0, 0.0}
```

```
},
new DataSample
{
   Health = 5.0,
   DifficultyLevel = 2.0,
   HitCount = 28.0,
   HitPerSecond = 0.2,
   output = new double[] {1.0, 0.0, 0.0, 0.0}
},
new DataSample
{
   Health = 20.0,
   DifficultyLevel = 2.0,
   HitCount = 20.0,
   HitPerSecond = 0.6,
   output = new double[] {1.0, 0.0, 0.0, 0.0}
},
new DataSample
{
   Health = 32.0,
   DifficultyLevel = 2.0,
   HitCount = 16.0,
   HitPerSecond = 0.68,
   output = new double[] {0.0, 1.0, 0.0, 0.0}
},
new DataSample
{
   Health = 48.0,
   DifficultyLevel = 2.0,
   HitCount = 12.0,
   HitPerSecond = 0.85,
   output = new double[] {0.0, 1.0, 0.0, 0.0}
},
new DataSample
{
   Health = 56.0,
```

```
    DifficultyLevel = 2.0,
    HitCount = 9.0,
    HitPerSecond = 0.98,
    output = new double[] {0.0, 1.0, 0.0, 0.0}
},
new DataSample
{
    Health = 79.0,
    DifficultyLevel = 2.0,
    HitCount = 7.0,
    HitPerSecond = 1.2,
    output = new double[] {0.0, 0.0, 1.0, 0.0}
},
new DataSample
{
    Health = 92.0,
    DifficultyLevel = 2.0,
    HitCount = 2.0,
    HitPerSecond = 2.2,
    output = new double[] {0.0, 0.0, 1.0, 0.0}
},
new DataSample
{
    Health = 8.0,
    DifficultyLevel = 3.0,
    HitCount = 35.0,
    HitPerSecond = 0.4,
    output = new double[] {0.0, 1.0, 0.0, 0.0}
},
new DataSample
{
    Health = 18.0,
    DifficultyLevel = 3.0,
    HitCount = 28.0,
    HitPerSecond = 0.7,
    output = new double[] {0.0, 1.0, 0.0, 0.0}
```

```
},
new DataSample
{
   Health = 28.0,
   DifficultyLevel = 3.0,
   HitCount = 22.0,
   HitPerSecond = 0.86,
   output = new double[] {0.0, 1.0, 0.0, 0.0}
},
new DataSample
{
   Health = 48.0,
   DifficultyLevel = 3.0,
   HitCount = 18.0,
   HitPerSecond = 1.0,
   output = new double[] {0.0, 0.0, 1.0, 0.0}
},
new DataSample
{
   Health = 57.0,
   DifficultyLevel = 3.0,
   HitCount = 12.0,
   HitPerSecond = 1.2,
   output = new double[] {0.0, 0.0, 1.0, 0.0}
},
new DataSample
{
   Health = 72.0,
   DifficultyLevel = 3.0,
   HitCount = 10.0,
   HitPerSecond = 1.5,
   output = new double[] {0.0, 0.0, 1.0, 0.0}
},
new DataSample
{
   Health = 81.0,
```

```
    DifficultyLevel = 3.0,
    HitCount = 8.0,
    HitPerSecond = 1.8,
    output = new double[] {0.0, 0.0, 0.0, 1.0}
},
new DataSample
{
    Health = 94.0,
    DifficultyLevel = 3.0,
    HitCount = 7.0,
    HitPerSecond = 2.6,
    output = new double[] {0.0, 0.0, 0.0, 1.0}
},
new DataSample
{
    Health = 2.0,
    DifficultyLevel = 4.0,
    HitCount = 41.0,
    HitPerSecond = 0.6,
    output = new double[] {0.0, 0.0, 1.0, 0.0}
},
new DataSample
{
    Health = 11.0,
    DifficultyLevel = 4.0,
    HitCount = 34.0,
    HitPerSecond = 0.95,
    output = new double[] {0.0, 0.0, 1.0, 0.0}
},
new DataSample
{
    Health = 32.0,
    DifficultyLevel = 4.0,
    HitCount = 28.0,
    HitPerSecond = 1.1,
    output = new double[] {0.0, 0.0, 1.0, 0.0}
```

```
    },
    new DataSample
    {
        Health = 45.0,
        DifficultyLevel = 4.0,
        HitCount = 24.0,
        HitPerSecond = 1.35,
        output = new double[] {0.0, 0.0, 0.0, 1.0}
    },
    new DataSample
    {
        Health = 53.0,
        DifficultyLevel = 4.0,
        HitCount = 20.0,
        HitPerSecond = 1.65,
        output = new double[] {0.0, 0.0, 0.0, 1.0}
    },
    new DataSample
    {
        Health = 75.0,
        DifficultyLevel = 4.0,
        HitCount = 14.0,
        HitPerSecond = 1.95,
        output = new double[] {0.0, 0.0, 0.0, 1.0}
    },
    new DataSample
    {
        Health = 98.0,
        DifficultyLevel = 4.0,
        HitCount = 8.0,
        HitPerSecond = 2.95,
        output = new double[] {0.0, 0.0, 0.0, 1.0}
    }
};

static Random random = new Random();
```

```
static float RandWeight(Random random)
{
   return (float)(random.NextDouble() - 0.5);
}

static int Difficulty(double[] vector)
{
   int sel = 0;
   double max = vector[sel];

   for(int i = 1; i < OUTPUT_NEURONS; i++)
   {
      if(vector[i] > max)
      {
         max = vector[i];
         sel = i;
      }
   }

   return sel;
}

static void AssignRandomWeights(Random random)
{
   for(int i = 0; i < INPUT_NEURONS + 1; i++)
   {
      for(int j = 0; j < HIDDEN_NEURONS; j++)
      {
         wih[i, j] = RandWeight(random);
      }
   }

   for (int i = 0; i < HIDDEN_NEURONS + 1; i++)
   {
      for (int j = 0; j < OUTPUT_NEURONS; j++)
      {
```

```
            who[i, j] = RandWeight(random);
        }
    }
}

static double Sigmoid(double value)
{
    return 1.0 / (1.0 + Math.Exp(-value));
}

static double SigmoidDerivative(double value)
{
    return value * (1.0 - value);
}

static void FeedForward()
{
    double sum;

    for(int i = 0; i < HIDDEN_NEURONS; i++)
    {
        sum = 0.0;

        for(int j = 0; j < INPUT_NEURONS; j++)
        {
            sum += inputs[j] * wih[j, i];
        }

        sum += wih[INPUT_NEURONS, i];

        hidden[i] = Sigmoid(sum);
    }

    for(int i = 0; i < OUTPUT_NEURONS; i++)
    {
        sum = 0.0;
```

```
        for(int j = 0; j < HIDDEN_NEURONS; j++)
        {
            sum += hidden[j] * who[j, i];
        }

        sum += who[HIDDEN_NEURONS, i];

        actual[i] = Sigmoid(sum);
    }
}

static void BackPropagate()
{
    for(int i = 0; i < OUTPUT_NEURONS; i++)
    {
        erro[i] = (target[i] - actual[i]) * SigmoidDerivative(actual[i]);
    }

    for(int i = 0; i < HIDDEN_NEURONS; i++)
    {
        errh[i] = 0.0;

        for(int j = 0; j < OUTPUT_NEURONS; j++)
        {
            errh[i] += erro[j] * who[i, j];
        }

        errh[i] *= SigmoidDerivative(hidden[i]);
    }

    for(int i = 0; i < OUTPUT_NEURONS; i++)
    {
        for(int j = 0; j < HIDDEN_NEURONS; j++)
        {
            who[j, i] += LEARN_RATE * erro[i] * hidden[j];
```

```
            }

            who[HIDDEN_NEURONS, i] += LEARN_RATE * erro[i];
        }

        for(int i = 0; i < HIDDEN_NEURONS; i++)
        {
            for(int j = 0; j < INPUT_NEURONS; j++)
            {
                wih[j, i] += LEARN_RATE * errh[i] * inputs[j];
            }

            wih[INPUT_NEURONS, i] += LEARN_RATE * errh[i];
        }
    }

    static void SaveWeights(string fileName, double[,] weights)
    {
        try
        {
            using(StreamWriter writer = new StreamWriter(fileName))
            {
                foreach(double value in weights)
                {
                    writer.WriteLine(value.ToString(CultureInfo.InvariantCulture));
                }
            }

            Console.WriteLine("Weights saved");
        }
        catch(Exception ex)
        {
            Console.WriteLine($"Error saving {fileName}: {ex.Message}");
        }
    }
```

```
static void ReadWeights(string filePath, double[,] array)
{
    try
    {
        // Read all lines from the file
        string[] lines = File.ReadAllLines(filePath);

        // Determine the number of rows based on the number of lines
        int rows = lines.Length;

        // Populate the array
        for (int i = 0; i < rows; i++)
        {
            if (double.TryParse(lines[i], NumberStyles.Float, CultureInfo.InvariantCulture, out double value))
            {
                // Calculate the array indices based on the row and column
                int row = i / array.GetLength(1);
                int col = i % array.GetLength(1);

                array[row, col] = value;
            }
            else
            {
                Console.WriteLine($"Failed to parse value: {lines[i]}");
            }
        }

        Console.WriteLine($"Successfully loaded data from {filePath}");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error loading data from {filePath}: {ex.Message}");
    }
}

static void SaveNetworkAccuracy(string fileName, float accuracy)
```

```
{
    try
    {
        using (StreamWriter writer = new StreamWriter(fileName))
        {
            writer.WriteLine(accuracy.ToString(CultureInfo.InvariantCulture));
        }

        Console.WriteLine("Network accuracy saved");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error saving {fileName}: {ex.Message}");
    }
}

static float ReadNetworkAccuracy(string filePath)
{
    try
    {
        using (StreamReader reader = new StreamReader(filePath))
        {
            string line = reader.ReadLine();

            if (float.TryParse(line, NumberStyles.Float, CultureInfo.InvariantCulture, out float result))
            {
                return result;
            }
            else
            {
                Console.WriteLine($"Failed to parse the content of {filePath} as a float.");
                return float.NaN;
            }
        }
    }
    catch (Exception ex)
```

```
    {
        Console.WriteLine($"Error reading from {filePath}: {ex.Message}");
        return float.NaN;
    }
}

static void Main(string[] args)
{
    double err;
    int sample = 0;
    int iterations = 0;
    int sum = 0;

    Random random = new Random();
    AssignRandomWeights(random);

    ReadWeights("wih.txt", wih);
    ReadWeights("who.txt", who);

    int sampleCount = dataSamples.Length;
    while (true)
    {
        if (++sample == sampleCount) sample = 0;

        inputs[0] = dataSamples[sample].Health;
        inputs[1] = dataSamples[sample].DifficultyLevel;
        inputs[2] = dataSamples[sample].HitCount;
        inputs[3] = dataSamples[sample].HitPerSecond;

        target[0] = dataSamples[sample].output[0];
        target[1] = dataSamples[sample].output[1];
        target[2] = dataSamples[sample].output[2];
        target[3] = dataSamples[sample].output[3];

        FeedForward();
```

```
            err = 0.0;

            for (int i = 0; i < OUTPUT_NEURONS; i++)
            {
                err += Math.Pow(dataSamples[sample].output[i] - actual[i], 2);
            }

            err = 0.5 * err;

            Console.WriteLine("mse = " + err);

            if (iterations++ > 100000) break;

            BackPropagate();
        }

        for (int i = 0; i < sampleCount; i++)
        {
            inputs[0] = dataSamples[i].Health;
            inputs[1] = dataSamples[i].DifficultyLevel;
            inputs[2] = dataSamples[i].HitCount;
            inputs[3] = dataSamples[i].HitPerSecond;

            target[0] = dataSamples[i].output[0];
            target[1] = dataSamples[i].output[1];
            target[2] = dataSamples[i].output[2];
            target[3] = dataSamples[i].output[3];

            FeedForward();

            if (Difficulty(actual) != Difficulty(target))
            {
                Console.WriteLine(inputs[0] + "  " + inputs[1] + "  " + inputs[2] + "  " + inputs[3]);
                Console.WriteLine("actual: " + difficultyLevels[Difficulty(actual)] + ";  target: " +
difficultyLevels[Difficulty(target)]);
            }
```

```
    else
    {
        sum++;
    }
}

float previousNetworkAccuracy = ReadNetworkAccuracy("networkAccuracy.txt");

float networkAccuracy = (float)sum / (float)sampleCount * 100;
Console.WriteLine("Network is correct: " + networkAccuracy);

if (networkAccuracy > 60f && networkAccuracy > previousNetworkAccuracy)
{
    SaveWeights("wih.txt", wih);
    SaveWeights("who.txt", who);

    SaveNetworkAccuracy("networkAccuracy.txt", networkAccuracy);
}

inputs[0] = 7.0;
inputs[1] = 2.0;
inputs[2] = 23.0;
inputs[3] = 0.13;
FeedForward();
Console.WriteLine("H: 7, D: 2, HC: 23, HPS: 0.13, " + "Difficulty level: " + Difficulty(actual));

inputs[0] = 55.0;
inputs[1] = 1.0;
inputs[2] = 2.0;
inputs[3] = 1.12;
FeedForward();
Console.WriteLine("H: 55, D: 1, HC: 2, HPS: 1.12, " + "Difficulty level: " + Difficulty(actual));

inputs[0] = 92.0;
inputs[1] = 3.0;
inputs[2] = 8.0;
```

```
            inputs[3] = 2.2;
            FeedForward();
            Console.WriteLine("H: 92, D: 3, HC: 8, HPS: 2.2, " + "Difficulty level: " + Difficulty(actual));

            inputs[0] = 31.0;
            inputs[1] = 4.0;
            inputs[2] = 29.0;
            inputs[3] = 1.0;
            FeedForward();
            Console.WriteLine("H: 32, D: 4, HC: 22, HPS: 1.0, " + "Difficulty level: " + Difficulty(actual));

            Console.ReadLine();
        }
    }
}
```