Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження впливу патернів
штучного інтелекту на ігровий процес
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-3

Алєксєєв Д.Д.
(прізвище, ініціали)

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

Керівник д.т.н. Власенко Л.А.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

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

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Алєксєєву Дмитру Дмитровичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження впливу патернів штучного інтелекту на ігровий процес»

Затверджена наказом по університету від 29.03.2024р. № 250 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 17.06.2024

3. Вихідні дані до роботи опис досліджуваних патернів поведінки штучного інтелекту, загальний опис алгоритмів, патернів та структур даних, мова програмування C#, вимоги до дослідження, програмний застосунок з демонстрацією роботи патернів.

4. Перелік питань, що потрібно опрацювати в роботі

вступ, постановка задачі, аналіз предметної галузі, підготовка до проведення дослідження, аналіз та опис поведінки NPC, аналіз окремих патернів поведінки, опис та аналіз досліджуваних структур даних, алгоритмів та патернів, написання програмних рішень.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної області	15.01 – 14.02.24	*виконано*
2	Вибір патернів для дослідження	15.02 – 29.02.24	*виконано*
3	Аналіз та опис обраних патернів реалізації	01.03 – 28.03.24	*виконано*
4	Аналіз результатів досліджень та розробка рекомендацій	29.03 – 16.04.24	*виконано*
5	Написання та оформлення статті та тез доповіді	17.04 – 23.04.24	*виконано*
6	Підготовка пояснювальної записки	24.04 – 25.05.24	*виконано*
7	Підготовка презентації та доповіді	26.05 – 01.06.24	*виконано*
8	Нормоконтроль	05.05 – 08.05.24	*виконано*
9	Рецензування	09.05 – 12.06.24	*виконано*
10	Занесення диплома в електронний архів	14.06.2024	*виконано*
11	Попередній захист	14.06.2024	*виконано*
12	Допуск до захисту у зав. кафедри	15.06.2024	*виконано*

Дата видачі завдання 29 березня 2024р.

Студент (ка) _____ Алєксєєв Д.Д.
(підпис)

Керівник роботи _____ д.т.н. Власенко Л.А.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи магістра містить: 64 с., 10 рис., 3 таблиці, 8 джерел, 5 додатків.

АЛГОРИТМ, ГРА, ІГРОВИЙ ПРОЦЕС, ПАТЕРН, ПОВЕДІНКА, ПЕРСОНАЖ, ШТУЧНИЙ ІНТЕЛЕКТ, GAMEPLAY, NPC, UNITY.

Об’єктом дослідження є патерни поведінки неігрових персонажів на ігровий процес.

Метою роботи є дослідження впливу, ефективності та наслідків різних патернів поведінки та алгоритмів що їх реалізують на ігровий процес в різних ситуаціях.

Результатом роботи є підготовлений аналіз з детальним описом патернів реалізації та рекомендацій щодо них. Аналіз включає в себе деталі реалізації патернів, їх ключові характеристики та дослідження впливу який вони мають на ігровий процес. Також було створено програмний застосунок для демонстрації роботи патернів ШІ.

ALGORITHM, GAME, GAMEPLAY, PATTERN, BEHAVIOR, CHARACTER, ARTIFICIAL INTELLIGENCE, GAMEPLAY, NPC, UNITY.

The object of study is the influence of different behavioral patterns of non-playable characters on the gameplay.

The purpose of the study is to investigate the use, effectiveness, and consequences of different behavioral patterns and algorithms that implement them on the gameplay in different situations.

The result of the work is a prepared analysis with a detailed description of implementation patterns and recommendations regarding them. The analysis includes the details of the implementation of the patterns, their key characteristics and the study of the impact they have on the gameplay. Also, software for demonstration of AI patterns’ logic was created.

Я, Алєксєєв Дмитро Дмитрович, студент гр.ІПЗм-22-3, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя робота, що буде представлена для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Усі запозичення з друкованих та електронних джерел мають відповідні посилання.

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

ЗМІСТ

Перелік скорочень 7
Вступ 8
1 Постановка задачі 10
1.1. Виявлення проблематики 10
1.2. Постановка задачі 11
2 Аналіз предметної галузі 13
2.1. Історія розвитку ШІ в іграх 13
2.2. Різновиди алгоритмів ШІ в іграх 15
2.3. Технічні аспекти використання ШІ 16
2.4. Аналіз впливу ШІ на геймплей 17
2.5. Перспективи розвитку ШІ в ігровій індустрії 18
3 Підготовка до проведення дослідження 21
4 Опис та аналіз патернів ші 26
4.1. Finite State Machine 26
4.2. Behaviour Tree 30
4.3. Goal Oriented Action Planning 36
4.4. Utility System 39
4.5. Monte-Carlo Tree Search 44
Висновки 47
Перелік джерел посилання 48
Додаток А Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії 49
Додаток Б Звіт результатів перевірки на унікальність тексту в мережі інтернет та базі ХНУРЕ 50
Додаток В Слайди презентації 51
Додаток Г Апробація результатів 61
Додаток Д Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008:2015 63

ПЕРЕЛІК СКОРОЧЕНЬ

ШІ	Штучний інтелект
AI	Artificial Intelligence
CPU	Central Processing Unit
FSM	Finite State Machine
GOAP	Goal-Oriented Action Planning
MCTS	Monte Carlo Tree Search
NPC	Non-player character
RAM	Random Access Memory

ВСТУП

З розвитком ігрової індустрії та інформаційних технологій в цілому зростають й вимоги користувачів до якості продуктів. Сучасні ігри повинні не просто кидати виклик гравцям, а й постійно та швидко адаптуватись до рішень, що він приймає. Це правило стосується будь-яких жанрів: від класичних шахів до великих та складних стратегій в режимі реального часу. Процес досягнення мети (або ігровий процес) є ключовим елементом в проєктуванні та розробки будь-якої гри. Одним з найефективніших інструментів в його покращенні є штучний інтелект, використання якого може зробити ігровий процес різноманітним, складним та захоплюючим.

Розробка штучного інтелекту для проекту зазвичай є великим та складним етапом в життєвому циклі продукту. Це вимагає детального проектування та балансування параметрів патерну, заглиблення в деталі реалізації з урахуванням обчислювальних можливостей девайсів, а також детальне та пильне тестування ШІ з метою перевірки адекватності його реакції на дії майбутніх користувачів.

Данна робота має на меті дослідження впливу різних алгоритмів штучного інтелекту на ігровий процес в різних обставинах. Це включає в себе загальну оцінку ролі штучного інтелекту в іграх, дослідження ефективності роботи того чи іншого алгоритму в окремих жанрах, характеристики та можливості алгоритмів ШІ, аналіз різноманітних сценаріїв ігор та шляхи застосування AI в них.

В роботі було розглянуто окремі патерни реалізації штучного інтелекту в іграх. Для кожного з них було проведено детальний аналіз та зроблені висновки. Робота заглиблюється в деталі реалізації кожного патерну, їх слабкі та сильні сторони, проблеми патернів та приклади реальних проектів з їх використанням. На основі даного аналізу були розроблені рекомендації щодо їх використання в майбутніх проектах з урахуванням можливих проблем та ризиків.

Дане дослідження у сфері розробки ігрових застосунків має на меті поглиблення розуміння того, як застосування алгоритмів штучного інтелекту може підвищити якість кінцевого продукту. Основна ціль полягає в аналізі того, як різні алгоритми ШІ можуть впливати на ігровий процес, роблячи його більш

адаптивним, складним, та різноманітним. Ключовим елементом є дослідження впливу цих алгоритмів на різні функціональні елементи гри, такі як поведінка NPC, можливість адаптації до дій гравця та реакції на них, передбачення результату та здатність адаптуватися до стратегій та стилів гри різних гравців. Також, важливим аспектом є аналіз ефективності алгоритмів ШІ у різних ігрових жанрах та їх вплив на залученість гравців. Це дослідження допоможе визначити оптимальні стратегії використання ШІ для створення більш захоплюючих, інтерактивних та унікальних ігрових досвідів.

1 ПОСТАНОВКА ЗАДАЧІ

1.1. Виявлення проблематики

Застосування алгоритмів ШІ в іграх може ставити перед розробником велику кількість питання. Найбільші з них, що будуть описані в цій роботі, включають:

- швидкість реакції на дії користувача: швидкість реакції ШІ на дії гравців є ключовою для створення реалістичного ігрового досвіду. У динамічних іграх, таких як шутери або гонки, швидка реакція ШІ є вирішальною для забезпечення ефекту максимального занурення та створення конкуренції гравцю з боку NPC. З іншого боку, в стратегічних іграх або RPG, де планування та тактичні рішення відіграють більшу роль, швидкість реакції може мати значно менше значення при виборі необхідного алгоритму;
- технічні обмеження на використання ресурсів: використання ресурсів, таких як RAM та CPU, є важливим аспектом у реалізації ШІ в іграх. Великі обсяги даних та складні обчислення вимагають значної обчислювальної потужності, що може впливати на загальну продуктивність ігрової системи. Інтернет-з'єднання також може бути важливим для ігор, які використовують хмарні обчислення для ШІ;
- варіативність відповідей: важливим аспектом є здатність ШІ надавати різноманітні відповіді на дії користувачів. Деякі алгоритми, наприклад, генеративні моделі, можуть створювати багатий набір варіантів відповідей, що підвищує непередбачуваність і залученість користувача до гри. З іншої точки зору, більш прості моделі поведінки (як наприклад, State Machine[1]) можуть пропонувати хоч і обмеженішу кількість дій, але цього може бути достатньо, якщо цього не вимагає гра (наприклад, жанр Tower Defense). В таких ситуаціях простіші алгоритми з низкою варіативністю можуть бути адекватним вибором;
- масштабування: масштабування алгоритмів ШІ є важливим для підтримки різних рівнів складності та обсягів ігор. Це стає особливо критичним у багатокористувацьких іграх та великих відкритих світах (наприклад,

MMORPG, стратегії в реальному часу тощо), де ШІ має взаємодіяти з великою кількістю елементів одночасно;

– адаптивність: адаптивність ШІ полягає у здатності алгоритму налаштовуватися під конкретного гравця, змінюючи стиль гри, складність або навіть стратегію. Така адаптивність важлива для підтримки почуття виклику та інтересу у різних типів гравців та може бути реалізована через складніші механізми, такі як машинне навчання;
– доцільність використання в залежності від жанру: вибір алгоритму ШІ важливо підлаштовувати під специфіку жанру гри. Наприклад, дерева пошуку можуть бути ідеальними для стратегічних ігор, де потрібно аналізувати багато різних ходів, тоді як станові машини можуть бути кращим вибором для ігор з фіксованими сценаріями або обмеженими виборами дій.

Дані аспекти є критичними при виборі алгоритму ШІ та проєктуванні ігрового процесу. Помилка в будь-якому з них може призвести до самих різних небажаних сценаріїв в майбутньому, як наприклад негативний ефект в залученні гравця чи проблеми з оптимізацією.

1.2. Постановка задачі

Головна мета даної роботи – це підготовка до проведення магістерського дослідження. Для цього, необхідно все підготувати для проведення самого дослідження. Постановка задачі для роботи: “Дослідження впливу патернів штучного інтелекту на ігровий процес” має наступний вигляд:

– аналіз існуючих алгоритмів штучного інтелекту, які використовуються в іграх;
– вивчення їх впливу на різні аспекти ігрового процесу;
– розгляд ключових аспектів при виборі алгоритму;
– оцінка використання та впливу алгоритмів в розрізі окремих жанрів;
– розробка рекомендацій для використання алгоритмів ШІ у майбутньому розвитку ігор;

– зробити висновки щодо результатів дослідження.

Ця постановка задачі зосереджується на детальному аналізі алгоритмів штучного інтелекту в іграх, оцінює їх вплив на геймплей та розробляє стратегії для їх ефективного використання у майбутньому.

2 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

2.1. Історія розвитку ШІ в іграх

Історія ігрової індустрії розпочинається з 1950-х років, коли були створені перші комп'ютерні ігри. Важливим кроком у розвитку цієї сфери була поява перших неігрових персонажів (NPC). Спочатку їх поведінка була строго прописана за допомоги скриптів, але вже у 1970-х роках почали з’являтись персонажі які могли по-різному реагувати на дії гравця. Одним з найбільш відомих прикладів є гра "Pong", де простий алгоритм імітував поведінку опонента.

Далі, на початку 1980-х, ігри почали використовувати більш складні алгоритми, які дозволяли NPC вести себе реалістичніше. Одним із знакових прикладів цього періоду є гра "Pac-Man" (рис. 1), де кожен з привидів мав свій унікальний стиль гри.

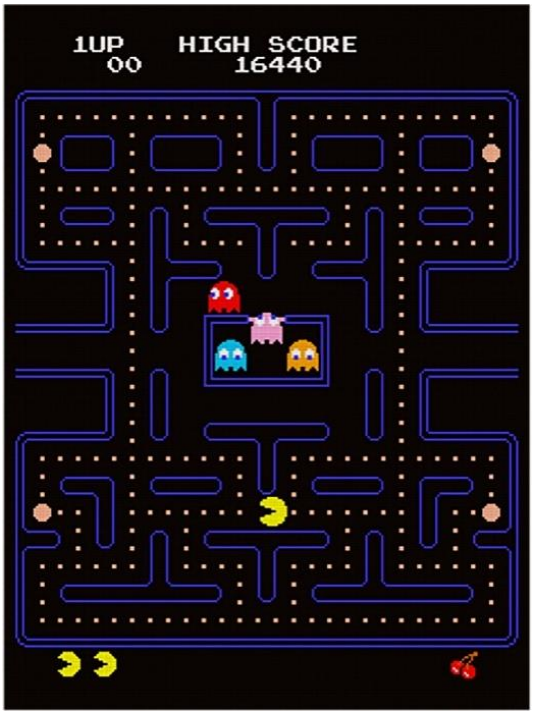

Рисунок 2.1 – гра Pac-Man (за даними [2])

Це була свого роду реалізація State Machine, кожний з привидів мав три стані (переслідувати, розбігатись, лякатись), а також свої особливі поведінки. Наприклад, Pinky та Inky намагались заганяти гравця у кут, а Blinky просто гнався

за ним. Це був один із перших випадків, коли ШІ не просто реагував на дії гравця, а й сам ініціював дії.

Через складності з обчислювальними можливостями та дуже обмеженою пам’яттю машин, AI в іграх розвивався дуже повільно. В 1998 році з’явилось перше за великий проміжок часу нововведення – в грі Half-Life було реалізовано Finite State Machine (схема показана на рисунку 2.2) з ієрархічною структурою. Стани ділились не просто на «Бій», «Патрулювання» тощо, а мали свої під-стани, що зробило гру набагато динамічнішою. NPC покращили свою навігацію по мапі, а в битвах могли використовувати ширший набір дій.

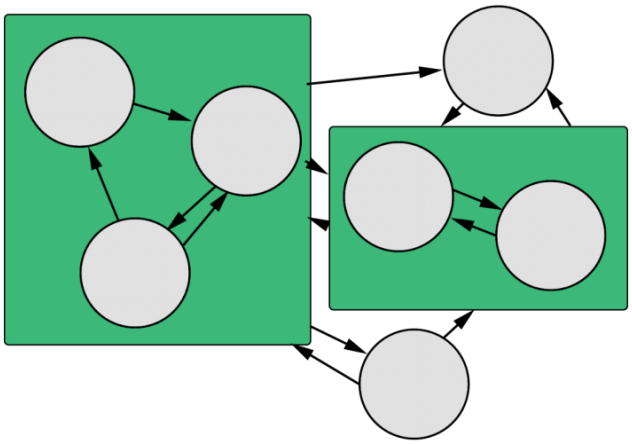

Рисунок 2.2 – схема ієрархічної Finite State Machine (за даними [3])

Приблизно з 1998 – початку 2000-х років, штучний інтелект в іграх починає стрімко розвиватись. Окрім звичних машин станів, розробники починають використовувати більш складні, вже досліджені алгоритми та структури даних – дерева, пошук шляху, Goal-Oriented Action Planning. Також винаходять й нові підходи, як наприклад Utility Systems. Штучний інтелект тепер може оперувати не просто діями гравця чи станом мапи. В комплексних стратегіях, як наприклад Total War, AI оперує економікою, дипломатією, торгівлею, розвитком, армією та війнами, а кожна серія гри поступово покращувала алгоритми в минулій. Останні версії використовували алгоритм Monte Carlo Tree Search (рис. 2.3), що зробили супротивника дійсно складним, реалістичним та непередбачуваним.

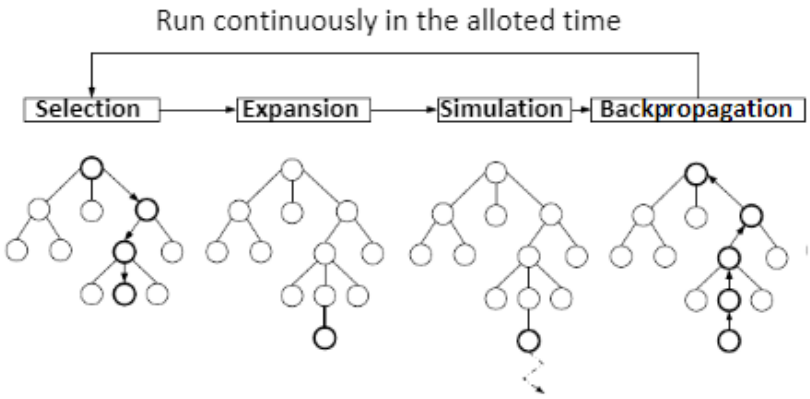

Рисунок 2.3 – схема алгоритму MCTS (за даними [4])

Сучасний ШІ досяг такого рівня складності, що може навіть конкурувати з професійними кіберспортсменами. Вражаючим прикладом цього є боти, розроблені компанією OpenAI, які грали проти кіберспортивної команди у грі Dota 2, демонструючи високий рівень стратегічного мислення та прийняття рішень. За допомоги машинного навчання вони змогли створити такий штучний інтелект, який з легкістю зміг обійти найкращих гравців у світі в чесній грі.

Історія розвитку ШІ у іграх є історією невпинного прогресу та інновацій. Вона напряму пов’язана з відкриттями у сфері математики, комп’ютерної інженерії та алгоритмів. З цього можна побачити, як нові алгоритми допомагали створювати все більш реалістичні та захоплюючі віртуальні світи, все сильніше посилюючи ефект «занурення» в користувача.

2.2. Різновиди алгоритмів ШІ в іграх

Розвиток штучного інтелекту в іграх характеризується великою різноманітністю підходів. Ігри на самому початку індустрії використовували найпростіші алгоритми, здебільшого на основі правил, де ШІ реагував на певні умови або дії гравця або взагалі просто копіював їх.

З розвитком технологій, зокрема збільшення потужності персональних комп’ютерів та ігровий консолей, з'явилися більш складні та адаптивні системи. Розробники ставали все менш залежними від обмежень на використання пам’яті та процесору, а зараз подекуди можна й зустріти використання хмарних розрахунків.

Вибір того чи іншого алгоритму залежить від багатьох факторів, наприклад це жанр, кількість дій що може виконувати NPC та наскільки далеко необхідно передбачувати дії гравця.

State Machine не може передбачувати дії гравця, але є чудовим вибором в таких іграх, як Tower Defense або Rogue Like, де кількість дій досить обмежена, тому за рахунок вибору цього алгоритму можна виграти в часі та в ресурсах. Дерева поведінки і їх варіації здатні легко масштабуватись та приймати рішення з подальшим прогнозуванням дій користувача. Інші підходи, як наприклад Utility Systems що приймає рішення на основі коефіцієнту корисності, використовуються в більш специфічних сценаріях.

Слід також зазначити й про моделі основані на машинному навчанні. Зазвичай вони показують надвисоку ефективність, але мають один досить великий недолік – необхідно збирати великі масиви даних вже на етапі релізу проєкту, оскільки самостійно згенерувати необхідну вибірку неможливо і потрібна велика кількість реальних гравців.

2.3. Технічні аспекти використання ШІ

Технічні аспекти використання штучного інтелекту в іграх включають низку викликів і обмежень, що значною мірою визначають можливості та ефективність ШІ. Одним з основних обмежень є складність масштабування, яка включає збільшення розмірів ігрового середовища та кількості агентів (NPC), з якими ШІ має взаємодіяти та оперувати. Великі ігрові світи з великою кількістю агентів вимагають значно більших обчислювальних ресурсів та ефективного розподілу завдань між ними.

Використання пам'яті та процесору є іншим критичним аспектом. Складні алгоритми ШІ можуть вимагати значного обсягу оперативної пам'яті та великої обчислювальної потужності, особливо в іграх з високою ступеню інтерактивності та деталізації. Це створює додатковий тиск на оптимізацію коду та ефективне використання системних ресурсів.

Ще однією важливою проблемою є розробка ШІ, який може виконувати велику кількість дій. Чим більше потенційних дій та варіантів поведінки передбачено для ШІ, тим складнішим стає процес його програмування та тестування, оскільки необхідно враховувати значно більшу кількість можливих сценаріїв взаємодії. Також, це напряму залежить від обраного алгоритму, оскільки деякі з них масштабуються значно легше ніж інші.

Останнім, але не менш важливим аспектом є використання машинного навчання для створення ШІ в іграх. Цей підхід вимагає збору великих обсягів даних для тренування моделей, що може бути складним на етапі розробки. Крім того, це ставить перед розробниками завдання забезпечення точності та об'єктивності цих даних, щоб уникнути упередженості або непередбачуваної поведінки NPC.

2.4. Аналіз впливу ШІ на геймплей

Штучний інтелект грає ключову роль у формуванні геймплею в сучасних відеоіграх, забезпечуючи багатогранність взаємодії з NPC та динаміку ігрового процесу. Вплив ШІ на геймплей можна аналізувати через кілька основних аспектів: швидкість відгуку, варіативність дій, непередбачуваність, створення ефекту "занурювання" та адаптація до гравця.

ШІ в іграх часто реагує на дії гравців у реальному часі, що забезпечує відчуття безперервної взаємодії та динаміки. Наприклад, у шутерах або бойових іграх NPC можуть миттєво реагувати на агресивні дії гравця, змінюючи свою поведінку або тактику. З іншого боку, покрокові стратегії, MMORPG або класичні настільні ігри не зазвичай не вимагають миттєвого відгуку та мають вдосталь часу на обробку інформації між раундами. Тому швидкість відгуку на дії користувача хоч і є дуже важливим фактором, але він критичним він є не для всіх жанрів.

Сучасний AI здатен виконувати широкий спектр дій, що робить гру більш непередбачуваною та захоплюючою. У рольових іграх це особливо важливо, оскільки чим більше дій може виконати NPC, тим більш реалістичним його відчуває гравець. В ідеальному варіанті будь-який ШІ повинен мати той самий набір дій, що й гравець.

Ефективний ШІ повинен бути непередбачуваним, створюючи унікальні ігрові ситуацій. Його дії повинні бути не просто випадковими, а мати логічний зв’язок і повинні вести його до перемоги над гравцем. Це особливо важливо у іграх де елемент змагання є головним. В таких ситуаціях непередбачуваність є ключовою стратегією для протидії гравцю.

Також, ШІ вносить великий вклад у створення реалістичного ігрового світу. Реалістична поведінка NPC, їх здатність до соціальної взаємодії та емоційних реакцій забезпечують глибоке "занурення" гравця в ігровий світ. Чим краще AI імітує реального гравця, тим захоплюючим є продукт.

Деякі сучасні ШІ можуть адаптуватися до стилю гри конкретного гравця, змінюючи рівень складності або поведінку в залежності від його дій і рішень. Така адаптація забезпечує персоналізований ігровий досвід, що є особливо важливим для різних груп користувачів. Хтось бажає від гри «виклик», хтось віддає перевагу дослідженню гри. Адаптація під різну когорти гравців дозволяє значно розширити користувацьку базу свого продукту.

2.5. Перспективи розвитку ШІ в ігровій індустрії

Вже зараз штучний інтелект застосовується в багатьох етапах проєктування та розробки застосунку. Це стосується не тільки тих елементів, які вже були описані в минулих пунктах роботи, а ще й в графічному та наративному дизайні, в генерації контенту, створення сюжетів та розробці людиноподібних, реалістичних персонажів з унікальною непередбачуваною поведінкою та можливістю передбачувати дії гравця та адаптуватись під них.

Штучний інтелект також відіграє важливу роль у процесі створення динамічних ігрових світів, де кожен елемент та кожен персонаж цього світу взаємодіє з гравцем унікальним чином. Наприклад, за допомогою ШІ можна генерувати унікальні ландшафти або за певними правилами та алгоритмами змінювати розташування об'єктів, що робить кожну ігрову сесію неповторною. Такі рішення вже можна побачити, наприклад, в новій версії ігрового движка Unreal Engine 5 (рис. 2.4). Генерація рівнів з алгоритмами від компанії Epic Games може

повністю змінити підхід до такої важливої в ігровій індустрії дисципліни як Level-дизайн. Такий підхід дозволяє створити більш багатогранні та різноманітні ігрові середовища, збільшуючи загальну привабливість та деталізованість гри.

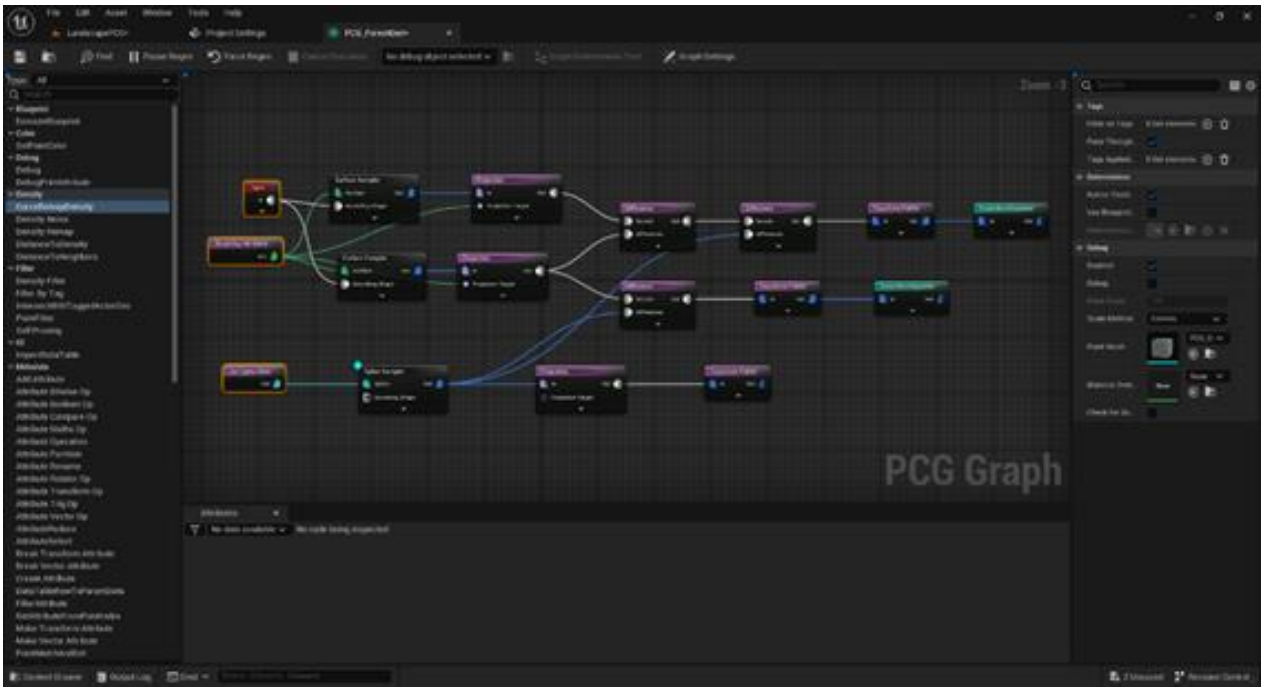

Рисунок 2.4 – процедурна генерація контенту в Unreal Engine 5 (за даними [5])

Застосування різних сучасних моделей ШІ у вказаних сферах відкриває принципово нові можливості для створення захоплюючих проєктів – підвищення реіграбельності, зменшення фінансових витрат та часу на створення не технічних елементів (сюжети, візуальний дизайн, музика та звуки тощо), розширення можливостей користувача, створення реалістичних ігрових персонажів для підвищення глибини ігрового досвіду та створення більш непередбачуваних для нього ситуацій.

В майбутньому можна очікувати ще більшої інтеграції алгоритмів ШІ у всі аспекти розробки ігор. Особливо цікавими є можливості штучного інтелекту в області віртуальної реальності та змішаної реальності, де ШІ може вносити вагомий вклад у створення інтерактивних, реалістичних віртуальних середовищ. Разом з цим, цікавим є підвищення реалістичності NPC за рахунок створення складних та непередбачуваних поведінок шляхом реалізації більш комплексних алгоритмів та використанням великих вибірок даних для їх навчання. Таким чином,

впровадження ШІ в індустрію розробки ігор надає не тільки підвищення якості ігрового процесу, але й відкриває нові горизонти у взаємодії ігрових світів з гравцями.

3 ПІДГОТОВКА ДО ПРОВЕДЕННЯ ДОСЛІДЖЕННЯ

Для виконання дослідження необхідно обрати алгоритми поведінки штучного інтелекту в ігрових застосунках. Було обрано наступні алгортми:

– finite state machine;
– behavior trees;
– goal-oriented action planning (GOAP);
– utility systems;
– Monte Carlo tree search (MCTS).

Далі наведемо короткий опис кожного з обраних алгоритмів:

– finite state machine – один з найпростіших алгоритмів реалізації штучного інтелекту в іграх. Поведінка сутності визначається його поточним станом (state), а переходи між станами відбуваються за допомоги строго прописаних розробником умов;
– behavior trees – більш комплексний алгоритм, найчастіше використовується для реалізації не ігрових персонажів (Non Playable Character, далі – NPC). Поведінка визначається деревом, де кожний його вузол є певною дією, рішенням чи набором завдань, а також має свій набір правил щодо переходу до наступного вузла. Дерева поведінки легко масштабувати, а також вони підтримують послідовності дій (на приклад, у вигляді структури даних черга (queue));
– goal-oriented action planning (GOAP) – даний патерн базується на діях ті цілях. NPC мають певну ціль та намагаються досягти її за допомоги виконання певних дій. В свою чергу, кожна дія має свої передумови та вплив при виконанні. Найбільшою перевагою цього патерну є можливість адаптації до середовища, що дає змогу справлятися з непередбаченими ситуаціями, на відміну від строго прописаних правил;
– utility systems – суть цього патерну полягає в розрахунку показника «корисності» тієї чи іншої дії. Сутності, для якої реалізований патерн, надається доступ до показників корисності – значення середовища, цілі

NPC, поточний стан гри тощо. За допомоги цих показників, а також спеціальних формул чи алгоритмів, розраховується яка наступна дія сутності була б максимальна корисна для досягнення необхідного результату;

– Monte Carlo tree search (MCTS) – на відмінну від Behavior Trees, даний патерн сам намагається збудувати дерево пошуку з метою передбачити можливі майбутні стани середовища. Даний патерн найчастіше використовується в стратегіях, оскільки в них присутній великий простір рішень. Найбільш відомий цей патерн в таких іграх, як шахи та Го.

Далі, визначимо критерії, за якими будемо порівнювати патерни:

– використання оперативної пам’яті (RAM, Mb);
– використання ресурсів центрального процесору (CPU, мілісекунди);
– швидкість відгуку на дії користувача (мілісекунди);
– варіативність алгоритму (додавання нових дій/рішень/станів);
– масштабування алгоритму – здатність патерну працювати з великою кількістю змін середовища, цілей, сутностей, високою складністю системи.

Тепер створимо таблицю з патернами та критеріями порівняння, та оцінимо їх (табл. 3.1).

Таблиця 3.1 – Значення критеріїв порівняння патернів штучного інтелекту (таблиця виконана самостійно)

	RAM	CPU	Швидкість відгуку	Варіативність	Масштабування
Finite state machine	Низьке споживання RAM	Низьке споживання CPU	Миттєвий відгук	Низька варіативність	Складно масштабується
Behavior trees	Середнє споживання RAM	Низьке споживання CPU	Затримка перед відгуком	Висока варіативність	Легко масштабується

Продовження таблиці 3.1

GOAP	Середнє споживання RAM	Низьке споживання CPU	Затримка перед відгуком	Висока варіативність	Легко масштабуєтьс я
Utility Systems	Середнє споживання RAM	Низьке споживання CPU	Миттєвий відгук	Середня варітивність	Середній рівень масштабованості
MCST	Високе споживання RAM	Низьке споживання CPU	Затримка перед відгуком	Висока варіативність	Легко масштабуєтьс я

Переведемо текстові значення критеріїв в оцінку по шкалі від 0 до 5 та отримаємо нову таблицю з числовими значеннями (таблиця 3.2).

Таблиця 3.2 – Чисельні значення (таблиця виконана самостійно)

	RAM	CPU	Швидкість відгуку	Варіативність	Масштабування
Finite state machine	5	5	5	1	1
Behavior trees	4	5	4	5	5
GOAP	4	5	4	5	5
Utility Systems	4	5	5	3	3
MCST	3	5	2	5	5

Далі, використаємо лінійну адитивну згортку з коефіцієнтами за для визначення важливості кожного з критеріїв. Маємо наступні коефіцієнти:

– RAM – 0.05;

– CPU – 0.1;

– швидкість відгуку – 0.2;

– варіативність – 0.35;

– масштабування – 0.3.

Маючи значення критеріїв та їх вагові коефіцієнти, можемо розрахувати (див. формула 3.1) який патерн реалізації штучного інтелекту є найбільш ефективним.

$$Z = \max_{i=1,m} \sum_{j=1}^{n} \alpha_j \beta_j a_{ij} \quad (3.1)$$

Проводимо розрахунки та отримали наступні результати в таблиці 3.3 (усі розрахунки були проведені в Excel):

Таблиця 3.3 – результати розрахунків (таблиця виконана самостійно)

	RAM	CPU	Швидкість відгуку	Варіативність	Масштабування	Z*
Finite state machine	5	5	5	1	1	0,229
Behavior trees	4	5	4	5	5	0,6569
GOAP	4	5	4	5	5	0,6569
Utility Systems	4	5	5	3	3	0,4514
MCST	3	5	2	5	5	0,6557
β	0,05	0,1	0,2	0,35	0,3	
α	0,025	0,04	0,0909	0,1842	0,1579	

Маємо наступні результати:

– Finite state machine – 0,229

– Behavior trees – 0,6569

– Goal-Oriented Action Planning (GOAP) – 0,6569

– Utility systems – 0,4514

– Monte Carlo Tree Search (MCTS) – 0,6557

З результатів дослідження видно, що найефективніше при обраних критеріях себе показують алгоритми, основані на деревах – дерево поведінки, Goal-Oriented Action Planning та Monte Carlo Tree Search. Важливо зазначати, що результати не є абсолютними, та реальна ефективність того чи іншого підходу до реалізації ШІ може бути інша (наприклад, у випадку якщо від NPC не очікується складна та непередбачувана поведінка, краще використовувати більш прості патерни, такий як Finite State Machine).

4 ОПИС ТА АНАЛІЗ ПАТЕРНІВ ШІ

4.1. Finite State Machine

Finite State Machine (FSM) є математичною моделлю обчислень, яка часто використовується для опису поведінки системи через набір станів і переходів між ними. Кожен стан відповідає певній конфігурації системи, а переходи визначають, як система змінює свій стан у відповідь на зовнішні чи внутрішні події. Зазвичай FSM складається з обмеженого набору станів, початкового стану, подій, які спричиняють зміни станів, переходів між станами та дій, що виконуються під час цих переходів.

Система починає свою роботу в початковому стані і переходить між станами, реагуючи на події. Кожен перехід може супроводжуватися певними діями, що виконуються під час переходу або при перебуванні в новому стані. Наприклад, у rogue-like іграх вороги можуть мати три основні стани: патрулювання, переслідування та атака. У стані патрулювання ворог просто рухається по визначеному маршруту, у стані переслідування він слідує за гравцем, а у стані атаки намагається завдати шкоди гравцеві. Умовами для переходу можуть слугувати відстань до цілі – при наближенні з патрулювання змінюється на переслідування, якщо занадто близько – то з переслідування на атаку. Якщо умови не виконуються, то повертаємося до попереднього стану.

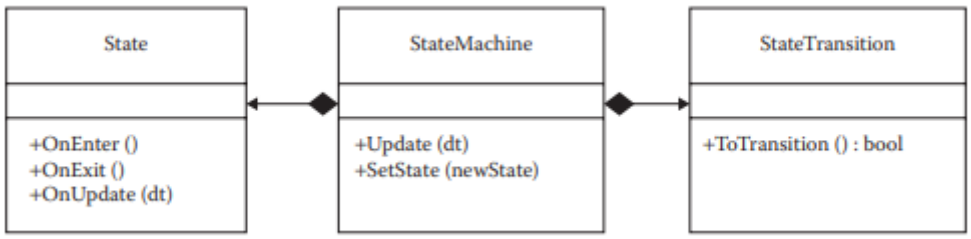

Рисунок 4.1 – схема патерну Finite State Machine (рисунок створено самостійно)

Один із ключових аспектів реалізації FSM — це чітке визначення станів і подій, щоб уникнути неоднозначності у поведінці системи. Поточний стан системи завжди зберігається, щоб точно знати, як реагувати на наступну подію. Ця модель дозволяє легко відстежувати та змінювати поведінку системи, що робить її

ідеальною для використання в ігровому AI, де потрібно моделювати відносно просту поведінку NPC, наприклад, патрулювання, атаку чи втечу.

Для демонстрації роботи патерну в програмній системі було створено наступні класи: State, Transition та FiniteStateMachine.

State відображає поточний стан. Як було вказано на рисунку 4.1, він має три методи Enter, Execute та Exit які і відображають його логіку.

```
public abstract class State
{
	public NavMeshAgent Owner { get; set; }
	public abstract void Enter();
	public abstract void Execute();
	public abstract void Exit();
}
```

Transition являє собою зв’язок між станами. Клас має посилання на стан, в який переходить автомат станів при виконанні умови, та сама умова.

```
public abstract class Transition
{
	public NavMeshAgent Owner { get; set; }
	public State TargetState { get; set; }

	public abstract bool ShouldTransition();
}
```

В якості прикладу приведемо реалізацію переходу від стану Patrolling до стану Evading. Стан Evading вмикається, якщо неподалік від AI-агенту з’являється пастка.

```
public class PatrollingToEvadingTransition : Transition
{
	public PatrollingToEvadingTransition(NavMeshAgent owner, State
targetState)
	{
		Owner = owner;
		TargetState = targetState;
	}

	public override bool ShouldTransition()
	{
		var results =
		Physics.SphereCastAll(Owner.transform.position,
```

```
            StateSettings.SphereCastRadiusForEvading,
            Owner.transform.forward,
            StateSettings.SphereCastRadiusForEvading);

            foreach (var hit in results)
            {
                  if (hit.collider.CompareTag("Trap"))
                  {
                        return true;
                  }
            }

            return false;
      }
}
```

Головний клас FiniteStateMachine відповідає за управління станами та переходами. Нижче приведено метод Update, який відпрацьовує кожний кадр (кількість кадрів в секунду визначається на девайсі кінцевого користувача).

```
private void Update()
{
      foreach (var (state, transition) in transitions)
      {
            if (currentState != state) continue;
            if (!transition.ShouldTransition()) continue;

            currentState.Exit();
            text.text = $"From
            {currentState.GetType().Name.Replace("State", "")} to
            {transition.TargetState.GetType().Name.Replace("State",
            "")}";
            currentState = transition.TargetState;
            currentState.Enter();
            break;
      }

      currentState.Execute();
}
```

Finite State Machine (FSM) має кілька значних переваг, які роблять його популярним вибором для ігрової розробки. Однією з головних переваг FSM є його простота і зрозумілість. Дизайнери та розробники можуть легко візуалізувати та моделювати поведінку персонажів за допомогою діаграм станів, що робить процес розробки інтуїтивно зрозумілим. Крім того, FSM забезпечує детерміновану

поведінку, що означає, що для кожного можливого стану і події результат буде завжди передбачуваним. Це дозволяє легко налагоджувати та тестувати систему.

Для ігрової розробки це особливо важливо, оскільки забезпечує передбачуваність поведінки NPC (неігрових персонажів), що робить ігровий процес більш надійним і керованим. Завдяки FSM розробники можуть легко реалізувати та відстежувати ці стани і переходи між ними.

В той же час, FSM має деякі недоліки. Один із основних недоліків полягає в тому, що з ростом складності системи кількість станів і переходів може значно збільшитися, що робить FSM важким для управління та розширення. Це особливо актуально для складних ігор з великою кількістю можливих станів і подій. Наприклад, у грі з десятками різних видів ворогів, кожен з яких має свої унікальні поведінкові патерни, кількість станів і переходів може стати дуже великою і складною для управління та масштабування.

Також, FSM споживає досить велику кількість ресурсів, якщо одночасно використовується декілька таких об’єктів. Найчастіше це намагаються вирішити переходом від постійних перевірок умов переходів на систему основану на подіях (event-driven system), але це все ще не покриває проблем з нераціональним використанням даних, необхідних для кожного об’єкту.

Ще один недолік FSM — це його обмежені можливості для моделювання складних поведінкових сценаріїв. Наприклад, для персонажів, які повинні враховувати багато факторів або умови для прийняття рішень, FSM може виявитися занадто спрощеним. У таких випадках інші патерни можуть бути більш ефективними. Проте, попри ці недоліки, FSM залишається одним із найпоширеніших патернів для ігрової розробки, особливо для простих та середньо складних поведінкових моделей. Цей патерн часто використовується платформерах та невеликих за масштабом проектах.

Загалом, FSM є потужним інструментом для ігрових розробників, дозволяючи створювати прості і зрозумілі моделі поведінки, які легко налагоджувати і підтримувати. Він особливо корисний у тих випадках, коли

потрібна детермінована і передбачувана поведінка персонажів, що робить його ідеальним вибором для багатьох ігрових жанрів.

4.2. Behaviour Tree

Behavior Tree (BT) — це гнучка і масштабована структура даних, яка використовується для реалізації штучного інтелекту в іграх. Вона являє собою дерево, де кожен вузол відповідає певній поведінці або дії. Корінь дерева є початковою точкою, від якої починається виконання поведінки, а кожен з вузлів дерева може бути складовим (тобто мати під вузли) або кінцевим (тобто виконувати конкретну дію).

Основна робота Behavior Tree полягає в проходженні дерева від кореня до листя, перевіряючи умови та виконуючи дії, доки не буде знайдена завершальна дія або виконані всі можливі варіанти. Це дозволяє моделювати складні поведінкові сценарії, розділяючи їх на більш прості та керовані компоненти.

Серед найголовніших елементів Behaviour Tree, слід виділити:

- задачі (tasks): це базові елементи дерева, які можуть бути або виконуваними діями, або умовами. Дії виконуються безпосередньо, тоді як умови перевіряють певні стани або ситуації в грі;
- умови (conditions): це логічні перевірки, які визначають, чи слід виконувати певну гілку дерева. Наприклад, умовою може бути перевірка наявності ворога в полі зору.
- дії (actions): це конкретні дії, які виконує персонаж. Наприклад, рух до певної точки, атака ворога або взаємодія з об'єктом.
- послідовності (sequences): це вузли, які виконують під вузли послідовно. Вони успішні лише тоді, коли всі під вузли успішно завершені.

Behaviour Tree став одним з найпоширеніших і найстабільніших варіантів реалізації штучного інтелекту в ігровій індустрії з кількох причин:

- гнучкість: BT дозволяє легко змінювати і розширювати поведінку персонажів. Оскільки кожен елемент дерева є окремим і самодостатнім,

розробники можуть додавати нові умови або дії, не впливаючи на вже існуючі частини дерева;

– масштабованість: ВТ легко масштабуються від простих до дуже складних поведінкових моделей. Це означає, що вони можуть використовуватися як для NPC з простою поведінкою, так і для складних боссів або головних персонажів з багаторівневими стратегіями;
– зручність у розробці та налагодженні: завдяки своїй структурі, ВТ легко візуалізувати та налагоджувати. Розробники можуть використовувати інструменти для побудови та тестування дерева, що дозволяє швидко знаходити і виправляти помилки;
– повторне використання: компоненти ВТ можна легко повторно використовувати у різних частинах дерева або в інших проектах. Це сприяє ефективному використанню ресурсів та зменшує час на розробку.

Behaviour Tree є критично важливими для сучасної ігрової індустрії через їх здатність моделювати складні та реалістичні поведінкові патерни. Вони дозволяють створювати персонажів, які можуть адаптуватися до змінних умов у грі, реагувати на дії гравця та приймати логічні рішення в реальному часі. Це робить ігровий процес більш захоплюючим і непередбачуваним.

Для демонстрації роботи ШІ в програмному застосунку було реалізовано Behaviour Tree з наступними основними класами: Node, BehaviourTree, ActionNode, SequenceNode, ConditionNode та інші.

Клас Node є головним абстрактним класом усієї системи і відображає вузол дерева рішень. В ньому відображено поточний стан вузла та функцію за яку він відповідає.

```
public abstract class Node
{
      public enum State
      {
            Running,
            Success,
            Failure
      }
```

```
        protected State state;

        public State NodeState => state;

        public abstract State Execute();
}
```

Клас ActionNode відповідає за конкретну дію, яку може виконувати агент. В дереві рішень цей клас є листям дерева.

```
public class ActionNode : Node
{
        public delegate State ActionDelegate();
        private ActionDelegate action;

        public ActionNode(ActionDelegate action)
        {
                this.action = action;
        }

        public override State Execute()
        {
                state = action();
                return state;
        }
}
```

Клас SequenceNode є послідовністю в дереві рішень. При виконанні методу Execute(), він також викликає цей метод в усіх свої під вузлах по черзі. Якщо хоча б один з вузлів має статус Failure (тобто дія не була виконана успішно), то уся послідовність також має статус Failure.

```
public class SequenceNode : Node
{
        private List<Node> children = new List<Node>();

        public void AddChild(Node node)
        {
                children.Add(node);
        }
```

```
        public override State Execute()
        {
            foreach (var child in children)
            {
                var childState = child.Execute();
                if (childState == State.Failure)
                {
                    state = State.Failure;
                    return state;
                }
            }
            state = State.Success;
            return state;
        }
    }
```

Клас condition відображає ще один вид вузлів дерева – умови. Якщо визначена для вузла умова виконується, то далі виконується уся гілка дерева.

```
public class ConditionNode : Node
{
    public delegate bool ConditionDelegate();
    private ConditionDelegate condition;

    public ConditionNode(ConditionDelegate condition)
    {
        this.condition = condition;
    }

    public override State Execute()
    {
        state = condition() ? State.Success : State.Failure;
        return state;
    }
}
```

Клас BehaviourTree є класом з якого стартує уся робота дерева. В собі він має лише корневий вузол.

```
public class BehaviourTree : MonoBehaviour
{
    private Node root;

    public void SetRootNode(Node rootNode)
    {
        root = rootNode;
    }

    void Update()
    {
```

```
            if (root != null)
            {
                  root.Execute();
            }
      }
}
```

У таких іграх, як Spore та Halo, Behaviour Tree використовуються для моделювання складної поведінки ворогів, що дозволяє їм координувати атаки, укриватися та працювати в команді. Це додає глибини і реалізму ігровому процесу, що важко досягти з іншими патернами.

Таким чином, Behaviour Tree стали стандартом у розробці ігрового штучного інтелекту, пропонуючи потужний інструмент для створення інтерактивних і захоплюючих ігрових світів. Нижче, на рисунку 4.2, можна побачити приклад схеми Behaviour Tree. В залежності від поточних умов, AI може обрати захват об’єкту, або його пошук в різних місцях. Тобто, в межах одного дерева можна створювати цілу колекцію ймовірних дій, які може виконати AI, тим самим роблячи його поведінку більш реалістичною та непередбачуваною.

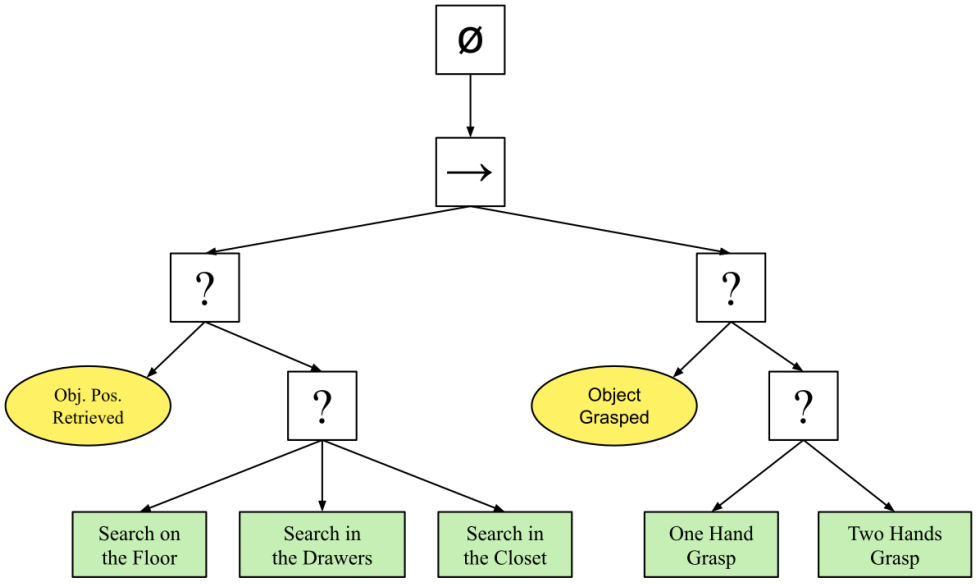

Рисунок 4.2 – схема Behaviour Tree (за даними [6])

Хоча Behaviour Tree (BT) є надзвичайно популярним і ефективним інструментом для розробки штучного інтелекту в іграх, вони мають і свої недоліки.

Розуміння цих обмежень допоможе розробникам вибрати найкращий підхід для реалізації AI у своїх проектах та уникнути потенційних проблем.

Одним із основних недоліків є складність налаштування та підтримки великих поведінкових дерев. У міру додавання нових вузлів і умов, поведінкові дерева можуть ставати дуже великими та складними. Це ускладнює їх налаштування та підтримку, особливо у великих проектах. Великі ігри з комплексними AI-системами можуть перетворити поведінкові дерева на заплутані конструкції, що ускладнює розуміння та відстеження логіки.

Ще одним недоліком є виконання повторних перевірок. У Behaviour Tree кожен раз при виконанні перевіряються всі умови знову. Це може призвести до неефективного використання обчислювальних ресурсів, особливо коли йдеться про складні дерева з багатьма умовами та діями, що повторюються. Постійна перевірка тих самих умов може уповільнювати виконання ігрового процесу, що впливає на загальну продуктивність гри.

Крім того, Behaviour Tree часто мають обмежену здатність реагувати на непередбачувані зміни в середовищі гри в реальному часі. Якщо дерево не містить специфічної логіки для обробки певної ситуації, AI може діяти неадекватно або неприродньо. Це може знизити якість геймплею, особливо в динамічних іграх з великим числом взаємодій. Поведінкові дерева можуть стати недостатньо гнучкими, коли йдеться про моделювання складних і непередбачуваних поведінок персонажів.

Ще однією проблемою є питання з garbage collection. У мовах програмування, які підтримують автоматичний збір сміття, використання Behaviour Tree може призвести до проблем з продуктивністю через часте створення та знищення об'єктів. Кожен новий вузол дерева або об'єкт, створений для виконання дії чи перевірки умови, може вимагати значних ресурсів для управління пам'яттю. Це може спричинити часті виклики збору сміття, що негативно впливає на плавність ігрового процесу, особливо на пристроях з обмеженими ресурсами.

Незважаючи на ці недоліки, Behaviour Tree залишаються важливим інструментом для розробки ігрового AI, завдяки своїй гнучкості та

масштабованості. Розуміння їх обмежень дозволяє розробникам приймати обґрунтовані рішення при виборі підходу до реалізації AI і знаходити способи пом'якшення негативних наслідків. У деяких випадках комбінування Behaviour Tree з іншими патернами або використання додаткових структур даних може допомогти вирішити вказані проблеми та створити ефективнішу систему штучного інтелекту.

4.3. Goal Oriented Action Planning

Goal Oriented Action Planning (GOAP) — це підхід до розробки штучного інтелекту, який фокусується на досягненні конкретних цілей за допомогою динамічного планування дій. GOAP дозволяє персонажам приймати більш розумні і адаптивні рішення, ніж традиційні патерни, такі як Finite State Machine або Behaviour Tree.

GOAP [7] працює шляхом визначення набору цілей, які AI може прагнути досягти, та набору дій, які можуть бути виконані для досягнення цих цілей. Кожна дія має свої передумови та ефекти, а AI використовує планувальник для визначення оптимальної послідовності дій, які приведуть до досягнення мети. Це дозволяє створювати більш складні та адаптивні поведінки, оскільки AI може реагувати на змінні умови в ігровому середовищі.

Головними елементами GOAP є:

– цілі (goals) – це кінцеві стани, яких AI прагне досягти. Наприклад, у грі це може бути знищення ворога, знаходження предмету або досягнення певної локації;
– дії (actions) – кожна дія має передумови та ефекти. Передумови — це умови, які повинні бути виконані для того, щоб дія могла бути виконана. Ефекти — це зміни в стані світу, які відбуваються в результаті виконання дії;
– планувальник (planner) – це алгоритм, який аналізує доступні дії та їхні передумови, щоб знайти оптимальний шлях досягнення мети. Планувальник шукає послідовність дій, які найефективніше приведуть до виконання цілі, враховуючи поточний стан світу.

Для реалізації алгортиму GOAP в програмній системі було реалізовані наступні класи: WorldState, Goal, Action, Planner та Agent.

WorldState відображає поточний стан середовища для оцінки бажаних результатів дій. Клас є абстрактним для його модифікації під будь-яке середовище чи ситуацію.

```
public abstract class WorldState
{
      public abstract void UpdateState();
      public abstract bool IsStateSatisfied(string stateName);
      public abstract int GetStateValue(string stateName);
      public abstract void SetStateValue(string stateName, int value);
}
```

Goal є ціллю. Ціль також має пріоритет для більш детального процесу планування.

```
public abstract class Goal
{
      public abstract int Priority { get; }
      public abstract bool IsAchieved(WorldState worldState);
}
```

Клас Action є дією, яка повинна мати певний ефект на середовище задля досягнення мети. Будь-яка дія може мати (чи не мати) свою ціну та певні передумови задля її виконання.

```
public abstract class Action
{
      public abstract int Cost { get; }
      public abstract bool ArePreconditionsMet(WorldState worldState);
      public abstract void ApplyEffects(WorldState worldState);
}
```

Клас Planner є планувальником. Через нього розробник може створити необхідні цілі та за допомоги класу WorldState збирати інформацію про ефекти дій AI та чи задовольняються для них умови.

```
public abstract class Planner
```

```
{
      public abstract List<Action> Plan(Agent agent, List<Goal> goals,
WorldState worldState);
}
```

Клас Agent відповідальний за виконання дій. Як і з FiniteStateMachine та BehaviourTree, за допомоги методу Update, що має певну періодичність протягом усієї роботи алгоритму, він оновлює інформацію про стан світу та обирає наступні дії.

```
public class Agent : MonoBehaviour
{
      public WorldState WorldState { get; set; }
      public List<Goal> Goals { get; set; }
      public List<Action> Actions { get; set; }
      public Planner Planner { get; set; }

      private void Update()
      {
            var plan = Planner.Plan(this, Goals, WorldState);
            foreach (var action in plan)
            {
                  if (action.ArePreconditionsMet(WorldState))
                  {
                        action.ApplyEffects(WorldState);
                  }
            }
      }
}
```

GOAP є одним з найпотужніших і гнучких підходів до розробки AI завдяки своїй здатності адаптуватися до змін у середовищі гри. Однією з основних переваг GOAP є його динамічний характер. AI може швидко переглядати свої плани, якщо умови змінюються, що робить його більш реалістичним та адаптивним. Це дозволяє створювати персонажів, які можуть реагувати на непередбачувані ситуації та приймати обґрунтовані рішення, що покращує загальний ігровий досвід.

Ще однією перевагою є масштабованість. GOAP може використовуватися як для простих, так і для дуже складних поведінкових моделей. У великих іграх з багатьма NPC це особливо корисно, оскільки дозволяє створювати складні взаємодії та сценарії без необхідності прописувати кожну можливу поведінку вручну. Наприклад, у грі серії F.E.A.R., AI ворогів використовує GOAP для

прийняття рішень, що дозволяє їм координувати атаки, використовувати укриття та адаптувати свою тактику в залежності від дій гравця.

Незважаючи на свої численні переваги, GOAP також має кілька недоліків. Один з основних недоліків полягає в тому, що реалізація GOAP може бути складною та вимагати значних обчислювальних ресурсів. Планування дій в реальному часі може бути дуже затратним, особливо у великих ігрових світах з великою кількістю NPC та можливих дій. Це може призвести до затримок і зниження продуктивності гри.

Крім того, GOAP може бути важко налаштувати та підтримувати, оскільки кожна дія має свої передумови та ефекти, які повинні бути точно визначені та скоординовані між собою. Помилки в цих визначеннях можуть призвести до непередбачуваної або некоректної поведінки AI. Тому, хоча GOAP пропонує величезні можливості для створення реалістичного та адаптивного AI, його реалізація вимагає ретельного планування та тестування.

4.4. Utility System

Utility System — це універсальний і адаптивний підхід, що використовується в штучному інтелекті для прийняття рішень на основі оцінки корисності або доцільності різних варіантів. Цей метод особливо ефективний у створенні AI, який може зважувати декілька факторів і вибирати найкращий курс дій, що призводить до більш нюансованої та реалістичної поведінки в іграх.

Основний принцип Utility System полягає у присвоєнні числового значення, або оцінки корисності, кожному потенційному дії або рішенню, яке може прийняти AI. Ці оцінки корисності обчислюються на основі різних факторів або критеріїв, що стосуються поточної ситуації. AI потім вибирає дію з найвищою оцінкою корисності, ефективно приймаючи найбільш "корисне" або "вигідне" рішення відповідно до визначених параметрів.

Процес починається з визначення різних факторів і критеріїв, які впливатимуть на процес прийняття рішень AI. Ці фактори можуть включати рівень здоров'я, відстань до цілі, рівень загрози, доступність ресурсів та інші елементи,

специфічні для контексту. Кожному фактору присвоюється вага або важливість відносно загального рішення. Для кожної потенційної дії обчислюється оцінка корисності шляхом оцінки відповідних факторів. Це часто включає створення функцій корисності для кожного фактора, які перетворюють значення фактора на оцінку корисності. Загальна оцінка корисності для дії зазвичай обчислюється шляхом комбінування індивідуальних оцінок за допомогою зваженої суми або іншого методу агрегування.

Після того, як всі потенційні дії оцінені та їм присвоєні оцінки корисності, AI порівнює ці оцінки та вибирає дію з найвищою оцінкою. Це дозволяє AI динамічно вибирати найкращу дію на основі поточного стану гри або середовища. Кожному фактору відповідає функція корисності, яка визначає, як значення фактора перетворюється на оцінку корисності. Наприклад, функція корисності для здоров'я може призначати вищі оцінки для вищих значень здоров'я, що спонукає AI віддавати перевагу діям, які зберігають здоров'я.

Щоб забезпечити можливість прямого порівняння оцінок корисності з різних факторів, їх часто нормалізують до спільної шкали. Крім того, кожному фактору присвоюється вага, що відображає його важливість. Зважені оцінки корисності потім комбінуються для формування загальної оцінки корисності для кожної дії. AI використовує алгоритм прийняття рішень для оцінки всіх можливих дій, обчислення їх оцінок корисності та вибору дії з найвищою оцінкою. Цей алгоритм зазвичай працює в реальному часі, що дозволяє AI адаптувати свої рішення у міру зміни умов.

Для розрахунку коефіцієнта корисності (Utility), найчастіше використовують наступні функції: лінійка, степенева, логістична та шматково-лінійна.

Лінійна функція (див. формула 4.1) найчастіше являє собою нормалізоване значення від 0 до 1 та відображає пряму залежність між факторами:

$$U = \frac{x}{m} \tag{4.1}$$

де x – вхідне значення,

m – максимальне.

Лінійна функція зручна у випадках, коли існує очевидна пряма залежність (див. рисунок 4.3). Наприклад, це може бути показник здоров'я – якщо він більший, то результат завжди корисніший.

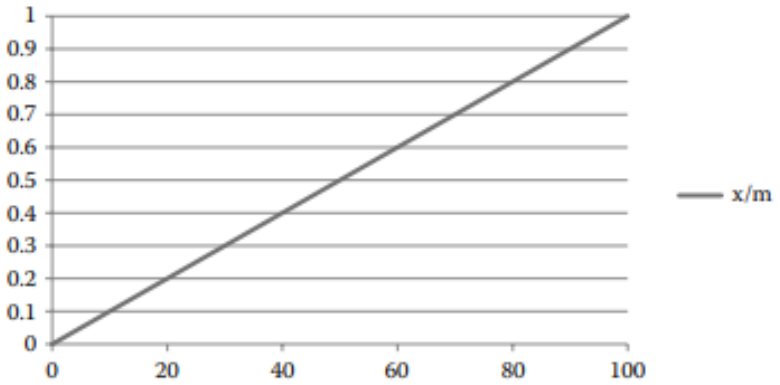

Рисунок 4.3 – графік лінійної функції (рисунок створено самостійно)

Степенева функція (див. формулу 4.2) найчастіше є простим додаванням степеню до лінійної функції:

$$U = \left(\frac{x}{m}\right)^k \tag{4.2}$$

де k – показник степеню.

Степенева функція використовується у випадках, коли середні значення найбільш оптимальні (див. рисунок 4.4). Одним з найяскравіших прикладів може бути дальність до цілі для атаки (у випадку, якщо радіус атаки необмежений).

Також, при необхідності можна використати показник k нижчий за нуль, тоді ми отримаємо інвертований результат і більш швидкий ріст на старті графіку.

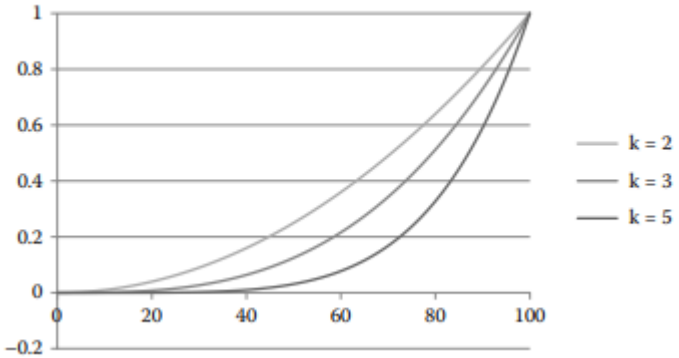

Рисунок 4.4 – графік квадратичної функції (рисунок створено самостійно)

Логістична функція (див. формулу 4.3) використовується у випадках, коли потрібно значно зменшити вплив факторів на результат Utility.

$$U = \frac{1}{1 + e^{-x}} \tag{4.3}$$

Також, даний метод підрахунку Utility часто використовують при роботі з граничними ефектами або насиченістю.

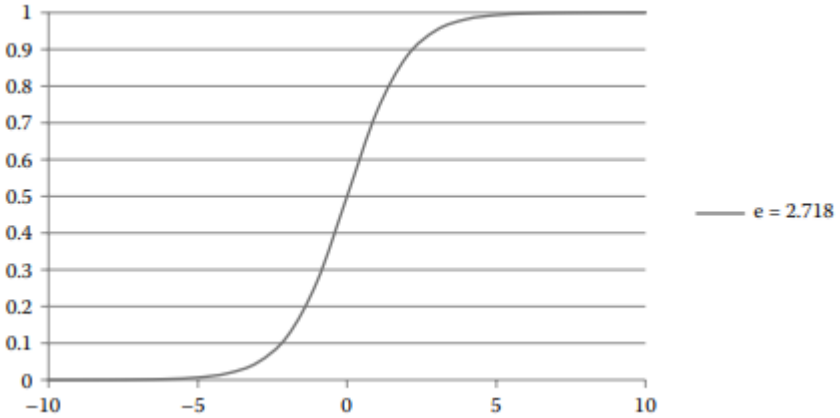

Рисунок 4.5 – графік логістичної функції (рисунок створено самостійно)

В програмній системі для демонстрації роботи патерну Utility System було реалізовано три головних класи: UtilityEvaluator, AIController та Action.

Клас AIController відповідає за стан агенту. В ньому зберігаються усі поточні параметри (як максимальні, так і теперішні), а також в методі Update виконується постійний прорахунок найкориснішої дії.

```
void Update()
{
    EvaluateUtilities();
    Action highestUtilityAction = SelectHighestUtilityAction();
    ExecuteAction(highestUtilityAction);
}
```

Клас UtilityEvaluator відповідає за розрахунок корисності кожної дії. Уся головна логіка зберігається в ньому. Наприклад, в програмі була реалізована дія «Просуватись до двері». Агент може відкрити двері, якщо має 5 монет та йому вистачає енергії. Ось приклад розрахунку Utility для цієї дії:

```
public float EvaluateMoveToDoorUtility(AIController aiController)
{
    if (aiController.coinCount < 5) return 0f;
    float distance = Vector3.Distance(aiController.position,
aiController.doorPosition);
    return 1f / (distance + 1f);
}
```

Клас Action є окремою дією та зберігає в собі реалізацію геймплейних елементів:

```
public abstract class Action
{
    public abstract void Execute(AIController aiController);
}
```

Незважаючи на численні переваги, Utility System також має кілька значних недоліків, які слід враховувати під час розробки штучного інтелекту.

Одним із найбільших недоліків Utility System є складність у виборі та налаштуванні пріоритетів між різними факторами, що впливають на прийняття рішень. Це може бути особливо складним у випадках, коли фактори мають складні взаємозв'язки або коли один і той самий фактор може мати різне значення в різних контекстах.

Іншим великим недоліком є складність у налаштуванні самого алгоритму. Для того, щоб Utility System працювала оптимально, необхідно налаштувати

численні параметри, такі як ваги факторів, функції обчислення корисності, та нормалізаційні константи. Це налаштування є дуже чутливим до змін і потребує ретельного балансу.

Utility System вимагає ретельного тестування та балансування, щоб забезпечити, що всі фактори враховуються правильно і результуюча поведінка AI є логічною та передбачуваною. Це може бути дуже складним і трудомістким процесом, особливо в великих ігрових проектах з багатьма різними ситуаціями та сценаріями.

4.5. Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS) — це підхід до реалізації штучного інтелекту в іграх, який використовує стохастичні симуляції для пошуку найперспективніших дій у грі. Основною метою MCTS є передбачення найкращих можливих дій шляхом багаторазового випадкового дослідження ігрового простору. Цей метод ефективний як для класичних настільних ігор, так і для сучасних настільних ігор та відеоігор.

MCTS [8] базується на симуляції ігор, де як керований AI об’єкт, так і його супротивники виконують випадкові або псевдовипадкові ходи. З однієї випадкової гри можна дізнатися мало, але симуляція безлічі випадкових ігор дозволяє визначити ефективну стратегію. Алгоритм MCTS будує і використовує дерево можливих майбутніх ігрових станів за допомогою наступних етапів:

– вибір (selection): коли стан знайдено в дереві, наступна дія вибирається відповідно до збереженої статистики таким чином, щоб балансувати між використанням і дослідженням. З одного боку, завдання часто полягає у виборі ігрової дії, яка наразі призводить до найкращих результатів (експлуатація). З іншого боку, менш перспективні заходи все ще повинні бути досліджені через невизначеність оцінки (розвідки);

– розширення : коли гра досягає першого стану, який неможливо знайти в дереві, цей стан додається як новий вузол. Таким чином, дерево розширюється на один вузол для кожної змодельованої гри;

– симуляція (simulation): протягом решти гри дії вибираються випадковим чином до кінця гри. Природно, що адекватне зважування ймовірностей вибору дії має значний вплив на рівень гри. Якщо всі можливі дії обрані з рівною ймовірністю, то стратегія часто є слабкою, а рівень алгоритму Монте-Карло неоптимальним. Ми можемо використовувати евристичні знання, щоб надати більшої ваги діям, які виглядають більш перспективними;

– зворотне поширення (backpropagation): після досягнення кінця симульованої гри ми оновлюємо кожен вузол дерева, який було пройдено під час цієї гри. Кількість відвідувань збільшується, а співвідношення перемог/програшів змінюється відповідно до результату.

Схему роботи алгоритму можна побачити на рисунку 4.6 нижче.

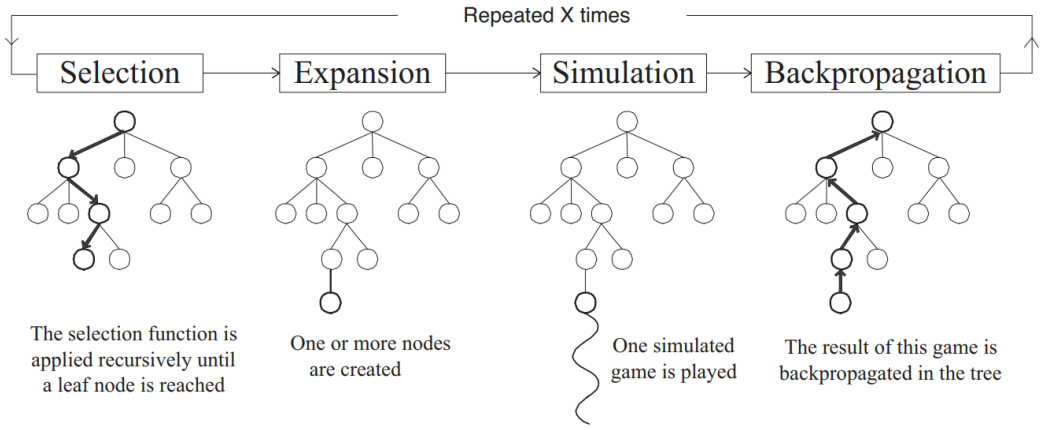

Рисунок 4.6 – схема роботи Monte Carlo Tree Search (рисунок створено самостійно)

Основна перевага MCTS полягає у його здатності передбачати ходи противника. Це досягається завдяки численним випадковим симуляціям, які дозволяють визначити найкращі можливі дії у грі.

Також, MCTS має високі вимоги до обчислювальних ресурсів. Алгоритм потребує великої кількості симуляцій для досягнення високої якості гри, що може бути проблематичним у складних іграх з численними можливими ходами. Також MCTS може стикатися з обмеженнями часу на прийняття рішень у реальному часі.

Крім того, у іграх з високою глибиною дерева можливих ходів MCTS може потребувати дуже багато симуляцій для достатньо глибокого аналізу.

Найпоширенішими прикладами ігор, де використовується даний підхід, є шахи, го та шашки.

ВИСНОВКИ

В результаті виконання даного проєкту були досягнуті визначені цілі, зокрема, проведено глибокий аналітичний огляд на тему «Дослідження впливу патернів штучного інтелекту на ігровий процес». Було зібрано та проаналізовано відомості про ігровий штучний інтелект, його вплив на ігровий процес, а також про технології та платформи, які використовуються для його реалізації. Особлива увага приділялася аналізу історичного розвитку та сучасних тенденцій використання ШІ в іграх, вивченню його різноманітних підходів та методів, а також оцінці перспектив його майбутнього розвитку.

Також було детально розглянуто технічні аспекти використання штучного інтелекту в ігровій індустрії, включаючи його потенційні обмеження та виклики.

Було проведено аналіз п’яти окремих патернів реалізації ШІ в іграх. Були оцінені їх слабкі та сильні сторони, розглянуті деталі реалізації та приклади реальних проектів, де вони використовувались. Також проведена оцінка проблем кожного з патерну, їх шляхи вирішення та рекомендацій щодо використання патернів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Application Design Patterns: State Machines – NI. URL: https://www.ni.com/en/support/documentation/supplemental/16/simple-state-machine-template-documentation.html (дата звернення: 16.03.2024)..

2. The Pac-Man Dossier. URL: https://www.gamedeveloper.com/design/the-pac-man-dossier (дата звернення: 18.03.2024).

3. History of AI in Games – HalfLife & Halo. URL: https://modl.ai/halflife-halo (дата звернення: 19.03.2024).

4. Monte-Carlo Tree Search, Mark H.M. Winands, 9 стр.;

5. Unreal Engine 5. URL: https://www.unrealengine.com/en-US/unreal-engine-5 (дата звернення: 22.03.2024).

6. Game AI Pro 2 / ed. by S. Rabin. A K Peters/CRC Press, 2015. URL: https://doi.org/10.1201/b18373 (дата звернення: 25.03.2024).

7. Game AI Pro 3 / ed. by S. Rabin. A K Peters/CRC Press, 2017. URL: https://doi.org/10.4324/9781315151700 (дата звернення: 28.03.2024).

8. A Study of Optimization Models for Creation of Artificial Intelligence for the Computer Game in the Tower Defense Genre / O. Mazurova та ін. 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T). 2020. С. 491-496.