

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____ Центр післядипломної освіти
(повна назва)
Кафедра _____ Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____ перший (бакалаврський)

_____ Моделі обчислювального інтелекту для адаптивного управління динамічним
_____ контентом в ігрових середовищах з відкритим світом
_____ (тема)

Виконав:
здобувач _____ другого _____ року навчання,
групи _____ ІТШІпз-24-1

_____ Вікторія ВОРОБІЙОВА
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки

(код і повна назва спеціальності)

Тип програми _____ освітньо-професійна
Освітня програма _____ Штучний інтелект

(повна назва освітньої програми)

Керівник _____ проф. Марина КУДРЯВЦЕВА
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ

(підпис)

_____ Лариса ЧАЛА
(власне ім'я, прізвище)

2026 р.

Харківський національний університет радіоелектроніки

Факультет _____ Центр післядипломної освіти _____

Кафедра _____ Штучного інтелекту _____

Рівень вищої освіти _____ перший (бакалаврський) _____

Спеціальність _____ 122 Комп'ютерні науки _____
(код і повна назва)

Тип програми _____ освітньо-професійна _____

Освітня програма _____ Штучний інтелект _____
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____

(підпис)

«_____» _____ 20 ____ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві _____ Воробйовій Вікторії Яківні _____
(прізвище, ім'я, по батькові)

1. Тема роботи Моделі обчислювального інтелекту для адаптивного управління динамічним контентом в ігрових середовищах з відкритим світом

затверджена наказом університету від 18 травня 2026 р. № 77Стз

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2026 р.

3. Вихідні дані до роботи наукові публікації з теорії потоку та адаптивного управління складністю в іграх, офіційна документація Unreal Engine 5 та його підсистем (World Partition, Behavior Trees, Significance Manager, NNE Plugin, UMG), офіційна документація мови програмування C++ та стандартної бібліотеки, офіційна документація Python та бібліотек PyTorch, NumPy і Stable-Baselines3, специфікація формату ONNX та документація інструментів експорту моделей, дослідження у галузі навчання з підкріпленням та алгоритмів DQN, публікації з проектування ігрових інтерфейсів та UI/UX для ігрових рушіїв, загальнодоступні набори даних ігрових сесій та поведінкових метрик гравців, веб-ресурси та академічні публікації з проблематики Dynamic Difficulty Adjustment

4. Перелік питань, що потрібно опрацювати в роботі _____

1) Аналіз предметної галузі адаптивного управління в іграх _____

2) Математичне та алгоритмічне забезпечення моделі обчислювального інтелекту _____

3) Програмна реалізація інтелектуальної системи та ігрового прототипу _____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / термін виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	18.05.2026	виконано
2	Аналіз предметної галузі	19.05.2026	виконано
3	Огляд існуючих DDA-систем та підходів	19.05.2026	виконано
4	Концептуальна розробка системи адаптивного управління	20.05.2026	виконано
5	Визначення вимог до системи та вибір інструментарію	21.05.2026	виконано
6	Формалізація задачі у термінах MDP та обґрунтування вибору DQN	22.05.2026	виконано
7	Розробка моделі профілювання гравця та класифікатора станів	22.05.2026	виконано
8	Дослідження та розробка алгоритму LoD-AI	23.05.2026	виконано
9	Порівняльний аналіз моделей (rule-based, fuzzy logic, DQN)	24.05.2026	виконано
10	Навчання RL-агента та експорт моделі у ONNX	25.05.2026	виконано
11	Програмна реалізація C++-компонентів у Unreal Engine 5	30.05.2026	виконано
12	Розробка Debug/Monitor Panel та системи hot reload	02.06.2026	виконано
13	Написання пояснювальної записки	03.06.2026	виконано
14	Попередній захист	12.06.2026	виконано
15	Захист перед екзаменаційною комісією	16.06.2026	

Дата видачі завдання 18 травня 2026 р.

Здобувач 
(підпис)

Керівник роботи _____
(підпис)

проф. Марина КУДРЯВЦЕВА
(посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 63 с., 2 табл., 4 дод., 26 джерел.

АДАПТИВНЕ УПРАВЛІННЯ, ВІДКРИТИЙ СВІТ, ДИНАМІЧНЕ МАСШТАБУВАННЯ СКЛАДНОСТІ, ІГРОВИЙ ШТУЧНИЙ ІНТЕЛЕКТ, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, НЕІГРОВІ ПЕРСОНАЖІ, ОБЧИСЛЮВАЛЬНИЙ ІНТЕЛЕКТ, ПРОФІЛЮВАННЯ ГРАВЦЯ, ТЕОРІЯ ПОТОКУ, UI/UX МОНІТОРИНГ.

Об'єктом дослідження виступає процес адаптивного управління ігровим контентом у середовищах відкритого типу та інструментальний інтерфейс моніторингу відповідної системи.

Предметом дослідження є моделі обчислювального інтелекту, алгоритми навчання агента з підкріпленням та UI/UX-рішення для задачі динамічного масштабування складності ігрового процесу.

Метою роботи є проектування й реалізація системи адаптивного управління контентом на базі RL-агента та профілювання поведінкових показників гравця (`skill_score`, `stress_index`, `flow_zone`), а також побудова Debug/Monitor UI для налагодження та моніторингу ШІ-агента.

Дослідження виконане шляхом проектування агента у рамках MDP, програмної реалізації на C++ (Unreal Engine 5) з Python-бекендом для машинного навчання, побудови Debug Panel за принципами UX та аналізу стратегій DDA.

ABSTRACT

Bachelor's thesis contains: 63 pp., 2 tabl., 4 ann., 26 references.

ADAPTIVE MANAGEMENT, COMPUTATIONAL INTELLIGENCE, DEBUG PANEL, DYNAMIC DIFFICULTY ADJUSTMENT, FLOW THEORY, GAME AI, NON-PLAYER CHARACTERS, OPEN WORLD, PLAYER PROFILING, REINFORCEMENT LEARNING, UI/UX.

The object of the research is the process of adaptive game content management in open environments and the instrumental monitoring interface of the corresponding system.

The subject of the research is computational intelligence models, reinforcement learning algorithms for an agent, and UI/UX solutions for the problem of dynamic scaling of the complexity of the game process.

The purpose of the work is to design and implement an adaptive content management system based on an RL agent and profiling player behavioral indicators (skill_score, stress_index, flow_zone), as well as building a Debug/Monitor UI for debugging and monitoring an AI agent.

The research was carried out by designing an agent within the MDP framework, software implementation in C++ (Unreal Engine 5) with a Python backend for machine learning, building a Debug Panel based on UX principles, and analyzing DDA strategies.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів	8
Вступ.....	9
1 Аналіз предметної галузі адаптивного управління в іграх	11
1.1 Архітектура ігрових середовищ з відкритим світом та класифікація контенту	11
1.2 Проблема адаптивного управління складністю (DDA)	12
1.2.1 Концепція теорії потоку як основа ігрового балансу.....	14
1.2.2 Вплив адаптації на ігровий досвід та утримання користувача .	15
1.3 Роль неігрових персонажів у формуванні динамічного оточення	16
1.4 Огляд існуючих рішень для адаптації контенту	17
1.5 Постановка задачі та обґрунтування вибору методів дослідження...	19
2 Математичне та алгоритмічне забезпечення моделі обчислювального інтелекту.....	21
2.1 Формалізація задачі управління контентом на основі навчання з підкріпленням	21
2.2 Побудова моделі профілювання гравця та оцінка емоційного стану	23
2.3 Алгоритмічне рішення для динамічного масштабування складності III (LoD-AI)	24
2.4 Проектування логіки взаємодії та інтерфейсу системи (UI/UX)	26
2.4.1 Сценарії взаємодії та візуалізація прихованих параметрів адаптації	26
2.4.2 Принципи побудови Debug-інтерфейсу для моніторингу III ..	27
2.5 Альтернативні моделі обчислювального інтелекту для задачі DDA	28
2.5.1 Rule-based контролер складності.....	28
2.5.2 Контролер на основі нечіткої логіки.....	30
2.6 Порівняльний аналіз моделей та обґрунтування вибору DQN	32
3 Програмна реалізація інтелектуальної системи та ігрового прототипу ...	36
3.1 Загальна архітектура системи та вибір інструментарію розробки	36

3.2 Реалізація модуля профілювання та класифікатора станів гравця	37
3.3 Програмна реалізація RL-агента та формування функції винагороди	38
3.4 Розробка ігрового прототипу у відкритому світі.....	39
3.5 Реалізація UI-компонентів та системи зворотного зв'язку	40
3.6 Інтеграція компонентів у єдиний ігровий цикл	41
Висновки	43
Перелік джерел посилання	44
Додаток А Код Double DQN агент для динамічного масштабування складності.....	48
Додаток Б Код модулю профілювання поведінкових показників гравця ...	53
Додаток В Код ігрової підсистеми інференсу та оркестрації DDA	58
Додаток Г Відомість кваліфікаційної роботи.....	63

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

DDA – Dynamic Difficulty Adjustment – динамічне масштабування складності;

DQN – Deep Q-Network – глибока Q-мережа;

HUD – Heads-Up Display – ігровий інформаційний оверлей;

LoD – Level of Detail – рівень деталізації;

MDP – Markov Decision Process – марковський процес прийняття рішень;

NNE – Neural Network Engine – рушій нейронних мереж (плагін UE5);

NPC – Non-Player Character – неігровий персонаж;

ONNX – Open Neural Network Exchange – відкритий формат обміну нейронними мережами;

RL – Reinforcement Learning – навчання з підкріпленням;

UMG – Unreal Motion Graphics – система побудови UI в Unreal Engine;

UE5 – Unreal Engine 5 – ігровий рушій п'ятого покоління;

UI – User Interface – інтерфейс користувача;

UX – User Experience – досвід користувача.

ВСТУП

Ігри з відкритим світом – це, мабуть, найцікавіший і водночас найскладніший жанр в сучасній ігровій індустрії. Коли вперше запускаєш Skyrim або Witcher 3, одразу відчувається простір – можна йти куди хочеш, робити що хочеш. Але якщо придивитись уважніше, за цим враженням стоїть колосальна інженерна проблема: як зробити так, щоб гравець з будь-яким рівнем досвіду відчував однаково захопливий виклик, незалежно від того, куди він пішов і що вирішив робити. Ця проблема і стала відправною точкою для даної роботи.

Один із підходів до вирішення цієї проблеми – це Dynamic Difficulty Adjustment, або DDA. Ідея проста: не фіксувати складність наперед, а дивитись на те, як грає конкретна людина, і підлаштовувати параметри гри під неї в реальному часі. Теоретичне обґрунтування цього підходу дає теорія потоку психолога Міхая Чіксентміхаї [1]. Він ще у 1975 році описав, що людина отримує максимальне задоволення від будь-якої діяльності тоді, коли складність завдання приблизно відповідає її рівню навичок. Якщо завдання надто просте – стає нудно. Якщо надто складне – з'являється тривога і бажання все кинути. І тільки десь посередині виникає той стан повного занурення, який він назвав «потокком».

Поєднати DDA з машинним навчанням намагались і до цього, але більшість таких розробок стосується лінійних ігор або ігор з рівнями, де все більш-менш передбачувано. У відкритому світі все інакше – гравець може піти куди завгодно, обрати будь-яку послідовність дій. Ніяка таблиця правил такого не передбачить. Тут потрібен агент, який сам навчається на основі того, що відбувається в грі [2].

Також існує проблема, яка полягає в тому, що розробник бачить тільки результат і не може нічого підкрутити без перезапуску всього. Тому в цій роботі окрема увага приділена розробці Debug/Monitor Panel – це інтерактивний оверлей прямо в Unreal Engine 5, де можна в реальному часі

бачити що робить агент, змінювати параметри без перезапуску гри та зберігати дані сесій для аналізу.

Мета роботи – розробити і реалізувати систему адаптивного управління контентом у відкритому ігровому світі, яка поєднує RL-агента, профілювання поведінки гравця через показники `skill_score`, `stress_index` і `flow_zone`, а також повноцінний Debug/Monitor UI.

Для досягнення цієї мети потрібно вирішити такі задачі: розглянути існуючі підходи до DDA в комерційних і дослідницьких проєктах і зрозуміти їх сильні та слабкі сторони; формалізувати задачу адаптації у термінах марковського процесу прийняття рішень і обґрунтувати чому саме DQN підходить для її вирішення; побудувати модель профілювання гравця і класифікатор його емоційного стану на основі поведінкових метрик; розробити алгоритм LoD-AI для управління складністю NPC через чотири рівні деталізації поведінки; спроектувати два інтерфейси – мінімалістичний HUD для гравця і насичену інформацією Debug Panel для розробника; реалізувати все це на C++ в Unreal Engine 5 з Python-бекендом для навчання агента.

Об'єкт дослідження – процес адаптивного управління складністю ігрового контенту в середовищах відкритого типу та інструментальний інтерфейс моніторингу відповідної системи.

Предмет дослідження – моделі обчислювального інтелекту, алгоритми навчання агента з підкріпленням та UI/UX-рішення для задачі динамічного масштабування складності ігрового процесу.

Практична цінність роботи в тому, що розроблені компоненти – особливо Debug Panel з hot reload і CSV-експортом – можна використовувати в будь-якому UE5-проєкті як готовий інструмент для налагодження адаптивного ІІІ без перезапуску гри.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ АДАПТИВНОГО УПРАВЛІННЯ В ІГРАХ

1.1 Архітектура ігрових середовищ з відкритим світом та класифікація контенту

Поняття «відкритий світ» настільки вкоренилось у сучасному ігровому дискурсі, що за ним легко не помітити реальну інженерну складність. Відкритий світ – це не просто «велика карта», це архітектурне рішення, що принципово змінює стосунки між дизайнером і гравцем: замість провадити гравця заздалегідь прокладеним маршрутом, розробник будує середовище, де гравець сам обирає свій шлях [3].

Сучасні відкриті світи влаштовані досить складно з точки зору архітектури. Якщо брати Unreal Engine 5 як приклад, то фізичний шар – тобто геометрія, рельєф, будівлі – базується на технології Nanite і системі World Partition. World Partition це по суті автоматизована система яка сама вирішує які шматки рівня завантажувати в пам'ять залежно від того де знаходиться гравець [4]. Раніше розробники робили це вручну, розбиваючи рівень на підрівні і прописуючи логіку переходів. Тепер це відбувається автоматично, що значно спрощує роботу з великими просторами.

Окрім фізичного шару є ще логічний – це правила взаємодії об'єктів, фізика, механіки бою, квестова система. Є шар контенту – персонажі, предмети, події. І є управляючий шар – підсистеми штучного інтелекту, які відповідають за поведінку NPC, спавн ворогів і адаптацію середовища до дій гравця. Саме цей останній шар і є головним предметом даної роботи.

Контент у відкритих світах можна розділити на кілька типів залежно від того як він взаємодіє з гравцем. Статичний контент – це все що не змінюється і не реагує на дії гравця: будівлі, гори, дерева, декоративні елементи. Динамічний контент – це те що живе своїм життям: NPC з розкладами, ресурси які відновлюються, погода яка змінюється.

Процедурний контент генерується в момент потреби – наприклад система може створити нове підземелля або підібрати набір ворогів прямо під час гри, без того щоб дизайнер прописував кожен варіант вручну [5].

З точки зору впливу на ігровий досвід виділяють три категорії. Наративний контент – квести, діалоги, записники, аудіозаписи – все що розповідає історію. Механічний контент – зброя, атрибути ворогів, рівні складності – все що впливає на баланс. І атмосферний контент – музика, освітлення, звуки навколишнього середовища – все що створює настрій [6]. Для задачі DDA найважливішим є механічний контент, бо саме через нього найпростіше керувати складністю.

Тут варто згадати принцип Level of Detail, або LoD, який зазвичай застосовують до графіки: об'єкти поблизу гравця відображаються детально, далекі – спрощено. В Unreal Engine 5 є Significance Manager який робить щось схоже, але для обчислень: близькі об'єкти отримують більше ресурсів процесора, далекі – менше [7]. В даній роботі ця ідея переноситься на рівень поведінки NPC – чим далі ворог від гравця, тим простіша його логіка і тим рідше вона оновлюється. Цей підхід отримав назву LoD-AI і став основним механізмом адаптації в розробленій системі.

Варто підкреслити: саме висока ентропія стану гравця відрізняє відкритий світ від лінійної гри і ускладнює DDA. У рівневому шутері дизайнер наперед знає послідовність зіткнень. У відкритому світі будь-яке таке припущення хибне. Звідси – принципова необхідність методів машинного навчання замість ручних правил.

1.2 Проблема адаптивного управління складністю (DDA)

Dynamic Difficulty Adjustment – це автоматичне, в ідеалі непомітне, налаштування параметрів ігрового процесу відповідно до поточного стану гравця, активне дослідницьке зацікавлення виникло з появою робіт

Yannakakis і Hallam [8], які показали, що показники залучення гравця можна використовувати як сигнал для навчання адаптивних систем.

Всі DDA-системи можна розділити на явні і неявні [9]. Явні – це коли гра відкрито говорить гравцю що змінює складність, наприклад пропонує перейти на легший рівень після кількох поразок підряд. Неявні – це коли все відбувається приховано: ворог непомітно починає бити трохи слабше, або корисні предмети починають випадати трохи частіше. Більшість сучасних ігор використовують саме неявний підхід, і причина проста: якщо гравець помічає що гра йому «підіграє», відчуття власних досягнень одразу зникає і це псує весь досвід.

Основні параметри, які зазвичай доступні для адаптації це: атрибути противників (здоров'я, шкода, швидкість реакції), щільність спавну ворогів, якість і кількість ресурсних дропів, складність поведінкових патернів NPC, тривалість ігрових таймерів Дослідження Hunicke і Chapman [10] показали, що навіть незначні зміни кількох параметрів одночасно суттєво впливають на суб'єктивне відчуття складності.

Сучасний розвиток DDA все більше рухається від простих правил і евристик у бік методів машинного навчання. Дослідники з Electronic Arts [11] показали що для того щоб RL-агент міг ефективно керувати складністю, стан гравця потрібно описувати не одним числом а набором метрик.

Для максимально точного управління складністю доцільно використовувати чотиривимірний простір метрик. Це дозволило би не просто реагувати на помилки гравця, а прогнозувати його поведінку. Skill Score – оцінка технічного рівня гравця, його точності і ефективності. Stress Index – рівень стресу, який відображає наскільки гравець перевантажений і наскільки хаотично він діє. Flow Status – бінарний показник того чи знаходиться гравець в зоні потоку. І Behavioral Profile – загальна класифікація стилю гри за його пріоритетами.

1.2.1 Концепція теорії потоку як основа ігрового балансу

Якщо спитати досвідченого геймера, що він відчуває у кращі моменти гри, – найчастіше почуєш щось на кшталт «я просто грав, не думаючи, і раптом виявилось, що пройшло три години». Саме цей стан угорсько-американський психолог Міхай Чіксентміхаї описав у своїй роботі 1975 року і назвав його «поток» [12]. За його словами це стан повного занурення в діяльність коли людина діє автоматично, не думаючи про зусилля, і отримує від самого процесу максимальне задоволення.

Головний висновок Чіксентміхаї полягає в тому що потік виникає тільки при певному співвідношенні складності завдання і рівня навичок людини [13]. Графічно це зображується як вузька «смуга потоку» у двовимірному просторі «навички – виклик». Нижче смуги – нудьга: завдання надто просте для наявних навичок. Вище – тривога: завдання переважає можливості виконавця. І лише всередині смуги – оптимальне залучення.

Sweetser і Wyeth у 2005 році перенесли цю модель безпосередньо у контекст відеоігор і розробили фреймворк GameFlow [14]. Вони показали що всі вісім компонентів потоку – концентрацію, виклик, навички, контроль, цілі, зворотний зв'язок, занурення і соціальну взаємодію – можна виміряти через реальні ігрові метрики. Це стало теоретичним фундаментом для побудови DDA-систем на основі теорії потоку.

Математично умова знаходження гравця в каналі потоку записується так:

$$|S(t) - C(t)| \leq \varepsilon \quad (1.1)$$

де $S(t)$ – це рівень навичок гравця в момент t ;

$C(t)$ – поточний рівень виклику який пред'являє гра;

ε – ширина каналу потоку.

В даній роботі $S(t)$ реалізований через показник `skill_score`, $C(t)$ через `challenge_level` яким керує RL-агент, а значення ϵ встановлено рівним 0.15 на основі емпіричних спостережень.

Важливо також розуміти що навички гравця $S(t)$ не стоять на місці. Чим більше людина грає тим краще вона розуміє механіки, виробляє ефективні тактики, звикає до поведінки ворогів. Тобто $S(t)$ постійно зростає, і DDA-система повинна це враховувати і відповідно підтягувати $C(t)$ вгору. Якраз ця необхідність постійно відстежувати динаміку показників і коригувати параметри в реальному часі і робить навчання з підкріпленням природним вибором для реалізації такої системи.

1.2.2 Вплив адаптації на ігровий досвід та утримання користувача

Для ігрових компаній одними з ключових показників є тривалість ігрових сесій і те наскільки часто гравці повертаються до гри. І хоча зв'язок між якісною DDA і цими показниками здається очевидним, є цілком конкретні дослідження які це підтверджують.

Наприклад Andrade та співавтори [15] провели експеримент де порівнювали сесії з активною DDA і без неї на 60 учасниках. Виявилось що в групі з DDA середня тривалість сесії зросла на 28 відсотків, а оцінка задоволеності від гри підвищилась на 1.8 бала з п'яти. Схожі результати отримали і Zohaib [16], хоча вони використовували дещо іншу методологію.

Przybylski, Rigby і Ryan [17] підійшли до цього питання з боку психології і спираючись на теорію самодетермінації показали що відчуття власної компетентності є одним з найсильніших предикторів довготривалої мотивації до гри. Якщо гравець відчуває що він справді вміє грати і що його успіхи заслужені – він буде повертатись до гри знову і знову. DDA яка правильно підтримує цей баланс формує саме таке відчуття.

Разом із тим є суттєве застереження. Дослідження [18] показують що погано реалізована DDA може дати прямо протилежний ефект. Якщо

гравець починає помічати що гра йому підіграє, наприклад якщо ворог раптом помітно став слабкішим, гравець починає сприймати свої успіхи як незаслужені. Цей ефект добре відомий по гоночних іграх де комп'ютерні суперники штучно сповільнюються коли гравець сильно відстає – так звана «гумова стрічка».

Саме тому в даній роботі зміни `challenge_level` відбуваються дуже плавно – не більше ніж на 0.05 за один крок агента що відповідає одній секунді. При такому темпі зміни відбуваються в межах природного варіювання ігрового середовища і гравець сприймає їх просто як особливості конкретного ворога або ситуації, а не як очевидне втручання системи [19].

1.3 Роль неігрових персонажів у формуванні динамічного оточення

NPC – це по суті головний інструмент через який гравець відчуває рівень складності гри. Ворог який б'є занадто сильно і занадто точно – це фрустрація і бажання вимкнути гру. Ворог який ледве рухається і не реагує на дії гравця – це нудьга і відчуття що час витрачено даремно. Знайти між ними баланс і є практичним завданням DDA стосовно NPC.

У відкритих світах NPC зазвичай поділяються на кілька типів. Ворожі агенти безпосередньо протистоять гравцю – в розробленому прототипі це три типи: Grunt з базовими характеристиками, Scout з підвищеною швидкістю і чутливістю, і Captain який може координувати атаки з іншими ворогами але тільки на повному рівні деталізації. Нейтральні агенти – торговці, жителі міст, перехожі – вони не воюють але створюють відчуття живого світу. Союзники якщо вони передбачені грою потребують окремої логіки кооперативної поведінки. І квестові NPC які дають завдання і просувають сюжет.

Поведінка NPC реалізується через дерева поведінки. В Unreal Engine 5 Behavior Trees є стандартним інструментом для цього, і одна з їх головних

переваг – модульна структура яка дозволяє просто «вимикати» складні гілки логіки коли вони не потрібні [20]. Саме це і використовується в концепції LoD-AI: залежно від відстані до гравця NPC перемикається на спрощене або повне дерево поведінки без необхідності переписувати всю логіку.

В даній роботі використовуються чотири рівні деталізації поведінки NPC. На рівні LoD-0 при відстані до гравця менше 15 метрів NPC використовує повне дерево поведінки з усіма можливостями – cover system, кооперативні атаки, реакція на звук і зір, оновлення 30 разів на секунду. На рівні LoD-1 від 15 до 40 метрів залишається тільки базове переслідування і атака, без тактики, оновлення 10 разів на секунду. На рівні LoD-2 від 40 до 80 метрів NPC просто патрулює свій маршрут з оновленням двічі на секунду. На рівні LoD-3 понад 80 метрів NPC взагалі заморожується і не витрачає ресурси процесора. Перехід між рівнями відбувається автоматично, але порогові відстані масштабуються параметром `challenge_level` [21].

Ключова деталь: порогові відстані між рівнями не фіксовані, а масштабуються залежно від `challenge_level`. Тобто коли агент підвищує складність, більша кількість NPC одночасно знаходиться на вищих рівнях деталізації з більш складною поведінкою – і гравець відчуває це як природне посилення тиску, а не як очевидне підкручування параметрів. Champandard [22] підтвердив що переходи між рівнями LoD залишаються непомітними для гравця якщо вони відбуваються достатньо далеко від центру його уваги.

1.4 Огляд існуючих рішень для адаптації контенту

Перш ніж пропонувати власне рішення, варто розглянути те, що вже існує в індустрії та дослідницькій літературі. Картина досить строката: від елегантних промислових систем до академічних прототипів – і кожен підхід відкриває щось корисне.

Найпростіший і найстаріший підхід – системи на основі жорстких правил. Логіка проста: якщо гравець програв тричі підряд – зменшити здоров'я ворогів на 20 відсотків. Серія Resident Evil використовує щось подібне ще з 1996 року, відстежуючи влучання і промахи і коригуючи параметри відповідно. Перевага цього підходу в тому що він простий і передбачуваний, легко налагоджується і не вимагає складної технічної реалізації. Проблема в тому що такі системи дуже крихкі: варто гравцю зробити щось що не передбачали дизайнери – і вся логіка ламається. В умовах відкритого світу де можливих сценаріїв тисячі це практично не працює [8].

Статистичні підходи роблять крок вперед: замість жорстких порогових правил вони агрегують метрики за ковзним вікном і оцінюють загальну тенденцію. Гарний комерційний приклад – FIFA Procedural Persona, яка відстежує ефективність гравця протягом матчу і динамічно змінює тактику і агресивність суперника. Дослідження Electronic Arts [11] показали що якщо описувати стан гравця не одним числом а набором метрик – skill, stress, behavioral profile – точність адаптації суттєво зростає порівняно з однопараметричними підходами.

Одна з найвідоміших промислових реалізацій DDA – це AI Director у Left 4 Dead від Valve 2008 року. Ця система в реальному часі аналізує стан всієї команди гравців і на основі цього керує хвилями ворогів, кількістю аптечок і загальним темпом подій. Мета – підтримувати постійне відчуття напруги не даючи грі ставати ні занадто легкою ні нестерпно складною.

В академічному середовищі найперспективнішим напрямком є поєднання навчання з підкріпленням і профілювання гравця. Яннакакіс і Тоджеліус [2] запропонували концепцію «experience-driven procedural content generation» – ідею про те що контент може генеруватись або підлаштовуватись у відповідь на поточний емоційний стан гравця. Це дуже близько до того що реалізується в даній роботі.

Але є проблема яка є спільною практично для всіх існуючих рішень – відсутність нормального інструменту для того щоб розробник міг спостерігати за тим що відбувається всередині системи. Комерційні DDA це чорні скриньки: можна подивитись агреговану статистику після сесії але не можна побачити в реальному часі чому агент прийняв конкретне рішення. Debug/Monitor Panel покликана вирішити саме цю проблему.

1.5 Постановка задачі та обґрунтування вибору методів дослідження

На основі проведеного аналізу предметної галузі задача формулюється так: необхідно розробити систему, яка в реальному часі спостерігає за поведінкою гравця через показники `skill_score`, `stress_index` і `flow_zone`, на основі цих даних приймає рішення про зміну параметрів ігрового середовища через механізм `challenge_level` і `LoD-AI`, і при цьому надає розробнику зручний інструмент для спостереження і налаштування через Debug/Monitor Panel.

Система повинна відповідати таким вимогам. Адаптація повинна бути непомітною для гравця – зміна `challenge_level` не більше ніж на 0.05 за один крок. Конфігурацію повинно бути можливим змінювати без перезапуску гри через механізм `hot reload`. Debug Panel повинна бути доступна тільки в dev-білдах і не потрапляти у фінальний реліз. Час відгуку інтерфейсу не більше 16 мілісекунд що відповідає 60 кадрам на секунду. Дані сесій повинні зберігатись у CSV для подальшого аналізу.

Вибір навчання з підкріпленням (конкретно DQN) як основного механізму прийняття рішень обумовлений тим, що функція переходів у відкритому світі невідома і стохастична – поведінка гравця принципово непередбачувана, тому `model-based` алгоритми непридатні; DQN добре справляється з дискретним простором дій і теоретично добре вивчений; навчена модель може бути експортована у формат ONNX і виконуватись у рушії без Python-залежностей у рантаймі, що критично для продуктивності

у відкритому світі. Додатково, метод Double DQN усуває систематичне переоцінювання Q-значень, що особливо важливо при нестационарному середовищі – яким і є поведінка гравця.

Вибір C++ і Unreal Engine 5 пояснюється промисловою продуктивністю при обробці сотень NPC, вбудованими Behavior Trees, Mass Entity Framework для DOTS-подібної обробки агентів, NNE Plugin для ONNX-інференсу та UMG для побудови Debug Panel. Технологія World Partition UE5 [4] безпосередньо підтримує архітектуру LoD-AI через Significance Manager [7]. Python залишається лише для фази навчання DQN-агента – тут його екосистема (PyTorch, Stable-Baselines3) не має конкурентів серед промислових інструментів.

2 МАТЕМАТИЧНЕ ТА АЛГОРИТМІЧНЕ ЗАБЕЗПЕЧЕННЯ МОДЕЛІ ОБЧИСЛЮВАЛЬНОГО ІНТЕЛЕКТУ

2.1 Формалізація задачі управління контентом на основі навчання з підкріпленням

Щоб навчити агента приймати правильні рішення потрібно спочатку формально описати середовище в якому він діє – що він може спостерігати, які дії може виконувати, і як оцінювати наскільки хороші або погані ці дії. Стандартний математичний інструмент для цього – марковський процес прийняття рішень або MDP. Концепцію оптимальності в таких процесах описав ще Беллман у 1950-х [23], а сучасна форма MDP для задач навчання з підкріпленням закріплена в підручнику Саттона та Барто [24]:

$$M = \langle S, A, P, \gamma \rangle \quad (2.2)$$

де S – множина можливих станів середовища;

A – множина дій які може виконати агент;

$P(s'|s,a)$ – ймовірність того що виконавши дію a зі стану s агент опиниться в стані s' ;

$R(s,a)$ – винагорода яку агент отримує за пару стан-дія;

γ – коефіцієнт дисконтування який визначає наскільки агент цінує майбутні винагороди порівняно з поточними.

Вектор стану який передається агенту в кожен момент часу складається з шести компонентів: $s(t) = [\text{skill_score}(t), \text{stress_index}(t), \text{flow_zone}(t), h(t), \text{kd}(t), \text{session_len}(t)]$.

Всі компоненти нормалізовані до діапазону від 0 до 1. Skill_score – рівень майстерності гравця. Stress_index – рівень стресу. Flow_zone – бінарний показник того чи гравець в зоні потоку. H – поточне здоров'я. Kd –

відношення знищень до смертей за останні 10 боїв. `Session_len` – нормалізована тривалість поточної сесії.

Простір дій дискретний і містить вісім варіантів як агент може вплинути на гру, які описані в таблиці 2.1.

Таблиця 2.1 – Простір дій RL-агента

Дія	Ідентифікатор	Вплив	Ефект
a ₁	INCREASE_CHALLENGE	підвищення <code>challenge_level</code> , збільшення радіусів LoD-AI	складніша гра
a ₂	DECREASE_CHALLENGE	зниження <code>challenge_level</code> , зменшення радіусів LoD-AI	простіша гра
a ₃	INCREASE_RESOURCE_SPAWN	збільшення частоти спавну ресурсних дропів	допомога гравцю
a ₄	DECREASE_RESOURCE_SPAWN	зменшення частоти спавну ресурсних дропів	більший виклик
a ₅	INCREASE_ENEMY_COUNT	активація додаткового патрулю ворогів	збільшення тиску
a ₆	DECREASE_ENEMY_COUNT	деактивація патрулю, зменшення кількості ворогів	зниження тиску
a ₇	ADJUST_NPC_ACCURACY	корекція точності атак та реакції NPC	тонке балансування
a ₈	NULL_ACTION	збереження поточних параметрів	утримання балансу

При точному збігу `skill_score` і `challenge_level` `r_flow` дорівнює 1 і поступово спадає в міру відхилення. Гауссова функція обрана замість лінійної бо вона дає плавну градацію – агент отримує хорошу винагороду не тільки при точному збігу але і при невеликому відхиленні, що відповідає ідеї «ширини каналу» в теорії потоку.

2.2 Побудова моделі профілювання гравця та оцінка емоційного стану

Якщо RL-агент це мозок системи, то модуль профілювання – це її органи чуття. Його задача перетворити сирі ігрові події – влучання, ухилення, смерті, використані предмети – на числовий вектор який агент може обробити.

Показник `skill_score` збирає три метрики які разом дають досить повне уявлення про технічний рівень гравця. Точність атак – це частка влучань серед усіх спроб атаки за останні 60 секунд. Ефективність ухилень – частка успішно відбитих або уникнутих атак. Економічність ресурсів – наскільки раціонально гравець використовує предмети відносно отриманої шкоди:

$$\text{skill_score} = 0.40 \cdot \text{accuracy} + 0.35 \cdot \text{evasion} + 0.25 \cdot \text{resource_eff.}$$

Ваги підібрані на основі експертних оцінок і відображають відносну важливість кожної складової. Точність і ухилення важливіші ніж економічність ресурсів – звідси і різниця у вагах.

Показник `stress_index` відображає рівень стресу через дві речі: як часто гравець помирає і наскільки хаотично він рухається. `Chaos_factor` вимірюється як стандартне відхилення кутів атак за вікно спостереження – якщо гравець б'є куди попало це добре видно по великій дисперсії кутів:

$$\text{stress_index} = \min(1.0, \text{death_rate}/\text{dr_max} + \text{chaos_factor}/\text{cf_max}).$$

Поріг `stress_index` більше 0.8 відповідає стану коли гравець явно втратив контроль над ситуацією – це тригер для класу `Frustration`.

Показник `flow_zone` бінарний і визначається одночасно через обидва попередні показники:

$$\text{flow_zone} = \text{TRUE} \Leftrightarrow |\text{skill_score} - \text{challenge_level}| \leq \varepsilon \text{ AND } \text{stress_index} \leq 0.8.$$

Для класифікації емоційного стану гравця використовується класифікатор `StateClassifier` який відносить поточний профіль до одного з трьох класів. `Flow` – гравець в балансі, `flow_zone = TRUE` і помірний стрес. `Frustration` – стрес зашкалює або складність сильно перевищує навички.

Boredom – навички сильно перевищують складність при низькому стресі, умови для класифікації вкладені в таблиці 2.2.

Таблиця 2.2 – Характеристики емоційних станів гравця

Стан	Умова активації	Реакція системи	Дії агента
Flow	$ \text{skill} - \text{challenge} \leq \varepsilon \text{ AND } \text{stress} \leq 0.8$	підтримати баланс	NULL_ACTION або мінімальне коригування
Frustration	$\text{stress} > 0.8 \text{ АБО } \text{skill} \ll \text{challenge}$	знижити виклик	a2, a3, a6 – полегшення умов
Boredom	$\text{skill} \gg \text{challenge} \text{ AND } \text{stress} < 0.2$	підвищити виклик	a1, a4, a5 – ускладнення умов

Класифікатор реалізований як k-NN з k рівним 3 і евклідовою метрикою у тривимірному просторі. Вибір k-NN обумовлений двома причинами: по-перше він дуже швидкий і не вимагає додаткових обчислювальних ресурсів, по-друге розробник може безпосередньо бачити в Debug Panel які прототипи стали найближчими – тобто розуміти логіку класифікації без розкриття «чорної скриньки».

Вихід класифікатора використовується двічі: як додатковий компонент вектора спостережень для агента, і як сигнал для Debug Panel де стан гравця відображається кольором – зелений це Flow, жовтий це Boredom, червоний це Frustration.

2.3 Алгоритмічне рішення для динамічного масштабування складності III (LoD-AI)

LoD-AI це той місток між рішеннями агента і реальними змінами в грі. Агент видає одне число від 0 до 1 – challenge_level, – а компонент ALoDAIController перекладає це число у конкретні зміни поведінки конкретного NPC. Підхід базується на розширенні класичної ідеї Level of

Detail з графіки на рівень штучного інтелекту, що дозволяє вирішити одразу дві проблеми: оптимізувати обчислювальне навантаження і управляти складністю [7].

Чотири рівні деталізації відрізняються між собою складністю логіки і частотою оновлення. LoD-0 при відстані менше 15 метрів – повне дерево поведінки з усіма можливостями, 30 Гц. LoD-1 від 15 до 40 метрів – базове переслідування і атака без тактики, 10 Гц. LoD-2 від 40 до 80 метрів – тільки патрулювання, 2 Гц. LoD-3 понад 80 метрів – NPC заморожений, оновлення не відбувається.

Ключова ідея в тому що порогові відстані не фіксовані, а масштабуються залежно від `challenge_level`:

$$r'_i = r_i \cdot (1 + 0.5 \cdot \text{challenge_level}).$$

При `challenge_level` рівному 1 відстані збільшуються в 1.5 рази – тобто більше NPC одночасно знаходиться на вищих рівнях деталізації з більш складною поведінкою. При `challenge_level` рівному 0 – навпаки, більшість NPC перемикається на спрощені режими.

Атрибути NPC теж масштабуються:

$$\text{HP_eff} = \text{HP_base} \cdot (0.7 + 0.6 \cdot \text{challenge_level}),$$

$$\text{DMG_eff} = \text{DMG_base} \cdot (0.6 + 0.8 \cdot \text{challenge_level}).$$

Це дає діапазон 70–130% від базового здоров'я і 60–140% від базової шкоди. Межі підібрані так щоб при плавному управлінні з кроком 0.05 зміни були в межах природного варіювання і не привертати уваги гравця.

Алгоритм оновлення рівнів виконується через Octree-просторовий індекс що дає складність $O(n \log n)$ замість $O(n^2)$ при наївному підході – це критично для сцен з сотнями активних NPC.

Алгоритм 2.1 – LoD-AI Update Procedure для кожного NPC у активній сцені (через Octree)

$$d = \text{distance}(\text{NPC.pos}, \text{Player.pos})$$

$$r_eff[i] = r_base[i] * (1 + 0.5 * \text{challenge_level})$$

```

new_lod = clamp(floor(d / r_eff), 0, 3)
if NPC.lod != new_lod:
    NPC.SwitchBehaviorTree(new_lod)
    NPC.SetTickRate(kTickRates[new_lod])
    NPC.ScaleAttributes(challenge_level)
    NPC.TickIfReady()

```

Переходи між рівнями залишаються непомітними для гравця якщо вони відбуваються достатньо далеко від центру його уваги [22]. Власне це і є основна причина чому LoD-AI обраний головним механізмом адаптації – він природний і не видає себе.

2.4 Проектування логіки взаємодії та інтерфейсу системи (UI/UX)

У системи є два принципово різних типи користувачів з різними потребами і різними контекстами використання. Перший – це гравець якому потрібен мінімалістичний HUD що не відволікає від гри і не розкриває механіки ШІ. Другий – це розробник або геймдизайнер якому потрібен максимально інформативний інструмент для спостереження і налаштування. Проектування обох інтерфейсів починалось з аналізу потреб цих двох типів користувачів.

2.4.1 Сценарії взаємодії та візуалізація прихованих параметрів адаптації

Гравець – вік від 15 до 45, будь-який рівень технічної підготовки. Він взаємодіє з грою в реальному часі через HUD і нічого не знає про те що всередині працює RL-агент. Головні вимоги: відгук UI не більше 16 мілісекунд, зміни складності без помітних стрибків, мінімум тексту на екрані, читабельність при будь-якому освітленні.

Розробник – технічний спеціаліст який одночасно використовує IDE і ігровий рушій. Йому потрібно розуміти що робить агент і швидко вносити правки. Головні вимоги: всі ключові метрики в одному місці в реальному часі, можливість змінювати параметри без перезапуску гри, захист від потрапляння в релізний білд, збереження даних сесій для аналізу.

2.4.2 Принципи побудови Debug-інтерфейсу для моніторингу ШІ

Перш ніж описувати конкретні елементи панелі, варто зрозуміти чому вона взагалі потрібна. Агент навчається офлайн у Python і в рантаймі просто виконує вже навчену модель – тобто сама панель на якість навчання не впливає ніяк. Але є інша проблема: коли навчена модель завантажена в рушій і починає приймати рішення в реальній грі, розробник бачить тільки результат – складність змінилась – але не причину. А причина може бути де завгодно: профайлер рахує skill неправильно через баг в обробці подій, reward function дає не той сигнал, або агент просто ще не навчився достатньо для конкретного сценарію поведінки гравця. Без інструменту який показує що відбувається всередині системи в реальному часі, знайти де саме щось пішло не так – це питання годин налагодження замість хвилин.

Друга проблема – ітеративність балансування. Налаштування адаптивної системи принципово відрізняється від налаштування звичайної гри: ефект від зміни параметра проявляється не одразу а через хвилини гри, бо кожне рішення агента залежить від накопиченої статистики. Якщо кожна така зміна вимагає зупинки і перезапуску – одна ітерація займає не хвилини а десятки хвилин. Hot reload вирішує саме це: можна змінити діапазон складності або стратегію NPC прямо під час сесії і одразу бачити реакцію агента. Поєднання цього з Q-значеннями на панелі дає ще один рівень діагностики – якщо Q-значення для INCREASE і DECREASE майже рівні, агент знаходиться в стані невизначеності і це сигнал що або вектор стану недостатньо інформативний для поточної ситуації, або варто перевірити що

профайлер дійсно передає коректні дані. Тобто Debug Panel це не частина готової гри і не частина навчання – це інструмент розробника який робить процес налагодження адаптивного ШІ керованим і зрозумілим.

Debug Panel виступає як UMG-оверлей і доступний тільки в dev-білдах – це забезпечується препроцесорною умовою `UE_BUILD_SHIPPING`. Панель організована у чотири вкладки за принципом Card Stack.

Вкладка `PLAYER PROFILE` показує поточний стан гравця в реальному часі. Вкладка `RL AGENT STATUS` – поточну нагороду зі знаком плюс або мінус і поточне значення `epsilon`. Вкладка `CONFIG PARAMS` – елементи керування для зміни параметрів без перезапуску. На панелі є лог подій з фільтрацією за типом через чекбокси і текстовим пошуком.

2.5 Альтернативні моделі обчислювального інтелекту для задачі DDA

Вибір конкретної моделі обчислювального інтелекту для задачі DDA не є очевидним і потребує порівняльного аналізу. Перш ніж перейти до обґрунтування вибору DQN як основного механізму управління складністю, доцільно розглянути два альтернативних підходи, які широко застосовуються в індустрії та академічних дослідженнях: системи на основі правил (rule-based контролери) та системи на основі нечіткої логіки (fuzzy logic controllers). Обидва підходи мають чітко описану математичну основу і реальну практику застосування в ігровій індустрії, що робить їх природними кандидатами для порівняння.

2.5.1 Rule-based контролер складності

Rule-based підхід є найстарішим і найпоширенішим методом реалізації DDA в комерційних іграх. Його ідея полягає у визначенні набору умов-правил виду «ЯКЩО ... ТО ...» (IF-THEN), які перевіряються в кожен

момент прийняття рішення. Незважаючи на простоту, такі системи функціонують у багатьох відомих іграх і дозволяють розробнику повністю контролювати логіку адаптації [8].

Формально rule-based контролер визначається як кінцева множина правил $R = \{r_1, r_2, \dots, r_n\}$, де кожне правило r_i має вигляд:

$$r_i: IF C_i(s) THEN a_i, i = 1, \dots, n \quad (2.3)$$

де $C_i(s)$ – булева умова над вектором стану s ;

$a_i \in A$ – відповідна дія.

Вектор стану s в контексті задачі DDA описаний у підрозділі 2.1 і включає компоненти `skill_score`, `stress_index`, `flow_zone` та інші поведінкові метрики гравця.

Механізм виконання контролера полягає у послідовній перевірці умов $C_i(s)$ для поточного стану s і виконанні першої дії a_i , умова якої є істинною. Якщо жодна умова не виконується, обирається дія за замовчуванням $a_0 = \text{NULL_ACTION}$.

Важливою характеристикою rule-based контролера є його детермінованість: для одного і того самого вектора стану s контролер завжди повертає одну і ту саму дію. Це суттєво спрощує тестування і налагодження – поведінку системи можна передбачити і перевірити аналітично без запуску симуляцій [9].

Проте rule-based підхід має принциповий недолік, який є критичним саме для задачі DDA у відкритому світі. Простір стану S у відкритому ігровому середовищі є неперервним і багатовимірним: вектор $s = [\text{skill_score}, \text{stress_index}, \text{flow_zone}, h, kd, \text{session_len}]$ визначає нескінченну кількість можливих конфігурацій. Априорне покриття цього простору кінцевим набором правил принципово неможливе: завжди знайдуться стани $s \notin \cup_i \{s : C_i(s) = \text{TRUE}\}$, в яких система не матиме обґрунтованої відповіді. Це явище відоме як проблема неповноти покриття (coverage gap) і є

основним аргументом проти rule-based підходу у динамічних нелінійних середовищах.

Додатково слід зазначити, що rule-based контролер є статичним: він не адаптується до індивідуальних особливостей конкретного гравця з часом. Два гравці з однаковим миттєвим значенням $\text{skill_score} = 0.6$ можуть мати принципово різний ігровий досвід і потребувати різних стратегій адаптації – але rule-based контролер не здатний це врахувати, оскільки не має механізму накопичення і використання історичної інформації.

2.5.2 Контролер на основі нечіткої логіки

Нечітка логіка (fuzzy logic), розроблена Лотфі Заде у 1965 році [26], є природним розширенням класичної булевої логіки на неперервний діапазон істинності $[0, 1]$. На відміну від rule-based контролера, де умова $C_i(s)$ є або повністю істинною, або повністю хибною, нечіткий контролер дозволяє частковий ступінь належності до кожного класу. Це принципово важливо для задачі DDA, оскільки поняття «гравець у стресі» або «гравець нудьгує» за своєю природою є розмитими категоріями без чітких меж.

Нечіткий контролер складності описується через три компоненти: лінгвістичні змінні, функції належності та нечіткі правила висновку. Лінгвістична змінна – це змінна, значення якої є словесним описом стану, наприклад змінна STRESS може приймати значення {«низький», «середній», «високий»}.

Для кожного такого значення задається функція належності $\mu: [0,1] \rightarrow [0,1]$, яка показує наскільки конкретне числове значення відповідає даному лінгвістичному терміну.

Для задачі управління складністю доцільно визначити нечіткі змінні на основі тих самих метрик, що використовуються у розробленій системі. Для змінної STRESS визначаються три трикутні функції належності:

$$\begin{aligned}
\mu_{lo}^W(x) &= \max(0, 1 - x / 0.3), \\
\mu_m^{ed}(x) &= \max(0, 1 - |x - 0.5| / 0.3), \\
\mu_h^{IG}(x) &= \max(0, (x - 0.6) / 0.4),
\end{aligned}
\tag{2.4}$$

де μ_{lo}^W , μ_m^{ed} , μ_h^{IG} – функції належності до термінів «низький», «середній» і «високий»;

$x \in [0, 1]$ – поточне значення stress_index.

На основі лінгвістичних змінних формуються нечіткі правила висновку. На відміну від rule-based контролера де правила є булевими, нечіткі правила мають ступінь спрацювання α_i , що обчислюється як мінімум ступенів належності для всіх умов правила (оператор AND Мамдані):

$$\alpha_i = \min(\mu_1(x_1), \mu_2(x_2), \dots, \mu_k(x_k)). \tag{2.5}$$

Наприклад, правило “ЯКЩО STRESS є HIGH ТА DELTA є NEGATIVE – ТО ЗМЕНШИТИ СКЛАДНІСТЬ” матиме ступінь спрацювання $\alpha = \min(\mu_h^{IG}(\text{stress_index}), \mu_n^{eG}(\text{delta}))$. Якщо stress_index = 0.85, то $\mu_h^{IG}(0.85) = 0.625$. Якщо delta = -0.25 (складність перевищує навички), то $\mu_n^{eG}(-0.25) = 1.0$. Тоді $\alpha = \min(0.625, 1.0) = 0.625$ – правило спрацьовує частково, з інтенсивністю 0.625.

Після обчислення ступенів спрацювання всіх правил виконується дефазифікація – перетворення нечіткого виводу у конкретне числове значення зміни складності. Найпоширенішим методом є метод центру мас (centroid defuzzification):

$$\Delta C = (\sum_i \alpha_i \cdot c_i) / (\sum_i \alpha_i), \tag{2.6}$$

де ΔC – результуюча зміна challenge_level;

c_i – центр нечіткого терміну вихідної змінної для правила i ;

α_i – ступінь спрацювання правила i .

Результат ΔC є неперервним числом, а не дискретним вибором з набору дій, що теоретично дає плавнішу адаптацію порівняно з rule-based підходом.

Нечітка логіка має суттєву перевагу: вона природно відображає розмитість самої теорії потоку. Канал потоку з умовою $|S(t) - C(t)| \leq \varepsilon$ формально визначає чітку межу, але психологічно ця межа є поступовою – немає різкого переходу між «гравець у потоці» і «гравець нудьгує». Нечіткий контролер моделює саме таку поступовість через часткові ступені спрацювання правил.

Разом з тим нечіткий контролер поділяє з rule-based підходом ключовий недолік: і функції належності, і самі правила задаються розробником апріорно і не оновлюються в процесі роботи системи. Параметри функцій належності (наприклад, центри і ширини трикутних функцій) потребують ретельного ручного калібрування і можуть виявитись невдалим для певних стилів гри. Крім того, зі зростанням числа вхідних змінних кількість правил зростає комбінаторно: при трьох термінах для кожної з трьох вхідних змінних базове число правил становить $3^3 = 27$, а при п'яти змінних – вже $3^5 = 243$ [10].

Важливо також те, що нечіткий контролер, як і rule-based, є реактивним: він реагує на поточний стан гравця, але не формує довгострокової стратегії адаптації. Відповідно він не може врахувати тренд розвитку навичок гравця – поступове зростання `skill_score` з часом, яке вимагає не просто реакції на поточне значення, а прогнозування майбутнього стану і завчасного коригування складності.

2.6 Порівняльний аналіз моделей та обґрунтування вибору DQN

Три розглянуті моделі – rule-based контролер, fuzzy logic контролер і Double DQN – представляють три різних парадигми управління: детерміновану на основі правил, детерміновану на основі нечіткого виводу

і навчальну на основі взаємодії з середовищем. Кожна з них реалізує різний компроміс між прозорістю, гнучкістю та вимогами до проектування. Для систематичного порівняння виділимо сім критеріїв, які є ключовими для задачі DDA у відкритому ігровому світі.

Перший критерій – покриття простору станів. Rule-based контролер покриває лише ті стани, для яких розробник явно визначив правило. Нечіткий контролер через інтерполяцію між правилами забезпечує ненульову відповідь у будь-якій точці простору станів, однак якість цієї відповіді у далеких від прототипів точках залишається під питанням. DQN апроксимує оптимальну функцію цінності $Q^*(s, a)$ у всьому просторі станів через нейронну мережу і не має зон «мовчання».

Другий критерій – адаптація до індивідуального гравця. Ані rule-based, ані fuzzy контролер не мають механізму «пам'яті»: вони реагують виключно на поточний вектор стану, не враховуючи того як змінювалась поведінка гравця протягом сесії. DQN через replay buffer накопичує досвід взаємодій і у процесі навчання формує стратегію, яка враховує довготривалі залежності між станами і винагородами. У виробничій системі це дозволяє агенту розпізнавати типові ігрові профілі і завчасно коригувати складність ще до того як стан гравця погіршиться.

Третій критерій – стійкість до нестаціонарності. Поведінка гравця принципово нестаціонарна: навички $S(t)$ зростають з часом, гравець навчається і змінює стиль гри. Rule-based і fuzzy контролери не мають механізму реагування на цей тренд – їх параметри фіксовані після проектування. DQN з дисконтуванням $\gamma = 0.97$, заданим у agent.py, явно моделює часову цінність майбутніх станів. Використання Double DQN замість стандартного DQN додатково усуває систематичне переоцінювання Q-значень, що є особливо важливим при нестаціонарному середовищі [24].

Четвертий критерій – прозорість і налагоджуваність. Тут rule-based контролер має очевидну перевагу: кожне рішення можна пояснити конкретним правилом. Нечіткий контролер дещо менш прозорий, але через

ступені спрацювання правил розробник може зрозуміти логіку виводу. DQN є класичною «чорною скринькою»: причину конкретного рішення пояснити аналітично неможливо. Саме ця властивість є основним мотивом для розробки Debug/Monitor Panel – без інструменту спостереження за Q-значеннями в реальному часі «чорна скринька» DQN стає практичною проблемою для розробника.

П'ятий критерій – обчислювальна вартість під час виконання. Rule-based контролер виконує просту послідовну перевірку умов – складність $O(n)$, де n – кількість правил. Нечіткий контролер вимагає обчислення функцій належності і дефазифікації – складність $O(n \cdot m)$, де m – кількість вхідних змінних. Інференс DQN на однопотоковому CPU для мережі з трьох лінійних шарів (128–128–8) займає порядку десятків мікросекунд – набагато менше порогу 16 мс і цілком прийнятно для запуску раз на секунду, як реалізовано в `UDDAGameSubsystem::RunInferenceStep`.

Шостий критерій – вимоги до проектування. Rule-based контролер вимагає повного ручного покриття простору станів – трудомісткий процес який не масштабується при збільшенні числа метрик. Нечіткий контролер вимагає ретельного підбору функцій належності – також ручна робота, хоча і більш стійка до прогалин. DQN переносить зусилля з проектування правил на проектування функції винагороди, що є суттєво компактнішою специфікацією: замість десятків правил потрібно одна функція `compute_reward`, яка реалізована у п'яти рядках коду у `agent.py`.

Сьомий критерій – сумісність з промисловим стеком. Rule-based і fuzzy контролери можуть бути реалізовані безпосередньо в C++ без зовнішніх залежностей. DQN вимагає фази навчання на Python з PyTorch, однак після експорту в ONNX виконується в Unreal Engine 5 через NNE Plugin без будь-яких Python-залежностей у рантаймі.

Rule-based контролер є прийнятним рішенням для лінійних ігор з обмеженим і передбачуваним простором станів, однак для відкритого світу з комбінаторно великим простором можливих сценаріїв він є принципово

недостатнім. Fuzzy logic контролер є кроком вперед завдяки інтерполяції і природному представленню розмитих понять теорії потоку, але зберігає фундаментальний недолік – статичність параметрів і відсутність довгострокової адаптації. Double DQN є єдиною з трьох моделей, яка одночасно забезпечує повне покриття простору станів, механізм адаптації до індивідуального гравця і стійкість до нестационарності середовища – ціною зниження прозорості, яка компенсується Debug/Monitor Panel.

Таким чином, вибір Double DQN як основної моделі обчислювального інтелекту є обґрунтованим виходячи з вимог задачі, а не є довільним. Rule-based і fuzzy контролери розглядаються як базові альтернативи, порівняння з якими дозволяє кількісно і якісно оцінити переваги обраного підходу. Обидва альтернативних підходи зберігають практичну цінність у певних умовах: rule-based – у прототипуванні і тестуванні де передбачуваність важливіша за оптимальність, fuzzy – у сценаріях де кількість метрик невелика і є чітке розуміння лінгвістичних термінів для їх опису.

Слід також відзначити, що три описані моделі не є взаємовиключними – їх можна поєднувати в гібридних архітектурах та ці напрями є перспективними для подальшого розвитку системи.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ТА ІГРОВОГО ПРОТОТИПУ

3.1 Загальна архітектура системи та вибір інструментарію розробки

Перед тим як переходити до деталей реалізації варто зазначити, що повний простір дій агента який описаний в розділі 2 в поточній реалізації спрощений до одного узагальненого параметра `challenge_level` з різними кроками зміни. Управління спавном ворогів, ресурсними дропами і точністю NPC як окремими діями залишено для подальшого розширення. Те що реалізовано повністю функціонує і демонструє всі ключові механізми системи – однак не в повному обсязі.

Поточна реалізація є функціональним прототипом, що демонструє архітектурні рішення та взаємодію компонентів системи. Навчання агента виконано на спрощеному симульованому середовищі з обмеженим покриттям простору станів. Повноцінне навчання потребує середовища симуляції що покриває всі три емоційні стани гравця та тривалих ігрових сесій для накопичення репрезентативного досвіду.

Система розбита на два технологічні шари. Перший шар – навчання агента – реалізований на Python з PyTorch і NumPy і працює офлайн. Другий шар – ігровий рантайм – реалізований на C++ в Unreal Engine 5 і не має жодних Python-залежностей. Комунікація між шарами відбувається через формат ONNX: навчена модель зберігається в цьому форматі і завантажується в UE5 через NNE Plugin.

Така архітектура вирішує класичну проблему несумісності між середовищем машинного навчання і промисловим ігровим рушієм: Python потрібен тільки на етапі навчання, а в готовій грі він взагалі не потрібен.

C++-рівень складається з чотирьох компонентів. `UPlayerProfilerComponent` збирає і агрегує поведінкові метрики гравця. `UDDAGameSubsystem` є центральним оркестратором – завантажує модель,

запускає інференс і передає `challenge_level`. `ALoDAIController` реалізує LoD-логіку і масштабування атрибутів для кожного NPC. `UDebugPanelWidget` надає розробнику інтерактивний оверлей.

Вибір UE5 обумовлений вбудованими Behavior Trees, Mass Entity Framework, Significance Manager, NNE Plugin і UMG – все що потрібно для реалізації системи є вже в рушії.

3.2 Реалізація модуля профілювання та класифікатора станів гравця

`UPlayerProfilerComponent` прикріплюється до персонажа гравця при спауні і постійно збирає ігрові події через метод `PushEvent`. Після кожної нової події він видаляє застарілі записи – старші за 60 секунд – і перераховує всі три показники через внутрішній метод `Recalc`.

Ковзне вікно в 60 секунд є принциповим архітектурним рішенням. Якщо робити вікно коротшим система стає занадто нервовою і реагує на кожну випадкову помилку. Якщо довшим – занадто інертною і не встигає реагувати на реальні зміни в грі.

Найцікавіше місце в `Recalc` – це обчислення `chaos_factor` (лістинг 3.1). Замість того щоб зберігати всі кути атак і рахувати дисперсію в кінці, компонент рахує її покроково через рекурентну формулу.

Лістинг 3.1 – Обчислення `chaos_factor` у `Recalc` (фрагмент, `PlayerProfilerComponent.cpp`)

```
float MeanAngle = 0.f;
for (const FRawGameEvent& E : EventWindow) MeanAngle +=
E.AimAngle;
MeanAngle /= N;
float Variance = 0.f;
for (const FRawGameEvent& E : EventWindow)
    Variance += (E.AimAngle - MeanAngle) * (E.AimAngle -
MeanAngle);
const float ChaosFactor = N > 1 ? FMath::Sqrt(Variance / N)
: 0.f;
```

Класифікатор k-NN реалізований прямо в методі `GetPlayerState`. Прототипи зберігаються як статичний масив структур `FProto` – це означає що вони живуть в секції `.rodata` і не можуть бути випадково зібрані збирачем сміття. Для кожного виклику рахуються відстані до всіх 15 прототипів, три найближчих дають голоси, клас з найбільшою кількістю голосів і є результатом.

Python-модуль `profiler.py` це по суті той самий компонент але на Python – використовується тільки під час навчання агента. Обидві реалізації мають однакові константи і однакові формули, що гарантує що вектор стану під час навчання і під час гри означає одне і те ж.

3.3 Програмна реалізація RL-агента та формування функції винагороди

Агент реалізований в модулі `agent.py` як Double DQN з двома мережами – основною і цільовою. Архітектура `QNetwork` проста: три повнозв'язних шари з ReLU між ними. Нічого складнішого тут не потрібно – простір стану невеликий, шість чисел, і складна архітектура тут тільки заважала б.

Головна деталь реалізації – метод `_train_once` де і реалізована ідея Double DQN (лістинг 3.2). Основна мережа вибирає яка дія найкраща для наступного стану, а цільова мережа оцінює цю дію. Це усуває систематичне переоцінювання Q-значень яке виникає в звичайному DQN і особливо шкодить при нестационарному середовищі.

Лістинг 3.2 – Double DQN update у `_train_once` (фрагмент, `agent.py`)

```
with torch.no_grad():
    best_a = self.q(sn).argmax(1, keepdim=True)
    q_tgt_v = self.q_tgt(sn).gather(1, best_a).squeeze(1)
    y = r + self.GAMMA * q_tgt_v * (1 - done)

q_pred = self.q(s).gather(1, a.unsqueeze(1)).squeeze(1)
loss = nn.functional.huber_loss(q_pred, y)
```

Функція винагороди `compute_reward` реалізує теорію потоку в коді. Основний компонент `r_flow` – гауссова функція від різниці між `skill` і `challenge`. Гауссова а не лінійна бо лінійна давала б нагороду 0 при будь-якому відхиленні від ідеалу, а гауссова плавно спадає – агент отримує хорошу нагороду навіть при невеликому відхиленні в межах каналу потоку.

Штрафний компонент `r_rep` карає за різкі стрибки `challenge_level`. Без нього агент міг би різко підіймати і опускати складність що гравець точно б помітив.

Навчання запускається через `test_agent.py` який симулює 2000 кроків і потім зберігає модель у ONNX через `torch.onnx.export` з `opset_version=13` – саме ця версія підтримується NNE Plugin в UE5.

3.4 Розробка ігрового прототипу у відкритому світі

Прототип побудований на двох C++ класах. `ADDABaseNPC` – базовий клас для всіх ворожих NPC. Він зберігає дві пари атрибутів: базові значення `BaseHealth` і `BaseDamage` які задаються в редакторі, і поточні масштабовані значення `CurrentHealth` і `CurrentDamage` якими керує контролер.

`ALoDAIController` успадковує від `AAIController` і додає до нього дві речі: автоматичне перемикавання між рівнями деталізації і масштабування атрибутів NPC при зміні `challenge_level`.

Найцікавіше рішення в `UpdateLoDLevel` – порогові відстані `L0`, `L1`, `L2` не константи а масштабуються через `Scale` залежно від `challenge_level` (лістинг 3.3). Тобто агент не просто перемикає якийсь перемикач – він реально змінює те наскільки далеко від гравця NPC залишається «небезпечним».

Лістинг 3.3 – Динамічне масштабування порогів LoD (фрагмент, `LoDAIController.cpp`)

```
const float Scale = 1.f + 0.5f * ChallengeLvl;
ELoDLevel NewLoD;
```

Продовження лістингу 3.3

```

        if (Dist < LoDThreshold::L0 * Scale) NewLoD =
        ELoDLevel::LoD0;
        else if (Dist < LoDThreshold::L1 * Scale) NewLoD =
        ELoDLevel::LoD1;
        else if (Dist < LoDThreshold::L2 * Scale) NewLoD =
        ELoDLevel::LoD2;
        else NewLoD =
        ELoDLevel::LoD3;

```

В `ApplyLoDTransition` для `LoD3` викликається `BrainComponent->StopLogic` що заморожує NPC не знищуючи об'єкт. Це важливо для продуктивності коли на сцені багато далеких ворогів. Для рівнів `LoD0–LoD2` активується відповідне дерево поведінки з масиву `BehaviorTrees` де індекс відповідає числовому значенню `enum` – завдяки цьому не потрібен `switch`-оператор.

В `ScaleNPCAttributes` масштабування застосовується до `BaseHealth` і `BaseDamage` а не до поточних значень. Це принципово – інакше кожен наступний виклик множив би вже помножений результат і характеристики поповзли б в непередбачуваному напрямку.

3.5 Реалізація UI-компонентів та системи зворотного зв'язку

`UDebugPanelWidget` успадковує від `UUserWidget` і весь його код загорнутий в `#if !UE_BUILD_SHIPPING / #endif`. Це найнадійніший спосіб гарантувати що налагоджувальний код не потрапить у реліз навіть якщо про це забудуть.

Підписка на події агента відбувається в `NativeConstruct` (лістинг 3.4).

Лістинг 3.4 – Підписка `Debug Panel` на подію зміни складності (`DebugPanelWidget.cpp`)

```

void UDebugPanelWidget::NativeConstruct()
{
    Super::NativeConstruct();
    #if !UE_BUILD_SHIPPING

```

Продовження лістингу 3.4

```

        auto*          Sub          =          GetGameInstance() -
>GetSubsystem<UDDAGameSubsystem>();
        if (Sub) Sub->OnChallengeUpdated.AddDynamic(
            this, &UDebugPanelWidget::OnChallengeUpdated);
    #endif
}

```

ApplyConfig реалізує hot reload в чотири кроки: зчитати значення слайдерів, перевірити що DiffMin менше DiffMax, серіалізувати конфіг у JSON через FJsonObjectConverter, записати у файл і передати в підсистему. Якщо будь-який крок не вдається – операція скасовується повністю без часткового застосування.

CSV-експорт в ExportSessionCSV зберігає файл у директорію Saved/DDA/ з міткою часу в імені. Без мітки часу кожна нова сесія перезаписувала б попередню.

AppendSessionEntry розв'язує UI від деталей реалізації профайлера: панель не знає нічого про EventWindow чи алгоритм Recalc – вона отримує готову структуру FSessionEntry і просто відображає її значення.

3.6 Інтеграція компонентів у єдиний ігровий цикл

UDDAGameSubsystem це місце де всі компоненти системи збираються разом. Вибір GameInstanceSubsystem як базового класу не випадковий: на відміну від Actor або Component підсистема живе весь час поки існує GameInstance – тобто вона переживає завантаження рівнів і переходи між локаціями. В умовах відкритого світу де гравець може переходити між зонами це критично для накопичення статистики.

Таймер інференсу запускається раз на секунду. Це свідоме рішення: частіше не потрібно бо зміна challenge_level на 0.05 раз на секунду вже є межею непомітності для гравця, а рідше – система втрачає реактивність.

В RunInferenceStep весь цикл прийняття рішення відбувається в одному місці (лістинг 3.5).

Лістинг 3.5 – Цикл інференсу агента (фрагмент, DDAGameSubsystem.cpp)

```
TArray<float> State = ProfilerComp->GetStateVector();
// ...
ModelInstance->RunSync(Inputs, Outputs);

int32 Best = 0;
for (int32 i = 1; i < 8; ++i)
    if (QValues[i] > QValues[Best]) Best = i;

const float Delta = FMath::Clamp(
    ActionToChallengeDelta(Best), -0.05f, 0.05f);
CurrentChallenge = FMath::Clamp(
    CurrentChallenge + Delta, MinDifficulty,
    MaxDifficulty);
```

Після цього нове значення `challenge_level` розповсюджується через `TActorIterator` по всіх `ALoDAIController` на рівні. Для прототипу це нормальне рішення, але для повноцінної гри з сотнями NPC варто розглянути підхід на основі подій або кешованого списку.

`OnChallengeUpdated` після кожного кроку сповіщає `Debug Panel` передаючи нове `challenge_level`, індекс обраної дії і весь масив Q-значень. Завдяки цьому розробник бачить не тільки результат рішення але і логіку яка до нього привела.

В цілому ігровий цикл виглядає так: щосекунди підсистема зчитує стан гравця, запускає інференс, розраховує нове `challenge_level` і передає його всім контролерам. Кожен контролер оновлює рівень LoD і атрибути свого NPC. `Debug Panel` відображає все це в реальному часі. Гравець взаємодіє зі зміненими NPC і генерує нові події – і цикл починається знову.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи вирішено задачу адаптивного управління складністю в ігровому середовищі з відкритим світом на основі навчання з підкріплення. Задача формалізована у термінах марковського процесу прийняття рішень що дозволило застосувати DQN-агента без необхідності явно знати функцію переходів середовища.

Побудовано модель профілювання гравця на основі трьох поведінкових показників – майстерності, стресу і зони потоку – і k-NN класифікатор що відносить поточний стан до одного з трьох емоційних класів. Завдяки цьому теорія потоку Csíkszentmihályi з абстрактної психологічної концепції перетворилась на конкретну технічну реалізацію.

Розроблено алгоритм LoD-AI що управляє складністю NPC через чотири рівні деталізації поведінки. Масштабування порогових відстаней разом з `challenge_level` дає плавне і непомітне для гравця підсилення або послаблення тиску з боку ворогів.

Реалізовано повний технічний стек: офлайн-навчання агента на Python з PyTorch, збереження моделі у ONNX і запуск в Unreal Engine 5 через NNE Plugin без Python-залежностей в рантаймі. Архітектура з чотирьох слабо зв'язаних C++-компонентів дозволяє підключити систему до будь-якого UE5-проекту з мінімальними змінами.

Debug/Monitor Panel з hot reload і CSV-експортом закриває практичну проблему яка є у всіх відомих комерційних DDA-системах – відсутність зручного інструменту для спостереження за тим що відбувається всередині адаптивного ІІІ в реальному часі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Csikszentmihalyi M. Flow: the psychology of optimal experience. New York : Harper & Row, 1990. 303 p.
2. Yannakakis G. N., Togelius J. Artificial intelligence and games. 2nd ed. New York : Springer, 2024. URL: <https://gameaibook.org/wp-content/uploads/2024/08/book2.pdf> (date of access: 07.06.2026).
3. Adams E. Fundamentals of game design. 3rd ed. Berkeley : New Riders, 2014. URL: https://www.academia.edu/81247135/Fundamentals_of_game_design (date of access: 07.06.2026).
4. Epic Games. Unreal Engine 5 documentation: world partition. Epic Games, 2023. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/world-partition-in-unreal-engine> (date of access: 07.06.2026).
5. Shaker N., Togelius J., Nelson M. J. Procedural content generation in games. Cham : Springer International Publishing, 2016. URL: <https://doi.org/10.1007/978-3-319-42716-4> (date of access: 07.06.2026).
6. Salen K., Zimmerman E. Rules of play: game design fundamentals. Cambridge : MIT Press, 2004. URL: http://files.onearmedman.com/fordham/salen_zimmerman2004.pdf (date of access: 07.06.2026).
7. Epic Games. Significance manager in unreal engine. Unreal Engine Documentation, 2023. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/significance-manager-in-unreal-engine> (date of access: 07.06.2026).
8. Yannakakis G. N., Hallam J. Game and player feature selection for entertainment capture. *2007 IEEE symposium on computational intelligence and games*, Honolulu, HI, USA, 1–5 April 2007. 2007. URL: <https://doi.org/10.1109/cig.2007.368105> (date of access: 07.06.2026).

9. Zook A., Riedl M. A temporal data-driven player model for dynamic difficulty adjustment. *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment*. 2021. Vol. 8, no. 1. P. 93–98. URL: <https://doi.org/10.1609/aiide.v8i1.12504> (date of access: 07.06.2026).
10. Hunicke R., Chapman V. AI for Dynamic Difficulty Adjustment in Games. *Proceedings of the AIIDE Workshop on Challenges in Game AI*. 2004. URL: <https://users.cs.northwestern.edu/~hunicke/pubs/Hamlet.pdf> (date of access: 07.06.2026).
11. Dynamic difficulty adjustment for maximized engagement in digital games / S. Xue et al. *The 26th international conference*, Perth, Australia, 3–7 April 2017. New York, New York, USA, 2017. URL: <https://doi.org/10.1145/3041021.3054170> (date of access: 07.06.2026).
12. Csíkszentmihályi M. *Beyond boredom and anxiety*. San Francisco : Jossey-Bass Publishers, 1975. 231 p. URL: <https://archive.org/details/beyondboredomanx0000csik> (date of access: 07.06.2026).
13. Csikszentmihalyi M., LeFevre J. Optimal experience in work and leisure. *Journal of personality and social psychology*. 1989. Vol. 56, no. 5. P. 815–822. URL: <https://doi.org/10.1037/0022-3514.56.5.815> (date of access: 07.06.2026).
14. Sweetser P., Wyeth P. GameFlow. *Computers in entertainment*. 2005. Vol. 3, no. 3. P. 3. URL: <https://doi.org/10.1145/1077246.1077253> (date of access: 07.06.2026).
15. Dynamic game balancing: an evaluation of user satisfaction / G. Andrade et al. *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment*. 2021. Vol. 2, no. 1. P. 3–8. URL: <https://doi.org/10.1609/aiide.v2i1.18739> (date of access: 07.06.2026).
16. Zohaib M. Dynamic difficulty adjustment (DDA) in computer games: a review. *Advances in Human-Computer Interaction*. 2018. Vol. 2018.

- P. 1–12. URL: <https://doi.org/10.1155/2018/5681652> (date of access: 07.06.2026).
17. Przybylski A. K., Rigby C. S., Ryan R. M. A motivational model of video game engagement. *Review of general psychology*. 2010. Vol. 14, no. 2. P. 154–166. URL: <https://doi.org/10.1037/a0019440> (date of access: 07.06.2026).
18. Lopes R., Bidarra R. Adaptivity challenges in games and simulations: a survey. *IEEE transactions on computational intelligence and AI in games*. 2011. Vol. 3, no. 2. P. 85–99. URL: <https://doi.org/10.1109/tciaig.2011.2152841> (date of access: 07.06.2026).
19. Hunicke R. The case for dynamic difficulty adjustment in games. *The 2005 ACM SIGCHI international conference*, Valencia, Spain, 15–17 June 2005. New York, New York, USA, 2005. URL: <https://doi.org/10.1145/1178477.1178573> (date of access: 07.06.2026).
20. Millington I., Funge J. Artificial intelligence for games. 2nd ed. Burlington : Morgan Kaufmann, 2009. URL: http://www.matt-versaggi.com/mit_open_courseware/GameAI/ArtificialIntelligenceforGamesSecondEdition.pdf (date of access: 07.06.2026).
21. Isla D. Handling complexity in the Halo 2 AI. Proceedings of GDC. San Francisco, 2005. URL: <https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai> (date of access: 07.06.2026).
22. Champandard A. J. Behavior trees for next-gen game AI. Proceedings of GDC. 2007. URL: <https://gdcvault.com/play/1015317> (date of access: 07.06.2026).
23. Bellman R. Dynamic programming. Princeton : Princeton University Press, 1957. URL: <https://archive.org/details/dynamicprogrammi0000bell> (date of access: 07.06.2026).

24. Sutton R. S., Barto A. G. Reinforcement learning: an introduction. 2nd ed. Cambridge : MIT Press, 2018. URL: <http://incompleteideas.net/book/the-book-2nd.html> (date of access: 07.06.2026).
25. Watkins C. J. C. H. Learning from delayed rewards : PhD thesis / Cambridge University. Cambridge, 1989. URL: https://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf (date of access: 07.06.2026).
26. Zadeh L. A. Fuzzy sets. *Information and control*. 1965. Vol. 8, no. 3. P. 338–353. URL: [https://doi.org/10.1016/s0019-9958\(65\)90241-x](https://doi.org/10.1016/s0019-9958(65)90241-x) (date of access: 07.06.2026).