ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Інтелектуальна система симуляції поведінки ігрових агентів у реальному часі в ігровому рушії Unity»

на здобуття освітнього ступеня магістр
за спеціальності Інформаційні системи та технології
(код,найменування спеціальності)
освітньо-професійної програми 126 Інформаційні системи та технології
(назва)

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

_____ Іван Гойна
(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти
група ІСДМ-61
(ім'я, ПРІЗВИЩЕ)
Керівник Ольга Полоневич
к.т.н. *(ім'я, ПРІЗВИЩЕ)*
Рецензент: _____
(ім'я, ПРІЗВИЩЕ)

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційні системи та технології
Ступінь вищої освіти магістерський
Спеціальність Інформаційні системи та технології
Освітньо-професійна програма 126 Інформаційні системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедрою ІСТ

_____

“____” _____2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гойна Іван Сергійович
(прізвище, ім’я, по батькові здобувача)

1.Тема кваліфікаційної роботи: Інтелектуальна система симуляції поведінки ігрових агентів у реальному часі в ігровому рушії Unity
керівник кваліфікаційної роботи:Полоневич О.В. кандидат технічних наук, доцент
(ім’я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від “___” жовтня 2025 р. № ____

2. Строк подання кваліфікаційної роботи « » грудня 2025 р.

3. Вихідні дані кваліфікаційної роботи:

1. **Платформа розробки:** ігровий рушій Unity.
2. **Мова програмування:** C#.
3. **Ключові технології:** система навігації Unity NavMesh для побудови маршрутів та обходу перешкод.
4. **Типи агентів:** автономні NPC (Non-Player Characters) з ролями "охоронець" (патрульний) та "гравець".
5. **Методи моделювання:** архітектура скінченних автоматів (Finite State Machine — FSM) для керування станами поведінки.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно

розробити):

1. **Теоретичні основи ігрового ШІ:** аналіз понять і класифікація ігрових агентів (пасивні, реактивні, когнітивні).
2. **Огляд методів моделювання поведінки:** порівняння скінченних автоматів (FSM), дерев поведінки (BT) та планування дій (GOAP).
3. **Проєктування архітектури системи:** визначення вимог до системи симуляції, вибір алгоритмів прийняття рішень та опис структури програмних модулів.
4. **Практична реалізація в Unity:** створення інтерактивної сцени, налаштування навігаційної сітки NavMesh та написання C#-скриптів для керування логікою агентів.
5. **Тестування та аналіз:** проведення експериментальних запусків (патрулювання, виявлення, переслідування), оцінка продуктивності системи та аналіз отриманих результатів.

5.Перелік ілюстраційного матеріалу: *презентація*, *графічні схеми (структура ігрового агента, діаграми станів)*.

6. Дата видачі завдання «___» _____ 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Огляд сучасних методів моделювання ШІ	Тиждень 1-2	
2	Обґрунтування вибору Unity та NavMesh як основного інструментарію	Тиждень 2	
3	Розробка концептуальної схеми взаємодії ігрових агентів	Тиждень 3	
4	Налаштування навігаційних поверхонь та запікання (Baking) навігаційної сітки	Тиждень 3	
5	Реалізація скриптів C# для контролера ворога (EnemyAI)	Тиждень 4	
6	Створення системи сенсорів (Vision/Hearing) для детекції гравця	Тиждень 4	
8	Проведення серії тестів на швидкість реакції та коректність маршрутів	Тиждень 5	
9	Оформлення результатів дослідження та написання звіту	Тиждень 6	

Здобувач вищої освіти _____ _____
(підпис) *(ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ _____

(підпис) *(ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Кваліфікаційна робота присвячена розробленню інтелектуальної системи симуляції поведінки ігрових агентів у реальному часі в середовищі Unity з використанням мови програмування C#. Об’єктом дослідження є процес моделювання поведінки ігрових агентів у тривимірному інтерактивному середовищі, а предметом – методи та програмні засоби реалізації навігації, прийняття рішень і взаємодії агентів із середовищем за допомогою NavMesh і C#-скриптів.

Метою роботи є розроблення та експериментальна перевірка інструменту симуляції поведінки ігрових агентів у реальному часі, що забезпечує патрулювання, переслідування гравця та адаптацію до змін оточення. Для досягнення поставленої мети проаналізовано теоретичні основи ігрового штучного інтелекту, підходи до моделювання поведінки агентів і можливості рушія Unity, спроєктовано архітектуру інтелектуальної системи, реалізовано програмні модулі мовою C# та інтегровано їх із системою навігації NavMesh. Проведено тестування симуляції й оцінено ефективність розробленого інструменту.

Методологічну основу роботи становлять аналіз наукових і технічних джерел, методи об’єктно орієнтованого проєктування, моделювання та експериментальні дослідження в середовищі Unity. Практична значущість роботи полягає у можливості використання розробленого інструменту для навчальних цілей, прототипування ігрових проєктів і подальших досліджень у сфері ігрового штучного інтелекту.

Ключові слова: ІНТЕЛЕКТУАЛЬНА СИСТЕМА, ІГРОВІ АГЕНТИ, UNITY, C#, НАВІГАЦІЯ, NAVMESH, ШТУЧНИЙ ІНТЕЛЕКТ, СИМУЛЯЦІЯ, РЕАЛЬНИЙ ЧАС.

ABSTRACT

The qualification thesis focuses on the development of an intelligent system for real-time simulation of game agent behavior using the Unity game engine and the C# programming language. The object of the study is the process of modeling game agent behavior in a three-dimensional interactive environment, while the subject includes methods and software tools for implementing navigation, decision-making, and agent–environment interaction using NavMesh and C# scripts.

The aim of the thesis is to develop and experimentally evaluate a real-time simulation tool that enables patrolling, player pursuit, and adaptation to environmental changes. To achieve this goal, the theoretical foundations of game artificial intelligence, behavior modeling approaches, and the capabilities of the Unity engine were analyzed. An intelligent system architecture was designed, software modules were implemented in C#, and their integration with the NavMesh navigation system was performed. The simulation was tested and the effectiveness of the developed tool was assessed.

The methodological basis of the research includes analysis of scientific and technical sources, object-oriented software design methods, modeling and simulation tools within Unity, and experimental evaluation of agent behavior. The practical significance of the work lies in the applicability of the developed system for educational purposes, game prototyping, and further research in the field of game artificial intelligence.

Keywords: INTELLIGENT SYSTEM, GAME AGENTS, UNITY, C#, NAVIGATION, NAVMESH, ARTIFICIAL INTELLIGENCE, SIMULATION, REAL-TIME.

ЗМІСТ

ВСТУП

Розвиток індустрії комп’ютерних ігор і віртуальних симуляторів зумовлює зростання вимог до якості поведінки ігрових агентів. Користувачі очікують від ігрового середовища не лише привабливої графіки, а й правдоподібної, узгодженої з контекстом поведінки персонажів, здатності реагувати на дії гравця в режимі реального часу, адаптуватися до змін середовища, приймати рішення в умовах невизначеності. У цих умовах особливої актуальності набувають інтелектуальні системи, що забезпечують моделювання поведінки агентів на основі формалізованих моделей, алгоритмів прийняття рішень та механізмів навігації.

Сучасні ігрові рушії, зокрема Unity, надають розробникам широкий спектр вбудованих засобів для реалізації штучного інтелекту. Це, з одного боку, спрощує створення прототипів, а з іншого – висуває завдання методично правильного використання наявного інструментарію, поєднання стандартних компонентів зі спеціалізованими програмними модулями та адаптації відомих алгоритмів до конкретних сценаріїв гри чи симуляції. Попри значну кількість комерційних ігор із розвиненими системами ШІ, в освітній і науково-прикладній літературі бракує завершених, добре задокументованих прикладів побудови цілісного інструменту для моделювання поведінки агентів у середовищі Unity, який був би орієнтований саме на навчальні цілі та подальше розширення.

Актуальність теми кваліфікаційної роботи зумовлена необхідністю створення доступного, модульного інструменту для симуляції поведінки ігрових агентів у реальному часі, який поєднував би сучасні підходи до навігації, прийняття рішень і взаємодії з оточенням із простотою інтеграції в проєкти на базі Unity. Для України така тематика є важливою з огляду на динамічний розвиток ІТ-галузі, зростання попиту на фахівців з розроблення ігор і симуляцій, а також необхідність впровадження практико орієнтованих навчальних матеріалів у підготовку програмістів та інженерів з інформаційних технологій.

У вітчизняних та зарубіжних дослідженнях значну увагу приділено питанням штучного інтелекту в комп’ютерних іграх, класифікації ігрових агентів, застосуванню алгоритмів пошуку шляху, поведінкових дерев, систем станів та підкріплювального навчання. Окремі роботи розглядають можливості рушія Unity і його системи NavMesh для організації навігації, існують також сторонні бібліотеки та плагіни для розширення стандартного функціоналу. Водночас більшість публікацій або орієнтовані на окремі аспекти, або пов’язані з великими комерційними проєктами і не демонструють покрокову побудову навчальної інтелектуальної системи з чіткою архітектурою, простими інтерфейсами налаштування й можливістю подальшої модернізації.

Нерозв’язаною залишається частина загальної проблеми, пов’язана з побудовою компактного, але повноцінного інструменту, який на прикладі типової сцени Unity показує повний цикл: від теоретичного обґрунтування вибору моделей поведінки та алгоритмів до архітектурного проєктування, програмної реалізації на C#, інтеграції в рушій та аналізу отриманих результатів. Саме цій частині проблеми присвячено дану кваліфікаційну роботу.

Метою дослідження є розроблення та дослідження інтелектуальної системи симуляції поведінки ігрових агентів у реальному часі на базі ігрового рушія Unity з використанням мови програмування C#.

Для досягнення поставленої мети в процесі дослідження вирішувалися такі основні *завдання*. Було проаналізовано теоретичні основи побудови інтелектуальних систем у комп’ютерних іграх, класифіковано типи ігрових агентів і підходи до їх моделювання. Далі виконано огляд алгоритмів прийняття рішень і навігації в реальному часі, можливостей сучасних ігрових рушіїв, зокрема Unity, а також особливостей використання мови C# для реалізації поведінки агентів. На основі проведеного аналізу спроєктовано архітектуру інтелектуальної системи, визначено ролі основних компонентів та обґрунтовано вибір алгоритмів. Наступним етапом стало моделювання поведінки агентів у сцені Unity, розроблення програмних модулів мовою C#, інтеграція їх із системою навігації

NavMesh та налаштування середовища. Завершальним етапом було проведення тестування симуляції, аналіз поведінки агентів за різних сценаріїв, а також формулювання висновків щодо ефективності запропонованого інструменту та перспектив його розвитку.

Об'єктом дослідження є процес моделювання та симуляції поведінки ігрових агентів у реальному часі в тривимірному віртуальному середовищі.

Предметом дослідження є методи, програмні засоби та архітектурні рішення побудови інтелектуальної системи симуляції поведінки агентів у рушії Unity з використанням засобів навігації NavMesh та скриптів мовою C#.

У роботі застосовано комплекс *методів дослідження*. Теоретичний аналіз і узагальнення літературних джерел використано для вивчення існуючих підходів до моделювання поведінки ігрових агентів та можливостей ігрових рушіїв. Методи структурного й об'єктно орієнтованого проєктування застосовано для побудови архітектури системи. Програмна реалізація виконана з використанням мови C# та середовища Unity, що спирається на методи моделювання й симуляції, а також на експериментальні дослідження поведінки агентів під час тестування в режимі реального часу.

Наукова новизна одержаних результатів полягає у поєднанні теоретичних підходів до класифікації й моделювання ігрових агентів із практичною реалізацією модульного інструменту симуляції на базі Unity. У роботі запропоновано структуровану архітектуру інтелектуальної системи, що забезпечує чіткий поділ між рівнем моделювання поведінки та рівнем реалізації в рушії. Удосконалено методичний підхід до побудови навчальних симуляцій: створено набір універсальних C#-модулів для керування переміщенням гравця та патрулюванням охоронних агентів із можливістю параметричного налаштування сценаріїв. Набули подальшого розвитку практичні рекомендації щодо використання системи Unity AI Navigation для реалізації простих, але надійних моделей поведінки агентів.

Практична значущість роботи полягає в тому, що розроблений інструмент може бути використаний як основа для створення навчальних і демонстраційних проєктів, а також як стартова платформа для розширених досліджень у галузі ігрового штучного інтелекту. Створена сцена Unity з реалізованою системою патрулювання, переслідування та взаємодії з гравцем може бути інтегрована до лабораторних робіт, використана для відпрацювання навичок програмування на C# в ігровому середовищі й слугувати прототипом для більш складних ігор чи симуляцій.

Апробація результатів дослідження здійснювалася в процесі розроблення і тестування програмного проєкту в середовищі Unity, а також під час обговорення проміжних результатів у межах навчального процесу. Отримані рішення можуть бути рекомендовані для використання у навчальних дисциплінах, пов’язаних із розробленням комп’ютерних ігор, моделюванням систем та програмуванням мовою C#.

Структурно кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ У КОМП'ЮТЕРНИХ ІГРАХ

1.1. Поняття та класифікація ігрових агентів

Для побудови будь-якої інтелектуальної системи необхідно спершу чітко визначити її базовий компонент - агента. У класичній теорії штучного інтелекту, інтелектуальний агент (ІА) визначається як автономна сутність, яка сприймає своє оточення за допомогою сенсорів (perception), приймає рішення на основі цих сприйняттів та своєї внутрішньої логіки (reasoning), та діє у цьому оточенні за допомогою виконавчих механізмів (acting) для досягнення визначених цілей. Центральними характеристиками такого агента є автономність, адаптивність, цілеспрямованість та інтерактивність. Фактично, вся дисципліна ШІ може бути визначена як "вивчення та проектування інтелектуальних агентів". В контексті комп'ютерних ігор, термін "ігровий агент" часто є синонімом Non-Player Character (NPC)[11]. Історично, NPC - це будь-який персонаж у грі, що не контролюється людиною-гравцем. Важливо зазначити, що ранні NPC не обов'язково були "інтелектуальними" у вищезгаданому сенсі; їхня поведінка часто була суто детермінованою та жорстко заскриптовою, не маючи жодної автономії чи адаптивності.

Проте, сучасні ігри прагнуть до значно більшої глибини. Еволюція NPC демонструє перехід від простих скриптів до симуляцій. У таких іграх, як The Elder Scrolls V: Skyrim, NPC отримали власні розклади та рутини (робота, їжа, сон), що створювало ілюзію живого світу. У Red Dead Redemption 2 були використані більш складні системи ШІ для створення непередбачуваних патернів поведінки та взаємодій. Таким чином, сучасний "ігровий агент" - це NPC, що володіє справжньою автономією, здатністю приймати рішення та адаптуватися до дій гравця. Для системного аналізу, функціональні можливості агентів

класифікуються за рівнем їхньої складності. Ця класифікація є фундаментальною для розуміння архітектурних рішень при проектуванні ШІ. Агент виступає як **суб'єкт-діяч** у грі: він сприймає інформацію про стан ігрового світу через інтерфейс (датчики, камери тощо) і формує дії-реакції у відповідь. Таким чином ігровий агент – це відособлена одиниця з власною логікою прийняття рішень, яка керує віртуальним персонажем або об'єктом. У формальному сенсі: «ігровий агент визначено як уявний персонаж у світі гри, який може бути керованим гравцем (PC) або комп'ютерним (NPC)». У функціональному плані роль агента полягає у створенні і підтриманні гри (поведінки персонажів, появі супутників/ворогів тощо), наповненні світу подіями та забезпеченні взаємодії з гравцем [3].

Ігрові агенти можна класифікувати за кількома ознаками:

1. **За рівнем автономності:**

- **Пасивні агенти** – не мають власної поведінки та діють лише у відповідь на зовнішні події або скрипти, фактично не приймаючи рішень самостійно (наприклад, довільно розташовані статичні об'єкти, пастки чи примусове анімаційне «оживання» фону гри).
- **Реактивні агенти** – швидко відповідають на поточну ситуацію в ігровому середовищі без планування та з мінімальною пам'яттю. Їх рішення ґрунтуються на простих правилах «якщо–то» (пресетах умов-дій). Наприклад, вороги-наглядачі одразу атакують гравця при виявленні або втікають при низькому здоров'ї. Такі агенти імітують рефлекси (аналогічно до біологічних організмів) й зазвичай обмежені п'ятьма–десятьма рядками умов.
- **Когнітивні агенти** – мають внутрішню модель світу, пам'ять та можуть планувати свої дії на декілька кроків уперед. Вони враховують цілі та контекст, можуть адаптуватися та навчатися. У грі це реалізують через системи штучного інтелекту (наприклад, побудова планів, машинне навчання, нейромережі). Когнітивні агенти здатні, наприклад, запам'ятовувати попередні зустрічі з гравцем і міняти тактику (як у систéмі Nemesis із *Middle-earth: Shadow of Mordor* чи навчанні ботів у Dota 2).

2. **За роллю в ігровому процесі:**

- **NPC-супутники (дружні агенти)** – негрові персонажі, які допомагають або супроводжують гравця (союзники, наприклад, компаньйони у RPG: Лідія у *Skyrim*).
- **Ворожі агенти** – NPC-противники, що вступають у конфлікт з гравцем (монстри, бандити, поліцейські тощо).
- **Нейтральні агенти** – NPC, які не беруть безпосередньої участі у бою, їх реакція нейтральна або контекстна (наприклад, торговці, мирні жителі). Як зазначено: «NPC можуть бути дружніми, нейтральними та ворожими». Усі ці ролі розрізняються за поведінкою та метою: супутники прагнуть допомагати гравцю, вороги – перешкоджати, а нейтрали – виконують сторонні функції (торгівля, створення атмосфери).

3. **За методом прийняття рішень:**

- **На основі правил (rule-based)** – агенти керуються жорстко закодованими правилами, умовними операторами («якщо – то»). Це найбільш простий підхід: NPC дотримуються стабільної логіки.
- **Сценарні агенти (scripted)** – їх поведінка задається сценаріями: вони виконують передбачені сценарні дії у певному порядку (часто в сюжетних моментах). Такі агенти менш динамічні: їхня поведінка визначена заздалегідь розробником (наприклад, відомі боса-тактики чи репліки в квестах).
- **Агенти на базі ШІ (AI-based)** – використовують складнішу систему штучного інтелекту (пошук, евристики, дерева поведінки тощо) для прийняття рішень. Їхня дія залежить від моделі середовища. Наприклад, сучасні ігри застосовують алгоритми штучних нейронних мереж або планувальники для NPC (див. *Alien: Isolation*, де ворог-предмет реалізований через комбінацію AI-модулів).
- **Навчені агенти (learning)** – саме агенти з можливістю машинного навчання (зазвичай – за методом підкріплення). Вони аналізують попередній досвід (помилки та успіхи) для вдосконалення поведінки з часом. Такі агенти

здатні адаптуватись до стилю гравця, як-от боти OpenAI Five у Dota2 або експериментальні системи на основі глибинного RL.

Таблиця 1.1

Класифікація ігрових агентів

Критерій	Тип агента	Короткий опис	Приклад гри (агент у ній)
Рівень автономності	Пасивний агент	Не приймає власних рішень, реагує тільки на зовнішні тригери	Статичні об’єкти і пастки (Tower Defense тощо)
	Реактивний агент	Швидка реакція на події за фіксованими правилами	Вартові-вороги у Skyrim (пристосовують поведінку за шаблоном)
	Когнітивний агент	Містить модель світу, планує дії, може навчатися	Приватні компаньйони у RPG із адаптивними діалогами
Роль у грі	Дружній NPC (супутник)	Допомагає або супроводжує гравця	Лідія, супутниця в *Skyrim*
	Ворожий NPC (противник)	Перешкоджає гравцю, атакує або конфронтує	Бандити і дракони у *Skyrim*, поліція в *GTA*
	Нейтральний NPC	Не проявляє агресію до гравця (або поводиться залежно від контексту)	Торговці і мирні жителі у *Skyrim*; перехожі в *GTA*
Метод рішення	На правилах (rule-based)	Рішення заздалегідь задані у вигляді правил (детерміновані)	Ворожі NPC зі статичними стратегіями у старих іграх
	Сценарні агенти (scripted)	Підпорядковані сценарію, виконують фіксовану послідовність дій	Сцени появи босів, діалогові NPC у сюжетних квестах
	На базі ШІ (AI-based)	Використовують алгоритми ШІ (пошук, дерева поведінки) для динамічних рішень	NPC із деревами поведінки в Unreal Engine; вороги з адаптивним AI
	Навчені (Learning)	Застосовують машинне навчання (підкріплення, нейромережі) для покращення поведінки	Боти із RL (наприклад, OpenAI Five у Dota2) або системи із навчанням у пісочницях

Ігровий агент у роботі розглядається як **автономний програмний об’єкт**, що взаємодіє зі світом гри. Класифікація агентів передбачає розгляд їх автономності (від простих реактивних до когнітивних з плануванням), ролі в ігровому процесі (друзі, вороги, нейтральні NPC) та способу формування поведінки (прості правила, скрипти чи алгоритми ШІ/машинного навчання) [24]. Усі ці аспекти підтверджені сучасними дослідженнями та гральними рушіями:

прості агенти здатні миттєво реагувати на стимули, тоді як агенти з навчанням – адаптуються і змінюють модель поведінки з часом. Таким чином, концепція ігрового агента поєднує елементи штучного інтелекту та специфічних ігрових механік для створення достовірного ігрового досвіду. Важливо розуміти, що ця академічна класифікація (простий, модельний, цільовий, утилітарний, навчальний) не є простою теорією. Вона безпосередньо відображає еволюцію практичних інженерних архітектур, які будуть детально розглянуті в наступному розділі. Кожна архітектура ШІ є, по суті, інженерним рішенням для реалізації одного з цих типів агентів:

- Модельно-рефлексивні агенти реалізуються через Дерева Поведінки (BT), які використовують Blackboard (Чорну дошку) для зберігання внутрішнього стану (моделі світу).
- Цільові агенти є теоретичною основою для архітектури Планування дій на основі мети (GOAP).
- Утилітарні агенти реалізуються безпосередньо через архітектуру Утилітарного ШІ (Utility AI).
- Навчальні агенти реалізуються за допомогою фреймворків Машинного навчання (ML-Agents).

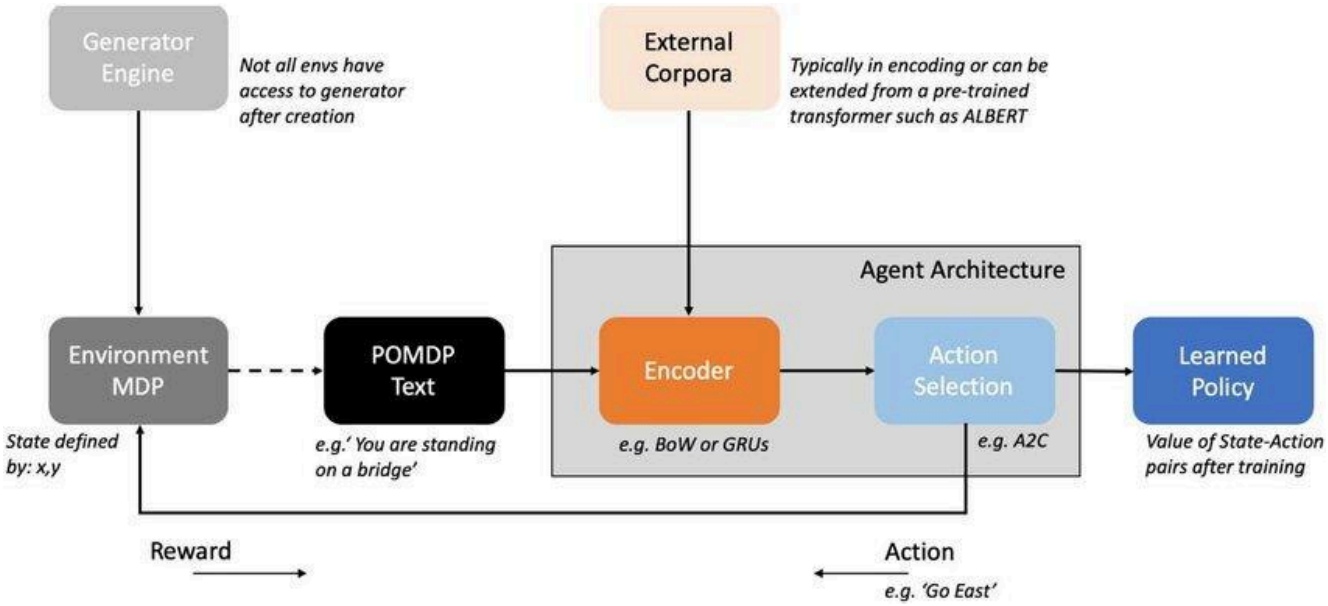

Рис.1.1. Загальна структура ігрового агента: агент сприймає стан середовища, обробляє інформацію та відправляє дії у світ гри.

Окрім цієї функціональної класифікації, існує також когнітивна архітектура Belief-Desire-Intention (BDI), яка моделює процес прийняття рішень, імітуючи людське "практичне міркування". Ця модель описує поведінку агента через три фундаментальні ментальні складові[5] :

− Beliefs (Вірування): Знання агента про світ (його внутрішній стан, дані сприйняття).

− Desires (Бажання): Кінцеві цілі або стани, яких агент прагне досягти.

− Intentions (Наміри): Конкретні цілі, які агент обрав і зобов'язався виконати, включно з планом для їх досягнення.

Модель BDI є потужною теоретичною основою для створення правдоподібних ігрових агентів, оскільки вона дозволяє створювати цілеспрямовану, але гнучку поведінку.

Рис.1.2. Репрезентативний приклад: оздоблення офіційної презентації The Sims 4 (агенти гри – віртуальні персонажі-симулятори повсякденного життя).

На горизонті з'являється нова парадигма: використання Генеративного ШІ та Великих Мовних Моделей (LLM) для створення динамічних NPC. Такі агенти не

обмежені заздалегідь написаними діалоговими деревами. Вони можуть пам'ятати минулі взаємодії з гравцем, вчитися на їх основі та мати власні, унікальні мотивації, що приводить до безпрецедентного рівня занурення та реалізму.

1.2. Методи моделювання поведінки ігрових агентів

Вибір правильної архітектури для кодування та управління поведінкою агента є фундаментальним рішенням у дизайні ігрового ШІ. Це завжди компроміс між простотою реалізації, обчислювальною вартістю під час виконання (runtime) та гнучкістю й правдоподібністю кінцевої поведінки. Нижче проаналізовано основні парадигми, що домінують у розробці ігор.

Теорія: Кінцевий автомат - це математична модель обчислень, що представляє поведінку системи через скінченну кількість станів (наприклад, "Патруль", "Атака", "Втеча", "Бездіяльність"). Система може перебувати лише в одному стані в будь-який момент часу. Поведінка визначається трьома елементами:

Режими поведінки, наприклад, шукати гравця, визначають, як саме агент реагує на події у середовищі. Переходи – це умови, що примушують систему змінити стан, наприклад, коли гравець потрапляє у зону видимості. Дії – це логіка, яка виконується під час перебування у певному стані. Переваги такого підходу полягають у його простоті: концепція FSM є надзвичайно легкою для розуміння та реалізації навіть без складних інструментів. Вона не потребує великих обчислювальних ресурсів і є дуже економною [1]. Крім того, FSM забезпечує чітке розділення логіки, що робить її ідеальною для сценаріїв із визначеною кількістю станів і конкретними переходами між ними.

Недоліки полягають у так званому вибуху станів – зі збільшенням кількості можливих ситуацій кількість переходів між ними зростає експоненціально, що

призводить до заплутаної структури коду. Система стає жорсткою, адже додавання нового стану або зміна логіки переходів часто потребує втручання у багато інших частин коду. Відсутність модульності також ускладнює повторне використання окремих частин логіки. Частковим рішенням є ієрархічні кінцеві автомати, які дозволяють вкладати один автомат у інший, створюючи більш гнучку структуру. Дерева поведінки стали промисловим стандартом для створення складного штучного інтелекту, прийшовши на зміну класичним кінцевим автоматам. Вони являють собою ієрархічну, спрямовану ациклічну структуру, де виконання починається з кореня і рухається вниз по гілках, зліва направо, у кожному кадрі. Основні елементи дерева – це корінь, композитні вузли, декоратори та вузли завдань [14].

Композитні вузли визначають послідовність або вибір дій: у першому випадку кожна дія виконується по черзі, доки всі не завершаться успіхом, у другому – система зупиняється, щойно одна дія спрацює успішно. Декоратори змінюють поведінку дочірнього вузла, наприклад, інвертують результат або повторюють виконання. Листові вузли – це конкретні завдання на кшталт переміститись, атакувати або запустити анімацію. Важливим компонентом є чорна дошка, на якій зберігається стан агента. Саме дерево не має власної пам’яті, тому звертається до цього спільного сховища для читання й запису даних, що дозволяє створити модельно-рефлексивного агента.

Перевагою дерев поведінки є їхня модульність: піддерева можна легко переносити або перевикористовувати. Вони добре масштабуються й мають візуальну наочність, що спрощує розробку у середовищах, таких як Unreal Engine. Недоліком є складність початкового налаштування – створення системи потребує великої кількості допоміжного коду. Крім того, дерева поведінки, як і FSM, залишаються переважно реактивними, тобто реагують на події, але не планують складних послідовностей дій наперед. Планування дій на основі мети, або GOAP, з’явилося як наступний крок у розвитку систем штучного інтелекту. Його концепція полягає у тому, що розробник визначає не алгоритм дій, а лише кінцеву

мету, а система сама знаходить оптимальну послідовність кроків для її досягнення [3].

GOAP працює із трьома ключовими поняттями: станом світу, метою та діями. Стан світу описує фактичні умови, наприклад, чи має агент зброю або де він перебуває. Мета – це бажаний результат, як-от отримати зброю. Дії мають передумови, ефекти та вартість виконання. Передумови визначають, що має бути істинним для виконання дії, ефекти – як зміниться світ після її виконання, а вартість – наскільки ця дія затратна. Планувальник, найчастіше алгоритм A*, шукає найкоротший або найдешевший шлях від поточного стану до цільового. Перевага GOAP у його гнучкості: агент може самостійно адаптуватися до змін у світі. Якщо одна дія стає недоступною, він знайде інший шлях до мети. Такий підхід роз'єднує логіку та дані, дозволяючи дизайнерам додавати нові дії без зміни коду планувальника [16].

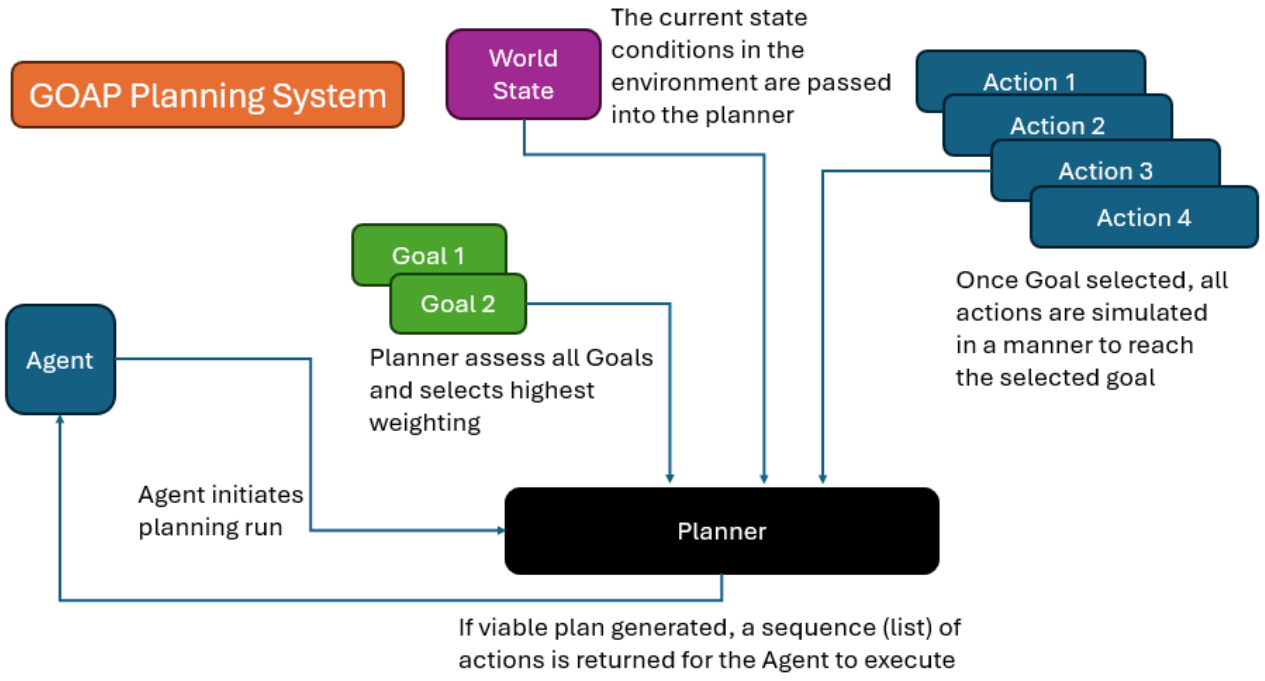

Рис.1.3. Схема роботи GOAP-системи планування дій ігрового агента (Goal-Oriented Action Planning)

Недоліки полягають у високій обчислювальній складності, адже пошук оптимального плану може займати значні ресурси. Крім того, система важко піддається налагодженню – іноді складно зрозуміти, чому агент обрав саме цей план. Ще однією проблемою є те, що під час виконання плану агент може ігнорувати нові події, доки не завершить поточну послідовність дій або поки вона не стане неможливою.

Утилітарний штучний інтелект є прямою реалізацією утилітарного агента. На відміну від кінцевих автоматів або дерев поведінки, система Utility AI не перемикає стани, а постійно оцінює всі доступні дії, вибираючи ту, що має найвищу корисність у конкретний момент часу. Основою є концепція Infinite Axis Utility Systems, у якій використовуються кілька послідовних етапів. Спочатку відбувається збір вхідних даних із середовища гри, наприклад, рівень здоров'я, кількість боєприпасів, відстань до ворога. Потім ці дані нормалізуються до єдиного діапазону від нуля до одного, що дозволяє оцінювати різнорідні показники за єдиною шкалою. Ключовим елементом є криві відгуку, які визначають, як саме певний параметр впливає на корисність дії [6]. Наприклад, дія «шукати аптечку» отримає високий рівень корисності, якщо здоров'я персонажа низьке, і майже нульовий – коли воно повне. Після цього результати міркувань об'єднуються, зазвичай множенням, щоб отримати фінальну оцінку дії. Агент вибирає ту, що має найвищий показник.

Перевага такого підходу полягає у здатності приймати тонкі, багатофакторні рішення. Агент може не просто діяти за схемою «так чи ні», а враховувати баланс між різними потребами, наприклад, ухилятися від ворога, коли здоров'я низьке, але атакувати, якщо ворог близько. Така система гнучка – легко додавати нові дії або параметри без руйнування логіки. Проте вона складна в налаштуванні. Через численні математичні криві важко передбачити точну поведінку агента. Ще однією проблемою є «смикання», коли агент починає хаотично перемикатися між кількома діями з близькими оцінками, якщо не реалізувати механізм пріоритетів або інерції. Ієрархічні мережі завдань, або HTN, представляють ще один підхід до

планування, який став альтернативою GOAP. Якщо GOAP шукає будь-яку можливу послідовність дій для досягнення мети, то HTN розкладає складне завдання на підзавдання. Розробник задає високорівневі завдання, наприклад, «влаштувати засідку», і визначає методи для їх подальшого поділу на конкретні кроки – знайти укриття, дочекатися ворога, атакувати [28]. Процес триває доти, доки не залишаться лише найпростіші завдання, які можна виконати безпосередньо. Підхід GOAP працює знизу вгору, генеруючи нові, неочікувані комбінації дій, тому є гнучким і емергентним. HTN, навпаки, рухається зверху вниз, і тому є передбачуванішим, швидшим і простішим для контролю з боку дизайнера. Це, по суті, сценарій, який система динамічно обирає під час виконання.

Таблиця 1.2

Порівняльний аналіз архітектур моделювання поведінки ШІ

Критерій	Кінцевий Автомат (FSM)	Дерево Поведінки (BT)	Планування (GOAP)	Планування (HTN)	Утилітарний ШІ (Utility AI)
Парадигма	Реактивна	Реактивна	Проактивна (Планувальна)	Проактивна (Планувальна)	Реактивна (Оціночна)
Основна логіка	Переходи між станами	Ієрархічне дерево (Seq/Sel)	Пошук A* в просторі станів	Декомпозиція завдань	Оцінка за кривими корисності
Обчислювальна вартість	Дуже низька	Низька	Висока (під час планування)	Середня/Висока (швидше за GOAP)	Середня (залежить від к-сті дій)
Легкість налагодження	Легко (на початку), неможливо (при масштабуванні)	Середня (візуально)	Складно ("Чому цей план?")	Середня ("Чому цей метод?")	Дуже складно ("Чорна скринька")
Гнучкість / Емергентність	Дуже низька	Середня	Дуже висока	Висока	Висока (але хаотична)

Основний випадок використання	Прості NPC, UI, стани гравця	Стандартний тактичний ШІ (вороги, союзники)	Складні NPC (напр. *F.E.A.R.*), симуляції	Скриптові боси, стратегії	Складні NPC з суперечливими потребами (напр. *The Sims*)

Жодна з цих архітектур не є універсальною. FSM не масштабуються, дерева поведінки обмежені у гнучкості, GOAP вимагає значних обчислювальних ресурсів, а Utility AI важко тонко налаштувати. Саме тому сучасні ігрові системи штучного інтелекту часто поєднують кілька підходів, створюючи гібридну архітектуру. На стратегічному рівні визначається, що саме потрібно зробити – для цього застосовуються GOAP або Utility AI, які дозволяють агенту вибрати найдоцільнішу мету, наприклад, вирішити, що пошук аптечки зараз важливіший за атаку. На тактичному рівні реалізується, як саме виконати цю мету, і тут ефективно працюють дерева поведінки або кінцеві автомати, які забезпечують детермінованість і простоту налагодження [17]. На виконавчому рівні виконуються конкретні дії, такі як пошук шляху або взаємодія з об’єктами, що реалізуються за допомогою алгоритмів на кшталт A*. Такий підхід поєднує гнучкість і здатність до адаптації високорівневих систем із точністю і контрольованістю низькорівневих структур, створюючи збалансовану, ефективну й передбачувану модель поведінки ігрового штучного інтелекту.

1.3. Алгоритми прийняття рішень і симуляції у реальному часі

Алгоритми пошуку шляху є основою будь-якої системи штучного інтелекту, яка потребує переміщення у просторі. Їхнє завдання полягає у знаходженні найкоротшого або найменш затратного маршруту між двома точками на карті, що зазвичай моделюється як граф, де вузли – це позиції, а ребра – можливі переходи. Пошук у ширину є одним із найпростіших алгоритмів. Він рівномірно досліджує

всі напрямки від стартової точки й завжди знаходить найкоротший шлях за кількістю кроків. Однак він не враховує вартість руху, тому в складних картах є неефективним. Алгоритм Дейкстри, на відміну від BFS, бере до уваги вартість кожного переходу. Наприклад, рух через болото може коштувати більше, ніж по твердій дорозі. Такий підхід гарантує пошук найдешевшого шляху, проте є «сліпим» – досліджує простір у всіх напрямках, не враховуючи розташування цілі. Жадібний алгоритм Best-First Search намагається виправити цю недоліковість, використовуючи евристику – оцінку відстані до фінішу. Він рухається до вузлів, які здаються найближчими до мети, завдяки чому працює швидше, але не завжди знаходить оптимальний шлях, оскільки може потрапити у локальний мінімум [4].

Алгоритм A* об’єднує точність Дейкстри з ефективністю жадібного підходу. Його ключова формула $f(n) = g(n) + h(n)$ складається з двох частин: реальної вартості шляху до поточного вузла і евристичної оцінки вартості від нього до цілі. Якщо евристика є допустимою, тобто не переоцінює відстань, A* гарантує оптимальний результат. Для розрахунків зазвичай використовують евклідову або манхеттенську відстань. Алгоритм працює через дві множини – відкриту, де зберігаються вузли, що ще потрібно перевірити, і закриту, де зберігаються вже досліджені. Саме тому A* став стандартом у відеоіграх і симуляціях. Алгоритми прийняття рішень, або змагальний пошук, застосовуються у випадках, коли потрібно знайти найкращий хід у грі з двома противниками, наприклад, у шахах або шашках. Вони працюють у середовищі, де кожен гравець намагається максимізувати свій результат і мінімізувати результат супротивника. Основою таких систем є алгоритм Minimax. Він рекурсивно будує дерево можливих ходів до певної глибини, де один гравець – це MAX, який намагається збільшити свій виграш, а інший – MIN, який намагається зменшити його. Штучний інтелект вибирає хід, що призводить до найкращого результату за умови, що суперник діє оптимально. Проблема полягає у величезній кількості можливих ходів: при

середньому коефіцієнті розгалуження b та глибині пошуку m кількість комбінацій становить $O(b^m)$, що робить алгоритм обчислювально дорогим.

Щоб скоротити час обчислень, використовується альфа-бета відсічення. Цей метод не змінює результат, але відсікає гілки дерева, які не впливають на кінцеве рішення. Він підтримує два параметри: альфа – найкраще значення для MAX, і бета – найкраще для MIN. Якщо під час аналізу гілки стає зрозуміло, що мінімальний гравець може змусити оцінку бути нижчою за альфа, ця гілка більше не розглядається [5]. Така оптимізація дозволяє значно збільшити глибину аналізу без втрати продуктивності. Алгоритми симуляції груп, або Flocking AI, вирішують іншу проблему – реалістичне відтворення руху великих скупчень агентів. Це можуть бути зграї птахів, косяки риб чи натовпи людей. Класичний підхід – алгоритм Boids, розроблений Крейгом Рейнольдсом у 1986 році. Його принцип полягає у децентралізації: жоден агент не є лідером, кожен реагує лише на сусідів у певному радіусі, і складна, узгоджена поведінка виникає природним шляхом.

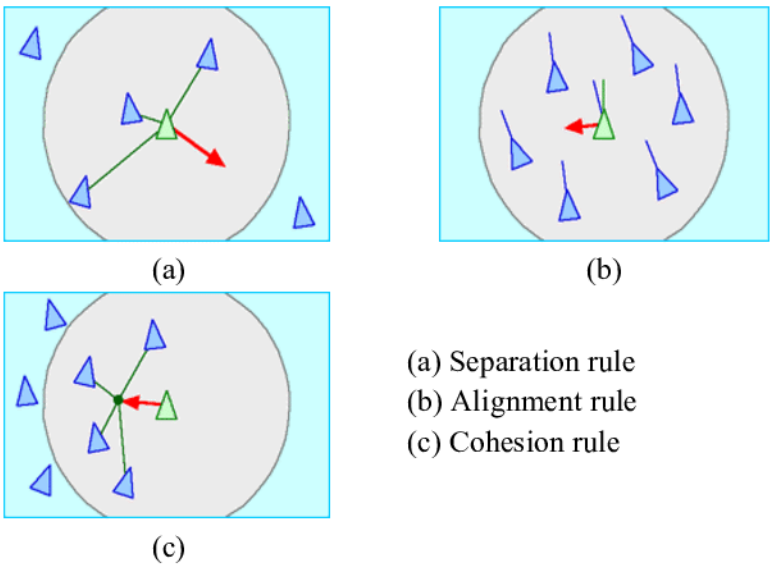

Рис. 1.4. Правила зграйної поведінки: розділення (a), вирівнювання (b), згуртованість (c)

Boids базується на трьох простих правилах. Перше – розділення, коли агент уникає зіткнень і тримає дистанцію від сусідів. Друге – вирівнювання, коли агент орієнтується у тому ж напрямку, що й сусіди. Третє – згуртованість, коли агент

прагне рухатися до центру групи, зберігаючи єдність. У кожному кадрі агент розраховує три вектори, зважує їх і коригує свій рух. Основна складність полягає у високій обчислювальній вартості, оскільки кожен агент повинен перевіряти відстань до всіх інших. Тому сучасні реалізації виконують пошук сусідів на графічних процесорах, що дозволяє симулювати тисячі об’єктів у реальному часі. Алгоритми цього рівня не існують самі по собі. Вони є основними механізмами дії для вищих архітектур штучного інтелекту. Коли листовий вузол дерева поведінки віддає команду «рухатися до гравця», він фактично викликає алгоритм A* для побудови шляху[15]. Коли планувальник GOAP оцінює вартість дії, він розраховує її за допомогою того самого A*. А правила Boids можуть бути реалізовані як міркування в системі Utility AI, де підсумковий вектор руху визначається на основі оцінок корисності. Таким чином, складні архітектури поведінки не могли б існувати без цих фундаментальних алгоритмів. Вони становлять основу будь-якої інтелектуальної системи, дозволяючи поєднувати планування, координацію та прийняття рішень у єдину динамічну модель.

1.4. Огляд сучасних ігрових рушіїв та їх можливостей

Ігрові рушії стали фундаментом сучасної розробки штучного інтелекту, адже саме вони перетворили створення складної поведінки агентів на доступний і керований процес. Завдяки вбудованим інструментам і високому рівню інтеграції розробники можуть не витрачати час на реалізацію базових алгоритмів, а зосередитися на створенні логіки та глибини поведінки персонажів.

Unreal Engine вирізняється філософією «батарейки в комплекті» – він пропонує розробнику повноцінну екосистему для побудови ШІ, де все вже з’єднано між собою. Основою системи є Behavior Trees – візуальні дерева, у яких логіка поведінки описується через вузли завдань і умов. Разом із ними працює Blackboard – центральна пам’ять агента, де зберігаються змінні на кшталт координат гравця, стану здоров’я чи останньої цілі. Найпотужніший інструмент

рушія – Environment Query System, який дозволяє ШІ аналізувати оточення через запити типу: «знайди найближче укриття» або «вибери найкращу точку для атаки». Цей модуль створює сітку точок, оцінює їх за низкою критеріїв і повертає найоптимальніший варіант. У типовому циклі Behavior Tree викликає EQS, отримує найкращу позицію, записує її у Blackboard, а потім система навігації NavMesh прокладає шлях до неї. Завдяки такій побудові Unreal забезпечує стабільність, передбачуваність і високу продуктивність навіть у великих проєктах[7].

Unity підходить до ШІ інакше – це радше набір інструментів, з яких розробник може створити власну архітектуру. Тут панує філософія «toolbox»: рушій не нав’язує структуру, а лише дає модулі, які можна вільно комбінувати. Вбудована система NavMesh забезпечує навігацію агентів, тоді як Animator може виконувати роль простого кінцевого автомата, керуючи станами персонажів. Офіційний пакет Unity Behavior додає Behavior Graph – візуальний редактор дерев поведінки. У нових версіях рушія з’явилася ціла екосистема Unity AI Suite, до якої входять Sentis, Muse і ML-Agents. Sentis дозволяє виконувати вже навчені нейронні моделі у форматі .onnx прямо на пристрої гравця. Muse допомагає розробникам, генеруючи код, текстури та навіть анімації за допомогою генеративного ШІ. ML-Agents відкриває шлях до навчання агентів через підкріплення, що робить можливим створення адаптивних, «живих» поведінкових моделей, які неможливо було б запрограмувати вручну. Unity дає простір для експериментів і гібридних підходів, поєднуючи класичні алгоритми з технологіями машинного навчання.

Godot Engine уособлює філософію відкритого коду, простоти та повного контролю. Він не нав’язує структуру, а надає фундаментальні інструменти, з яких можна побудувати будь-яку систему. У версії 4.x рушій має NavigationServer і NavigationAgent, які забезпечують пошук шляху у дво- та тривимірному просторі. Також доступні вузли Astar2D і Astar3D, що дозволяють створювати власні графи для пошуку маршрутів. Повноцінного редактора поведінки в рушії немає, проте

спільнота розробила кілька популярних плагінів, серед яких LimboAI і Beehave. Перший поєднує Behavior Trees і FSM, дозволяючи створювати складні гібридні системи, а другий зосереджується виключно на деревах поведінки[24]. Завдяки відкритому коду та гнучкій архітектурі Godot ідеально підходить для навчальних, експериментальних або інді-проєктів, де потрібен максимальний контроль над логікою ШІ.

Таблиця 2.

Основні можливості ШІ в ігрових рушіях

Можливість	Unreal Engine 5	Unity (2023.x+)	Godot Engine 4.x
Візуальний редактор поведінки	Вбудований: Behavior Trees	Вбудований (пакет): Behavior Graph	Плагіни: LimboAI (BT + FSM)
Система навігації (NavMesh)	Вбудована (NavMesh)	Вбудована (NavMesh)	Вбудований: NavigationServer
Аналіз оточення (EQS)	Вбудований EQS	Відсутній, потребує кастомної розробки	Відсутній
Виконання ML-моделей (Inference)	Плагіни	Вбудований: Sentis	Плагіни, наприклад Godot-Tensorflow
Навчання ML-моделей (Training)	Плагіни (Learning Agents)	Пакет: ML-Agents	Відсутній, використовуються зовнішні інструменти

Таким чином, вибір рушія визначає не лише технічні можливості, а й саму філософію створення штучного інтелекту. Unreal Engine пропонує цілісну, майже військову структуру, де все працює «з коробки» і з максимальною ефективністю. Unity надає свободу та інструменти, які можна адаптувати під будь-який тип гри, поєднуючи класичні й нейромережеві методи. Godot же залишає повну автономію, дозволяючи розробнику самому вирішувати, як мислитиме його агент.

1.5. Особливості мови програмування C# у Unity для реалізації ШІ

Unity використовує C# як основну мову скриптування. Особливості рушія Unity, зокрема його компонентно-орієнтований дизайн, безпосередньо впливають на те, як архітектури ШІ проектуються та реалізуються на практиці.На відміну від

традиційного об'єктно-орієнтованого програмування з глибокими ієрархіями успадкування, Unity використовує компонентно-орієнтований дизайн (COD). Сутність у грі (GameObject) – це просто порожній контейнер. Вся поведінка та дані додаються до нього шляхом приєднання компонентів. Кожен компонент є C#-скриптом, що успадковує клас MonoBehaviour [26]. Це дозволяє створювати NPC шляхом композиції: GameObject "Ворог" = Компонент Transform (позиція) + Компонент NavMeshAgent (рух) + Компонент Health (дані про здоров'я) + Компонент EnemyAI (логіка).

Аналіз ефективних практик розробки ШІ в Unity виявляє потужний архітектурний патерн, який можна назвати "Тріадою Інструментів C#". Він вирішує три ключові проблеми ШІ (Логіка, Дані та Час) за допомогою трьох різних інструментів Unity.

1. "Логіка": MonoBehaviour як мозок MonoBehaviour є "мозком" агента – він містить логіку Update(), обробляє події та приймає рішення.

Реалізація FSM: Як зазначалося, MonoBehaviour можна використовувати для реалізації станів. Але більш елегантний підхід – використання компонента Animator як візуального FSM. У цьому випадку Animator керує не анімацією, а логічними станами, активуючи відповідні скрипти StateMachineBehaviour для кожного стану.

2. "Дані": ScriptableObject як особистість

Проблема: Зберігання даних (наприклад, maxHealth = 100, attackDamage = 10, patrolSpeed = 3) безпосередньо у полях MonoBehaviour (наприклад, EnemyAI.cs) створює жорсткий зв'язок між логікою та даними. Це призводить до дублювання даних у префабах і робить балансування гри кошмаром.

Рішення: ScriptableObject – це спеціальний клас Unity, призначений для зберігання даних як окремих асетів у проекті.

Ключова перевага: Розділення Даних та Логіки (Separation of Concerns).

EnemyAI.cs (MonoBehaviour) містить логіку ("як патрулювати", "як атакувати").

GoblinConfig.asset (ScriptableObject) містить дані ("швидкість патрулювання: 2.5", "агресивність: 0.8").

OrcConfig.asset (ScriptableObject) містить дані ("швидкість патрулювання: 1.5", "агресивність: 1.0").

Застосування: Один скрипт EnemyAI.cs може керувати і "Гобліном", і "Орком", просто отримуючи різні асети ScriptableObject як конфігурацію. Це робить систему надзвичайно гнучкою для гейм-дизайнерів, яким не потрібно торкатися коду для зміни поведінки NPC.

3. "Час": Coroutines (Корутини) як секвенсор

Проблема: Логіка ШІ часто є послідовною та залежною від часу (наприклад, "йти до точки А, почекати 3 секунди, повернути голову, почекати 1 секунду, йти до точки Б"). Реалізація цього у функції Update(), яка викликається щокадру, вимагає складних таймерів та змінних стану ("спагеті-код").

Рішення: Корутини. У C# це методи, що повертають IEnumerator. Це, по суті, функції, які можуть призупиняти своє виконання в певному місці за допомогою оператора yield return і відновлювати його з того ж місця пізніше.

Ідіома для ШІ:

```
// C#
IEnumerator PatrolBehavior()
{
    agent.SetDestination(pointA);
    // Призупинити виконання цієї функції,
    // доки агент не дійде до точки
    yield return new WaitUntil(() => agent.remainingDistance < 0.1f);

    // Призупинити виконання на 3 секунди
    yield return new WaitForSeconds(3.0f);

    agent.SetDestination(pointB);
}
```

Рис.1.5. Фрагмент C#-коду корутини PatrolBehavior для патрулювання агента NavMeshAgent

Важливо: Корутини не є паралельними потоками (multithreading). Вони виконуються в основному потоці Unity, але дозволяють писати асинхронний, послідовний код у чистому, синхронному стилі. Корутини можна зупинити в будь-який момент (StopCoroutine). C# SDK для ML-Agents є ще одним прикладом компонентної архітектури. Розробник створює скрипт, що успадковує Agent, і реалізує методи C# для[30]:

CollectObservations(): Збір даних для "мозку" (позиція, швидкість, вороги).

OnActionReceived(): Отримання команди від "мозку" та її виконання (застосування сили, зміна швидкості).

SetReward(): Надання позитивного/негативного зворотного зв'язку.

Таким чином, C# у Unity надає потужний, ідіоматичний набір інструментів (MonoBehaviour, ScriptableObject, Coroutines), який ідеально підходить для реалізації модульних, гнучких та керованих даними архітектур ШІ.

1.6. Аналіз існуючих інструментів моделювання поведінки агентів

Unity NavMesh використовується для забезпечення можливості агентів знаходити шлях у складному тривимірному або двовимірному середовищі. Це фундаментальна система, на якій базується будь-яка поведінка, пов’язана з рухом. NavMesh – це полігональна сітка, що апроксимує всі прохідні поверхні у сцені. Вона створюється через процес “запікання” (Baking), під час якого враховується геометрія сцени. Розробник обирає всі статичні елементи, які мають бути враховані при побудові маршруту – підлоги, сходи, стіни, перешкоди – та позначає їх як Navigation Static. Далі налаштовуються параметри агента: радіус, висота, максимальний кут нахилу, висота кроку. Саме ці значення визначають, чи включатимуться у навігаційну сітку певні вузькі проходи або низькі арки. Після запікання Unity створює фінальну синю полігональну сітку[17]. У процесі виконання до об’єкта додається компонент NavMeshAgent, який відповідає за переміщення. Його параметри – швидкість, прискорення, кут повороту, дистанція

зупинки – визначають, як саме агент пересуватиметься. Керування маршрутом здійснюється простим викликом у коді C#: agent.SetDestination(Vector3 target).

Візуальні редактори дерев поведінки забезпечують розробникам зручний інтерфейс для створення складних поведінкових схем без необхідності писати великий обсяг коду. Unreal Engine має вбудований редактор Behavior Tree, який глибоко інтегрований із Blackboard – системою зберігання змінних, та EQS – системою просторового аналізу. Така інтеграція дозволяє створювати складні сценарії з урахуванням сприйняття агентом оточення. У Unity існує офіційний пакет Behavior Graphs, який підтримує нелінійну архітектуру, де гілки можуть знову з'єднуватись у спільний потік, що робить можливим створення більш складних логічних структур. У Godot популярним рішенням став плагін LimboAI – відкритий інструмент, що поєднує вузли як для Behavior Tree, так і для FSM у єдиному візуальному середовищі. Це дозволяє створювати гібридні системи: наприклад, дерево поведінки керує загальною логікою, а всередині одного з вузлів запускається FSM для управління бойовими станами.

Unreal Engine має окремий потужний інструмент для аналізу оточення – EQS (Environmental Query System), який надає агентам можливість оцінювати ситуацію і знаходити оптимальні рішення. Замість прямолінійного руху до гравця, агент аналізує навколишнє середовище та вибирає найкращу позицію, наприклад, укриття. Система працює за принципом генерації кандидатів, визначення контекстів і тестування. Генератори створюють точки для перевірки – наприклад, усі укриття або певну сітку навколо цілі. Контексти визначають точки відліку, якими можуть бути сам агент або гравець. Тести оцінюють кожну точку за певними параметрами – відстань, наявність прямої видимості, можливість дістатись до точки за допомогою пошуку шляху. EQS проводить фільтрацію, обчислює бали й повертає найкращий результат у Blackboard, після чого дерево поведінки використовує цю інформацію для прийняття рішення.

Unity ML-Agents Toolkit представляє перехід від детермінованого проектування поведінки до навчання агентів через досвід. У його основі лежить

навчання з підкріпленням (Reinforcement Learning) та імітаційне навчання. Агент у симуляційному середовищі спостерігає стан світу, виконує дії та отримує винагороду – позитивну або негативну залежно від результату. Мета полягає у тому, щоб навчити модель максимально збільшувати сумарну винагороду. Процес навчання здійснюється у зв'язці C# SDK, який визначає параметри сцени, та Python API, що виконує власне тренування нейронної мережі через PyTorch. Після завершення навчання отримана модель (.onnx) імпортується назад у Unity та використовується у грі через Unity Sentis для прийняття рішень у реальному часі. Такий підхід особливо ефективний для завдань, які важко запрограмувати традиційними методами: балансування роботів, командна координація, складна бойова поведінка[6].

Сучасний ігровий штучний інтелект можна розділити на дві філософії – детермінований дизайн і емергентне навчання. У першому випадку розробник виступає архітектором поведінки, який явно прописує всі реакції та дії агента. До цього підходу належать FSM, Behavior Trees, NavMesh, EQS. Він забезпечує передбачуваність, контроль і простоту налагодження, оскільки завжди можна простежити, яка логіка привела до певного рішення. У другому підході – Trainer Stack – розробник виступає тренером, який не прописує дії безпосередньо, а створює середовище, сенсори та систему винагород. Поведінка виникає природно, як результат навчання моделі. Така система менш передбачувана, але значно більш адаптивна. Майбутнє ігрового ШІ, ймовірно, лежить у поєднанні цих двох підходів, коли навчальні агенти формують високорівневу стратегію, а детерміновані системи забезпечують точне виконання її елементів.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

2.1. Постановка задачі та визначення вимог

Першим кроком у проєктуванні будь-якої складної системи є чітке визначення проблеми, яку вона має вирішувати, та формулювання набору конкретних, вимірюваних вимог. Цей підрозділ каталогізує функціональні та нефункціональні вимоги, що слугують основою для всіх подальших архітектурних рішень. Основною проблемою, що стоїть перед розробниками ігрового ШІ, особливо в іграх із механіками прихованості (стелс), є *поведінкова передбачуваність*. Сучасні ігри досягли вражаючої графічної достовірності, однак ця візуальна складність часто затьмарюється непереконливою та рудиментарною поведінкою ШІ. Коли супротивники суворо дотримуються жорстких маршрутів патрулювання або миттєво "забувають" про гравця після нетривалої тривоги, ігровий процес зводиться не до перехитрування думаючого супротивника, а до простої експлуатації запрограмованої послідовності. Ця розбіжність між візуальною та поведінковою складністю руйнує занурення та знецінює досягнення гравця.

Таким чином, центральне завдання полягає у проєктуванні інтелектуальної системи, яка долає цю поведінкову "крихкість". Система повинна генерувати поведінку, яка є:

1. Правдоподібною: Дії агента мають бути логічними та відповідати його ролі у світі.
2. Складною: Агент повинен демонструвати ширший спектр реакцій, аніж бінарна логіка "спокій/тривога".
3. Адаптивною: Агент повинен адекватно реагувати на динамічні зміни в оточенні та дії гравця.

4. Керованою: Поведінка не має бути хаотичною. Вона повинна бути достатньо структурованою, щоб гравець міг її вивчати, прогнозувати (з певною похибкою) та, зрештою, перехитрити.

Вирішення цієї задачі вимагає ретельного балансу між реалістичною симуляцією та створенням контрольованого, захоплюючого ігрового досвіду.

Функціональні вимоги визначають, що саме має робити система, тобто які операції й поведінкові сценарії повинен виконувати ШІ-агент. Підсистема сприйняття включає набір сенсорів, що імітують людські органи чуття. Агент повинен мати зорову систему у вигляді «конуса зору», у межах якого він здатен помічати гравця або інших персонажів, причому видимість має блокуватися стінами та іншими фізичними об’єктами. Слух повинен дозволяти реагувати на різні звуки – кроки, постріли, падіння предметів – і визначати їхнє джерело, що запускає поведінку розслідування. Фізичний контакт або отримання шкоди з боку гравця має миттєво переводити агента у стан підвищеної тривоги.

Поведінка агента управляється моделлю ментальних станів. У стані спокою він або патрулює визначеним маршрутом, або випадково переміщується у межах заданої зони. У стані пошуку агент реагує на сигнали, що викликають підозру, або на втрату гравця з поля зору. У цьому режимі він повинен рухатися до останньої відомої позиції гравця та перевіряти кілька потенційних укриттів, де той міг сховатися. Рівень пильності має спадати поступово[15]. У стані тривоги агент активно переслідує гравця, орієнтуючись на його актуальну або останню відому позицію.

У бойовому режимі агент повинен застосовувати складні тактики. Він має вміти знаходити укриття та використовувати їх для атаки чи уникнення шкоди. Його поведінка повинна залежати від ролі: стрільці можуть вести придушувальний вогонь або використовувати гранати, снайпери – шукати вигідні височини, а берсеркери – стрімко скорочувати дистанцію. За необхідності агенти можуть координувати дії між собою, наприклад, коли один відволікає ворога вогнем, а інший атакує з флангу.

Нефункціональні вимоги описують не те, які дії виконує система, а те, наскільки якісно вона здатна це робити. Це ключові архітектурні обмеження, що визначають ефективність, стабільність і життєздатність усієї системи ШІ. До них належить вимога продуктивності: логіка штучного інтелекту, включно зі сприйняттям, навігацією й ухваленням рішень, повинна працювати в межах дуже малого процесорного бюджету на кожен кадр, зазвичай у межах двох-трьох мілісекунд для всіх агентів разом. Лише так можна підтримувати стабільну високу частоту кадрів, наприклад 60 FPS, що критично для плавного ігрового процесу.

Масштабованість також має велике значення. Система повинна залишатися стабільною при збільшенні кількості агентів – наприклад, від десяти до сотні одночасно – і не допускати експоненційного падіння продуктивності. Вона також повинна дозволяти ускладнювати логіку окремого агента, наприклад додавати нові бойові дії чи поведінкові шаблони, без того щоб це призводило до неконтрольованого зростання обчислювальних витрат. Гнучкість і модульність є ще одним важливим аспектом: архітектура має дозволяти легко додавати нові типи поведінки, сенсорів або ворогів, не порушуючи вже створених систем. Логіка прийняття рішень повинна бути зрозумілою для розробників, бажано з можливістю візуалізації, щоб спростити налагодження та підтримку[1]. Монолітні структури або жорсткі зв’язки між компонентами у цьому контексті є надзвичайно небажаними.

Поєднання всіх вимог створює природний конфлікт між бажанням зробити поведінку агента складною, багатошаровою й непередбачуваною та необхідністю залишатися в межах суворих технічних обмежень. Такі елементи, як активний пошук гравця чи розгалужені бойові тактики, потребують значних обчислювальних ресурсів, тоді як вимоги до продуктивності та масштабованості встановлюють чіткі межі того, що може собі дозволити система. Саме баланс між цими двома полюсами визначає якість архітектурних рішень у сучасних ігрових ШІ-системах. Тому, центральним викликом проєктування, який буде розглянуто в наступних підрозділах, є не просто *реалізація* функціональних вимог, а

пошук *архітектури*, яка дозволяє реалізувати їх таким чином, щоб задовольнити нефункціональні обмеження. Вибір між такими парадигмами, як Машини скінченних станів, Дерева поведінки або Планування на основі цілей, є не академічним вибором "найкращої" технології, а прагматичним пошуком оптимального компромісу в цьому конфлікті.

2.2. Архітектура системи та вибір алгоритмів

У сучасному геймдеві домінують три головні парадигми: машини скінченних станів, дерева поведінки та планування на основі цілей. Машини скінченних станів є найпростішою й найстарішою моделлю. Агенти в таких системах перебувають у чітко визначених станах, між якими переходять, реагуючи на події чи умови. Цей підхід приваблює простотою і надзвичайно високою продуктивністю, адже в будь-який момент часу виконується лише один активний стан і перераховується дуже обмежений набір можливих переходів. Водночас масштабованість є головною проблемою FSM. Щойно система починає рости, стани перетворюються на заплутаний клубок залежностей. Додавання навіть одного нового стану вимагає модифікації багатьох інших, що фактично призводить до появи некерованої структури, перенасиченої умовами. Ієрархічні FSM частково пом’якшують цю проблему, але не усувають її повністю, тому що переходи залишаються жорстко пов’язаними всередині кожного стану. Модульність та повторне використання логіки в такій архітектурі є обмеженими, а це прямо суперечить нефункціональним вимогам на зразок гнучкості та підтримуваності.

Дерева поведінки сьогодні вважаються галузевим стандартом для складних ігрових ШІ. Цей підхід уперше здобув широку популярність після впровадження в Halo 2. Дерево поведінки – це ієрархічна структура, яка кожну секунду або кожен кадр повторно оцінює ситуацію, від кореня до листків. ВТ відокремлюють логіку ухвалення рішення від логіки виконання дій: композитні вузли визначають, коли і

як слід переходити між поведінками, тоді як листки містять окремі дії. Саме ця архітектура забезпечує високий рівень модульності, легкість розширення та можливість візуалізації – все це робить поведінку агентів значно зрозумілішою для розробників і дизайнерів. Проте така гнучкість має ціну. На відміну від FSM, дерева поведінки постійно проходять частину або всю свою логіку на кожному тіку. Це означає, що при великій кількості агентів або глибоких деревах витрати процесорного часу можуть зростати досить суттєво. Хоча це не позбавляє BT статусу найзручнішої парадигми, воно вимагає акуратного проєктування, щоб відповідати суворим вимогам до продуктивності.

Планування на основі цілей (GOAP) – це підхід, у якому агент сам будує план дій, щоб досягти заданої мети. Розробник задає лише цілі та дії з передумовами й ефектами, а агент за допомогою алгоритмів пошуку (як-от A*) знаходить оптимальний шлях. Метод став популярним після F.E.A.R. Перевага GOAP у тому, що він створює адаптивну та емерджентну поведінку: якщо один шлях недоступний, агент знайде інший. Додавання нової дії автоматично розширює можливості персонажа[13]. Недоліки – складність реалізації, високе навантаження на продуктивність через постійний пошук у просторі станів та менший контроль дизайнера над конкретною поведінкою агента, що проблемно для ігор зі строгими сценаріями, наприклад стелс-проєктів.

Таблиця 2.1. Порівняльний аналіз парадигм прийняття рішень

Критерій	Машини скінченних станів (FSM)	Дерева поведінки (BT)	Планування на основі цілей (GOAP)
Модульність (NFR3)	Низька. Переходи жорстко пов'язані зі станами.	**Висока.** Легке додавання/видалення гілок поведінки.	Висока (на рівні дій), але низька (на рівні логіки).
Продуктивність (NFR1)	**Дуже висока.**	Висока, але повільніша за FSM (вимагає оптимізації).	Низька. Висока обчислювальна вартість планування.
Масштабованість (NFR2)	Низька (вертикальна). "Код-спагеті"	**Висока.** Структура підтримує складність.	Низька (горизонтальна).

	при ускладненні.		Вартість зростає з кількістю агентів.
Контроль/Передбач.	Високий. Поведінка жорстко задана.	**Високий.** Дизайнер контролює потік та пріоритети.	Низький. Важко передбачити *як* буде досягнута ціль.
Складність реалізації	Низька (для простих систем).	Середня.	**Дуже висока**.
Основний сценарій	Простий ШІ, UI, анімації.	**Стандарт для ШІ-агентів у сучасних іграх**.	Складні симуляції, стратегії, емерджентна поведінка.

Обґрунтування вибору парадигми прийняття рішень у системі штучного інтелекту приводить до висновку, що саме Дерева поведінки є найбільш відповідним рішенням для цього проєкту. Порівняння трьох основних підходів – FSM, GOAP та BTs – показує, що лише Дерева поведінки здатні витримати технічні й дизайнерські вимоги, які висуває стелс-гра з численними агентами, складною тактикою та високими очікуваннями щодо продуктивності.

Відмовляючись від FSM, ми стикаємося з їхньою структурною негнучкістю. Кінцеві автомати природно розростаються у «код-спагеті» при додаванні нових станів або переходів. Вони суперечать вимогам до модульності та підтримуваності, адже кожне доповнення впливає на всю систему. Будь-яка спроба моделювати нюансовану поведінку, особливо стан «Пошук» чи комбінації поведінкових реакцій, стає складною, громіздкою й непридатною для масштабування.

GOAP, хоч і є вражаючим підходом із точки зору емерджентності, не відповідає вимогам до продуктивності й масштабованості. Динамічне планування в реальному часі – надто дорога операція для системи, де одночасно працює багато автономних агентів. Окрім цього, непередбачуваність такого підходу суперечить природі стелс-ігрового дизайну, де поведінка охорони повинна залишатися керованою, передбачуваною й логічно узгодженою із задумом

геймдизайнера[7]. У таких жанрах автономна творчість ШІ часто більше шкодить, ніж допомагає.

У цьому контексті Дерева поведінки стають компромісом, який поєднує контрольованість, гнучкість і продуктивність. Вони дозволяють отримати реактивну логіку, що ідеально підходить для ролі охоронця: агент може миттєво відгукуватися на зміни у світі, дотримуючись чіткої ієрархії пріоритетів. Ієрархічна структура забезпечує природну модульність, дозволяючи легко комбінувати та розширювати поведінку. Основні перевірки – помітив гравця, знижене здоров’я, наявність укриття – можуть бути розташовані у верхівці дерева, а базові шаблони поведінки, як-от патрулювання, діятимуть як поведінка за замовчуванням. Дерева поведінки виявляються візуально зрозумілими, легко підтримуваними й значно ефективнішими за GOAP у плані системного навантаження, що дає можливість оптимізувати їх під продуктивність.

Вибір BTs визначає не лише логіку прийняття рішень, а й особливості програмної архітектури, яка має відповідати компонентному підходу Unity. Система повинна бути розбитою на незалежні підсистеми, які можна гнучко комбінувати та перевикористовувати. Замість одного великого класу, що містить усю поведінку, агент складається з набору спеціалізованих компонентів MonoBehaviour. Центральним модулем виступає AIAgentController, який відповідає за координацію, ініціалізацію компонентів та менеджмент життєвого циклу ШІ. Перцептивна система зосереджується на логіці виявлення, аналізує поле зору та звукові тригери. Навігаційна система інкапсулює роботу NavMeshAgent і відповідає за рух. Бойова система відповідає за тактичні рішення, вибір укриттів і зброї. Нарешті, BehaviorTreeRunner забезпечує виконання дерева поведінки, синхронізуючи логіку прийняття рішень із іншими компонентами[19].

Ще однією важливою архітектурною ідеєю стає відокремлення логіки та даних за допомогою ScriptableObjects. У Unity часто виникає проблема, коли поведінка і конфігураційні параметри змішуються у MonoBehaviour. Це ускладнює редагування й створює непотрібні зв’язки, що знижує модульність. Рішенням є

перенесення всіх даних агента – швидкість, здоров’я, час реакції, дальність зору, точність – до одного конфігураційного SO-класу. Кожен тип ворога отримує власний .asset файл, що дозволяє дизайнерам коригувати статистики без втручання в логіку, а програмістам – працювати з чистою структурою. Під час гри кожен компонент, що потребує цих параметрів, звертається саме до AgentConfig, забезпечуючи єдиний центр даних та прекрасну масштабованість.

Ще однією ключовою частиною архітектури стає система подій на основі ScriptableObjects. Вона вирішує проблему роз’єднання компонентів, які мають комунікувати, але не повинні знати один про одного безпосередньо. До прикладу, коли система сприйняття помічає гравця, вона не повинна напряму викликати метод в конкретному компоненті дерева поведінки. Натомість вона передає сигнал через Event Channel – ScriptableObject, що містить подію. Компонент, який відповідає за поведінку, підписується на цей канал і реагує, коли сигнал надходить. Такий підхід майже повністю усуває жорстку зв’язність і перетворює архітектуру на гнучку систему, де будь-яку частину можна замінити або змінити, не порушуючи інші модулі.

В сукупності всі ці рішення дають чисту, модульну, розширювану й продуктивну архітектуру, яка повністю узгоджується з вимогами системи. BTs забезпечують передбачувану логіку для жанру стелс-ігри, компонентний поділ робить систему легко підтримуваною, ScriptableObjects гарантують роз’єднання логіки й даних, а подієві канали дозволяють компонентам комунікувати без створення жорстких залежностей. Це створює надійну основу для складної поведінки агентів, яку можна масштабувати, розвивати і налаштовувати без ризику втратити керування над системою.

У цьому підрозділі сформульовано мету та вимоги до інтелектуальної системи «охоронець-протагоніст» у середовищі Unity. Метою проєкту є розробка простого штучного інтелекту охоронця, який патрулює визначений маршрут, виявляє гравця (протагоніста) та реагує переслідуванням. Актуальність такої системи обґрунтовується її застосуванням у комп’ютерних іграх жанру stealth та

симуляторах безпеки, де правдоподібна поведінка NPC-охоронців підвищує реалізм ігрового процесу. Система повинна демонструвати базові можливості штучного інтелекту: автономне пересування сценою, реагування на появу гравця у полі зору та прийняття рішень залежно від ситуації. Вимоги до системи сформульовано з урахуванням сценарію «охоронець і порушник»:

− Функціональні можливості охоронця: NPC-охоронець повинен патрулювати задані точки маршруту (waypoints) у сцені, рухаючись між ними по оптимальних шляхах. Він має володіти “зором” – виявляти гравця у визначеному радіусі та полі зору. При виявленні гравця охоронець переходить від патрулювання до переслідування цілі, тобто починає наздоганяти гравця. Якщо гравець зникає з поля зору (наприклад, сховався за перешкодою або вийшов за межі радіуса), охоронець через певний час повертається до патрулювання. Охоронець також не повинен виходити за межі доступної території і обходити перешкоди на шляху до цілі.

− Характеристики середовища: Середовище моделюється в Unity як сцена з навігаційною сіткою NavMesh для забезпечення руху персонажів. Локація містить статичні об’єкти (стіни, перешкоди), які служать укриттями та обмежують зону прямої видимості. Необхідно налаштувати NavMesh таким чином, щоб він охоплював усі області, де може переміщатися охоронець (наприклад, підлога, коридори) і виключав непрохідні зони (стіни, бар’єри). Сцена має містити кілька ключових точок патрулювання для охоронця, розташованих так, щоб забезпечити огляд більшої частини території. Наприклад, у випадку лабіринту контрольні точки варто ставити на перехрестях і поворотах, щоб охоронець оглядав кожен коридор.

− Взаємодія з користувачем: Користувач керує персонажем-гравцем (порушником), що рухається сценою, намагаючись не потрапити в поле зору охоронця. Вимоги до ігрового процесу включають зручне керування гравцем (наприклад, з клавіатури та миші для переміщення та огляду), наявність базового інтерфейсу (UI) для відображення важливої інформації (наприклад, повідомлення

про те, що гравця помічено, або стану тривоги). Система повинна реагувати на дії гравця в реальному часі – якщо гравець наближається до охоронця або перетинає його зону спостереження, то це одразу впливає на поведінку NPC. Після завершення переслідування (успішного чи ні) бажано передбачити сценарій перезапуску або продовження симуляції для повторного тестування поведінки.

Перераховані вимоги визначають критерії успішності розроблюваної системи. Вони забезпечують досягнення поставленої мети – створення інтерактивної сцени з правдоподібною поведінкою охоронця на основі алгоритмів штучного інтелекту. Наступні підрозділи присвячені вибору архітектури та алгоритмів, що дозволять реалізувати ці вимоги, а також деталям практичної реалізації в Unity.

2.3. Моделювання поведінки агентів у середовищі Unity

Архітектура C#-класів агента ШІ зручно описується за допомогою діаграм UML, які відображають як статичну структуру класів, так і компонентну збірку префабу в Unity. На UML-діаграмі класів (рис. 2.1) демонструються ключові зв’язки між основними елементами системи. Усі компоненти, що кріпляться до GameObject, наслідуються від базового класу Unity MonoBehaviour. Усі асети-дані створюються на основі ScriptableObject, який слугує базовим класом для конфігурацій і подієвих каналів.

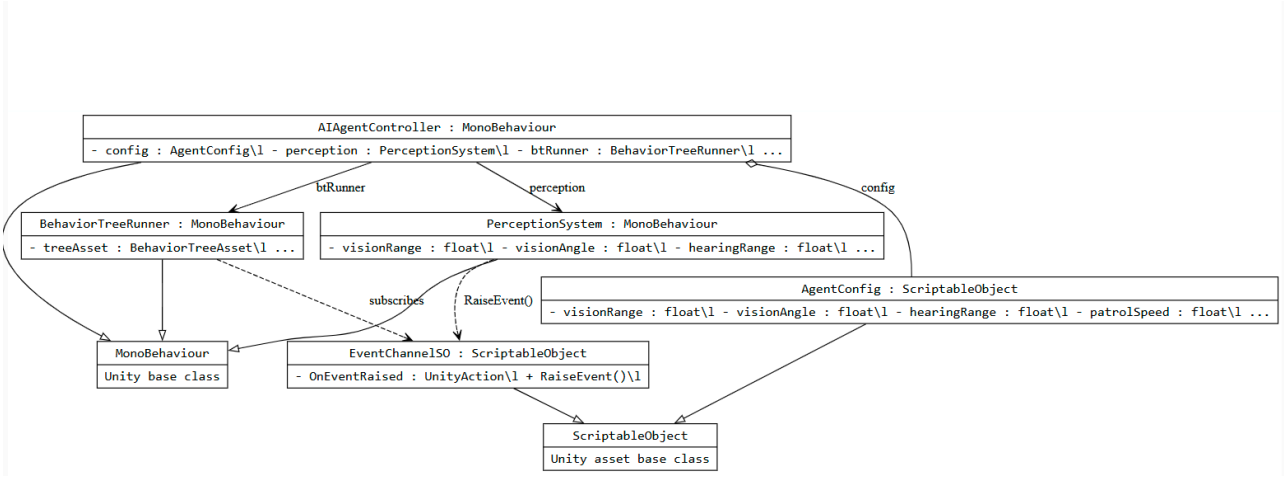

Рисунок 2.1. UML-діаграма класів архітектури агента

Центральним класом виступає AIAgentController, який є нащадком MonoBehaviour. Він має зв’язок агрегації з AgentConfig: це саме агрегація, а не композиція, оскільки AgentConfig реалізовано як ScriptableObject-асет, що існує незалежно від життєвого циклу конкретного екземпляра AIAgentController, зберігається в проєкті й може бути перевикористаний різними агентами. Крім того, AIAgentController має асоціації (або композицію, якщо він створює компоненти самостійно) з іншими компонентами, прикріпленими до того самого GameObject, зокрема з PerceptionSystem і BehaviorTreeRunner, а також, за потреби, з NavigationSystem і CombatSystem.

Клас AgentConfig успадковується від ScriptableObject і виступає чистим контейнером даних. У ньому зберігаються всі конфігураційні параметри: наприклад, поля типу float visionRange, float visionAngle, float hearingRange, float patrolSpeed, а також додаткові числові чи логічні параметри, пов’язані з боєм, швидкістю реакції, точністю тощо. Клас PerceptionSystem є MonoBehaviour-компонентом, що відповідає за реалізацію сенсорів, насамперед зору та слуху. Він має залежність від EventChannelSO – ця залежність у UML позначається пунктирною стрілкою: PerceptionSystem не зберігає EventChannelSO як складову частину, але використовує його для виклику методу RaiseEvent і генерації подій “гравця помічено”, “звук почуто” тощо. Клас EventChannelSO, у свою чергу, наслідується від ScriptableObject і виступає базовим класом для подій. Усередині нього визначається, наприклад, публічне поле типу UnityAction OnEventRaised, а також метод public void RaiseEvent(), який викликає делегат і сповіщає всіх підписників. BehaviorTreeRunner, як ще один MonoBehaviour-компонент, також має залежність від EventChannelSO, але вже в ролі слухача: він підписується на події відповідних каналів і реагує на їх виклик, змінюючи стан дерева поведінки або активуючи відповідні гілки.

На UML-діаграмі компонентів (рис. 2.2) показано, як формується префаб агента, наприклад GuardAgent.prefab. Цей префаб можна розглядати як реалізацію

деякого інтерфейсу IAgent, який визначає базові методи взаємодії агента з рештою системи. Усередині префаб містить стандартні Unity-компоненти: Transform як базову просторову прив’язку, Rigidbody для взаємодії з фізикою, CapsuleCollider для визначення фізичного тіла, а також NavMeshAgent – вбудований компонент Unity AI, що відповідає за навігацію по NavMesh. До того ж на префабі розміщуються C#-скрипти: AIAgentController.cs як керуючий “мозок”, який зв’язує всі системи й оперує конфігураційними даними; PerceptionSystem.cs, який реалізує сенсорну систему (зір, слух, дотик); BehaviorTreeRunner.cs, що відповідає за виконання ітерацій (“тіків”) дерева поведінки; CombatSystem.cs, який інкапсулює бойову логіку – вибір укриття, зброї, типу атаки тощо. Таким чином, префаб агента у вигляді GuardAgent.prefab є композицією з низки компонентів, кожен із яких несе свою чітку відповідальність у межах загальної архітектури.

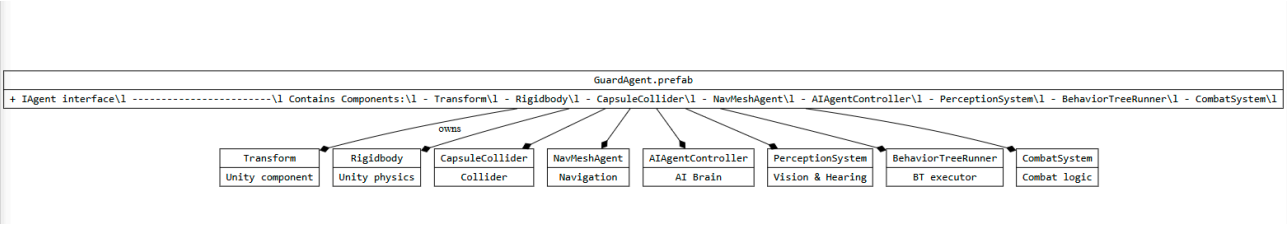

Рисунок 2.2 UML-діаграма компонентів префабу агента

Реалізація підсистеми сприйняття є критичною для правдоподібної поведінки та водночас чутливою до продуктивності. Особливо це стосується зору, адже наївна реалізація може швидко “з’їсти” обчислювальний бюджет, суперечачи вимогам до продуктивності (NFR1). Якщо використовувати лише Physics.Raycast у кожному кадрі, це дає надто вузьку картину – промінь перевіряє фактично одну точку й не дає повного розуміння того, хто перебуває в полі зору. Якщо ж обмежитися лише перевіркою кута між напрямком погляду агента і напрямком до потенційної цілі, система ігноруватиме реальні перешкоди, як-от стіни чи інші об’єкти, що блокують видимість. Тому застосовується оптимізований трьохетапний підхід, який зазвичай виконується не щокадру, а з нижчою частотою, наприклад у методі FixedUpdate або раз на певний інтервал (умовно 5

разів на секунду), щоб зменшити навантаження на CPU і відповідати заданому бюджетові часу.

Перший етап – це так звана широкофазна перевірка (Broad Phase), яка фокусується на колізіях у межах радіуса зору. Мета цього етапу – якнайшвидше зібрати список усіх потенційних цілей, які фізично знаходяться в зоні досяжності сенсора. Для цього використовується метод Physics.OverlapSphere, який виконує сферичну перевірку навколо поточного положення агента. Типовий виклик виглядає як Physics.OverlapSphere(transform.position, visionRange, targetLayerMask). Тут transform.position задає центр сфери, visionRange – радіус, а targetLayerMask – бітова маска шарів, які вважаються потенційними цілями (наприклад, шар Player або Allies). Результатом стає масив Collider-ів, що представляють усі об’єкти потенційних цілей у радіусі зору.

Другий етап – кутова перевірка (Angle Check), що визначає, чи знаходяться знайдені об’єкти в межах заданого конуса зору. Для цього використовується скалярний добуток (dot product) між напрямком погляду агента і напрямком на ціль. Для кожної цілі спершу обчислюється нормалізований вектор directionToTarget: Vector3 directionToTarget = (target.position - transform.position).normalized;. Далі визначається скалярний добуток: float dotProduct = Vector3.Dot(transform.forward, directionToTarget);. Оскільки скалярний добуток двох нормалізованих векторів дорівнює косинусу кута між ними, його можна використати як ефективний критерій без виклику відносно дорожчого Vector3.Angle(). Перевірка виглядає, наприклад, так: if (dotProduct > Mathf.Cos(visionAngle * 0.5f * Mathf.Deg2Rad)) – тобто ціль вважається у полі зору, якщо косинус кута між напрямком уперед та напрямком на ціль більший за косинус половини кута огляду. Таким чином, ми залишаємо лише ті об’єкти, що знаходяться не тільки в радіусі, а й у межах заданого конуса зору.

Третій етап – перевірка перешкод (Obstruction Check), яка гарантує, що між “очима” агента й ціллю немає непрозорих об’єктів. Цей крок виконується лише для тих цілей, які пройшли попередні два фільтри, що суттєво скорочує кількість

дорогих Raycast-операцій. Реалізація може виглядати так: спершу розраховується вектор directionToTarget = target.position - eyes.position;, де eyes.position – точка, що імітує позицію “очей” агента (часто це окрема дочірня трансформа). Далі викликається Physics.Raycast(eyes.position, directionToTarget, out RaycastHit hit, visionRange, obstructionLayerMask). Якщо промінь “влучає” в перший об’єкт на шляху й цей об’єкт відповідає самій цілі (hit.transform == target), робиться висновок, що між агентом і ціллю немає перешкод, а отже, об’єкт справді є видимим. Якщо ж Raycast спершу зустрічає інший колайдер (стіна, ящик, товста колона), ціль вважається прихованою, навіть якщо вона перебуває в радіусі й у межах кута зору.

У сумі ці три етапи дозволяють отримати збалансовану систему зору: швидка широкофазна фільтрація через OverlapSphere, дешева кутова перевірка через скалярний добуток та точна, але обмежена за кількістю Raycast-перевірка на перешкоди. Така архітектура не лише забезпечує доволі реалістичну поведінку “бачення”, але й відповідає суворим обмеженням NFR1 щодо продуктивності, оскільки обчислювально важкі операції істотно скорочуються за рахунок поетапного відсіювання зайвих цілей.

```
C#
using UnityEngine;

public class VisionSensor : MonoBehaviour
{
    public float visionRange = 20f;

    public float visionAngle = 90f;

    public LayerMask targetLayer;
    public LayerMask obstructionLayer;

    public Transform detectedTarget = null;
    private float visionAngleCosine;

    private void Start()
    {
        // Попередньо обчислюємо косинус для оптимізації
        visionAngleCosine = Mathf.Cos(visionAngle * 0.5f * Mathf.Deg2Rad);
    }

    // Викликати цей метод з фіксованою частотою, а не щокадру
    public bool CheckVision()
    {
        detectedTarget = null;
        Collider targetsInRadius = Physics.OverlapSphere(transform.position, visionRange,
targetLayer);

        foreach (Collider targetCollider in targetsInRadius)
        {
            Vector3 directionToTarget = (targetCollider.transform.position -
transform.position).normalized;

            // Етап 2: Перевірка кута (Dot Product)
            if (Vector3.Dot(transform.forward, directionToTarget) > visionAngleCosine)
            {
                float distanceToTarget = Vector3.Distance(transform.position,
targetCollider.transform.position);

                // Етап 3: Перевірка перешкод (Raycast)
                if (!Physics.Raycast(transform.position, directionToTarget, distanceToTarget,
obstructionLayer))
                {
                    // Ціль видима
                    detectedTarget = targetCollider.transform;
                    return true;
                }
            }
        }
        return false;
    }
}
```

Рис.2.3. C#-реалізація конуса зору агента (перевірка кута за допомогою Dot Product і Raycast для виявлення перешкод)

Слух у системі ШІ реалізується через «імітацію», оскільки використання реального аудіо-рушія робить поведінку агента непередбачуваною, що суперечить природі стелс-ігор. Такі ігри працюють як головоломки, тому гравець повинен інтуїтивно розуміти, де проходять межі «радіуса слуху» ворога. Замість складних аудіо-моделей застосовується фізична симуляція шуму. У проєкті створюється окремий NoiseManager, зазвичай у вигляді статичного класу або Singleton. Коли гравець генерує шум, наприклад кидає камінь чи наступає на пляшку, він викликає

метод на кшталт EmitNoise(position, radius). Цей метод формує сферу пошуку за допомогою Physics.OverlapSphere, охоплюючи всі об’єкти зі шаром ШІ в заданому радіусі. Кожен агент, якого знайдено всередині цієї сфери, отримує виклик OnSoundHeard зі вказаною позицією джерела звуку. Уже дерево поведінки або інша система прийняття рішень переходить у стан пошуку, записує координати почутого шуму й спрямовує NavMeshAgent до цієї точки, виконуючи вимоги функціоналу FR2.2.

```
C#
using UnityEngine;

// Статичний клас для керування шумом
public static class NoiseManager
{
    public static void EmitNoise(Vector3 position, float radius, LayerMask aiLayer)
    {
        Collider agentsInRange = Physics.OverlapSphere(position, radius, aiLayer);

        foreach (Collider agentCollider in agentsInRange)
        {
            // Використовуємо інтерфейс для гнучкого зв'язку
            IAgentHearing listener = agentCollider.GetComponent<IAgentHearing>();
            if (listener!= null)
            {
                listener.OnSoundHeard(position);
            }
        }
    }
}

// Інтерфейс, який повинен реалізувати агент ШІ
public interface IAgentHearing
{
    void OnSoundHeard(Vector3 soundPosition);
}

// Приклад на стороні агента (AIAgentController)
public class AIAgentController : MonoBehaviour, IAgentHearing
{
    //... інший код...
    private BehaviorTreeRunner btRunner; // Посилання на BT

    public void OnSoundHeard(Vector3 soundPosition)
    {
        // Передаємо інформацію в Дерево Поведінки
        // Наприклад, через "чорну дошку" (Blackboard)
        btRunner.Blackboard.Set("LastHeardSoundPosition", soundPosition);
        btRunner.Blackboard.Set("IsAlertedBySound", true);
    }
}
```

Рис.2.4. C#-реалізація імітації слуху ШІ-агентів на основі OverlapSphere і інтерфейсу IAgentHearing

Підсистема навігації та базових дій покладається на стандартний компонент Unity – NavMeshAgent, який забезпечує коректне пересування сценою за умови, що на ній попередньо створено та «запечено» NavMesh. Найтиповішою поведінкою для охоронця чи іншого простого ШІ є патрулювання за наперед визначеним маршрутом. Маршрут формується як набір точок, зазвичай у вигляді масиву Transform-об’єктів, які задають послідовність позицій, між якими пересувається агент. У скрипті патрулювання зберігається список таких точок і змінна індексу, що показує, до якої саме з них агент рухається зараз. Коли NavMeshAgent досягає поточної позиції (це перевіряється, коли залишкова дистанція стає меншою за stoppingDistance і при цьому шлях більше не оновлюється), він одразу переходить до наступної. Нова ціль задається через метод SetDestination, куди передається координата відповідного waypoint. Індекс рухається по колу, що дозволяє агенту нескінченно патрулювати маршрут, змінюючи точки за принципом циклічного масиву.

```
C#
using UnityEngine;
using UnityEngine.AI;


public class AgentPatrol : MonoBehaviour
{
    public Transform waypoints;
    private int destPoint = 0;
    private NavMeshAgent agent;

    void Start()
    {
        agent = GetComponent<NavMeshAgent>();

        // Вимикаємо авто-гальмування для плавного переходу між точками
        agent.autoBraking = false;

        GotoNextPoint();
    }

    void GotoNextPoint()
    {
        if (waypoints.Length == 0) return;

        // Встановлюємо нову ціль
        agent.destination = waypoints[destPoint].position; [62]

        // Оновлюємо індекс для наступної точки (циклічно)
        destPoint = (destPoint + 1) % waypoints.Length; [62]
    }

    // Цей метод буде викликатися з Дерева Поведінки
    public void UpdatePatrol()
    {
        // Перевіряємо, чи агент досяг точки (з урахуванням stoppingDistance)
        if (!agent.pathPending && agent.remainingDistance < 0.5f)
        {
            GotoNextPoint();
        }
    }
}
```

Рис.2.5. C#-скрипт патрулювання NavMeshAgent з використанням набору точок і циклічного обходу для вузла дерева поведінки

Переслідування є однією з найпростішої поведінки для агента: відповідний вузол дерева поведінки просто встановлює нову точку призначення для NavMeshAgent, використовуючи поточну позицію гравця. На кожному тіку дерево викликає SetDestination, і агент безперервно оновлює маршрут, спрямовуючись просто до player.transform.position.

Побудова власного фреймворку дерева поведінки дає змогу повністю контролювати логіку ШІ та глибше зрозуміти принципи роботи Behavior Trees,

навіть якщо у Unity існують готові комерційні рішення або спрощені вбудовані інструменти. У центрі цієї архітектури лежить абстрактний клас Node та перерахування NodeState. NodeState визначає три основні результати, які може повертати будь-який вузол: успіх, невдачу або стан виконання, що триває. Останнє варто використовувати для довготривалих дій на кшталт руху до цілі чи очікування певної події.

Сам абстрактний Node виступає базою для будь-якого типу вузлів – дій, умов, селекторів, послідовностей. Його можна реалізувати як звичайний C# клас або як ScriptableObject, якщо потрібне редагування в інспекторі. На ньому тримається вся логіка поведінкового дерева, адже кожен конкретний вузол перевизначає метод evaluate чи tick, який повертає відповідний стан і таким чином визначає подальшу поведінку агента.

```C#
public enum NodeState
{
    RUNNING,
    SUCCESS,
    FAILURE
}

public abstract class Node
{
    // Базовий метод "тіку" дерева
    public abstract NodeState Update();
}
```

Рис.2.6. Базовий абстрактний клас Node та перелік можливих станів вузла дерева поведінки

Композитні вузли керують потоком виконання своїх дочірніх вузлів.

1. SequenceNode **(Послідовність):** Виконує дочірні вузли по черзі.

- Повертає FAILURE, як тільки один з дочірніх вузлів повертає FAILURE.
- Повертає RUNNING, якщо дочірній вузол повертає RUNNING.
- Повертає SUCCESS, *тільки* якщо *всі* дочірні вузли повернули SUCCESS.
- *Використання:* Для лінійних завдань (наприклад, "Підійти до дверей", *потім* "Відкрити двері").

2. SelectorNode **(Вибір/Пріоритет):** Виконує дочірні вузли по черзі, доки один з них не поверне SUCCESS.
   - Повертає SUCCESS, як тільки один з дочірніх вузлів повертає SUCCESS.
   - Повертає RUNNING, якщо дочірній вузол повертає RUNNING.
   - Повертає FAILURE, *тільки* якщо *всі* дочірні вузли повернули FAILURE.
   - *Використання:* Для вибору поведінки за пріоритетом (наприклад, "Атакувати гравця" *АБО* "Патрулювати").

```
C#
using System.Collections.Generic;

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

    //... (конструктор для додавання дочірніх вузлів)...

    public override NodeState Update()
    {
        foreach (Node node in children)
        {
            switch (node.Update())
            {
                case NodeState.FAILURE:
                    return NodeState.FAILURE;
                case NodeState.RUNNING:
                    return NodeState.RUNNING;
                case NodeState.SUCCESS:
                    continue; // Переходимо до наступного
            }
        }
        return NodeState.SUCCESS; // Всі завершились успішно
    }
}

public class SelectorNode : Node
{
    protected List<Node> children = new List<Node>();

    //... (конструктор для додавання дочірніх вузлів)...

    public override NodeState Update()
    {
        foreach (Node node in children)
        {
            switch (node.Update())
            {
                case NodeState.SUCCESS:
                    return NodeState.SUCCESS;
                case NodeState.RUNNING:
                    return NodeState.RUNNING;
                case NodeState.FAILURE:
                    continue; // Пробуємо наступний
            }
        }
        return NodeState.FAILURE; // Жоден не впорався
    }
}
```

Рис.2.7. Реалізація композиційних вузлів SequenceNode та SelectorNode для дерева поведінки

Вузли-дії є тим рівнем дерева поведінки, де виконується справжня робота агента. Саме вони відповідають за конкретні дії, і саме вони найчастіше стикаються з проблемою тривалого виконання. Дерево оновлюється щокадру, але якщо дія займає час, наприклад очікування кілька секунд чи поступовий рух до точки, вона не повинна блокувати головний потік гри. Це вирішується поєднанням

стану RUNNING та корутин Unity, які дозволяють призупиняти виконання без будь-яких блокувань. Корутина відпрацьовує свою частину роботи у фоновому режимі, повертаючи управління між кадрами, і таким чином Node залишається легким і керованим.

Реалізація будується навколо того, що вузол або BehaviorTreeRunner, який цим вузлом керує, має бути компонентом MonoBehaviour, адже тільки він може запускати корутини. Під час першого звернення до вузла він ініціює корутину і одразу повертає стан RUNNING. Корутина виконує свою тривалу логіку, наприклад чекає певний час або крок за кроком рухає агента. Увесь цей час вузол продовжує повертати RUNNING, даючи зрозуміти дереву, що дія ще триває. Коли корутина завершує роботу, вона змінює внутрішній стан вузла. Після цього наступне оновлення дерева бачить, що дія завершена, і повертає або SUCCESS, або FAILURE залежно від результату. Це створює елегантну і контрольовану модель тривалих дій, повністю сумісну з оновленням дерева у реальному часі.

```
C#
// Компонент, що запускає дерево і керує корутинами
public class BehaviorTreeRunner : MonoBehaviour
{
    private Node rootNode;
    //... (код ініціалізації дерева)...

    void Update()
    {
        if (rootNode!= null)
        {
            rootNode.Update();
        }
    }

    // Вузли можуть викликати це для асинхронних операцій
    public Coroutine StartActionCoroutine(IEnumerator coroutine)
    {
        return StartCoroutine(coroutine);
    }
}

// Вузол-дія "Чекати"
public class WaitNode : Node
{
    private float waitTime;
    private BehaviorTreeRunner runner; // Посилання на MonoBehaviour

    private bool isWaiting = false;
    private float timer = 0f;

    public WaitNode(BehaviorTreeRunner runner, float waitTime)
    {
        this.runner = runner;
        this.waitTime = waitTime;
    }

    public override NodeState Update()
    {
        if (!isWaiting)
        {
            // Почати очікування
            timer = 0f;
            isWaiting = true;
            // Повертаємо RUNNING *негайно*
            return NodeState.RUNNING;
        }
        else
        {
            // Ми вже в процесі очікування
            timer += Time.deltaTime;
            if (timer >= waitTime)
            {
                // Очікування завершено
                isWaiting = false;
                return NodeState.SUCCESS;
            }
            else
            {
                // Очікування триває
                return NodeState.RUNNING;
            }
        }
    }
    // Альтернативна реалізація з корутиною (більш складна в керуванні станом)
    //...
}
```

Рис.2.8. C#-реалізація BehaviorTreeRunner і вузла-дії WaitNode з використанням корутин для тривалих операцій

(Примітка: Приклад вище використовує простий таймер замість корутини для спрощення керування станом, що є поширеною та ефективною практикою для простих затримок. Більш складні дії, як MoveTo, використовували б аналогічний патерн, повертаючи RUNNING, доки agent.remainingDistance > agent.stoppingDistance.)

Цей модульний, керований даними та подіями підхід до проєктування створює надійний фундамент для інтелектуальної системи, яка є одночасно складною, продуктивною та гнучкою для майбутніх розширень. На основі вимог спроєктовано архітектуру інтелектуальної системи, що складається з кількох взаємопов'язаних модулів. Головними компонентами є: модуль навігації, модуль виявлення цілі, модуль керування поведінкою та модуль керування гравцем. Кожен з них реалізує окремі аспекти поведінки системи, а разом вони утворюють цілісну систему "охоронець–гравець".

− Модуль навігації: відповідає за прокладання маршруту та переміщення охоронця по сцені. В ролі цього модуля використано вбудовану систему навігації Unity NavMesh. Було прийнято рішення використати компонент NavMeshAgent для NPC, що дозволяє йому самостійно рухатися по сітці NavMesh до вказаної цілі, автоматично обходячи перешкоди. Вибір NavMesh обґрунтований тим, що Unity надає ефективну реалізацію шляхfinding-алгоритму (як показують дослідження, NavMesh використовує модифікований алгоритм A* з оптимізаціями) і забезпечує динамічне реагування на оточення (уникає зіткнень з іншими агентами, коригує шлях при виявленні перешкод). Таким чином, навігаційний модуль позбавляє необхідності розробляти алгоритм пошуку шляху з нуля і гарантує оптимальні маршрути на основі заданої карти прохідних зон.

− Модуль виявлення гравця: забезпечує "почуття" охоронця – визначає, коли гравець потрапляє в поле зору або діапазон чутливості NPC. Це реалізовано шляхом визначення у охоронця конусоподібної зони видимості (кут огляду, наприклад 60°–90° перед собою) і максимальної дистанції виявлення. При кожному кроці симуляції (Update) або за допомогою подій тригерів перевіряється:

якщо гравець знаходиться в полі зору (тобто потрапив у кут огляду та не заслонений стінами) і достатньо близько – вважається, що охоронець його помітив. Для перевірки перешкод може використовуватися raycasting (випускання променя від охоронця до гравця) чи система колайдерів-тригерів (невидимі області-визначники). Коли умови виявлення виконані, модуль генерує сигнал (наприклад, встановлює певний прапорець стану або викликає подію), що гравець знайдений – це слугує вхідними даними для модуля поведінки.

− Модуль керування поведінкою: реалізує логіку прийняття рішень NPC залежно від ситуації. За основу поведінкової логіки було обрано модель скінченного автомата станів (Finite State Machine, FSM). FSM є моделлю, що описує зміну станів об’єкта залежно від поточного стану та зовнішньої інформації. Для охоронця визначено скінченну множину станів, таких як: “Патрулювання” (нормальний обхід маршруту), “Переслідування” (активне переслідування гравця), а також допоміжні стани “Тривога” або “Пошук” (можуть використовуватися, якщо гравець зник з виду, і охоронець деякий час обшукує район). Перехід між станами здійснюється на основі рішень/умов: наприклад, умова переходу з патрулювання в переслідування – *гравця виявлено в полі зору*[22]; з переслідування назад в патрулювання – *гравця втрачено з поля зору на певний час*. У межах кожного стану охоронець виконує відповідні дії: у стані патрулювання – рухається між точками маршруту; у стані переслідування – прямує до поточного місця розташування гравця, постійно оновлюючи ціль. FSM було обрано через простоту реалізації та наочність поведінки для заданої кількості сценаріїв. На відміну від більш складних моделей (дерева поведінки, нейронні мережі тощо), скінченний автомат з двома-трьома станами легко програмується та достатньо для моделювання базової охоронної AI-логіки.

− Модуль керування гравцем: відповідає за рух та дії ігрового персонажа під керуванням користувача. Він не є центральним з точки зору “інтелектуальності” системи, але необхідний для інтерактивності симуляції. Даний модуль обробляє ввід користувача (клавіатура/миш) і переміщує гравця

сценою (наприклад, змінюючи позицію CharacterController або фізичного Rigidbody, залежно від способу реалізації). Важливо, що переміщення гравця обмежене тими ж навігаційними межами (перешкодами), що і для охоронця, аби забезпечити коректність сценарію переслідування. Також модуль може генерувати події, корисні для інших компонентів – наприклад, повідомляти систему, що гравець здійснив певну дію (але в межах цього проєкту основною дією є переміщення)[1].

Взаємодія між модулями відбувається таким чином: модуль керування гравцем рухає персонажа; модуль виявлення у охоронця моніторить простір і при появі гравця в зоні видимості повідомляє про це модуль поведінки; той у свою чергу змінює стан NPC на “переслідування” і через модуль навігації спрямовує охоронця до гравця. Коли умови змінюються (гравець утік, або NPC досяг гравця), модуль поведінки знову коригує стан і дії. Така архітектура дозволяє чітко розділити завдання: навігація (рух) відділена від логіки вирішення, виявлення цілі відділене від власне погоні, що підвищує модульність і спрощує налагодження системи.

На етапі проєктування визначені компоненти було відображено у середовищі Unity у вигляді конкретних ігрових об’єктів, компонентів та налаштувань. **Агентом-охоронцем** у сцені є окремий GameObject, якому призначено необхідні компоненти для реалізації поведінки. Перш за все, на охоронця додано компонент NavMeshAgent, що дозволяє йому пересуватися по заздалегідь згенерованій навігаційній сітці. Сітка NavMesh була згенерована (пробейкана) для сцени на основі статичних об’єктів: підлоги (доступна поверхня для руху) та стін/перешкод (непрохідні області). Параметри NavMeshAgent (швидкість руху, кут повороту, радіус колайдера агента тощо) були налаштовані так, щоб охоронець рухався плавно і з реалістичною швидкістю патрулювання та погоні. Зокрема, швидкість патрулювання може бути встановлена помірною, а при переході в стан переслідування – дещо вищою, аби NPC міг наздоганяти гравця. Навігаційна система Unity автоматично керує уникненням зіткнень між агентом і

оточенням, тому охоронець обходить перешкоди на шляху до точки призначення без додаткового коду логіки обходу[2].

Для **патрюлювання** середовище забезпечене набором контрольних точок. У Unity кожна така точка реалізована як порожній об’єкт (Empty), розташований у ключовій позиції (наприклад, кут кімнати, дверний проріз, кінець коридору). Ці точки занесені в список (масив) у скрипті AI охоронця, і NPC обирає наступний пункт призначення послідовно або за певним алгоритмом. Наприклад, реалізовано послідовний обхід: індекс точки збільшується на 1 при кожному досягненні, а коли охоронець дійде до останньої точки – маршрут циклічно починається знову з першої. Такий підхід відповідає логіці типового патрулювання охоронця по замкненому маршруту. Для більшої реалістичності передбачено невелику паузу в кінцевих точках маршруту (імітація того, що охоронець оглядається або відпочиває), яка реалізована через лічильник часу або корутину, що затримує перехід до наступної точки.

Виявлення гравця змодельовано за допомогою комбінації колайдера-тригера та перевірки кута огляду. Безпосередньо перед NPC встановлено невидимий конус або циліндр (наприклад, сферичний колайдер) з радіусом дії, прив’язаний до об’єкта голови або тулуба охоронця. Коли гравець потрапляє в цей радіус, спрацьовує подія OnTriggerEnter (або аналогічна), але остаточне рішення про “помічання” приймається після перевірки напрямку – чи знаходиться гравець попереду охоронця. Для цього порівнюється вектор напряму від охоронця до гравця з напрямком погляду NPC (forward vector): якщо кут між ними менший за заданий кут огляду, вважаємо, що гравець у полі зору. Додатково виконується **raycast**: випромінюється промінь від охоронця до гравця, і перевіряється, чи не блокується він перешкодою. Таким чином досягається моделювання реального поля зору – охоронець “не бачить” крізь стіни. У разі успішного виявлення система встановлює внутрішній стан NPC “Знайдено ціль”. Це може бути реалізовано як зміна змінної стану у скрипті або виклик методу на кшталт OnPlayerSpotted(), що змінює поведінку.

Переслідування в Unity здійснюється через оновлення цілі NavMeshAgent'а. У скрипті AI передбачено, що у стані "Переслідування" агенту постійно оновлюється пункт призначення – координати поточного місця гравця (наприклад, agent.SetDestination(player.transform.position)). Агент автоматично прораховує найкоротший шлях по сітці NavMesh до рухомого гравця і прямує за ним. Якщо гравець змінив напрям чи швидкість, охоронець завдяки регулярному оновленню цілі коригує свій маршрут. Важливо відзначити, що частота оновлення цілі може бути оптимізована: немає потреби робити це кожен кадр, достатньо декілька разів на секунду, аби зменшити навантаження (Unity NavMeshAgent сам продовжує рух до останньої заданої позиції). У разі, коли гравець зник з поля зору (вийшов з тригер-зони або за перешкоду), система може або негайно припинити погоню, або перейти в режим *"пошуку"*: охоронець останнім зусиллям йде до останнього відомого місця гравця. Це надає поведінці більшої правдоподібності – NPC як би перевіряє, куди зник порушник. Якщо і там гравця не знайдено, охоронець скасовує тривогу і повертається до режиму патрулювання, продовжуючи свій маршрут з найближчої контрольної точки.

Модельована таким чином поведінка агентів у Unity поєднує наперед задані алгоритми (FSM для перемикання станів, A* для навігації) з можливостями ігрового рушія (система NavMesh, компоненти-тригери, фізика для raycast). Проєктування на рівні Unity-сцени дозволило відразу закладати необхідні параметри (розміри зон, швидкості, точки маршруту) і вбудувати їх у середовище, що спростило подальшу реалізацію та налагодження.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1. Створення програмних модулів мовою C#

Практична реалізація здійснювалася мовою C# у вигляді скриптів, прикріплених до об’єктів у Unity. Основними програмними модулями стали скрипти GuardAI (логіка NPC-охоронця) та PlayerController (керування гравцем), а також декілька допоміжних скриптів для зв’язку та UI. Нижче детально розглянуто структуру цих скриптів та їх взаємодію.

Скрипт GuardAI: цей клас відповідає за весь цикл поведінки охоронця. У ньому оголошені необхідні змінні: посилання на компонент NavMeshAgent; масив або список точок патрулювання (Transform[] waypoints); індекс поточної цілі патруля; параметри виявлення гравця (радіус огляду, кут зору); посилання на об’єкт гравця (Transform або посилання на скрипт гравця). Також визначено перелічення станів (enum) або інша структура, що містить можливі стани FSM: наприклад, State { Patrol, Chase, Alert }. Логіка GuardAI побудована таким чином, що різні методи відповідають різним аспектам поведінки:

- У методі Start() скрипт ініціалізує змінні. Наприклад, отримує компонент NavMeshAgent з об’єкта (agent = GetComponent<NavMeshAgent>()), встановлює початковий стан (звичайно Patrol), та відправляє агента до першої точки маршруту (agent.SetDestination(waypoints[0].position)). Можливо, відключається автоблокування (agent.autoBraking = false) для плавного руху між точками. Якщо передбачено, що охоронець самостійно знаходитиме гравця, на цьому етапі також можна знайти об’єкт гравця за тегом (player = GameObject.FindWithTag("Player")).

- Метод Update() викликається кожен кадр і реалізує фінцевий автомат: в залежності від поточного стану виконує відповідні дії та перевіряє умови переходу. Наприклад:
- **Якщо стан Patrol:** скрипт перевіряє відстань до цільової точки патруля (agent.remainingDistance). Коли агент майже досяг точки (менше певного порогу, напр. 0.5 м), викликається метод переходу до наступної точки (GotoNextPoint()), який змінює agent.destination на координати наступного waypoint (при потребі здійснює вибір випадкової точки або циклічно наступної). Також в стані Patrol відбувається перевірка на наявність гравця: метод CheckForPlayer() може виконуватися щокадру чи з певним інтервалом. Цей метод реалізує описану раніше логіку перевірки кута та дистанції (може використовувати Physics.Raycast до гравця). Якщо умови виконані, змінюється стан на Chase (та за потреби викликаються дії ініціації переслідування, наприклад, видається звук або змінна “alerted = true” для анімації).
- **Якщо стан Chase:** скрипт спрямовує охоронця за гравцем. Для цього може використовуватися або безперервне оновлення позиції: agent.SetDestination(player.position) всередині Update, або, більш оптимально, оновлення раз на декілька кадрів через корутину. Паралельно перевіряється, чи не зник гравець: якщо Vector3.Distance(transform.position, player.position) перевищує граничну відстань або CheckForPlayer() повертає false протягом певного часу – вважаємо, що гравця втрачено. Тоді або змінюємо стан на Alert (NPC зупиняється і озирається/шукає кілька секунд), або одразу повертаємо стан Patrol (продовження маршруту з найближчої точки). Якщо ж агент досягне гравця (колайдери зіткнулись) – це може трактуватися як “спіймано” і викликати відповідний інцидент (наприклад, сповістити GameManager або вивести повідомлення).
- **Якщо стан Alert/Search:** (якщо реалізовано) – охоронець рухається до останньої відомої позиції гравця і зупиняється, оглядаючись. Це можна

реалізувати поворотами або просто таймером паузи. Після закінчення часу пошуку NPC повертається в Patrol і продовжує маршрут.

Переключення станів у коді може відбуватися через явні умовні інструкції (if/else або switch(state)) або через подієву модель. В даному проєкті для простоти використано умовні вирази та змінну стану. Втім, структура коду відповідає концепції FSM: у кожному кадрі виконується дія залежно від стану, а рішення про зміну стану приймається на основі умов (наявність гравця, досягнення цілі тощо).

− Метод (або функція) GotoNextPoint() використовується для вибору наступної точки патруля. Він інкрементує індекс точки і, якщо індекс вийшов за межі масиву, обнуляє його (циклічний обхід). Потім задає нову ціль для агента: agent.destination = waypoints[currentIndex].position. Цей метод викликається в кінці Start() для початку руху і потім кожного разу, коли чергова точка досягнута. Таким чином, патрулювання охоронця виконано повністю в автономному режимі.

− Можливі додаткові методи: OnTriggerEnter(Collider other) – якщо використано колайдер на охоронці для виявлення гравця, цей метод може обробляти появу об’єкта з тегом "Player" в зоні, встановлюючи змінну на кшталт playerInRange = true. Аналогічно, OnTriggerExit може позначати вихід гравця із зони. Ці події полегшують визначення моменту виявлення/втрати гравця. У нашій реалізації, ми комбінували ці події з перевіркою видимості, тому OnTriggerEnter лише сигналізує про потенційну ціль, а остаточна перевірка робиться в Update(). Також може бути метод OnDrawGizmos() для відладки – наприклад, малювати в редакторі радіус видимості.

Скрипт PlayerController: забезпечує керування персонажем гравця. Цей скрипт, як правило, простіший. У ньому опрацьовуються події вводу: наприклад, в методі Update() зчитуються осі керування (Input.GetAxis("Horizontal"), Input.GetAxis("Vertical")) або конкретні клавіші (стрілки/WASD для руху). Отримані значення використовуються для зміни положення гравця: через метод CharacterController.Move() або зміну компоненту Rigidbody (в залежності від способу руху). Для плавності руху може застосовуватися нормалізація вектора

напрямку та множення на швидкість і Time.deltaTime. Також скрипт може керувати поворотом камери або моделі гравця у напрямку руху. В нашому випадку достатньо забезпечити рух по площині XZ, щоб гравець міг ходити локацією, уникаючи охоронця. Важливо, що персонаж гравця має колайдер (капсула) для взаємодії з фізичним оточенням і можливо Rigidbody (із замороженою обертанням, щоб не падав), якщо використовується фізичний рух. PlayerController може не взаємодіяти з GuardAI напряму – зв'язок односторонній: охоронець сам "бачить" гравця. Втім, для UI чи менеджера гри PlayerController може повідомляти про деякі дії (наприклад, якщо передбачено натискання клавіші для присідання, щоб зменшити видимість – але це вже розширення, не обов'язкове у даному проєкті).

Зв'язок між скриптами: реалізований здебільшого через **спільні об'єкти сцени** і **події Unity**. Наприклад, GuardAI отримує трансформ гравця через тег або шляхом встановлення посилання в Інспекторі Unity (перетягування об'єкта гравця до поля скрипта). Взаємодія може бути і через глобальні події: можна мати об'єкт GameManager з подією OnPlayerCaught, яку викликає GuardAI при спійманні гравця, і на яку підписаний, скажімо, UI (вивести "Вас спіймано!") та скрипт гравця (зупинити рух). В рамках цієї роботи зроблено спрощено: при спрацюванні події затримання гравця викликається метод у скрипті GameManager, що просто перезавантажує сцену або зупиняє гру.

Використання подій та станів: У місцях, де це доречно, застосовано подійний підхід. Це стосується насамперед колайдерів-тригерів Unity (події OnTriggerEnter/Exit). Такі події зручні, бо не потребують постійного опитування і спрацьовують точно в момент входу або виходу об'єкта. Крім того, внутрішньо в коді GuardAI використано події/делегати для розділення логіки: наприклад, створено делегат OnStateChanged для відлагодження (логування) або для потенційного підключення анімації – коли стан охоронця змінився, метод-обробник може змінити анімаційний стан моделі (наприклад, перехід на біг). Основний акцент, однак, зроблено на **автомат станів**: це дозволило

структурувати код, уникнути хаотичних умовних переходів. FSM-підхід полегшив налагодження, бо в будь-який час NPC чітко перебуває в одному зі станів і реагує лише на визначене коло подій. За потреби додавання нових режимів (наприклад, стан “Спокій” якщо гравець далеко, чи “Атака” якщо NPC озброєний) легко інтегрується через розширення переліку станів і умов переходу.

3.2. Інтеграція та налагодження системи у Unity

Після написання скриптів, всі компоненти були об’єднані та налаштовані в редакторі Unity. Робота над інтеграцією включала створення сцени, розміщення об’єктів, присвоєння скриптів і встановлення параметрів, а також поетапне тестування кожного елемента.

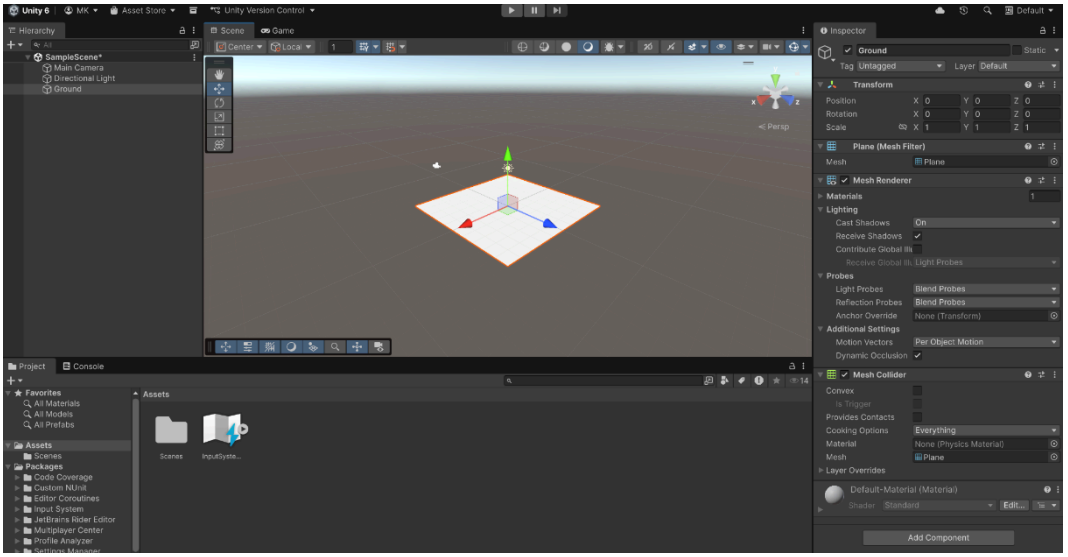

Рис.3.1 Створення базового середовища сцени в Unity (площина Ground та початкові об’єкти).

Налаштування сцени та об’єктів: Створено сцену “Warehouse” (умовна назва) розміром приблизно 20x20 метрів, що містить:

– **Статичне оточення:** кілька кімнат і коридорів, відмежованих стінами (3D-моделі приміщень або просто примітиви типу Cube). Всі стіни і підлога позначені як Static для навігації. На підлогу додано компонент NavMesh Surface, і

через інструмент Bake згенеровано навігаційну сітку – синій полігон, що покриває доступні для ходьби зони. Перевірено, що NavMesh коректно врахував перешкоди: немає “лазів” крізь стіни, сітка цілісна по коридорах. Якщо були виявлені проблеми (наприклад, надто вузький прохід не увійшов у NavMesh), коригувалися параметри генерації (Radius, Height, Step Height) та геометрія сцени.

- **Об’єкт охоронця:** на сцену додано капсулу (Capsule) або 3D-модель персонажа, яка представляє охоронця. Цьому об’єкту призначено компонент **NavMeshAgent** (із заданими швидкістю ~3.5 м/с для патрулювання, 5 м/с – прискорення для погоні, кут огляду повороту ~120°/с тощо) і колайдер (capsule collider) для фізичної присутності. Також додано наш скрипт GuardAI. В Інспекторі для GuardAI налаштовано публічні поля: масив waypoints заповнено перетягуванням до нього раніше створених пустих об’єктів-точок у потрібному порядку; поле радіуса огляду задано, наприклад, 10 метрів; кут огляду – 90°. Поле посилання на гравця встановлено на об’єкт гравця (або, альтернативно, скрипт сам знайде гравця за тегом під час Start). Для індикації зони виявлення навколо охоронця, йому як дочірній об’єкт додано сферу-тригер (Sphere Collider з позначкою **Is Trigger**, радіус відповідає радіусу виявлення). Цей тригер налаштований на шар (layer) “Vision”, який не взаємодіє з колізіями, лише для подій. У скрипті GuardAI підписка на події цього тригера реалізована через методи OnTriggerEnter/Exit. Отже, охоронець на сцені повністю налаштований: маршрут, сенсори, логіка поведінки.

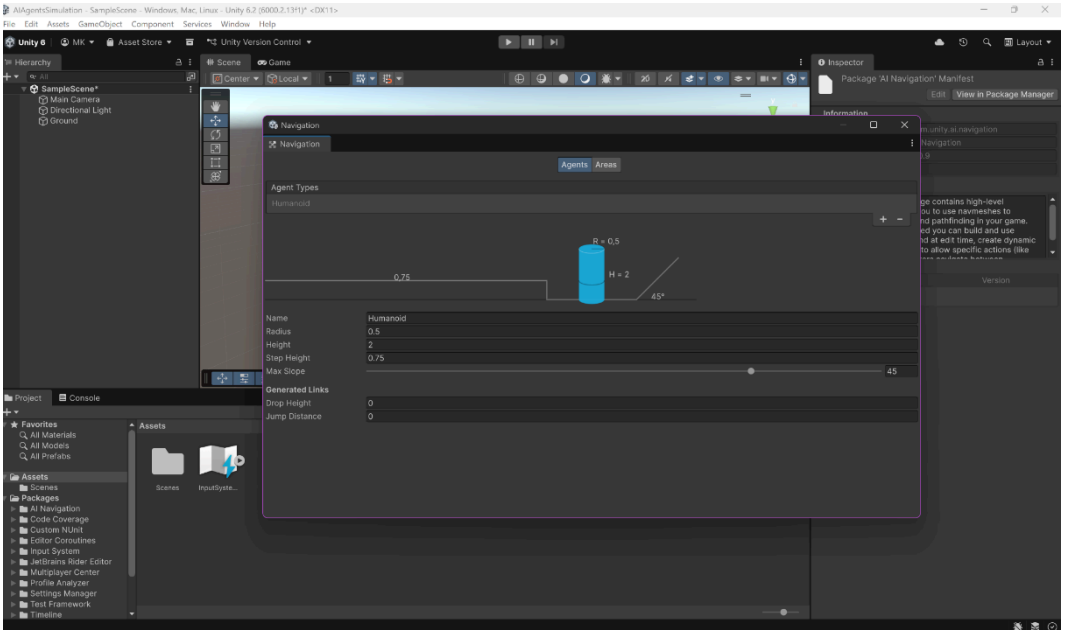

Рис.3.2 Налаштування параметрів агентів у вікні AI Navigation (визначення типів агентів для навігації).

– **Об’єкт гравця:** додано капсулу або модель персонажа на сцену, яку позначено тегом “Player”. Встановлено контролер: або компонент CharacterController, або Rigidbody для фізичного руху, та collider. Додано скрипт PlayerController, в полі якого налаштовано швидкість руху (наприклад, 5 м/с). Камеру сцени прив’язано до гравця (наприклад, як дочірній об’єкт для відстеження зверху/від третьої особи). Перевірено, що гравець може вільно пересуватися по сцені, зіткнення з оточенням коректні (не проходить крізь стіни, але може через дверні прорізи).

– **Менеджер та UI:** додано об’єкт GameManager (порожній Empty з нашим допоміжним скриптом або без скрипта, лише для зручності) з, можливо, компонентом AudioSource для звуків тривоги. На UI створено простий текстовий елемент чи панель, яка сповіщає статус (наприклад, “Вас помічено!”) – цей елемент за замовчуванням прихований. Скрипт GameManager може містити посилання на UI елементи і методи на кшталт ShowAlertMessage() та RestartLevel(). GuardAI при спійманні гравця викликає GameManager.ShowAlertMessage() (через знайдення об’єкта за тегом

“GameController” або передачу посилання), а потім RestartLevel() з затримкою у кілька секунд. Це забезпечує зупинку симуляції і демонстрацію результату.

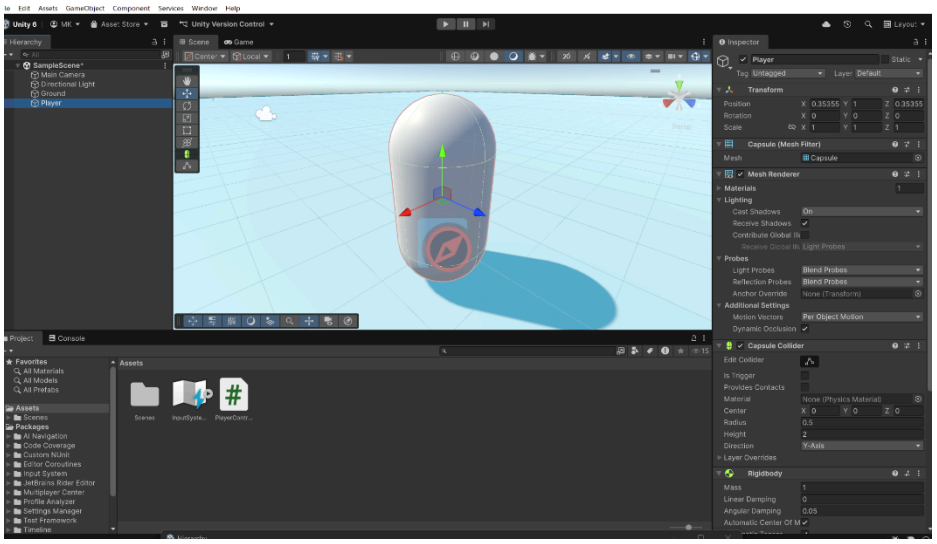

Рис.3.3. Додавання об’єкта Player та налаштування компонентів Capsule Collider і Rigidbody.

Об’єднання компонентів: Після розміщення всіх складових, сцена була запущена для перевірки взаємодії. На цьому етапі виявлялися дрібні нестикування, які усувалися шляхом налаштувань:

– Виправлено шари і теги: переконалися, що гравець має тег “Player” (інакше GuardAI не зможе знайти його чи визначити у тригері). Тригер охоронця розміщено на спеціальному шарі, щоб уникнути небажаних спрацювань з іншими об’єктами.

– Перевірено масштаб: радіус тригера і NavMeshAgent радіус повинні відповідати реальним розмірам приміщення. Спочатку радіус тригера було встановлено 5, але з’ясувалося, що це замало для кімнати 10x10, тому його збільшили до 8 для кращого реагування.

– Налаштовано параметри NavMeshAgent: змінено Stopping Distance до невеликого значення, щоб NPC підходив близько до точки, але не “тремтів” на місці. Також вимкнено Auto Braking для плавного руху між точками патруля (щоб агент не сповільнювався перед кожним waypoint, а рухався безперервно).

- Протестовано лінію видимості: помістили гравця за стіною від охоронця – переконались, що Raycast в CheckForPlayer правильно не знаходить гравця через перешкоду, тож переслідування не ініціюється поки гравець не вийде в прямий коридор.
- Скориговано таймери: у скрипті GuardAI параметр часу пошуку (Alert) встановлено, наприклад, 3 секунди – цього достатньо, щоб гравець міг сховатися далеко, але не настільки довго, щоб гравець просто чекав; пауза на waypoint'ах також підбиралася дослідним шляхом (~1-2 секунди, щоб помітно було зупинку, але не занадто довго).

Налагодження (debugging): У процесі інтеграції активно використовувалися засоби Unity для налагодження:

- **Debug.Log:** додано тимчасові повідомлення в код (наприклад, Debug.Log("Player spotted") при спрацюванні події виявлення, або Debug.Log("Switching to Chase state") при зміні стану). Це допомогло переконатися в правильності логіки – у консолі Unity було видно, коли саме охоронець "бачить" гравця і коли починає переслідування.
- **Gizmos:** для наочності в режимі сцени малювалися Gizmos – сфера радіусу виявлення навколо охоронця, лінія напряму погляду та ін. Це спростило підбір кута огляду та радіуса – по Gizmos було видно, чи охоплює зона потрібну область.
- **Покрокове виконання:** Unity дозволяє зупиняти гру (Pause) і переглядати стан об'єктів. Цим користувалися, щоб зупинити гру в момент погоні і перевірити значення змінних (через інспектор): стан FSM, поточний цільовий waypoint, координати цілі NavMeshAgent тощо.
- **Profiler (профілювання):** Хоча система досить проста, на фінальних етапах було використано профайлер, щоб впевнитися – немає непродуктивного використання ресурсів. Профайлер показав мінімальне навантаження: скрипти GuardAI і PlayerController займають незначний відсоток кадру, а більшість роботи

виконує движок (фізика та рендер). Це підтвердило ефективність обраної архітектури.

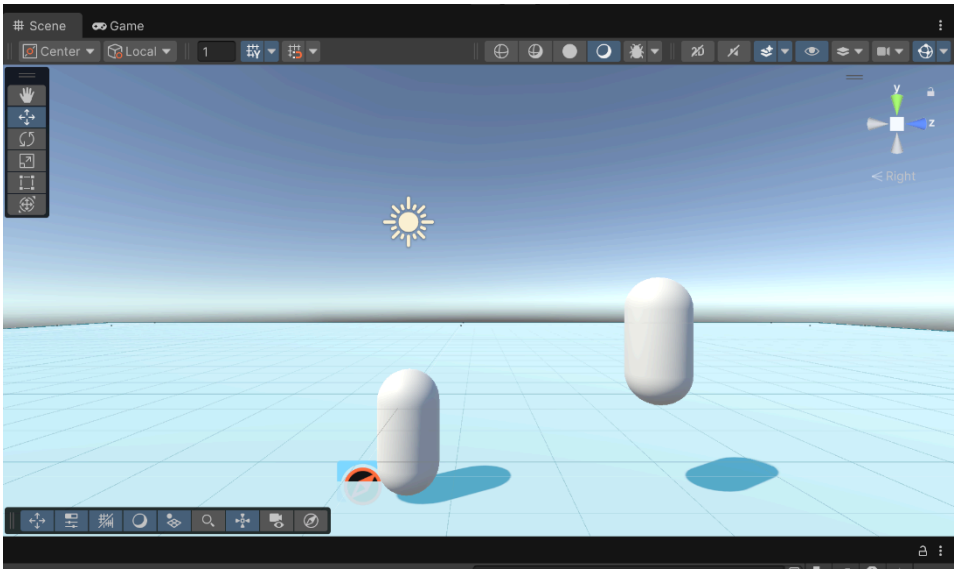

Рис.3.4. Розміщення гравця та AI-агента на сцені, створення патрульних точок.

Завдяки поступовій інтеграції та налагодженню, вдалося добитися узгодженої роботи всіх компонентів. На виході отримано повноцінну симуляцію: охоронець патрулює маршрут, реагує на появу гравця і переслідує його, а користувач може спостерігати або впливати на цю поведінку через керування гравцем.

3.3. Тестування симуляції та аналіз результатів

Після успішного об’єднання системи проведено серію тестувань, щоб переконатися у правильності та стійкості поведінки штучного інтелекту охоронця. Тестування охоплювало як функціональні сценарії, безпосередньо пов’язані з вимогами, так і граничні випадки, що могли виявити недоліки реалізації. Нижче наведено основні тестові сценарії та отримані результати:

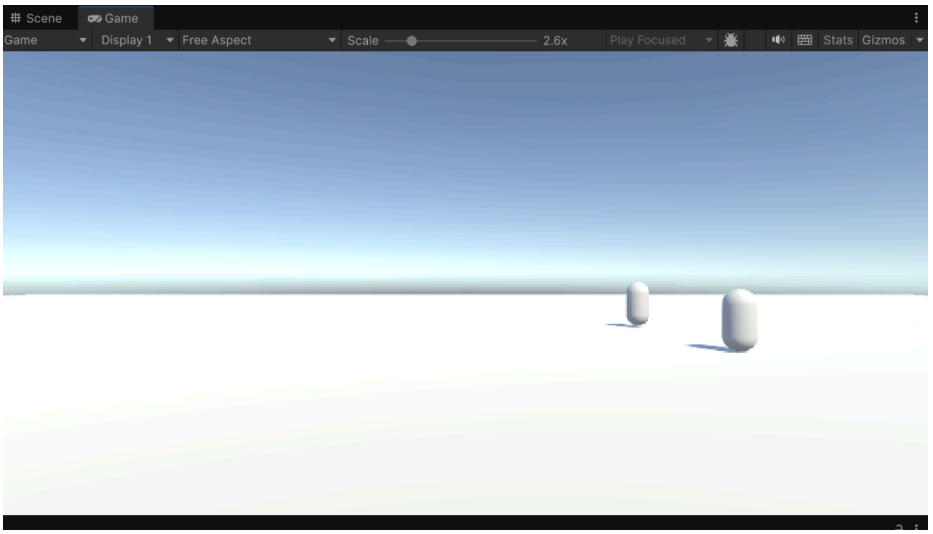

Рис.3.5. Робота симуляції в режимі Play: переміщення гравця та взаємодія з агентом.

- **Тест 1: Патрулювання маршруту.** *Сценарій:* Гравець відсутній у зоні видимості, охоронець безперешкодно ходить за маршрутом. *Очікування:* NPC послідовно обходить усі задані точки, цикл повторюється. *Результат:* Охоронець успішно пройшов маршрут з 3 точок і почав нове коло. Рух був плавним, без різких зупинок перед точками (завдяки вимкненому автогальмуванню). На кінцевих точках патруля спостерігалися короткі зупинки, як і задумано. Поведінка відповідала очікуванням – маршрут охоронця передбачуваний і відповідає заданій послідовності.
- **Тест 2: Виявлення гравця в полі зору.** *Сценарій:* Гравець рухається назустріч охоронцю у коридорі. *Очікування:* Як тільки гравець потрапить в конус зору охоронця, той повинен “помітити” його та перейти до переслідування. *Результат:* Приблизно на відстані ~8 метрів спереду охоронець раптово змінив поведінку: зупинив патрулювання і попрямував прямо до гравця. В консолі Unity з’явилося повідомлення Debug “Player spotted – switching to Chase”, підтвердивши спрацювання умови. Гравець почав тікати, і охоронець одразу прискорився. Цей тест підтвердив коректність модуля виявлення: NPC бачить гравця в межах заданого радіуса і кута, реакція відбувається негайно.

- **Тест 3: Переслідування та навігація.** *Сценарій:* Гравець тікає від охоронця складним маршрутом – оббігає навколо перешкод, заходить у сусідню кімнату. *Очікування:* Охоронець повинен слідувати за гравцем, використовуючи навігацію (обхід стін через двері), і не застрягати. *Результат:* NPC побіг за гравцем: спочатку через той самий дверний проріз, що й гравець, потім гравець зробив різкий поворот за кут. Охоронець трохи скоротив шлях (зрізав кут завдяки навміш-маршруту) і продовжив погоню. Коли гравець петляв між ящиками, охоронець трохи сповільнився, проте з часом знайшов обхід і наблизився. Жодного застрягання або спроб “йти напролом” крізь перешкоди не сталося – алгоритм A* на NavMesh коректно проклав шлях. Через деякий час, коли гравець вибіг на відкритий простір, NPC скоротив дистанцію і наздогнав його. Цей тест підтвердив ефективність навігаційного модуля: охоронець здатний переслідувати рухому ціль у складному оточенні, дотримуючись правил руху.
- **Тест 4: Втрата гравця та повернення до патрулювання.** *Сценарій:* Під час переслідування гравець зачиняє двері або перебігає за перешкоду, так що охоронець втрачає пряму видимість. *Очікування:* Охоронець певний час пошукає гравця і, не знайшовши, відновить патрулювання. *Результат:* Гравець забіг у кімнату та закрив за собою двері (двері позначені як перешкода на NavMesh). Охоронець добіг до дверей, але не міг пройти (двері відсутні як прохід). Він трохи постояв перед дверима (~3 секунди “Alert”), після чого розвернувся і пішов у напрямку найближчої патрульної точки. В консолі з’явилися повідомлення “Lost target – resuming patrol”. Охоронець повернувся на свій маршрут, як і очікувалося. Цей сценарій показав, що FSM логіка працює: по таймауту без контакту з гравцем AI переходить назад у режим патрулювання. Цікаво, що в процесі реалізації цього сценарію було виявлено необхідність: спочатку NPC одразу повертався до патруля без паузи, що виглядало неприродно – додавання короткого стану Alert усунуло цю проблему, зробивши поведінку реалістичнішою.
- **Тест 5: Керування гравцем і взаємодія UI.** *Сценарій:* Користувач свідомо дає спіймати себе охоронцю. *Очікування:* При зіткненні з NPC гра

повинна відреагувати (напр., показати повідомлення або перезапустити). *Результат:* Коли охоронець наздогнав гравця і їх колайдери перетнулися, в консолі з'явився запис "Player Caught". На екрані з'явилося червоне повідомлення "Вас спіймано!", програвся звук сирени. Через 2 секунди сцена перезапустилася (GameManager перезавантажив рівень). Усі ці дії відпрацювали за сценарієм, підтвердивши, що інтеграція з UI і ігровою логікою (GameManager) успішна. Керування гравцем також протестоване окремо – герой може рухатися з різною швидкістю, стрибати (якщо було реалізовано), і камера коректно слідує за ним. Це забезпечує необхідну інтерактивність для користувача.

Аналіз результатів: Результати тестування показали, що розроблена система в основному відповідає всім сформульованим вимогам. Охоронець продемонстрував правдоподібну поведінку: методично патрулює, реагує на гравця в межах досяжності та переслідує, виконуючи пошук та повернення до маршруту при втраті цілі. Навігація за допомогою NavMesh підтвердила свою ефективність – ні в одному тесті NPC не застряг і не поводився нелогічно, шлях завжди був оптимальним чи близьким до оптимального. Алгоритм A* і його похідні, закладені в Unity, показали високу продуктивність: навіть при складній геометрії сцени розрахунок шляху відбувався миттєво, без затримок помітних для гравця. FSM-алгоритм керування станами також підтвердив простоту розширення – додавання стану Alert суттєво покращило реалізм без значної переробки коду.

Виявлені труднощі та вдосконалення: Під час розробки виникли деякі труднощі. Наприклад, налаштування балансу між чутливістю виявлення та фальшивими спрацьовуваннями: спочатку кут зору був занадто великий, і охоронець "бачив" гравця навіть боковим зором, що робило гру дуже складною – цей параметр довелося зменшити. Також зіткнулися з проблемою, що охоронець міг продовжувати бігти за гравцем навіть коли той вже далеко попереду і втік (через те, що в тригері гравець все ще був певний час). Це вирішено коригуванням логіки: якщо NPC біжить понад N секунд без візуального підтвердження цілі, переслідування припиняється. Окрім цього, довелося врахувати колізії між

гравцем і охоронцем – інколи NPC міг зіштовхнути гравця (особливо якщо фізичний рух увімкнено). Вирішено зменшенням маси Rigidbody гравця та/або відключенням фізичного впливу NPC на гравця при контакті.

Оцінка ефективності: Проєкт продемонстрував, що навіть з використанням відносно простих алгоритмів (скінченний автомат, A*) можна досягти переконливої поведінки NPC у реальному часі. Система працює стабільно при 60 FPS, використання CPU мінімальне – основні витрати на рендеринг і фізику, а AI-логіка не створює відчутних навантажень. Це означає, що рішення є масштабованим: можна додати декілька охоронців на сцену, і кожен виконуватиме аналогічну логіку без значного падіння продуктивності (Unity дозволяє це, оскільки навігація оптимізована для багатьох агентів).

В підсумку, тестування підтвердило досягнення поставленої мети. Інтелектуальна система охоронця в Unity відповідає визначеним вимогам і реалізує базовий рівень штучного інтелекту для NPC. Отримані результати можуть бути основою для подальшого розширення – наприклад, додавання більш складних патернів поведінки, анімації персонажів, групової взаємодії кількох охоронців, що стане напрямом розвитку проєкту у майбутньому.

РОЗДІЛ 4. АНАЛІЗ РЕЗУЛЬТАТІВ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

4.1. Оцінка ефективності створеного інструменту

Розроблена система була протестована безпосередньо в середовищі Unity. Під час тестування перевірялися правильність реалізації алгоритмів навігації та поведінки агентів. В експериментах підтверджено, що агенти коректно слідують до призначених цілей по NavMesh без значних помилок чи блокувань. Unity NavMeshAgent використовує дані сітки (NavMesh) для прокладання оптимального шляху і забезпечує надійне рухання персонажів. У разі появи динамічних перешкод на шляху NavMeshObstacle створює «виріз» у сітці, і агенти автоматично змінюють траєкторію – вони оминають об’єкти або шукають інший маршрут. Крім того, в налаштуваннях NavMeshAgent активовано параметр autoRepath, що примусово запускає повторний розрахунок шляху при недоступності попереднього. Це гарантує адаптивність системи: агенти оперативно переплановують рух, якщо первинний маршрут став непрохідним (наприклад, після раптового з’їзду перешкоди).

Усі тестові симуляції продемонстрували стабільність роботи: помилок виконання чи збоїв не виявлено, а агенти весь час залишаються прив’язаними до навігаційної сітки. Рух агентів відбувається плавно, без раптових “стрибків” чи зависань – це підтверджується і візуальними засобами відлагодження. Візуалізація траєкторій (Gizmos) наочно показує, що маршрути завжди актуальні і не виходять за межі NavMesh. Ефективність навігації достатня для розглянутих сценаріїв: у умовах середньої складності сцени (кілька десятків агентів) частота кадрів залишалась прийнятною і обчислення шляхів не створювали помітних затримок. Зауважимо, що в літературних прикладах подібних симуляцій застосування паралельних алгоритмів та оптимізацій дозволяло підтримувати обробку сотень агентів на звичайному апаратному забезпеченні. Отже, розроблена система

демонструє високу стабільність і коректність роботи в рамках поставлених завдань.

4.2. Порівняльний аналіз із наявними рішеннями

Unity NavMeshAgent є стандартним механізмом для навігації персонажів у іграх і симуляціях. Він автоматично прокладає маршрут по створеній NavMesh і надає вбудовані можливості уникнення зіткнень, що робить його простим у застосуванні. Альтернативні рішення включають самостійні алгоритми пошуку шляху (наприклад, A* на графі), де розробник формує власну сітку та обчислення. У кількох відомих проєктах комбінували A*-пошук на регулярній ґратці з локальним алгоритмом уникнення зіткнень (наприклад, ORCA) для забезпечення гнучкості поведінки агентів. Такі системи зазвичай складніші в реалізації, але дозволяють тонше контролювати рух у складних середовищах. Існують також готові рішення (наприклад, A* Pathfinding Project або техніка «flow field»), які спеціально оптимізовані для сотень одночасних агентів. Вони здатні прискорювати пошук шляху в навантажених умовах, проте вимагають окремої інтеграції.

Порівняно з цими підходами, реалізована система (NavMesh + C#-скрипти) відзначається простотою використання та надійністю. Використання NavMeshAgent є типовим галузевим рішенням, яке швидко дозволяє отримати працездатну навігацію. З іншого боку, існуючі дослідження показують, що **Unity ML-Agents** (машинне навчання) можуть давати більш адаптивну і «розумну» поведінку, але потребують суттєвої попередньої підготовки та тривалого тренування. У наведеному дослідженні відзначено, що ML-Agents забезпечують інтеграцію «суперлюдського» AI значно простіше, ніж повна ручна розробка логіки, проте за рахунок складнішої конфігурації. Таким чином, наше рішення є компромісним: воно менш гнучке за підходи з навчанням, але працює «із коробки» для стандартних задач навігації. Загалом реалізація ближча до традиційних

методів у Unity і схожа на аналогічні проєкти, де інтегруються NavMesh для руху і прості сценарії поведінки на C#.

4.3. Можливості подальшого вдосконалення та розширення системи

Перспективною функцією є впровадження багатокористувацького режиму (мережевої симуляції). У цьому випадку потрібно додати мережеві компоненти (наприклад, Unity Netcode, Mirror тощо) для синхронізації стану агентів між клієнтами і сервером. Така розробка дозволить проводити спільні експерименти чи ігри за участю кількох гравців, але водночас спричинить складнішу логіку узгодження та затримок передачі даних.

Розширення логіки поведінки агентів може передбачати впровадження складніших стратегій. Замість простого пересування за точками можна реалізувати **поведінкові дерева** (Behaviour Tree) або системи переходів станів (FSM), які керуватимуть режимами патрулювання, розшуку, атаки тощо. Наприклад, для симуляції бою чи ухиляння можна використати класичні стежні моделі (алгоритми *Seek*, *Flee*, *Pursuit*, *Evade*), відомі з літератури з ІІ. Існують також готові засоби – наприклад, Unity-пакети чи плагіни з Asset Store, які забезпечують візуальне редагування поведінкових дерев і розширених AI. Також можна реалізувати додаткові системи сенсорів (зорове та слухове сприйняття) та кооперації між агентами.

Окремим напрямом є використання **Unity ML-Agents** або інших API для навчання агентів. ML-Agents – це відкритий набір інструментів для навчання інтелектуальних агентів у ігрових і симуляційних середовищах. За допомогою алгоритмів підкріплення агенти можуть самостійно знаходити оптимальні стратегії без явного програмування всіх правил. Дослідження показують, що такий підхід дозволяє значно підвищити «розумність» поведінки: ML-агенти можуть навчитись «суперлюдських» стратегій, які важко реалізувати звичайним кодом. Наприклад, у спрощеній бойовій симуляції агент навчився ефективно

атакувати вогняним шаром для перемоги над ворогом. Із застосуванням машинного навчання можна також використовувати передові підходи (напр. GAIL, curiosity-driven learning) та паралельно тренувати агентів на клауд-інстансах. Крім того, технологія Unity DOTS (Data-Oriented Tech Stack) може бути задіяна для оптимізації масштабованості: вона призначена для високопродуктивної обробки великої кількості об’єктів в реальному часі. У цілому, наведені напрямки (мережевий режим, розширені моделі поведінки, ML Agents) дозволять зробити систему більш гнучкою та функціонально багатою, відповідаючи сучасним вимогам симуляційних середовищ.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано науково-прикладну задачу розроблення інтелектуальної системи симуляції поведінки ігрових агентів у реальному часі на базі ігрового рушія Unity з використанням мови програмування C#. Сформульована мета – створити та експериментально дослідити інструмент, який забезпечує патрулювання, переслідування гравця та адаптацію агентів до змін середовища, – досягнута шляхом послідовного виконання теоретичних, проєктних і практичних етапів роботи.

На першому етапі проведено теоретичний аналіз предметної області. Узагальнено підходи до класифікації ігрових агентів, визначено основні типи поведінки, що реалізуються в комп’ютерних іграх, і окреслено місце інтелектуальних систем у сучасних ігрових застосунках. Розглянуто методи моделювання поведінки агентів – від простих правил і скінченних автоматів станів до поведінкових дерев та алгоритмів машинного навчання. Проаналізовано алгоритми прийняття рішень та навігації в реальному часі, зокрема пошук шляху на основі навігаційних сіток. Окрему увагу приділено можливостям сучасних ігрових рушіїв, насамперед Unity, а також особливостям мови C# у контексті реалізації штучного інтелекту й інтеграції з вбудованими засобами навігації NavMesh. Здійснено огляд існуючих інструментів моделювання поведінки агентів, що дозволило виявити їх сильні сторони та обмеження.

На другому етапі було виконано проєктування інтелектуальної системи. Сформульовано постановку задачі та визначено вимоги до розроблюваного інструменту: підтримка патрулювання визначеним маршрутом, виявлення гравця в зоні видимості, переслідування з урахуванням перешкод, повернення до базового режиму при втраті цілі, а також інтерактивна взаємодія в режимі реального часу. Обґрунтовано архітектуру системи із чітким поділом на модуль навігації, модуль виявлення гравця, модуль керування поведінкою агентів та модуль керування гравцем. Для реалізації логіки поведінки охоронця обрано скінченний автомат

станів, що дозволяє прозоро описати переходи між станами патрулювання, переслідування та пошуку. Запроєктовано модель поведінки агентів у середовищі Unity із використанням NavMesh для навігації, контрольних точок патрулювання та сенсорів виявлення гравця.

На третьому етапі здійснено практичну реалізацію системи в середовищі Unity. Створено тривимірну сцену з прохідними та непрохідними зонами, згенеровано навігаційну сітку NavMesh. Розроблено програмні модулі мовою C#: скрипт керування гравцем, який забезпечує переміщення персонажа сценою, та скрипт штучного інтелекту охоронця, що реалізує патрулювання між заданими точками, виявлення гравця на основі відстані й кута огляду, зміну станів та переслідування цілі із використанням компонента NavMeshAgent. Виконано інтеграцію скриптів з елементами сцени, налаштовано параметри агентів, сенсорів, таймінгів і логіки переходу між станами.

Проведено тестування розробленої симуляції за низкою сценаріїв: патрулювання без гравця, наближення гравця до охоронця, переслідування з обходом перешкод, втрата цілі та повернення до маршруту, навмисне потрапляння “в руки” охоронця. Результати випробувань засвідчили стабільну роботу системи: агенти коректно рухаються по NavMesh, адекватно реагують на появу гравця, не застрягають на перешкодах і відновлюють базовий режим роботи після завершення погоні. Проаналізовано ефективність побудованого інструменту та встановлено, що використання стандартних засобів Unity у поєднанні з власними C#-модулями дозволяє реалізувати правдоподібну базову поведінку ігрових агентів без надмірних витрат ресурсів.

У четвертому розділі виконано узагальнений аналіз отриманих результатів та окреслено перспективи розвитку системи. Порівняння з іншими підходами показало, що обрана архітектура на базі NavMesh та FSM займає проміжне місце між простими скриптовими рішеннями й складними системами на основі навчання з підкріпленням. Запропоновано напрями подальшого вдосконалення інструменту: розширення набору станів і поведінкових сценаріїв, запровадження

кооперації кількох агентів, реалізація складніших сенсорних моделей, інтеграція механізмів Unity ML-Agents для навчання агентів на основі досвіду, а також підтримка багатокористувацьких та мережевих сценаріїв.

Узагальнюючи, можна зробити висновок, що поставлена в роботі мета досягнута, усі основні завдання, сформульовані у вступі, послідовно розв’язані. На теоретичному рівні систематизовано підходи до побудови інтелектуальних систем у комп’ютерних іграх та визначено вимоги до поведінки ігрових агентів у режимі реального часу. На проєктному рівні сформовано архітектуру інтелектуальної системи та обґрунтовано вибір засобів реалізації. На практичному рівні створено працездатний інструмент симуляції поведінки агентів у Unity, проведено його тестування і сформульовано висновки щодо ефективності й напрямів розвитку.

Результати роботи мають як теоретичне, так і прикладне значення. Теоретична частина може бути використана як основа для подальших досліджень у галузі ігрового штучного інтелекту, а розроблений програмний інструмент – як навчальний та демонстраційний приклад для дисциплін, пов’язаних із розробленням комп’ютерних ігор, моделюванням систем та програмуванням у середовищі Unity.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Вачнадзе, А. М. О. (2020). *Розробка FPS гри на ігровому рушії Unity* [Thesis, Чернігів]. Електронний архів Чернігівського національного технологічного університету (IRChNUT). http://ir.stu.cn.ua/123456789/23439

2. Григоров, О. В., Аніщенко, Г. О., Стрижак, В. В., Петренко, Н. О., Турчин, О. В., Окунь, А. О., & Пономарьов, О. Е. (2019). Artificial intelligence. Machine learning. *Vehicle and Electronics. Innovative Technologies*, (15), 17. https://doi.org/10.30977/veit.2226-9266.2019.15.0.17

3. Журавчак, Л. М., & Кашевка, В. В. (2024). Інтелектуальна комп'ютерна рольова гра на ігровому рушії Unreal Engine 5. *Scientific Bulletin of UNFU*, *34*(8). https://doi.org/10.36930/40340814

4. Ситніков, Д., & Єлтишев, П. (2024). Розробка гри на ігровому рушії unity. In *Радіоелектроніка та молодь у XXI столітті. Т. 6 : Конференція "Інформаційні інтелектуальні системи"*. Press of the Kharkiv National University of Radioelectronics. https://doi.org/10.30837/iyf.iis.2024.861

5. Терещук, С. О., Панаріна, І. В., & Вольський, Р. А. (2023). Побудова ігрового інтелекту за допомогою патерну State в ігровому рушії Unity. *Технічна інженерія*, (2(92)), 166–173. https://doi.org/10.26642/ten-2023-2(92)-166-173

6. Яцик, М., & Обершев, В. (2024). Розробка гри на ігровому рушії unreal engine 5 за допомогою плагіну gameplay ability system. In *Радіоелектроніка та молодь у XXI столітті. Т. 4 : Конференція "Перспективи розвитку інфокомунікацій та інформаційно-вимірювальних технологій"*. Press of the Kharkiv National University of Radioelectronics. https://doi.org/10.30837/iyf.pdicimt.2024.142

7. Addoum, M. A., Mekhaemar, J., Rouffet, M., & Jacopin, E. (2021). Khaldun: GOAP for both Procedural Level generation and NPC Behaviors. In *2021 IEEE conference on games (cog)*. IEEE. https://doi.org/10.1109/cog52621.2021.9619062

8. Agent-Based simulation. (2014). In *Discrete and continuous simulation* (pp. 255–276). CRC Press. https://doi.org/10.1201/b17127-17

9. Alaliyat, S., Yndestad, H., & Sanfilippo, F. (2014). Optimisation of boids swarm model based on genetic algorithm and particle swarm optimisation algorithm (comparative study). In *28th conference on modelling and simulation*. ECMS. https://doi.org/10.7148/2014-0643

10. Angelides, M. C., & Paul, R. J. (n.d.). Towards a framework for integrating intelligent tutoring systems and gaming simulation. In *1993 winter simulation conference - (WSC '93)*. IEEE. https://doi.org/10.1109/wsc.1993.718392

11. Barlow, A. (2011). Web technologies and supply chains. In *Supply chain management - new perspectives*. InTech. https://doi.org/10.5772/23018

12. Beri, R., & Behal, V. (2015). Cloud computing: A survey on cloud computing. *International Journal of Computer Applications*, *111*(16), 19–22. https://doi.org/10.5120/19622-1385

13. Buxton, G. (1975). Warehouse management and materials handling. In *Effective marketing logistics* (pp. 125–144). Palgrave Macmillan UK. https://doi.org/10.1007/978-1-349-02101-7_6

14. Chayalakshmi, C. L., Kakkasageri, M. S., Pujar, R. S., & Hegde, N. (2023). IoT sensors for smart automation. In *AI and blockchain applications in industrial robotics* (pp. 141–170). IGI Global. https://doi.org/10.4018/979-8-3693-0659-8.ch006

15. Choi, J., Shin, D., & Shin, D. (2006). Research on design and implementation of adaptive physics game agent for 3D physics game. In *Agent computing and multi-agent systems* (pp. 614–619). Springer Berlin Heidelberg. https://doi.org/10.1007/11802372_68

16. Condò, M. (2024). Sb-goap. In *Game AI uncovered* (pp. 175–185). CRC Press. https://doi.org/10.1201/9781003323549-21

17. Costa, C. J. (2019). ERP – enterprise resource planning. *OAE – Organizational Architect and Engineer Journal*. https://doi.org/10.21428/544e68e8.1b5cd585

18. Engelbrecht, D. (2023). Unity ML-Agents. In *Introduction to Unity ML-Agents* (pp. 87–135). Apress. https://doi.org/10.1007/978-1-4842-8998-3_6

19. Feriyadi, N. (2018). The augmented reality 3D object augmented reality. *KOPERTIP : Jurnal Ilmiah Manajemen Informatika dan Komputer*, *2*(2), 76–83. https://doi.org/10.32485/kopertip.v2i2.46

20. Frazelle, E. (n.d.). Design problems in automated warehousing. In *1986 IEEE international conference on robotics and automation*. Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/robot.1986.1087686

21. Game theory. (2021). In *Self-Organising Multi-Agent Systems* (pp. 71–99). WORLD SCIENTIFIC (EUROPE). https://doi.org/10.1142/9781800610439_0003

22. Kambhampati, S. (1995). AI planning. *ACM Computing Surveys*, *27*(3), 334–336. https://doi.org/10.1145/212094.212118

23. Kamir, D., & Diskin, S. (2023). Artificial intelligence and machine learning. *Medical Writing*, 2–4. https://doi.org/10.56012/vrpa5453

24. Lawlor, B. (2021). Artificial intelligence and machine learning. *Chemistry International*, *43*(1), 8–13. https://doi.org/10.1515/ci-2021-0103

25. Lin, Z., & Chen, G. (2020). Multi-agent planing based on causal graph. *Journal of Physics: Conference Series*, *1601*, 042019. https://doi.org/10.1088/1742-6596/1601/4/042019

26. Luis, G., Suárez, D., & Mateos, A. J. (2018). Multi-Agent word guessing game. *Adcaij: Advances in Distributed Computing and Artificial Intelligence Journal*, *7*(4), 17. https://doi.org/10.14201/adcaij2018741726

27. McCallum, E. B., Yarbrough, J. L., & Schmitt, A. J. (2021). Game-Based cooperative learning. In T. A. Collins & R. O. Hawkins (Eds.), *Peers as change agents* (pp. 76–83). Oxford University Press. https://doi.org/10.1093/med-psych/9780190068714.003.0007

28. Milov, O., Kostyak, M., Milevsky, S., & Pogasiy, S. (2019). Засоби моделювання поведінки агентів в інформаційно-комунікаційних система. *Системи*

управління, навігації та зв'язку. Збірник наукових праць, *6*(58), 63–70. https://doi.org/10.26906/sunz.2019.6.063

29. Nandy, A., & Biswas, M. (2018). Unity ML-Agents. In *Neural networks in unity* (pp. 27–67). Apress. https://doi.org/10.1007/978-1-4842-3673-4_2

30. Principal-Agent models. (2009). In *Game theory evolving* (pp. 162–178). Princeton University Press. https://doi.org/10.2307/j.ctvcm4gjh.10

31. Ratajczak-Ropel, E. (2017). Agent-Based optimization. In *Population-Based Approaches to the Resource-Constrained and Discrete-Continuous Scheduling* (pp. 7–23). Springer International Publishing. https://doi.org/10.1007/978-3-319-62893-6_2

32. Raut, U., Galchhaniya, P., Nehete, A., Shinde, R., & Bhoite, A. (2024). Unity ml-agents: Revolutionizing gaming through reinforcement learning. In *2024 2nd world conference on communication & computing (WCONF)* (pp. 1–7). IEEE. https://doi.org/10.1109/wconf61366.2024.10692314

33. Rugaber, S., Goel, A. K., & Martie, L. (2013). GAIA: A CAD environment for model-based adaptation of game-playing software agents. *Procedia Computer Science*, *16*, 29–38. https://doi.org/10.1016/j.procs.2013.01.004

34. Tan, C., & Cheng, H.-l. (2021). Personality-Based adaptation for teamwork in game agents. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *3*(1), 37–42. https://doi.org/10.1609/aiide.v3i1.18779

35. Ting, T. O., Wong, K. P., & Chung, C. Y. (2008). Hybrid constrained genetic algorithm/particle swarm optimisation load flow algorithm. *IET Generation, Transmission & Distribution*, *2*(6), 800. https://doi.org/10.1049/iet-gtd:20070224

36. Wang, X. (2012). Improvement and implementation of Boids cluster animation rules. In *2012 7th international conference on system of systems engineering (sose)*. IEEE. https://doi.org/10.1109/sysose.2012.6333534

37. Yoo, H.-J., Lee, M.-J., & Kim, K.-N. (2010). Flocking implementation for NPC AI. *Journal of the Korea Academia-Industrial Cooperation Society*, *11*(12), 5083–5088. https://doi.org/10.5762/kais.2010.11.12.5083

38. Young, J. (2024, April 29). *NPC AI planning with GOAP | excaliburjs*. Blog. https://excaliburjs.com/blog/goal-oriented-action-planning/

39. Zheng, S., He, K., Yang, L., & Xiong, J. (2024). MemoryRepository for AI NPC. *IEEE Access*, 1. https://doi.org/10.1109/access.2024.3393485

40. Zhong, Y., Ning, J., & Zhang, H. (2012). Multi-agent simulated annealing algorithm based on particle swarm optimisation algorithm. *International Journal of Computer Applications in Technology*, *43*(4), 335. https://doi.org/10.1504/ijcat.2012.047158

Тема роботи:

Інтелектуальна система симуляції поведінки ігрових агентів у реальному часі в ігровому рушії Unity

АКТУАЛЬНІСТЬ

Актуальність теми зумовлена зростаючими вимогами до правдоподібної поведінки ігрових агентів у комп'ютерних іграх та симуляторах. Попри широке використання Unity, у навчальній практиці бракує цілісних, методично завершених прикладів побудови інтелектуальних систем для моделювання поведінки агентів. Створення доступного, модульного інструменту симуляції є важливим для підготовки фахівців у галузі розробки ігор та інформаційних технологій, що особливо актуально для динамічного розвитку IT-сектору України.

НАУКОВА НОВИЗНА

1. Удосконалено методичний підхід до побудови навчальних симуляцій через структуровану архітектуру з універсальними C#-модулями та параметричним налаштуванням сценаріїв.

2. Удосконалено класифікацію ігрових агентів за критеріями рівня автономності, ролі у грі та методу прийняття рішень, що на відміну від існуючих підходів забезпечує систематизований вибір архітектури ШІ та спрощує проєктування поведінкових моделей для навчальних симуляцій.

Метою кваліфікаційної роботи є розроблення та дослідження інтелектуальної системи симуляції поведінки ігрових агентів у реальному часі в середовищі Unity з використанням мови програмування C#.

Об'єктом дослідження є процес моделювання та симуляції поведінки ігрових агентів у реальному часі в тривимірному віртуальному середовищі.

Предметом дослідження є методи, програмні засоби та архітектурні рішення побудови інтелектуальної системи симуляції поведінки агентів у рушії Unity з використанням NavMesh і C#-скриптів.

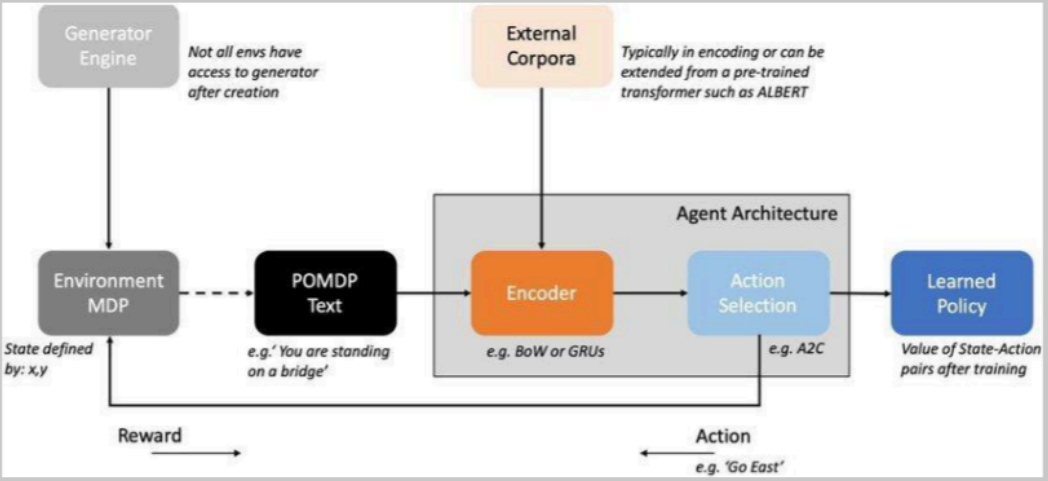

Рис.1.1. Загальна структура ігрового агента: агент сприймає стан середовища, обробляє інформацію та відправляє дії у світ гри.

Завдання дослідження

У межах роботи було проаналізовано теоретичні основи побудови інтелектуальних систем у комп'ютерних іграх, розглянуто підходи до моделювання поведінки агентів і алгоритми прийняття рішень. Також було досліджено можливості рушія Unity та особливості використання мови C# для реалізації ігрового штучного інтелекту, спроєктовано архітектуру системи, реалізовано програмні модулі та проведено тестування симуляції

У теоретичній частині роботи розглянуто поняття ігрових агентів, їх класифікацію та основні підходи до моделювання поведінки. Проаналізовано алгоритми прийняття рішень і навігації в реальному часі, а також можливості сучасних ігрових рушіїв. Окрему увагу приділено особливостям використання мови C# у середовищі Unity для реалізації штучного інтелекту.

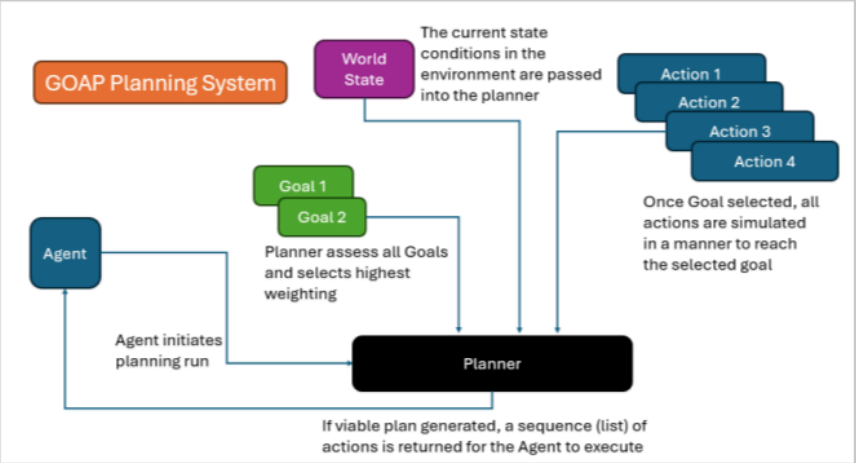

Рис.1.3. Схема роботи GOAP-системи планування дій ігрового агента (Goal-Oriented Action Planning)

Класифікація ігрових агентів

Критерій	Тип агента	Короткий опис	Приклад гри (агент у ній)
Рівень автономності	Пасивний агент	Не приймає власних рішень, реагує тільки на зовнішні тригери	Статичні об’єкти і пастки (Tower Defense тощо)
	Реактивний агент	Швидка реакція на події за фіксованими правилами	Вартові-вороги у Skyrim (пристосовують поведінку за шаблоном)
	Когнітивний агент	Містить модель світу, планує дії, може навчатися	Приватні компаньйони у RPG із адаптивними діалогами
Роль у грі	Дружній NPC (супутник)	Допомагає або супроводжує гравця	Лідія, супутниця в *Skyrim*
	Ворожий NPC (противник)	Перешкоджає гравцю, атакує або конфронтує	Бандити і дракони у *Skyrim*, поліція в *GTA*
	Нейтральний NPC	Не проявляє агресію до гравця (або поводиться залежно від контексту)	Торговці і мирні жителі у *Skyrim*; перехожі в *GTA*
Метод рішення	На правилах (rule-based)	Рішення заздалегідь задані у вигляді правил (детерміновані)	Ворожі NPC зі статичними стратегіями у старих іграх
	Сценарні агенти (scripted)	Підпорядковані сценарію, виконують фіксовану послідовність дій	Сцени появи босів, діалогові NPC у сюжетних квестах
	На базі ШІ (AI-based)	Використовують алгоритми ШІ (пошук, дерева поведінки) для динамічних рішень	NPC із деревами поведінки в Unreal Engine; вороги з адаптивним AI
	Навчені (Learning)	Застосовують машинне навчання (підкріплення, нейромережі) для покращення поведінки	Боти із RL (наприклад, OpenAI Five у Dota2) або системи із навчанням у пісочницях

Розроблена інтелектуальна система має модульну архітектуру, що забезпечує чіткий поділ логіки поведінки агентів і механізмів навігації. Прийняття рішень здійснюється на основі поточного стану середовища, а переміщення агентів реалізується через систему NavMesh. Такий підхід забезпечує стабільну роботу симуляції та можливість подальшого розширення системи.

```
C#
using UnityEngine;
using UnityEngine.AI;


public class AgentPatrol : MonoBehaviour
{
    public Transform waypoints;
    private int destPoint = 0;
    private NavMeshAgent agent;

    void Start()
    {
        agent = GetComponent<NavMeshAgent>();

        // Вимикаємо авто-гальмування для плавного переходу між точками
        agent.autoBraking = false;

        GotoNextPoint();
    }

    void GotoNextPoint()
    {
        if (waypoints.Length == 0) return;

        // Встановлюємо нову ціль
        agent.destination = waypoints[destPoint].position; [62]

        // Оновлюємо індекс для наступної точки (циклічно)
        destPoint = (destPoint + 1) % waypoints.Length; [62]
    }

    // Цей метод буде викликатися з Дерева Поведінки
    public void UpdatePatrol()
    {
        // Перевіряємо, чи агент досяг точки (з урахуванням stoppingDistance)
        if (!agent.pathPending && agent.remainingDistance < 0.5f)
        {
            GotoNextPoint();
        }
    }
}
```

Рис.2.5. C#-скрипт патрулювання NavMeshAgent з використанням набору точок і циклічного обходу для вузла дерева поведінки

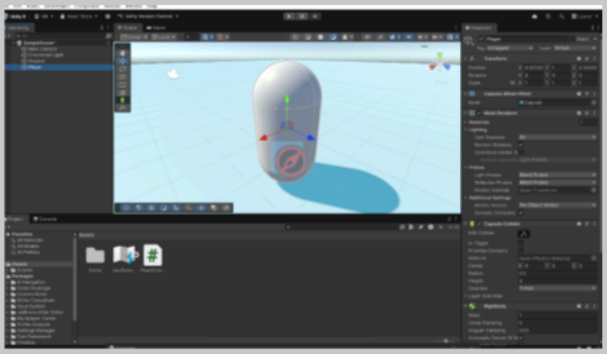

Рис.3.3. Додавання об’єкта Player та налаштування компонентів Capsule Collider і Rigidbody.

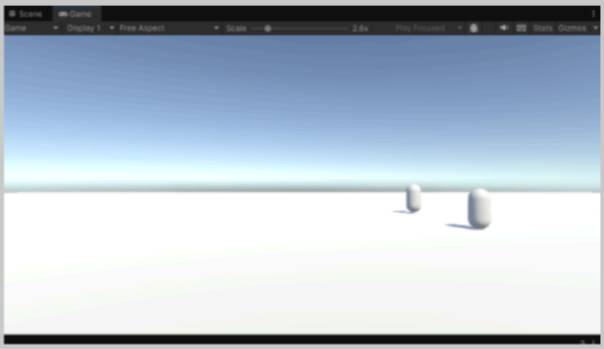

Рис.3.5. Робота симуляції в режимі Play: переміщення гравця та взаємодія з агентом.

Практична частина роботи включає створення сцени в Unity, налаштування навігаційної сітки та розроблення програмних модулів мовою C#. Реалізовано поведінку агентів, що включає патрулювання, виявлення гравця та переслідування. Система інтегрована з Unity AI Navigation і демонструє коректну роботу в режимі реального часу.

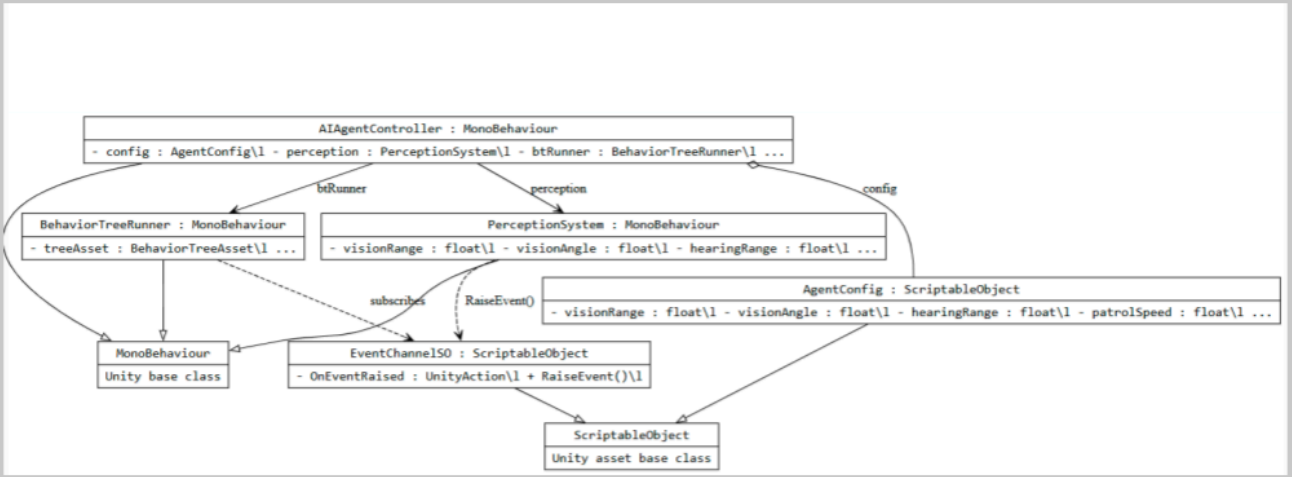

Рисунок 2.1. UML-діаграма класів архітектури агента

Проведене тестування показало коректну роботу розробленої системи за різних сценаріїв. У ході дослідження виявлено типові проблеми, пов’язані з налаштуванням навігації та логікою переходів між станами поведінки. Запропоновано рекомендації щодо їх усунення шляхом модульної організації коду, параметричного налаштування агентів і систематичного тестування сцен.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було створено інтелектуальну систему симуляції поведінки ігрових агентів у Unity, яка поєднує теоретичні підходи та практичну реалізацію. Розроблений інструмент може використовуватися в навчальних і демонстраційних проєктах, а також слугувати основою для подальшого розвитку, зокрема шляхом ускладнення моделей поведінки та використання Unity ML-Agents.

ДЯКУЮ ЗА УВАГУ