Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

РОЗРОБЛЕННЯ АДАПТИВНОГО ШТУЧНОГО ІНТЕЛЕКТУ NPC ДЛЯ ТАКТИЧНИХ СИМУЛЯТОРІВ З ВИКОРИСТАННЯМ МЕТОДУ GOAP
(тема)

Виконав:
здобувач 4 року навчання,
групи ІТІНФ-22-3
Веліканов О.Р.
(прізвище, ініціали)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник проф. Шафроненко А.Ю.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти перший (бакалаврський)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Веліканов Олександр Романович
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення адаптивного штучного інтелекту NPC для тактичних симуляторів з використанням методу GOAP

затверджена наказом університету від 26 травня 2026 року № 435Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 19 червня 2026 р.
3. Вихідні дані до роботи науково-методична та науково-технічна література, матеріали конференцій, дані інтернет-мережі, ігровий рушій Unity; мова програмування C#; алгоритм евристичного пошуку *A**; підсистеми просторової навігації (NavMesh) та фізичного моделювання середовища.

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз сучасного стану систем штучного інтелекту для ігрових NPC (FSM, дерева поведінки) та обґрунтування переваг методу GOAP для тактичних симуляторів CQB.
2. Розроблення модульної архітектури системи ШІ та алгоритмічної моделі зворотного планування у просторі станів за допомогою алгоритму *A**.
3. Програмна реалізація сенсорного модуля, підсистеми командної координації та пошуку тактичних позицій (укриттів) у середовищі Unity та дослідження їх обчислювальної ефективності.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) граф зворотного пошуку плану дій за методом GOAP, структурно-функціональну схему системи штучного інтелекту, узагальнену діаграму класів системи GOAP (UML), блок-схему роботи планувальника, графіки аналізу продуктивності та обчислювальної складності алгоритму пошуку.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	13.04.2026	
2	Аналіз завдання, підбір літератури	14.04.26-18.04.26	
3	Аналіз літератури з проблеми, що вирішується	18.04.26-19.04.26	
4	Аналіз існуючих методів розпізнавання	26.04.26-27.04.26	
5	Формування кроків вирішення проблеми	28.04.26-05.05.26	
6	Програмна реалізація	30.04.26-20.05.26	
7	Аналіз отриманих результатів	21.05.26-04.06.26	
8	Оформлення пояснювальної записки	05.06.26-09.06.26	
9	Перевірка на нормоконтроль	10.06.26-11.06.26	
10	Перевірка на плагіат	11.06.26-12.06.26	
11	Рецензування	20.06.26-24.06.26	
12	Підготовка презентації та доповіді	25.06.26-28.06.26	
13	Занесення роботи в електронний архів	30.06.26	
14	Попередній захист кваліфікаційної роботи	30.06.26	

Дата видачі завдання 13 квітня 2026 р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. Шафроненко А.Ю.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 75 с., 13 табл., 21 рис., 31 джерело.

ШТУЧНИЙ ІНТЕЛЕКТ, NPC, GOAP, ПЛАНУВАННЯ ДІЙ, ТАКТИЧНИЙ СИМУЛЯТОР, UNITY, C#, АЛГОРИТМ *A**.

Об’єктом роботи є система інтелектуальної поведінки неігрових персонажів (NPC) у тривимірному тактичному середовищі.

Метою роботи є розроблення адаптивної системи штучного інтелекту NPC для тактичних симуляторів на основі архітектури GOAP (Goal-Oriented Action Planning), яка забезпечить динамічне планування поведінки агентів у реальному часі без жорсткого сценарного програмування.

Під час роботи використано: методи штучного інтелекту та планування (алгоритм *A**, пошук у просторі станів, багатоцільове планування), методи розробки ігрового програмного забезпечення (модульна архітектура, компонентно-орієнтоване програмування в Unity).

Практична цінність полягає у розробці прототипу тактичного симулятора з інтелектуальними агентами, здатними до адаптивної тактичної поведінки: флангових обходів, використання укриттів, вогню на придушення та координованих командних дій.

ARTIFICIAL INTELLIGENCE, NPC, GOAP, ACTION PLANNING, TACTICAL SIMULATOR, UNITY, C#, *A** ALGORITHM.

The object of the work is the intelligent behavior system of non-player characters (NPCs) in a 3D tactical environment.

The aim of the work is to develop an adaptive NPC artificial intelligence system for tactical simulators based on the GOAP (Goal-Oriented Action Planning) architecture, which will ensure dynamic, real-time planning of agent behavior without rigid scripted programming.

During the work, the following were used: artificial intelligence and planning methods (*A** algorithm, state-space search, multi-goal planning), and game software development methods (modular architecture, component-based programming in Unity).

The practical value lies in the development of a tactical simulator prototype featuring intelligent agents capable of adaptive tactical behavior: flanking maneuvers, utilizing cover, suppressive fire, and coordinated team actions.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 7
Вступ 8
1 Аналіз сучасного стану систем штучного інтелекту для ігрових NPC та постановка задачі 10
1.1 Класифікація та аналіз підходів до розробки ШІ NPC 10
1.2 GOAP: принципи роботи та переваги над FSM 14
1.3 Огляд існуючих реалізацій GOAP у комерційних та академічних проєктах 17
1.4 Аналіз вимог до тактичних симуляторів і ігрового ШІ 19
1.5 Обґрунтування вибору інструментальних засобів 20
1.6 Постановка задачі 22
2 Архітектура та алгоритмічна модель системи GOAP 24
2.1 Загальна модульна структура системи штучного інтелекту 24
2.2 Модель планування у просторі станів та алгоритм *A** 28
2.3 Проєктування сенсорного модуля 32
2.4 Проєктування бібліотеки атомарних дій 34
2.5 Підсистема пошуку укриттів і тактичних позицій 37
2.6 Принципи координації командної взаємодії агентів 39
2.7 Моделювання системи слуху та короткочасної пам’яті 40
2.8 Інтеграція анімацій з планувальником поведінки 41
3 Програмна реалізація та функціональне тестування системи 42
3.1 Обґрунтування вибору середовища та інструментальних засобів реалізації 42
3.2 Загальна організація програмного коду проєкту 44
3.3 Реалізація моделі стану світу 46
3.4 Реалізація сенсорного модуля 48
3.5 Реалізація планувальника на основі алгоритму *A** 50
3.6 Реалізація бібліотеки атомарних дій 53

3.7 Реалізація динамічної системи цілей 57
3.8 Реалізація підсистеми пошуку укриттів 58
3.9 Реалізація підсистеми командної координації 60
3.10 Реалізація координуючого агента 61
3.11 Інтеграція алгоритмічного ядра GOAP зі збройовою підсистемою та інвентарем агента 64
3.12 Управління просторовим переміщенням агентів на базі навігаційної сітки 65
3.13 Оптимізація обчислювального навантаження та керування пам’яттю в системі ШІ 66
3.14 Інтеграція компонентів та цикл прийняття рішень 68
Висновки 70
Перелік джерел посилання 72

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

*A** – Алгоритм евристичного пошуку у графі

API – Application Programming Interface (програмний інтерфейс застосунку)

BT – Behavior Tree (дерево поведінки)

CQB – Close Quarters Battle (бойові дії у замкнених просторових середовищах)

CPU – Central Processing Unit (центральний процесор)

FSM – Finite State Machine (кінцевий автомат)

FPS – Frames Per Second (кадрів за секунду)

GOAP – Goal-Oriented Action Planning (планування дій, орієнтоване на досягнення цілей)

HFSM – Hierarchical Finite State Machine (ієрархічний кінцевий автомат)

IDE – Integrated Development Environment (інтегроване середовище розробки)

LoS – Line of Sight (лінія прямої видимості)

ML – Machine Learning (машинне навчання)

NavMesh – Navigation Mesh (навігаційна сітка для пошуку шляху)

NPC – Non-Player Character (неігровий персонаж)

ООП – Об’єктно-орієнтоване програмування

STRIPS – Stanford Research Institute Problem Solver (класична парадигма планування)

UML – Unified Modeling Language (уніфікована мова моделювання)

ШІ – Штучний інтелект

ВСТУП

Сучасна ігрова індустрія – один із найбільших і найбільш динамічних секторів цифрової економіки, що за обсягом валової виручки давно перевершив кінематограф та музичну індустрію разом узяті. За даними аналітичних звітів компаній Newzoo та Statista, у 2024 році глобальний ринок відеоігор сягнув позначки 188 млрд доларів США [1], причому значну частку зростання забезпечують саме жанри тактичних шутерів, командних мережевих ігор і реалістичних бойових симуляторів. Спільною рисою цих жанрів є висока вимогливість гравців до якості штучного інтелекту неігрових персонажів (NPC), що нерідко стає визначальним чинником комерційного успіху продукту.

Альтернативу класичним підходам пропонує метод GOAP (Goal-Oriented Action Planning), який бере свій початок ще з класичної системи STRIPS, розробленої у 1971 році [2]. Перша комерційна реалізація GOAP у грі F.E.A.R. (Monolith Productions, 2005) [3] продемонструвала, що відносно простий механізм автоматичного планування здатний породжувати вражаючі тактичні рішення: координовані атаки, фланкові обходи, відхід під вогневим прикриттям тощо. Усі ці прояви поведінки виникали без явного сценарного програмування, виключно як наслідок взаємодії атомарних дій та системи цілей агента. Понад двадцять років по тому метод GOAP залишається предметом активного академічного й інженерного інтересу, регулярно з’являючись на конференціях GDC, AIIDE та у фахових публікаціях.

Актуальність обраної теми зумовлена кількома факторами. По-перше, попри тривалу історію методу, відкритих, добре задокументованих реалізацій GOAP для сучасних ігрових рушіїв катастрофічно мало: більшість комерційних рішень є пропрієтарними, а доступні відкриті реалізації застаріли або обмежені у функціональності. По-друге, специфіка тактичних симуляторів CQB висуває особливі вимоги до системи планування, які не покриваються класичними реалізаціями: необхідність роботи з тривимірною геометрією

приміщень, миттєвої реакції на втрату лінії прямої видимості, координації командних дій. По-третє, розвиток сучасних ігрових рушіїв (зокрема Unity з компонентно-орієнтованою архітектурою) створює сприятливі умови для модульної інтеграції алгоритмічних рішень планування у високопродуктивний ігровий цикл.

Практична цінність роботи полягає у створенні відкритої, модульної та добре задокументованої архітектурної основи для розробки інтелектуальних NPC у тактичних симуляторах та подібних застосунках. Отриманий програмний продукт може бути використаний як основа для комерційних ігрових проєктів, навчальних симуляторів для силових структур, а також як методологічна платформа для подальшої роботи у галузі мультиагентних систем планування.

1 АНАЛІЗ СУЧАСНОГО СТАНУ СИСТЕМ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ІГРОВИХ NPC ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Класифікація та аналіз підходів до розробки ШІ NPC

Еволюція систем штучного інтелекту для неігрових персонажів у відеоіграх пройшла кілька ключових етапів, від простих детермінованих алгоритмів на основі кінцевих автоматів до складних систем, здатних до динамічного планування та координованої командної поведінки. Узагальнена хронологія розвитку технологій ігрового ШІ наведена на рисунку 1.1. Для визначення напрямів власної роботи необхідно провести детальний аналіз найпоширеніших підходів, що використовуються в сучасній ігровій індустрії, з’ окресленням їх сильних і слабких сторін.

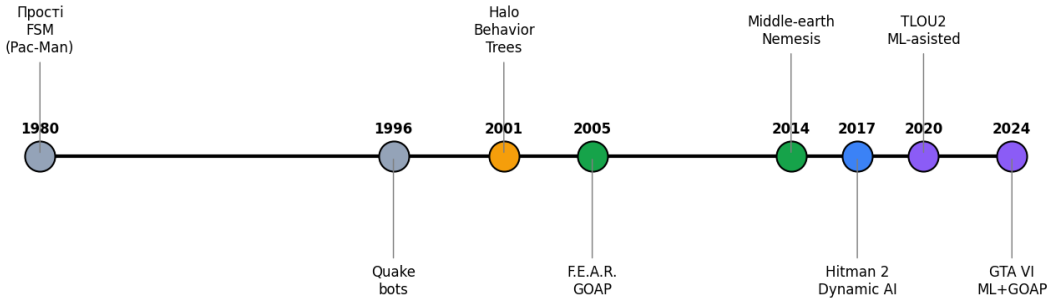

Рисунок 1.1 – Еволюція технологій штучного інтелекту в ігровій індустрії

Кінцевий автомат (Finite State Machine, FSM) є найбільш традиційним та широко вживаним підходом до реалізації ігрового ШІ. Принцип його роботи полягає у визначенні скінченного набору станів (наприклад, «патрулювання», «переслідування», «атака», «відступ», «укриття», «перезарядка») та умов переходу між ними. Агент у кожен момент часу перебуває рівно в одному стані

та реагує на зовнішні події відповідно до заздалегідь визначених правил. Загальна графічна модель такого FSM для базової поведінки тактичного NPC наведена на рисунку 1.2.

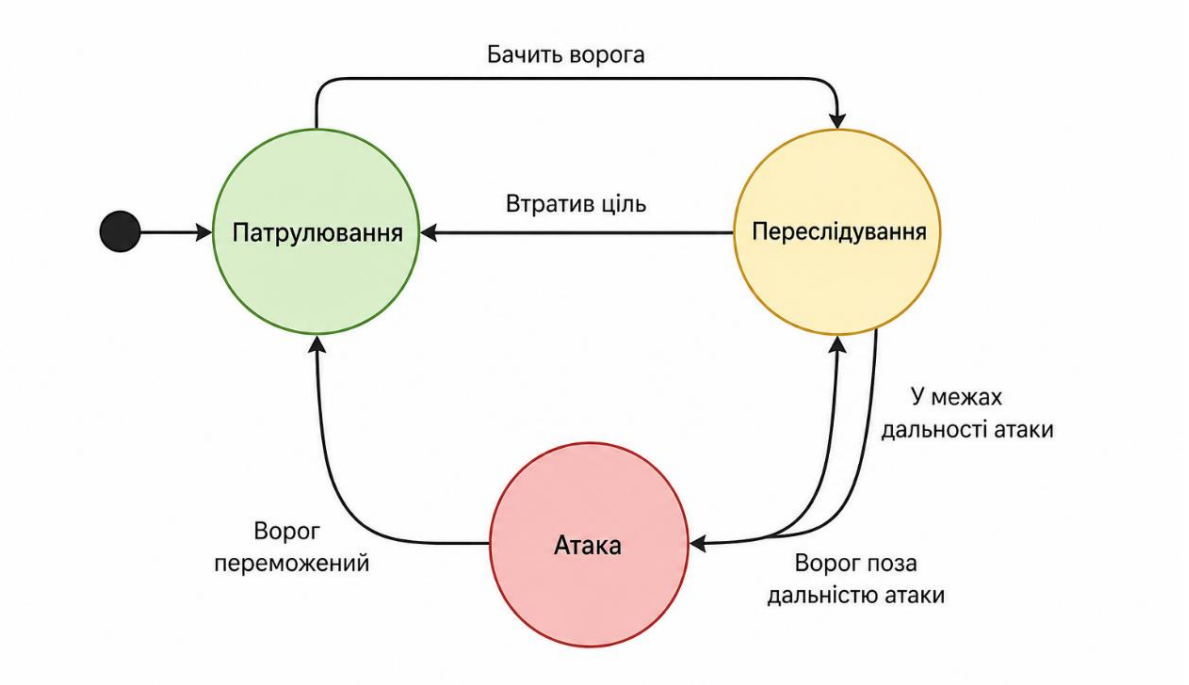

Рисунок 1.2 – Схема кінцевого автомата (FSM) для базової поведінки NPC

До переваг FSM відносять: простоту реалізації навіть для розробників-початківців; повну прозорість логіки роботи, що спрощує налагодження; незначні обчислювальні витрати, які дозволяють підтримувати десятки агентів одночасно з мінімальним впливом на FPS; зручність формальної верифікації поведінки за допомогою класичних методів теорії автоматів. Завдяки цим перевагам FSM було покладено в основу ШІ численних класичних відеоігор, від Pac-Man (1980) до серій Counter-Strike, Quake та Half-Life.

Однак FSM має принципове обмеження – так званий комбінаторний вибух кількості станів і переходів при ускладненні поведінки. Для тактичних сценаріїв, де агент повинен одночасно враховувати рівень здоров’я, наявність союзників, позицію противника, запас боєприпасів, геометрію рівня та погодні умови, кількість необхідних станів зростає експоненційно, що робить підтримку та розширення такої системи практично неможливими. Якщо для

опису простої лінійної поведінки достатньо 4–6 станів, то для повноцінного тактичного ШІ їх кількість може сягати кількох сотень, а перевірка коректності переходів стає окремою інженерною проблемою.

Дерева поведінки (Behavior Tree, BT) набули широкого поширення в ігровій індустрії протягом 2000-х років як більш гнучка альтернатива FSM. BT являє собою ієрархічну деревоподібну структуру, у якій листові вузли є атомарними діями або умовами, а внутрішні вузли визначають логіку управління: послідовне виконання (Sequence), вибір першої придатної гілки (Selector), паралельне виконання декількох гілок (Parallel) та модифікатори (Decorators). Узагальнена структура типового дерева поведінки для тактичного NPC наведена на рисунку 1.3.

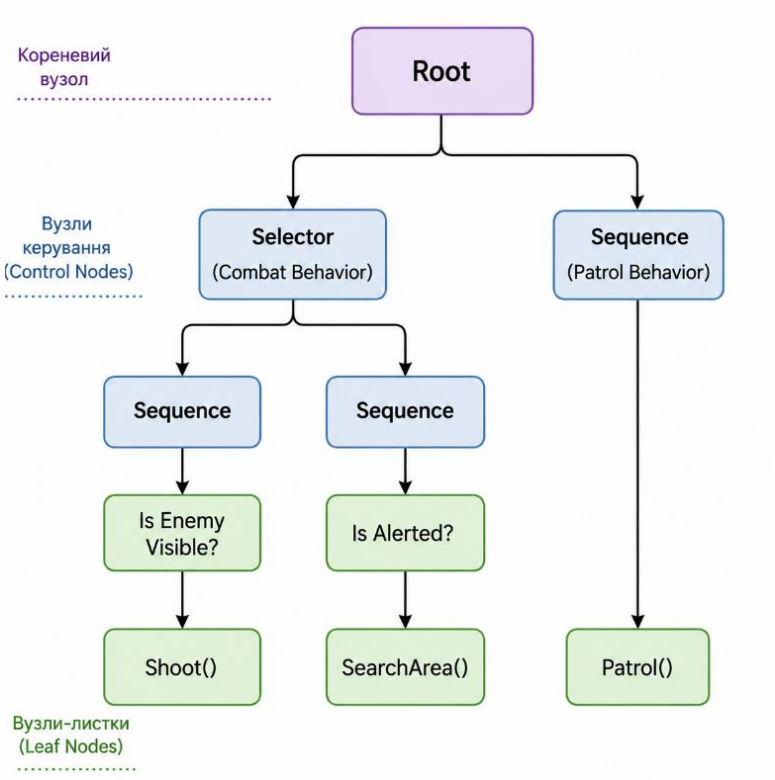

Рисунок 1.3 – Ієрархічна структура дерева поведінки (Behavior Tree)

Такий підхід дозволяє більш природно описувати складну поведінку та легше модифікувати окремі гілки дерева без перепроєктування всієї системи. Кожна гілка є самодостатньою одиницею поведінки, що інкапсулює умови запуску та послідовність дій. Дерева поведінки активно використовуються у

таких відомих іграх, як Halo 2, Halo 3, серія The Last of Us, Tom Clancy's The Division. Завдяки можливості візуального редагування у спеціалізованих інструментах (Behavior Designer для Unity, AI Behavior Tree Designer для Unreal Engine) BT стали де-факто стандартом для розробників середнього рівня.

Незважаючи на свою гнучкість, дерева поведінки також мають суттєве обмеження: кожна гілка дерева, що відповідає певному тактичному сценарію, повинна бути явно запрограмована розробником. Це означає, що агент не здатний самостійно знаходити нові способи досягнення мети в непередбачених ситуаціях – він лише виконує наявні сценарії. При зіткненні з ситуацією, що не передбачена деревом поведінки, NPC або «застигає», або повертається до найбільш пріоритетної гілки, що часто виглядає неприродньо. Окрім того, при великих за обсягом BT (понад 200–300 вузлів) їх читабельність і підтримуваність також помітно знижуються.

Для кількісної оцінки відмінностей між підходами доцільно провести експертну оцінку основних характеристик за десятибальною шкалою. Результати такого порівняння для трьох основних підходів – FSM, BT та GOAP – наведені на рисунку 1.4 та в таблиці 1.1.

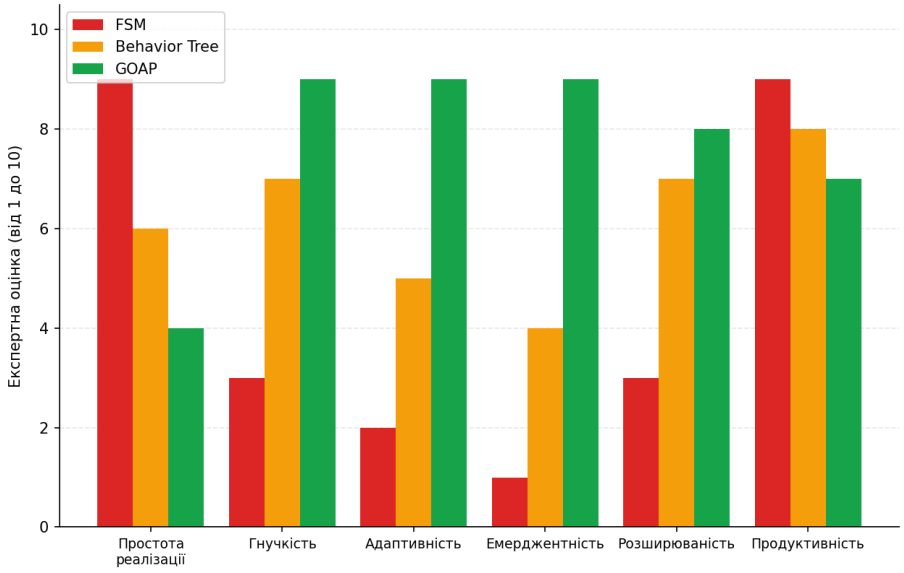

Рисунок 1.4 – Порівняльна оцінка підходів до реалізації ШІ NPC

Таблиця 1.1 – Порівняльна характеристика підходів до реалізації ШІ NPC

Характеристика	**FSM**	**Behavior Tree**	**GOAP**
Простота реалізації	Висока	Середня	Низька
Час розробки	Невеликий	Середній	Великий
Адаптивність	Низька	Середня	Висока
Емерджентність	Відсутня	Обмежена	Висока
Обчислювальні витрати	Мінімальні	Невеликі	Помірні
Підтримуваність коду	Низька	Висока	Висока
Розширюваність	Низька	Середня	Висока

Наведені дані переконливо свідчать, що метод GOAP найбільш повно відповідає вимогам тактичних симуляторів, де ключовими є адаптивність поведінки та здатність агента самостійно знаходити оптимальні тактичні рішення в непередбачених ситуаціях. Платою за ці переваги є більш висока складність початкової реалізації та відносно більші обчислювальні витрати, які, втім, можна суттєво зменшити за допомогою методів кешування та евристичних оптимізацій (детально розглянуті у підрозділі 2.7 цієї пояснювальної записки).

1.2 GOAP: принципи роботи та переваги над FSM

Метод GOAP (Goal-Oriented Action Planning) – це підхід до планування поведінки агентів, що базується на принципах класичного STRIPS-планування, розробленого Файксом і Нільсоном у Стенфордському дослідницькому інституті ще у 1971 році. Сама ідея автоматичного планування у штучному інтелекті виникла значно раніше, на ранніх етапах розвитку галузі, у роботах Маккарті, Ньюела, Саймона. Однак саме комбінація

трьох елементів – формального опису дій через передумови та ефекти, евристичного пошуку у просторі станів і динамічної системи цілей – зробила можливим практичне застосування цих ідей у комерційних ігрових проєктах.

На відміну від FSM та BT, GOAP не вимагає від розробника явного визначення всіх можливих сценаріїв поведінки. Замість цього розробник визначає набір атомарних дій (MoveToEnemy, Attack, Reload, TakeCover, FlankAction, SuppressFire тощо), кожна з яких описується трьома атрибутами: передумовами виконання (Preconditions), ефектами (Effects) та вартістю виконання (Cost). Передумови описують стан світу, який має бути виконаний для запуску дії, ефекти описують зміни стану світу після завершення дії, а вартість виступає метрикою для порівняння альтернативних планів планувальником.

Центральним компонентом системи є планувальник (GOAP Planner), який виконує пошук оптимальної послідовності дій, що переводить поточний стан світу до бажаного (цільового) стану. Цей пошук здійснюється не у фізичному просторі рівня, як у задачах патфайндингу, а у просторі станів – абстрактному графі, де вузли відповідають можливим логічним станам системи, а ребра відповідають можливим діям з певною вартістю виконання. Загальна схема такого графа наведена на рисунку 1.5.

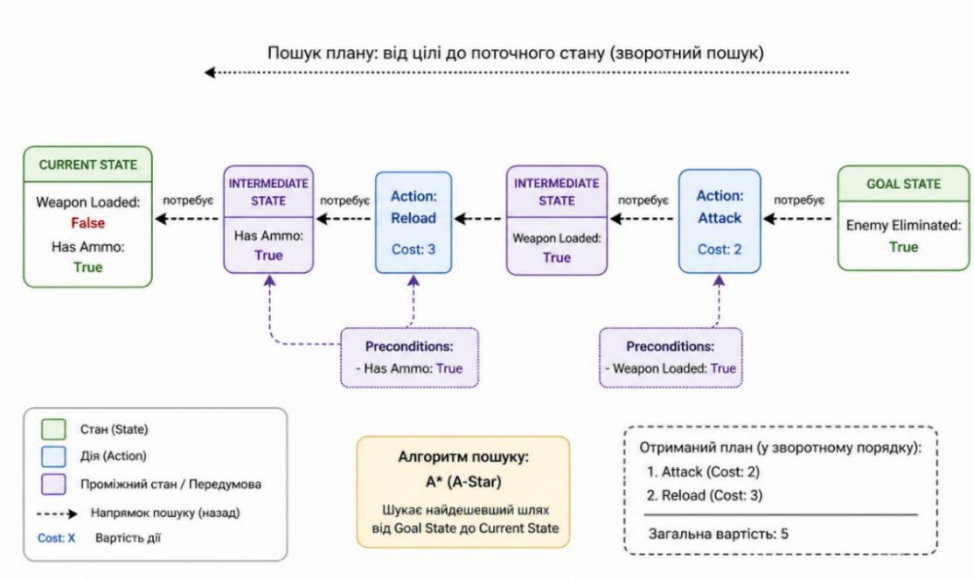

Рисунок 1.5 – Граф зворотного пошуку плану дій за методом GOAP

Алгоритм пошуку, як правило, реалізується на основі *A** або алгоритму Дейкстри. Планувальник виконує пошук у зворотному напрямку: від цільового стану до поточного, підбираючи на кожному кроці дії, ефекти яких відповідають передумовам попередньо запланованих дій. Такий підхід має кілька важливих переваг порівняно з прямим пошуком: по-перше, фактор галуження у просторі станів агента, як правило, менший саме у напрямку до цільового стану; по-друге, зворотний пошук природним чином зосереджується на цілеспрямованих діях, відсікаючи нерелевантні гілки; по-третє, такий підхід дозволяє планувальнику знаходити непередбачені розробником комбінації дій, що саме і породжує емерджентну, непередбачувану поведінку агентів.

Прикладом класичної реалізації GOAP у комерційній грі є шутер F.E.A.R. (Monolith Productions, 2005), де агенти демонстрували вражаюче тактичне мислення: займали фланкові позиції, використовували координовані атаки та відступали під вогневим прикриттям. Ці результати були досягнуті без жодного явного сценарного програмування, виключно через систему цілей та атомарних дій GOAP. Більш сучасні приклади включають систему Nemesis у Middle-earth: Shadow of Mordor (Monolith, 2014), деякі механізми ШІ в Tomb Raider (2013) та експериментальну реалізацію в Deus Ex: Mankind Divided (2016).

Порівняно з BT, метод GOAP вимагає дещо більших обчислювальних витрат на фазі планування, проте на практиці ці витрати є цілком прийнятними завдяки кешуванню планів і виконанню повторного планування лише при суттєвій зміні стану світу. На середньому ігровому обладнанні витрати на планування для агента із бібліотекою з 5–10 дій становлять у середньому 0,2–0,5 мс на запит, що становить менше 3 % від тривалості одного ігрового кадру за 60 FPS і дозволяє підтримувати до 20–30 агентів одночасно на одному CPU-ядрі.

Основна перевага GOAP полягає у тому, що розробник описує «що може робити агент» та «чого він хоче досягти», а система сама визначає «як саме»

це зробити оптимальним чином у кожній конкретній ситуації. Така парадигма називається декларативною (на відміну від імперативної у FSM/BT) і має низку важливих наслідків для процесу розробки:

– розширення поведінки агента відбувається через додавання нових атомарних дій або цілей, а не через переписування громіздких автоматів;

– агент здатний знаходити нові тактичні комбінації, не передбачені розробником, що подовжує життєвий цикл гри та підвищує реіграбельність;

– команда розробників може працювати паралельно: дизайнери описують цілі та поведінкові правила, програмісти імплементують атомарні дії;

– забезпечується принципова можливість автоматичної оптимізації поведінки через коригування вартостей дій (наприклад, методами машинного навчання);

– логіка планування є чітко окресленою алгоритмічною компонентою, що піддається формальному аналізу та модульному тестуванню.

1.3 Огляд існуючих реалізацій GOAP у комерційних та академічних проєктах

Метод GOAP має тривалу та насичену історію практичних реалізацій, які корисно розглянути для розуміння еволюції підходу та визначення перспективних напрямів власної роботи. Такий ретроспективний аналіз дозволяє не лише простежити тенденції розвитку систем планування, але й виявити «вузькі місця» класичних підходів. Здобутий досвід критично важливий для формування обґрунтованих вимог до створюваного ШІ, щоб уникнути типових архітектурних проблем. Систематизований огляд відомих реалізацій GOAP наведено у таблиці 1.2 із зазначенням року, технологічної основи, ключових інновацій та виявлених обмежень.

Таблиця 1.2 – Хронологія ключових реалізацій GOAP

Рік	**Проєкт**	**Платформа**	**Ключова інновація**
2005	F.E.A.R.	Lithtech (C++)	Перша комерційна реалізація, координовані тактичні дії
2008	F.E.A.R. 2	Lithtech (C++)	Розширена бібліотека дій (понад 30 атомарних дій)
2011	Brink	idTech 4	Поєднання GOAP з системою класів і ролей
2013	Tomb Raider	Foundation	GOAP для емерджентної поведінки тварин
2014	Middle-earth	LithTech Firebird	Інтеграція GOAP з системою Nemesis
2016	Deus Ex MD	Dawn Engine	GOAP для NPC у відкритому світі
2019	ReGoap (OSS)	Unity / C#	Перша широко поширена відкрита реалізація для Unity
2022	CrashGoap	Unity / C#	Асинхронне планування, інтеграція з ECS
2024	GOAP.NET	Unity / Unreal	Узагальнена крос-рушійна реалізація

Як видно з таблиці 1.2, основним мейнстрімом протягом останнього десятиліття стала міграція реалізацій GOAP з закритих комерційних рушіїв до відкритих платформ Unity та Unreal Engine. Це створює сприятливі умови для подальшої уніфікації підходів та активної академічної роботи у відповідній галузі. Серед відкритих реалізацій особливо помітне зростання популярності проєкту ReGoap (Luca Tomei, 2017) [8], який нараховує понад 1500 зірок на GitHub та використовується у численних інді-проєктах.

1.4 Аналіз вимог до тактичних симуляторів і ігрового ШІ

Тактичний симулятор ближнього бою (Close Quarters Battle, CQB) висуває до системи штучного інтелекту особливі вимоги, які суттєво відрізняються від вимог до інших жанрів. Якщо у класичному шутері або у real-time стратегії агенти оперують переважно у відкритому просторі з добре передбачуваною геометрією, то у CQB-сценаріях – як правило, всередині приміщень, у вузьких коридорах, на сходах – геометрія є складною, а тактичні рішення мають прийматися у фракції секунди. Типова конфігурація такого сценарію зображена на рисунку 1.6.

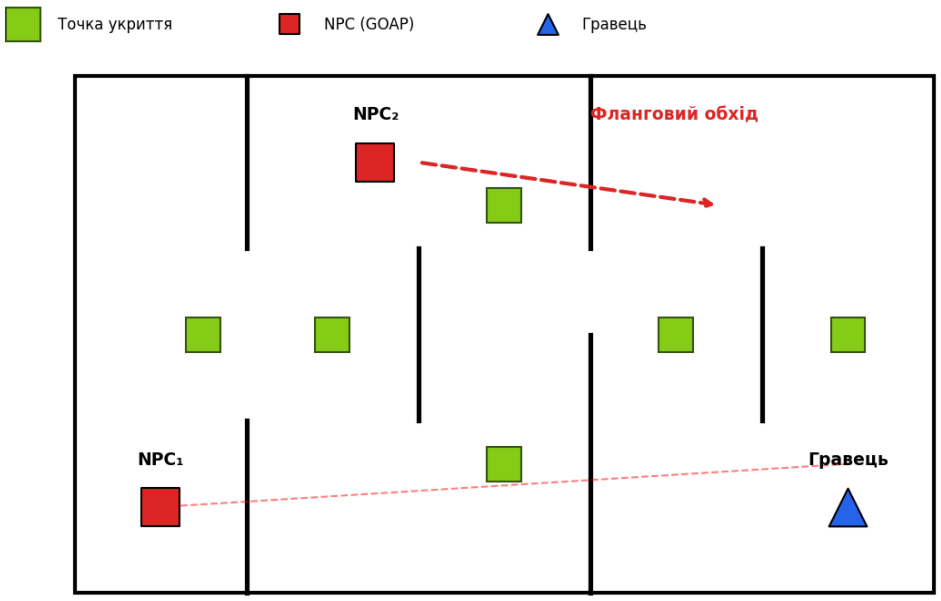

Рисунок 1.6 – Типовий CQB-сценарій тактичного симулятора

Аналіз сучасних комерційних CQB-симуляторів (Rainbow Six Siege, Ready or Not, Ground Branch, SWAT 4) дозволяє виокремити наступні специфічні вимоги до системи ШІ:

– миттєва реакція на втрату лінії прямої видимості: при появі агента на відкритій ділянці супротивник повинен реагувати протягом 200–300 мс, що приблизно відповідає часу реакції людини;

– тактично грамотний вибір укриття з урахуванням позиції загрози: укриття не повинно бути «прозорим» для вогню супротивника, а також повинно забезпечувати можливість швидкого переходу до сусідніх позицій;

– координація командної поведінки: типовий CQB-сценарій передбачає атаку груп з 2–5 агентів, які повинні взаємодоповнювати один одного (наприклад, один прикриває вогнем, інший здійснює фланговий маневр);

– адаптація до дій гравця: ШІ не повинен дотримуватися жорстко заданих сценаріїв, оскільки досвідчені гравці швидко вчаться їх передбачати, що знижує ігрову цінність;

– реалізм поведінки під вогнем: агенти повинні демонструвати «природну» реакцію на отримання шкоди (укриття, виклик підкріплення), а не продовжувати атаку до повного знищення;

– врахування обмежень спорядження: запас боєприпасів, час перезарядки, спеціальні засоби (гранати, флешбенги) повинні бути інтегровані у логіку планування.

Жоден із класичних підходів (FSM, BT) не може задовольнити цей перелік вимог у повному обсязі без значного ускладнення логіки. Натомість метод GOAP природним чином підходить для розв’язання таких задач: динамічна система цілей забезпечує реакцію на зміни ситуації, евристичний пошук у просторі станів породжує оптимальні тактичні рішення, а декларативний опис дій спрощує опис специфіки спорядження та обмежень. Виходячи з цього, для розробки тактичного симулятора CQB було обрано саме GOAP як архітектурну основу системи штучного інтелекту.

1.5 Обґрунтування вибору інструментальних засобів

Для реалізації системи GOAP-планування у тактичному симуляторі необхідно обрати ігровий рушій та мову програмування, що забезпечать

зручну архітектурну основу, достатню продуктивність і широкі можливості для тестування та відлагодження.

Ігровий рушій Unity було обрано як основну платформу розробки. Unity посідає одне з провідних місць серед рушіїв у сфері незалежної та напівпрофесійної розробки ігор, підтримує кросплатформне розгортання (Windows, Linux, macOS, консолі, мобільні пристрої) та надає розгалужену екосистему інструментів для роботи з тривимірним середовищем. Вбудована компонентна архітектура Unity (Entity-Component system) добре узгоджується з модульним підходом до реалізації GOAP, де кожен модуль (Perception, Planner, Actions) може бути реалізований як окремий компонент MonoBehaviour. Зведений порівняльний аналіз основних ігрових рушіїв стосовно задачі розробки тактичного симулятора наведено у таблиці 1.3.

Таблиця 1.3 – Порівняльний аналіз ігрових рушіїв для тактичного симулятора

Критерій	**Unity**	**Unreal Engine**	**Godot**	**Custom (C++)**
Мова скриптів	C#	C++ / Blueprints	GDScript / C#	C++
Поріг входження	Низький	Високий	Низький	Дуже високий
Документація	Дуже повна	Повна	Достатня	Власна
ECS / DOTS	Так (DOTS)	Частково	Ні	За потреби
NavMesh	Вбудований	Вбудований	Базовий	Ні
Сторонні бібліотеки GOAP	Багато (ReGoap)	Декілька	Поодинокі	Власна реалізація

Отже, обраний стек Unity + C# + Rider дає оптимальне поєднання зручності розробки, продуктивності й візуальних інструментів для тестування,

що є критично важливим при проєктуванні та відлагодженні поведінки інтелектуальних агентів.

1.6 Постановка задачі

Аналіз сучасного стану ігрового ШІ, проведений у попередніх підрозділах, виявив незаповнену нішу між простими детермінованими підходами (FSM, BT), що є широкодоступними та добре задокументованими, та складними комерційними системами планування, які є пропрієтарними і погано задокументованими. Більшість доступних відкритих реалізацій GOAP є або застарілими (розраховані на Unity 4–5), або надто спрощеними й не включають повноцінної тактичної моделі поведінки для сценаріїв CQB.

Об’єктом роботи обрано систему інтелектуальної поведінки NPC у тривимірному тактичному симуляторі на базі ігрового рушія Unity.

Метою роботи є проєктування та реалізація повнофункціональної системи GOAP для тактичних сценаріїв ближнього бою (CQB), що забезпечить адаптивну, непередбачувану та тактично обґрунтовану поведінку агентів. Для досягнення поставленої мети необхідно виконати такі завдання:

– провести аналіз традиційних підходів до реалізації ШІ NPC та визначити переваги методу GOAP для тактичних застосунків;

– розробити архітектуру модульної системи GOAP, що включає сенсорний модуль, планувальник та бібліотеку атомарних дій з чітким розмежуванням відповідальностей;

– реалізувати алгоритм *A**-планування у просторі станів для побудови оптимальних послідовностей дій агента;

– розробити підсистему пошуку та керування тактичними позиціями (точками укриття), що враховує лінії прямої видимості, відстані та зайнятість позицій союзниками;

– реалізувати динамічну систему цілей агента, у якій пріоритети цілей залежать від поточного стану світу (рівня здоров’я, наявності боєприпасів, видимості противника);

– реалізувати прототип тактичного симулятора в Unity, що демонструє тактичні маневри агентів: флангові обходи, вогонь на придушення, координований відступ;

– провести функціональне тестування системи та порівняльний аналіз поведінки агентів GOAP та FSM/BT у ідентичних тактичних сценаріях;

– оформити технічну документацію та керівництво користувача до розробленої системи.

Критеріями успішності роботи є: продуктивність системи (час планування для одного агента не повинен перевищувати 1 мс на середньому ігровому обладнанні); масштабованість (підтримка щонайменше 20 агентів одночасно зі збереженням 60 FPS); тактична обґрунтованість поведінки (агенти повинні демонструвати флангові маневри, використання укриттів, координовану взаємодію); надійність системи (відсутність крашів і повисань протягом тривалого ігрового сеансу).

Результати виконання поставлених завдань представлені у наступних розділах цієї пояснювальної записки: у другому розділі описано архітектурне проєктування системи; у третьому – програмну реалізацію та результати тестування; у висновках узагальнено основні результати роботи та окреслено напрями подальшого розвитку розробленої системи.

2 АРХІТЕКТУРА ТА АЛГОРИТМІЧНА МОДЕЛЬ СИСТЕМИ GOAP

2.1 Загальна модульна структура системи штучного інтелекту

Розроблена система штучного інтелекту неігрових персонажів побудована за модульним архітектурним принципом, що забезпечує незалежність окремих підсистем, гнучкість їх модифікації та можливість незалежного тестування кожного компонента. Такий підхід відповідає принципам сучасної ігрової інженерії та полегшує підтримку коду проєкту в довгостроковій перспективі. Зокрема, він дозволяє виокремити алгоритмічне ядро системи планування від платформозалежного коду ігрового рушія, що в подальшому надає можливість перенесення алгоритмічних модулів на інші платформи без істотної доробки.

Загальна архітектура системи складається з трьох ключових функціональних модулів: сенсорного (Perception Module), модуля планування (GOAP Planner) та модуля виконання атомарних дій (Actions Library). Координація між цими модулями здійснюється через центральний компонент GOAPAgent, який реалізує загальний життєвий цикл NPC та виступає сполучною ланкою між алгоритмічним ядром системи і ігровим рушієм Unity. Структурно-функціональна схема системи наведена на рисунку 2.1.

Сенсорний модуль (Perception Module) виконує функцію збору інформації про стан навколишнього середовища та формування глобального представлення стану світу (World State). Модуль здійснює такі основні операції: виявлення супротивників у заданому радіусі сприйняття за допомогою процедури Physics.OverlapSphere; перевірку лінії прямої видимості (Line of Sight) через виклик Physics.Raycast; пошук потенційних точок укриття на основі аналізу геометрії рівня; визначення поточних параметрів агента, таких як рівень здоров’я та кількість боєприпасів у магазині.

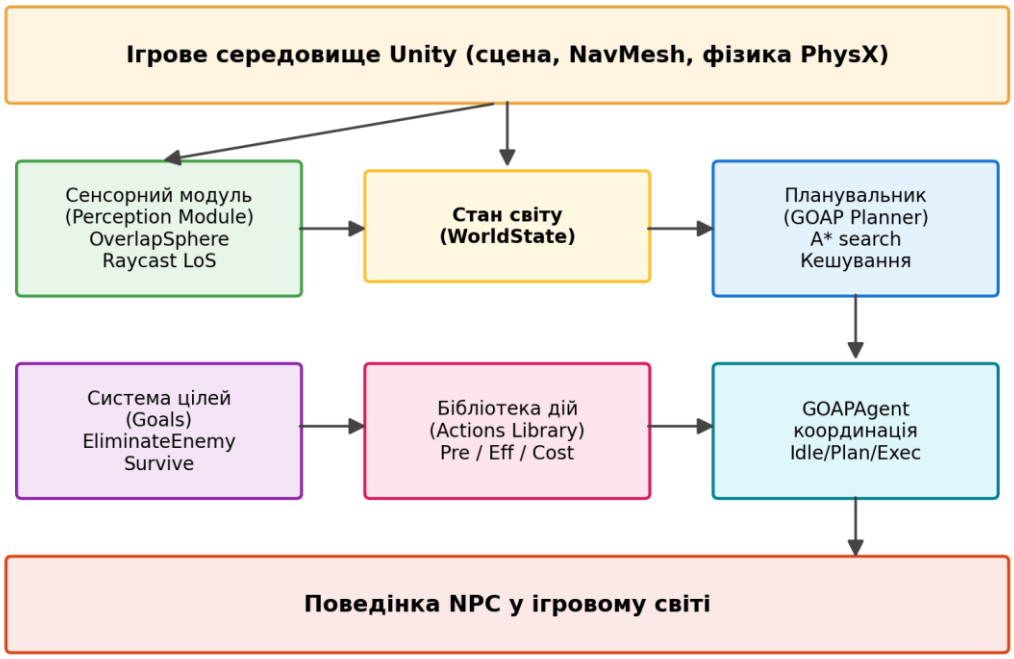

Рисунок 2.1 – Структурно-функціональна схема системи GOAP

Модуль планування (GOAP Planner) є центральним обчислювальним компонентом системи. Він отримує на вхід поточний стан світу WorldState, набір активних цілей з пріоритетами та бібліотеку доступних атомарних дій. На основі цих даних планувальник будує граф станів та виконує пошук оптимального плану – послідовності дій, що переводить систему від поточного стану до цільового з мінімальною сумарною вартістю виконання.

Модуль виконання атомарних дій (Actions Library) містить набір базових поведінкових примітивів, з яких планувальник будує плани вищого рівня. Кожна атомарна дія характеризується трьома атрибутами: передумовами виконання (Preconditions) – умовами стану світу, що мають бути виконані перед запуском дії; ефектами (Effects) – змінами стану світу після завершення дії; вартістю виконання (Cost) – числовим значенням, що використовується планувальником для порівняння альтернативних планів.

Центральним координуючим компонентом є клас GOAPAgent, що реалізує спрощену кінцеву машину з трьох станів верхнього рівня: Idle (очікування нової цілі), Planning (запит до планувальника та одержання плану) та Executing (послідовне виконання дій плану). Переходи між цими станами

ініціюються подіями нижчого рівня: завершення дії, виявлення супротивника, суттєва зміна параметрів агента або примусове скасування плану. Загальна схема цього автомата наведена на рисунку 2.2.

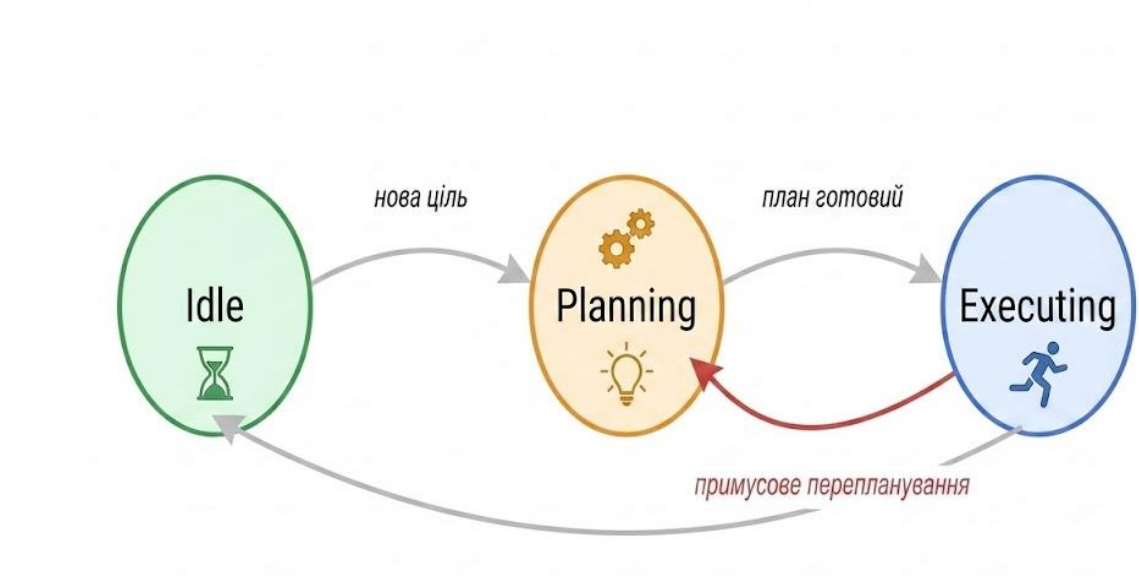

Рисунок 2.2 – Кінцева машина станів верхнього рівня GOAPAgent

Перелік модулів системи з визначенням їх призначення, вхідних та вихідних даних наведено у таблиці 2.1. Така формалізація меж відповідальності між модулями забезпечує можливість їх паралельної розробки декількома виконавцями та полегшує написання модульних тестів.

Завдяки чіткому розмежуванню функцій між модулями досягається висока гнучкість архітектури. Зокрема, доповнення бібліотеки новими атомарними діями не потребує модифікації планувальника, а зміни алгоритму планування не впливають на роботу сенсорного модуля. Така архітектурна декомпозиція також спрощує юніт-тестування окремих компонентів незалежно від ігрового рушія: алгоритмічне ядро (Planner, World State, Actions) може бути протестоване у звичайному консольному середовищі .NET без запуску Unity Editor.

Таблиця 2.1 – Перелік модулів системи штучного інтелекту та їх функції

Модуль	**Призначення**	**Вхідні дані**	**Вихідні дані**
PerceptionModule	Збір даних про стан світу	Стан ігрової сцени, параметри NPC	WorldState (Dictionary)
GOAPPlanner	Пошук оптимального плану	WorldState, Goal, ActionsList	Послідовність дій (Plan)
ActionsLibrary	Виконання атомарних дій	Plan, ігровий контекст	Зміна стану світу / агента
CoverSystem	Пошук точок укриття	Позиція агента, позиція ворога	CoverPoint (або null)
GoalsManager	Динамічне керування цілями	WorldState	Активна ціль з пріоритетом
GOAPAgent	Координація життєвого циклу	Усі модулі	Поведінка NPC у сцені

Окремо слід відзначити переваги обраної архітектури з точки зору розширюваності системи. Інтерфейс IGOAPAction дозволяє додавати нові поведінкові примітиви без модифікації планувальника та координуючого агента. Аналогічно, через інтерфейс IGoal можлива реалізація нових типів цілей з нестандартною логікою динамічної переоцінки пріоритетів. У майбутньому це створює фундамент для інтеграції методів машинного навчання, наприклад, для автоматичної корекції вартостей дій на основі попереднього досвіду агента або для генерації нових цілей з прикладів демонстрації поведінки.

2.2 Модель планування у просторі станів та алгоритм A^*

Центральною концепцією методу GOAP є представлення задачі планування як пошуку у просторі станів. На відміну від навігаційного планування, де простір пошуку відповідає фізичній геометрії рівня та реалізується засобами NavMesh, у GOAP простір пошуку є абстрактним графом, у якому вузли відповідають можливим логічним станам ігрового світу, а ребра – атомарним діям з певною вартістю переходу. Такий рівень абстракції дозволяє планувальнику оперувати поняттями «що має бути зроблено», а не «куди потрібно піти», що принципово розширює спектр тактичних рішень, доступних агенту.

Стан світу формалізується як кінцева множина пар «ключ–значення», у якій кожна пара описує одну логічну характеристику ситуації. У реалізації системи стан представлено асоціативним контейнером Dictionary<string, bool> мови C#. Приклад типового представлення поточного стану наведено у виразі:

$$S = \{\ enemyVisible: true,\ hasAmmo: true,\ inCover: false,\ healthCritical: false,\ magazineEmpty: false\ \}. \quad (2.1)$$

Хоча у поточній реалізації використано виключно булевий тип значень, архітектура передбачає розширення до загальної моделі States як Dictionary<string, object>, де значення можуть бути цілочисловими (наприклад, ammoCount), дійсними (healthPercent) чи перерахуваннями (threatLevel). Це створює потенціал для подальшого вдосконалення системи без перепроєктування алгоритмічного ядра.

Кожна атомарна дія A_i описується трійкою параметрів за формулою:

$$A_{\text{i}} = (\ Pre(A_{\text{i}}), Eff(A_{\text{i}}), c(A_{\text{i}})\ ), \quad (2.2)$$

де $Pre(A_i)$ – множина передумов виконання, тобто підстан, які повинні бути виконані безпосередньо перед запуском дії;

$Eff(A_i)$ – множина ефектів, тобто змін у стані світу після завершення дії;

$c(A_i)$ – числова вартість виконання, що використовується для порівняння альтернативних планів та керування «природністю» поведінки агента.

Граф планування будується у зворотному напрямку, від цільового стану до поточного. Такий вибір зумовлений декількома міркуваннями: по-перше, в задачах планування поведінки агента кількість можливих кінцевих станів зазвичай менша за кількість початкових варіантів, що зменшує коефіцієнт галуження; по-друге, зворотний пошук природним чином зосереджується на цілеспрямованих діях, відсікаючи нерелевантні гілки; по-третє, такий підхід природно поєднується з ідеєю декларативного опису цілей через бажаний стан світу. Схема процесу пошуку проілюстрована на рисунку 2.3.

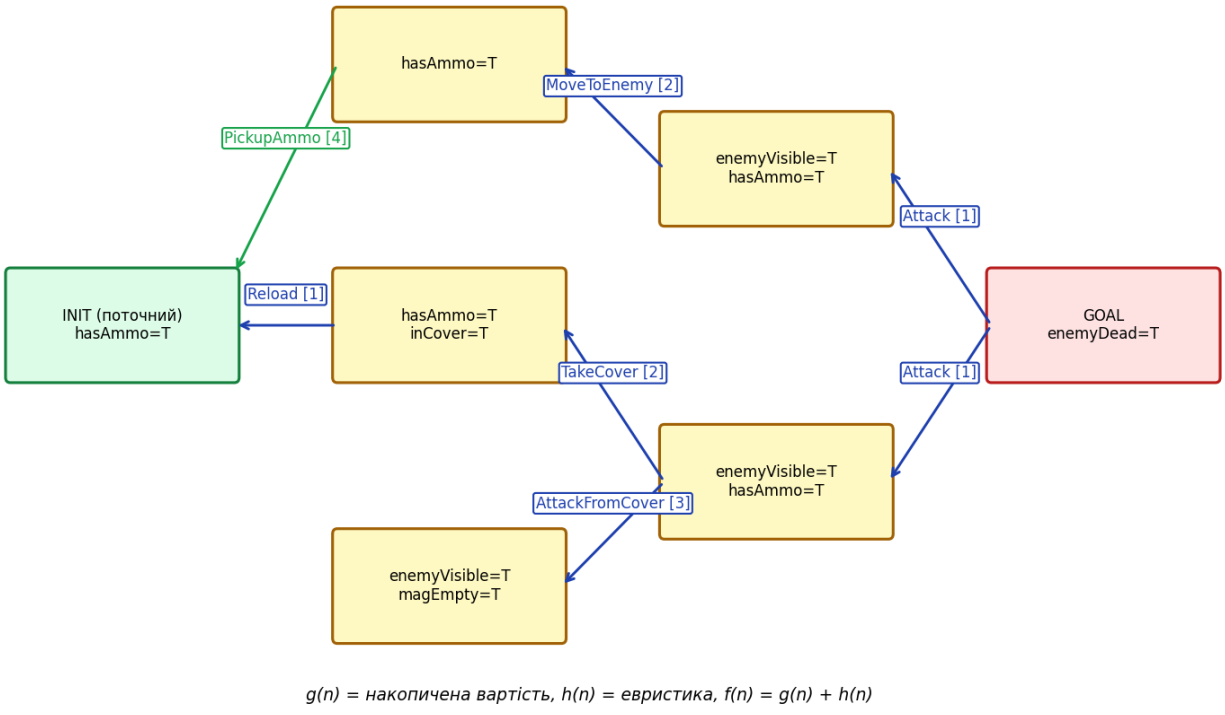

Рисунок 2.3 – Граф зворотного пошуку плану у просторі станів

Для пошуку оптимального плану застосовується алгоритм *A**, що поєднує переваги пошуку у ширину (повнота, оптимальність при допустимій

евристиці) та жадібного пошуку (швидкість). Функція оцінки f(n) для кожного вузла n визначається класичною формулою:

$$f(n) = g(n) + h(n), \tag{2.3}$$

де *g(n)* – сумарна вартість шляху від цільового стану до вузла *n*, накопичена через додавання значень $c(A_i)$;

h(n) – евристична функція, що оцінює мінімальну вартість шляху від вузла n до поточного стану.

Як евристика використовується кількість незадоволених літералів цільового стану, що є природним нижнім обмеженням реальної вартості та відповідає вимозі допустимості евристики *A**. Це гарантує, що знайдений план буде глобально оптимальним за критерієм сумарної вартості, що формально доведено у класичній статті Харта, Нільсона і Рафаеля [4].

Для оптимізації обчислювальних витрат планувальника реалізовано механізм кешування планів. Якщо до планувальника надходить запит з ідентичними параметрами (поточний стан світу та ціль, що збігаються з попередніми у межах допустимої похибки), повертається кешований план без повторного запуску *A**-пошуку. Це особливо ефективно в сценаріях, де декілька агентів одночасно опиняються в подібних тактичних ситуаціях. Додатково, реалізовано асинхронне планування за допомогою механізму async/await у C#, що виносить обчислення плану в окремий потік і не блокує ігровий цикл рендерингу.

Узагальнений псевдокод алгоритму планування для метода GOAP подано у лістингу 2.1.

Лістинг 2.1 Псевдокод алгоритму *A**-планування у просторі станів:

```
function Plan(currentState, goalState, actions):
    openSet = PriorityQueue()    // черга з пріоритетом за f(n)
```

```
while openSet not empty:
    n = openSet.PopMin()      // вузол з мінімальним f(n)
    if Satisfies(n.state, currentState):
        return ReconstructPlan(n)   // план знайдено
    foreach action in actions:
        if EffectsMatch(action, n.state):
            newState = ApplyBackward(action, n.state)
            if newNode not in closedSet:
                openSet.Add(newNode)
return null   // план не знайдено
```

Наведена реалізація має обчислювальну складність у найгіршому випадку $O(b^d)$, де b – середній коефіцієнт галуження простору дій (кількість дій, ефекти яких можуть бути застосовані до поточного вузла), d – довжина оптимального плану. Залежність кількості вузлів пошуку від цих параметрів наведена на рисунку 2.4.

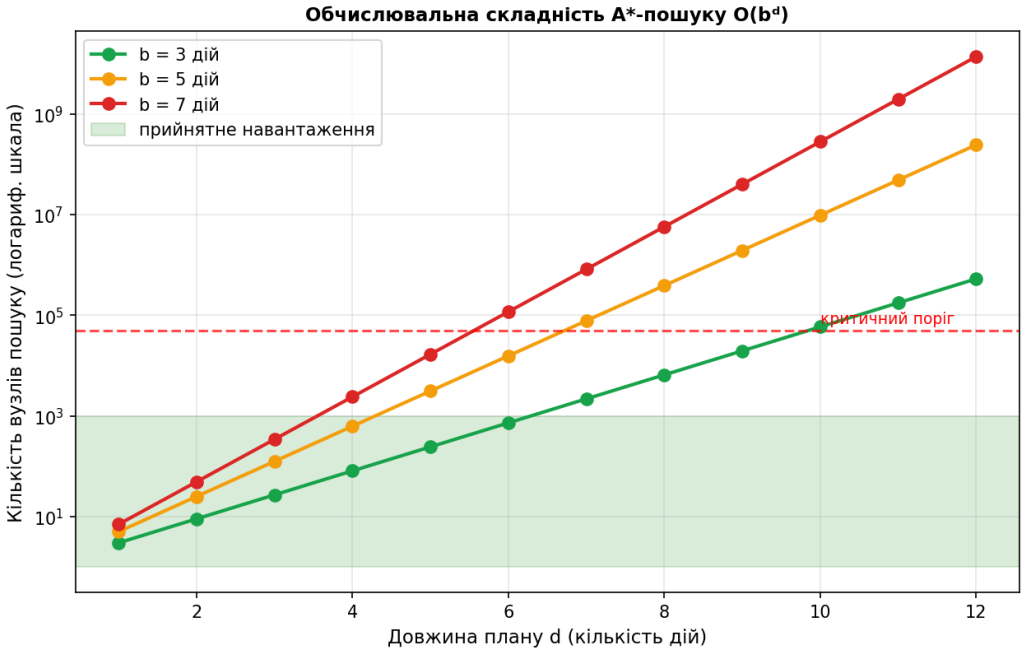

Рисунок 2.4 – Обчислювальна складність *A**-пошуку для різних параметрів задачі

На практиці завдяки обмеженій кількості атомарних дій (у системі реалізовано сім дій) та невеликій довжині планів (5–10 кроків у типових сценаріях) ця складність є цілком прийнятною для виконання у реальному часі. Експериментальні вимірювання, наведені у третьому розділі цієї роботи, підтверджують, що типовий час планування для одного агента у CQB-сценарії не перевищує 0,4 мс на середньому ігровому обладнанні.

2.3 Проєктування сенсорного модуля

Сенсорний модуль є першою ланкою у конвеєрі прийняття рішень агентом. Його основна функція – перетворення сирих даних ігрового середовища у формалізоване подання WorldState, придатне для використання модулем планування. Архітектурно модуль реалізований як C#-клас PerceptionModule, що успадковує MonoBehaviour ігрового рушія Unity і прикріплюється до ігрового об’єкта NPC поряд із компонентами GOAPAgent та GOAPPlanner.

Принцип роботи модуля реалізовано як обчислювальний цикл, що виконується з фіксованою частотою, за замовчуванням 10 разів на секунду. Така частота свідомо обрана значно нижчою за частоту кадрів (60 FPS) з міркувань продуктивності: операції Raycast та OverlapSphere є відносно дорогими в обчислювальному сенсі, тому їх виконання на кожному кадрі для кожного агента на сцені призвело б до неприйнятного навантаження на CPU. Зниження частоти до 10 Гц забезпечує цілком прийнятну реактивність NPC за умови суттєвого зниження обчислювальних витрат. Узагальнений алгоритм одного циклу сенсорного модуля наведений на рисунку 2.5.

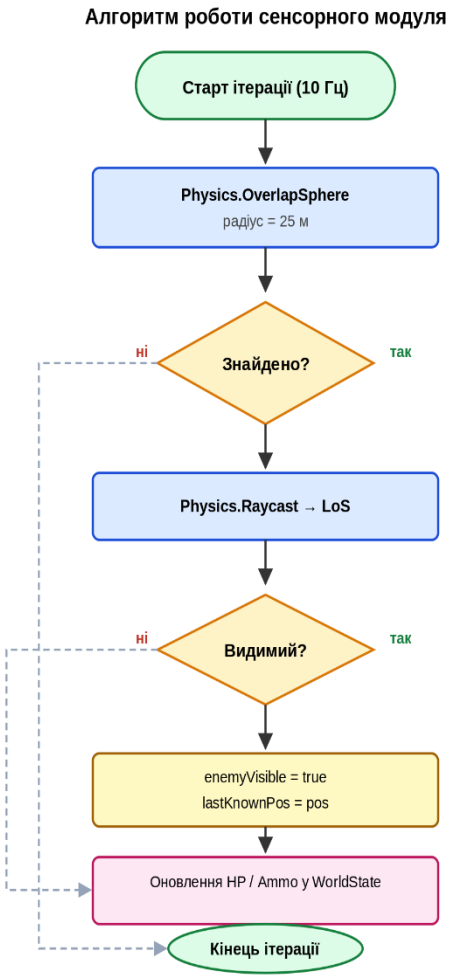

Рисунок 2.5 – Алгоритм роботи сенсорного модуля

Виявлення супротивників відбувається у два послідовні етапи. На першому етапі, широкому скануванні, використовується процедура Physics.OverlapSphere ігрового рушія Unity для пошуку всіх ігрових об’єктів з тегом «Enemy», що знаходяться у заданому радіусі сприйняття агента (за замовчуванням 25 метрів). Цей етап є обчислювально дешевим, оскільки використовує просторове кешування фізичної системи Unity і не вимагає трасування променів.

На другому етапі, точковій перевірці, для кожного знайденого кандидата виконується перевірка лінії прямої видимості (Line of Sight) шляхом викидання променя Physics.Raycast з точки розташування «очей» агента (eyeTransform) до точки прицілювання цілі. Цей етап враховує проміжні фізичні перешкоди: стіни, колони, контейнери, інших агентів. Тільки якщо

обидві перевірки успішні, об’єкт вважається «видимим» і відповідна властивість enemyVisible встановлюється у значення true.

Сенсорний модуль також відповідає за моніторинг внутрішніх параметрів агента: рівня здоров’я (healthPercent), залишку боєприпасів (ammoInMagazine), стану перезарядки тощо. Зміна цих параметрів автоматично відображається у відповідних ключах WorldState (healthCritical при healthPercent < 30, hasAmmo при ammoInMagazine > 0, magazineEmpty як інверсний прапор тощо).

2.4 Проєктування бібліотеки атомарних дій

Бібліотека атомарних дій є набором базових поведінкових примітивів, з яких планувальник будує плани вищого рівня. Кожна атомарна дія реалізується як окремий C#-клас, що успадковує спільний абстрактний клас GOAPAction. Він, у свою чергу, реалізує інтерфейс IGOAPAction, що визначає контракт взаємодії з планувальником. Загальна схема ієрархії класів наведена на рисунку 2.6.

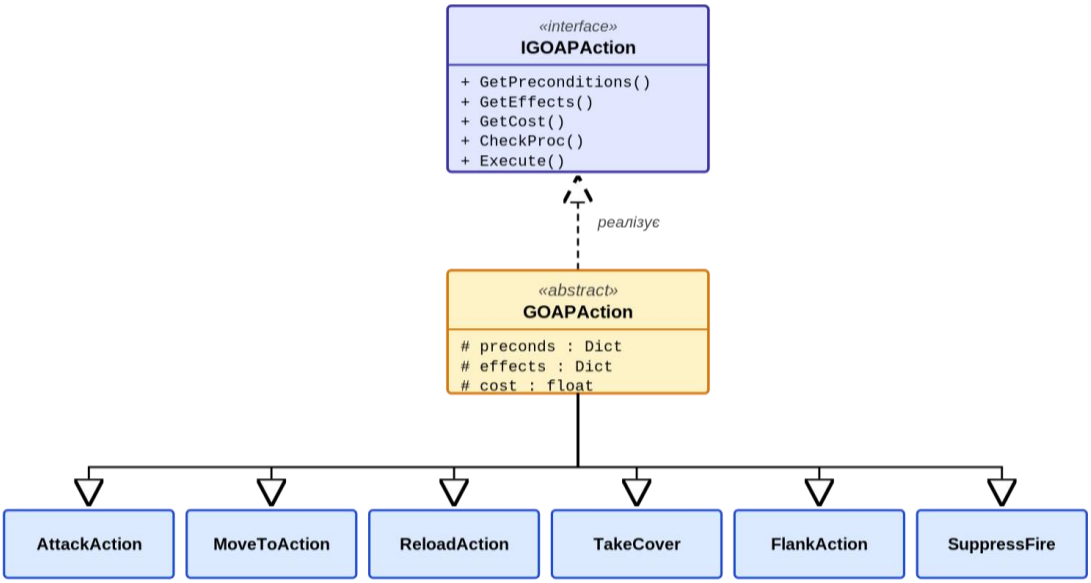

Рисунок 2.6 – Ієрархія класів бібліотеки атомарних дій (UML)

Інтерфейс IGOAPAction містить п’ять обов’язкових методів: GetPreconditions() – повертає словник передумов виконання; GetEffects() – повертає словник ефектів дії; GetCost() – повертає числову вартість виконання дії в поточному контексті; CheckProceduralPrecondition() – додатковий метод для перевірки контекстних передумов, які неможливо виразити статично (наприклад, наявність вільної точки укриття у радіусі досяжності); Execute() – власне виконання дії в ігровому циклі (метод викликається повторно до завершення дії).

Розроблена бібліотека включає сім базових атомарних дій, які забезпечують достатній набір тактичних примітивів для сценаріїв CQB.

Перелік дій з їх передумовами, ефектами та вартістю наведено у таблиці 2.2. Вартості підібрані емпіричним шляхом таким чином, щоб у типових ситуаціях планувальник віддавав перевагу більш «природним» з тактичної точки зору варіантам, наприклад, перезарядці в укритті перед атакою замість прямого штурму з порожнім магазином.

Таблиця 2.2 – Бібліотека атомарних дій системи GOAP

Дія	**Передумови**	**Ефекти**	**Cost**
MoveToEnemy	enemyKnown=T	enemyVisible=T	2
Attack	enemyVisible=T, hasAmmo=T	enemyDead=T	1
Reload	magazineEmpty=T	hasAmmo=T, magazineEmpty=F	1
TakeCover	coverAvailable=T	inCover=T	2
FlankAction	enemyVisible=T, flankRouteOK=T	flankPosReached=T	5
SuppressFire	hasAmmo=T, allyFlanking=T	enemySuppressed=T	3

Як видно з таблиці 2.2, дії FlankAction та SuppressFire мають найвищу вартість (5 та 3 відповідно), що відображає їх характер як складних тактичних маневрів, які потребують перенесення сил агента та зайняття вразливих позицій. Натомість дії з найнижчою вартістю (Attack, Reload) є базовими реактивними поведінковими актами. Така ієрархія вартостей створює природний тактичний пріоритет: агент віддасть перевагу атаці у простій ситуації, але вдасться до фланкового маневру у разі, коли пряма атака блокована.

Система цілей агента (Goals System) побудована за принципом динамічного пріоритету. На відміну від статичних систем, де порядок розгляду цілей фіксований розробником, у цій архітектурі пріоритет кожної цілі є функцією поточного стану світу.

У реалізації системи передбачено дві основні цілі вищого рівня: EliminateEnemy (ліквідувати противника) та Survive (вижити). Кожна ціль реалізована як C#-клас, що успадковує абстрактний клас GOAPGoal та надає реалізацію двох методів: GetDesiredState() – повертає цільовий стан світу, який слід досягти; CalculatePriority(WorldState ws) – обчислює поточний пріоритет цілі як функцію стану світу. Динаміка зміни пріоритетів цих цілей залежно від ключового параметра агента – рівня здоров’я – наведена на рисунку 2.7.

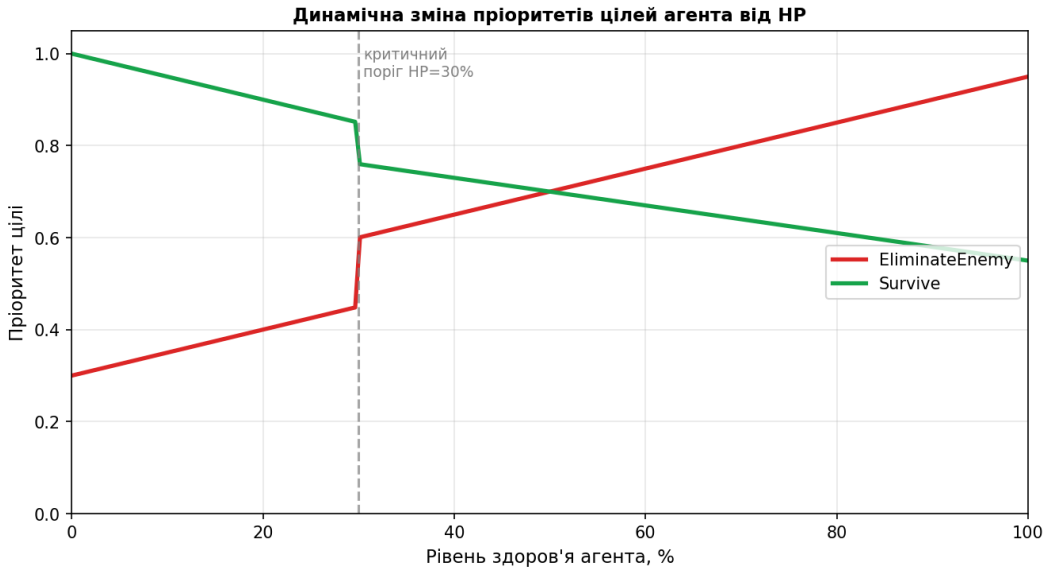

Рисунок 2.7 – Динамічна зміна пріоритетів цілей агента від рівня HP

Дискретний опис правил динамічної зміни пріоритетів цілей наведено у таблиці 2.3. У цій таблиці представлено основні умови, за яких відбувається переключення пріоритетів, а також приклади ситуацій, у яких ці правила спрацьовують.

При зміні стану світу планувальник обирає ціль з найвищим пріоритетом серед активних і будує план її досягнення. Якщо в процесі виконання поточного плану пріоритети змінюються, наприклад, рівень здоров'я знижується нижче критичного порогу (30 %), то агент перериває виконання поточного плану та переходить до планування під нову ціль.

Таблиця 2.3 – Динаміка пріоритетів цілей агента

Умова стану світу	**EliminateEnemy**	**Survive**
healthPercent ≥ 70 %	Високий (0.9)	Низький (0.2)
30 % ≤ healthPercent < 70 %	Середній (0.6)	Середній (0.5)
healthPercent < 30 %	Низький (0.3)	Високий (1.0)
magazineEmpty = true	Призупинено	Активна
allyDown = true (поруч)	Знижено (0.4)	Підвищено (0.7)

Цей механізм забезпечує необхідну реактивність та реалізм поведінки, не вимагаючи явного програмування переходів між тактичними станами, як у класичних FSM-системах.

2.5 Підсистема пошуку укриттів і тактичних позицій

Однією з ключових інновацій розроблюваної системи є підсистема пошуку укриттів (CoverSystem), яка забезпечує тактично грамотний вибір позицій для агентів у CQB-сценаріях. Принципова відмінність цієї підсистеми від тривіальних реалізацій полягає у тому, що при виборі точки укриття

враховується не лише її просторова доступність, але й тактична доцільність – зокрема, забезпечення фізичного захисту від вогню супротивника та запобігання зайняттю однієї точки декількома союзниками.

Підсистема CoverSystem реалізована як окремий менеджер, що зберігає реєстр зареєстрованих на рівні точок укриття типу CoverPoint. Кожна така точка попередньо розставляється дизайнером рівнів через спеціальний компонент Unity та реєструється у єдиному реєстрі через метод OnEnable(). Така архітектура дозволяє ефективно зберігати точки у просторовій структурі (наприклад, у k-d дереві) для швидкого пошуку найближчих кандидатів.

При запиті агента CoverSystem виконує фільтрацію точок за такими критеріями: по-перше, відстань до агента не перевищує заданого порогу (за замовчуванням 15 метрів); по-друге, точка забезпечує фізичний захист від вогню супротивника (перевіряється раздаванням Raycast від точки укриття до останньої відомої позиції супротивника, лінія має бути блокована геометрією рівня); по-третє, точка не зайнята іншим агентом-союзником (бронюється через прапор isReserved).

Окремою важливою задачею є динамічне резервування точок укриття. Коли агент А приймає рішення зайняти певну точку, ця точка позначається прапором isReserved до моменту фактичного прибуття агента. Це запобігає ситуаціям, коли декілька агентів одночасно планують зайняти одну й ту саму точку, що в подальшому призвело б до колізій руху та неприродного скупчення NPC у одному місці.

Для підвищення тактичної грамотності поведінки агентів реалізовано також систему оцінки якості укриття. Кожна точка отримує метрику CoverScore, що враховує: висоту перешкоди (повне або половинне укриття); кількість напрямків, з яких точка прострілюється; відстань до найближчого виходу для подальшого маневрування; можливість швидкого переходу до сусідніх точок. Така багатовимірна оцінка дозволяє планувальнику обирати не просто будь-яку доступну точку, а саме найкращу з тактичної точки зору.

2.6 Принципи координації командної взаємодії агентів

Одним із ключових вимог до системи ШІ для тактичних симуляторів CQB є координація командної взаємодії декількох агентів. Класичні комерційні реалізації, такі як F.E.A.R., досягали цього шляхом використання спільної структури даних (Squad Blackboard), на якій агенти могли публікувати та зчитувати інформацію про спільні наміри та поточні дії. Аналогічний підхід реалізовано і в розроблюваній системі.

Архітектурно командна координація реалізована через клас SquadCoordinator, що групує агентів у тактичні підрозділи (по 2–5 NPC у кожному) та підтримує спільну дошку даних (Blackboard). На цій дошці агенти публікують такі типи інформації: поточна тактична позиція; намір зайняти певну точку укриття (запобігає колізіям); намір виконати фланговий маневр (інші агенти можуть скоригувати свою поведінку). Принципова схема такого механізму наведена у таблиці 2.4.

Таблиця 2.4 – Інформаційні поля Squad Blackboard

Поле	**Тип**	**Призначення**
activeRole[agent]	enum {Leader, Flanker, Support}	Тактична роль агента у підрозділі
reservedCovers	List<CoverPoint>	Список зарезервованих точок укриття
plannedFlank[agent]	Vector3 (позиція)	Запланована точка флангового маневру
lastKnownEnemyPos	Vector3	Остання відома позиція ворога (спільна)
suppressionActive	bool	Чи активний зараз вогонь на придушення
alarm	enum {None, Engaged, Critical}	Загальний рівень тривоги підрозділу

Окремою важливою задачею є розподіл ролей у підрозділі. На етапі інкапсуляції підрозділу SquadCoordinator призначає кожному агенту одну з трьох тактичних ролей: Leader (керівник, що приймає ключові рішення), Flanker (відповідає за фланкові маневри) та Support (вогнева підтримка). Ці ролі впливають на ваги цілей агента та порядок розгляду альтернативних дій. Перерозподіл ролей відбувається при суттєвих змінах ситуації, наприклад, при загибелі Leader'а одного з агентів типу Support автоматично «підвищується» до ролі керівника.

2.7 Моделювання системи слуху та короткочасної пам'яті

У контексті тактичних симуляторів ближнього бою (CQB), де лінія прямої видимості часто перекрита складною геометрією приміщень, виключно візуального сприйняття, яке забезпечує розроблений сенсорний модуль, виявляється недостатньо. Для забезпечення повноцінної емерджентної поведінки систему сприйняття необхідно розширити підсистемою слуху та моделлю короткочасної пам'яті.

Підсистема слуху реалізується через реєстрацію звукових подій (Audio Events) у тривимірному просторі. Кожна дія, що генерує шум (постріл, біг, падіння предмета), створює у подієвій системі рушія сферичний імпульс із заданим радіусом поширення. Якщо NPC потрапляє в радіус дії цього імпульсу, сенсорний модуль фіксує напрямок та орієнтовну відстань до джерела шуму. Оскільки в умовах CQB звук може відбиватися від стін, для підвищення реалізму розраховується довжина шляху звукової хвилі за допомогою побудови найкоротшого маршруту на NavMesh, замість звичайної евклідової відстані. Це дозволяє агентам коректно реагувати на звуки, що лунають з інших кімнат чи поверхів.

Короткочасна пам'ять реалізується шляхом введення часових міток (timestamps) для збережених фактів у стані світу WorldState. Наприклад,

параметр lastKnownEnemyPos, який наразі оновлюється при візуальному контакті, доповнюється показником впевненості (Confidence Level). У момент втрати прямої видимості впевненість дорівнює 100 %, але з плином часу вона експоненційно спадає. Коли рівень впевненості падає нижче критичного порогу, планувальник автоматично генерує нову ціль — розвідку (Investigate), змушуючи агента рухатися до останньої відомої позиції для перевірки наявності загрози. Такий підхід усуває проблему «амнезії» NPC та робить тактичну поведінку більш природною.

2.8 Інтеграція анімацій з планувальником поведінки

Окремим завданням є синхронізація логічних дій планувальника з візуальним відображенням агента через компонент Animator.

Для вирішення цього завдання застосовано шаблон проєктування «Посередник» (Mediator), який реалізовано у вигляді додаткового компонента AnimationController. Цей компонент прослуховує події зміни стану в GOAPAgent (перехід між етапами Idle, Planning, Executing) та зчитує тип поточної атомарної дії, яка виконується. Відповідно до типу дії, контролер встановлює необхідні тригери та булеві змінні у стейт-машині Mecanim.

Критичним аспектом інтеграції є обробка тривалості виконання атомарних дій. На відміну від миттєвих логічних операцій, ігрові дії (наприклад, перезарядка зброї або перехід за укриття) потребують часу на програвання відповідної анімації. Для забезпечення консистентності методу Execute() атомарних дій використано механізм Animation Events рушія Unity. Анімаційний кліп містить маркери подій (наприклад, OnReloadComplete), які викликають відповідні колбеки у коді агента. Лише після отримання такого сигналу від аніматора атомарна дія позначається як успішно завершена (повертає true), і GOAPAgent переходить до наступного кроку плану.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ФУНКЦІОНАЛЬНЕ ТЕСТУВАННЯ СИСТЕМИ

3.1 Обґрунтування вибору середовища та інструментальних засобів реалізації

Вибір інструментальних засобів реалізації безпосередньо впливає на продуктивність розробки, якість кінцевого продукту та можливості його подальшого супроводу. Узагальнене обґрунтування вибору ігрового рушія наведено у підрозділі 1.5; у цьому підрозділі розглядаються конкретні версії, пакети та конфігурація програмного середовища, у якому виконано реалізацію системи.

Як основну платформу розробки обрано ігровий рушій Unity версії 2022 LTS. Вибір саме редакції з довготривалою підтримкою (Long-Term Support) зумовлений вимогою стабільності програмного інтерфейсу впродовж усього циклу розробки кваліфікаційної роботи: на відміну від технічних потоків Unity, редакції LTS не отримують змін API і виправляються лише в частині помилок та безпеки. Це гарантує відтворюваність результатів і відсутність необхідності адаптації коду до нових версій рушія в процесі роботи.

Рушій Unity надає низку вбудованих підсистем, що безпосередньо використовуються розробленою системою: підсистему навігації NavMesh (пакет AI Navigation) для побудови шляху агентів у тривимірному середовищі; систему фізичного моделювання NVIDIA PhysX, методи якої Physics.Raycast та Physics.OverlapSphere застосовуються сенсорним модулем для перевірки лінії прямої видимості та виявлення об’єктів у радіусі сприйняття; компонентну модель MonoBehaviour, що природно узгоджується з модульною архітектурою системи. Для введення з боку користувача застосовано новий пакет Input System, а для візуалізації сцени – конвеєр рендерингу Universal Render Pipeline (URP), який забезпечує прийнятну продуктивність на середньому ігровому обладнанні [9].

Мовою програмування обрано C# – основну мову скриптування Unity, що належить до сімейства об’єктно-орієнтованих мов із суворою статичною типізацією та автоматичним керуванням пам’яттю. Можливості мови: інтерфейси, узагальнення (generics), асинхронне програмування (async/await), делегати й лямбда-вирази, а також засоби LINQ дозволили реалізувати алгоритмічне ядро системи у вигляді чистих, незалежних від ігрового рушія класів, придатних до автономного модульного тестування у консольному середовищі .NET. Сувора типізація суттєво зменшує кількість помилок, що виявляються лише під час виконання, а вбудований механізм збирання сміття звільняє розробника від ручного керування пам’яттю [11].

Як інтегроване середовище розробки (IDE) використано JetBrains Rider. Це середовище забезпечує глибокий аналіз коду C#, інтелектуальне автодоповнення, виявлення потенційних помилок «на льоту» та розвинені засоби рефакторингу. Загальний вигляд середовища розробки з відкритим проєктом тактичного симулятора наведено на рисунку 3.1 [12].

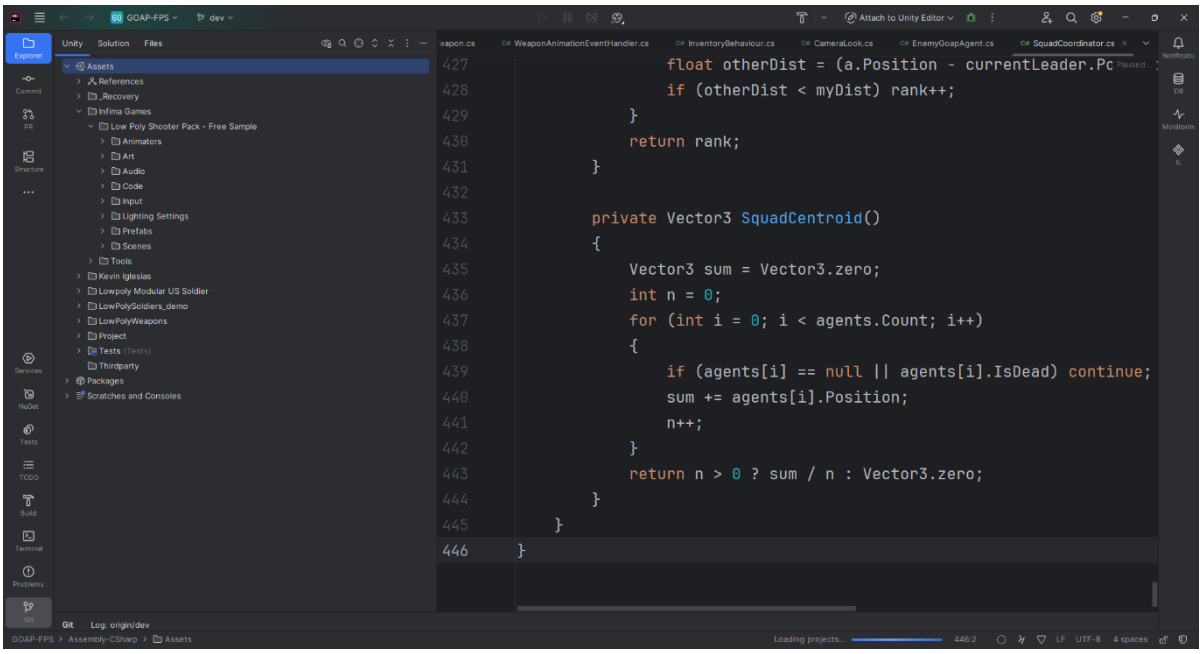

Рисунок 3.1 – Проєкт тактичного симулятора у середовищі розробки JetBrains Rider

Зведені відомості про використаний інструментарій, його версії та призначення наведено у таблиці 3.1. Обраний стек технологій є типовим для сучасної незалежної та напівпрофесійної розробки ігор і забезпечує оптимальне поєднання зручності розробки, продуктивності та широких можливостей для тестування й відлагодження поведінки інтелектуальних агентів.

Таблиця 3.1 – Інструментальні засоби реалізації системи

Засіб	**Версія**	**Призначення**
Unity	6.3 LTS	Ігровий рушій, фізика, навігація, рендеринг
C# / .NET	C# 9 / .NET Standard 2.1	Мова та платформа реалізації
JetBrains Rider	2026.1.2	Інтегроване середовище розробки
Git	2.42	Контроль версій вихідного коду

3.2 Загальна організація програмного коду проєкту

Програмний код проєкту організовано відповідно до модульної архітектури, обґрунтованої у другому розділі. Ключовим принципом організації є відокремлення алгоритмічного ядра системи планування від платформозалежного коду ігрового рушія. Завдяки цьому ядро (планувальник, модель стану світу, базові абстракції дій і цілей) не містить прямих залежностей від простору імен UnityEngine і може бути скомпільоване й протестоване у звичайному консольному застосунку .NET без запуску редактора Unity.

Фізично код розподілено за функціональними каталогами, кожному з яких відповідає окремий простір імен. Загальну структуру каталогів проєкту наведено на рисунку 3.2.

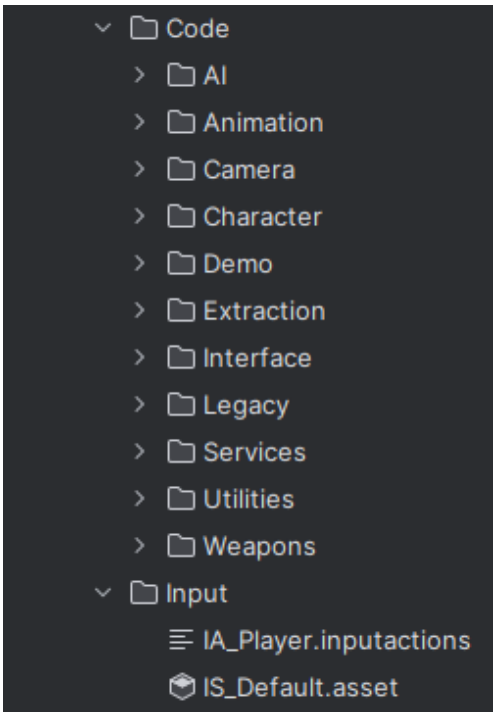

Рисунок 3.2 – Структура каталогів проєкту тактичного симулятора

Перелік основних програмних компонентів системи з зазначенням їх простору імен та призначення наведено у таблиці 3.2.

Таблиця 3.2 – Основні програмні компоненти системи

Компонент	**Модуль**	**Призначення**
WorldState	Core	Подання стану світу як набору пар «ключ–значення»
GOAPPlanner	Core	Зворотний пошук плану за алгоритмом *A**
GOAPAction	Core	Абстрактний базовий клас атомарної дії
GOAPGoal	Core	Абстрактний базовий клас цілі агента
SquadCoordinator	Tactics	Командна координація підрозділу
GOAPAgent	Agent	Координація життєвого циклу NPC

Взаємозв’язки між основними класами системи зображено на узагальненій діаграмі класів (рис. 3.3). Діаграма відображає відношення успадкування (конкретні дії успадковують GOAPAction, конкретні цілі – GOAPGoal), реалізації інтерфейсів (IGOAPAction) та композиції (GOAPAgent агрегує посилання на планувальник, сенсорний модуль і набір дій).

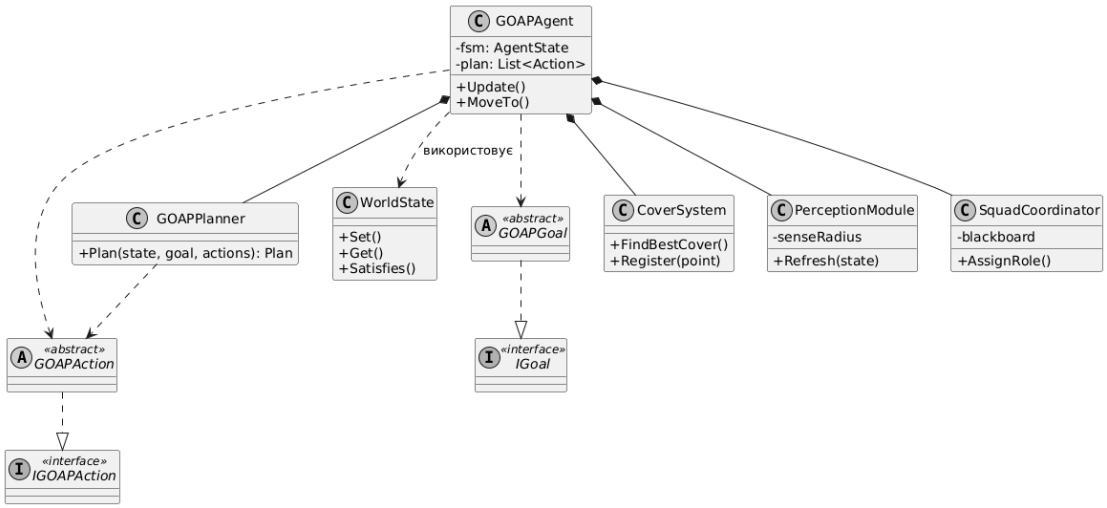

Рисунок 3.3 – Узагальнена діаграма класів системи GOAP (UML)

3.3 Реалізація моделі стану світу

Модель стану світу (World State) є базовою структурою даних, навколо якої побудовано всю систему планування. Стан світу формалізовано як скінченну множину пар «ключ–значення», де кожна пара описує одну логічну характеристику поточної ситуації. У реалізації стан представлено асоціативним контейнером на основі словника Dictionary<string, bool>, інкапсульованим у класі WorldState, що надає типобезпечний інтерфейс доступу та порівняння станів.

Клас WorldState реалізує методи встановлення й читання окремих ключів, перевірки виконання підстану (тобто чи задовольняє поточний стан множину необхідних умов) та клонування стану, що використовується

планувальником під час побудови графа пошуку. Узагальнений вигляд цього класу наведено у лістингу 3.1.

Лістинг 3.1 Реалізація моделі стану світу WorldState:

```
public class WorldState
{
    private readonly Dictionary<string, bool> _facts = new();
    public void Set(string key, bool value) => _facts[key] = value;
    public bool Get(string key) => _facts.TryGetValue(key, out var v) && v;
    public bool Satisfies(WorldState goal)
    {
        foreach (var kv in goal._facts)
            if (Get(kv.Key) != kv.Value) return false;
        return true;   // усі цільові літерали задоволені
    }

    public WorldState Clone() => new WorldState(_facts);
}
```

Перелік основних ключів стану світу, що використовуються системою у сценаріях ближнього бою, наведено у таблиці 3.3.

Таблиця 3.3 – Основні ключі стану світу агента

Ключ	**Тип**	**Семантика**
enemyKnown	bool	Відома позиція супротивника
enemyVisible	bool	Супротивник у прямій видимості
hasAmmo	bool	У магазині є боєприпаси
magazineEmpty	bool	Магазин порожній
inCover	bool	Агент перебуває в укритті
healthCritical	bool	Рівень здоров’я нижчий за критичний

3.4 Реалізація сенсорного модуля

Сенсорний модуль реалізовано у класі PerceptionModule, що успадковує MonoBehaviour і прикріплюється до ігрового об’єкта NPC поряд із компонентами GOAPAgent та GOAPPlanner. Основне завдання модуля – перетворення сирих даних ігрового середовища у формалізоване подання WorldState, придатне для використання планувальником.

Обчислювальний цикл модуля виконується з фіксованою частотою (за замовчуванням 10 Гц), що свідомо обрано значно нижчою за частоту кадрів з міркувань продуктивності. Виявлення супротивників відбувається у два етапи: широке сканування процедурою Physics.OverlapSphere у заданому радіусі сприйняття та точкова перевірка лінії прямої видимості викиданням променя Physics.Raycast до кожного знайденого кандидата. Лише за умови успішності обох перевірок відповідний об’єкт вважається видимим. Спрощений фрагмент реалізації сенсорного циклу наведено у лістингу 3.2.

Лістинг 3.2 Фрагмент циклу оновлення сенсорного модуля :

```
private void Tick()
{
   var hits = Physics.OverlapSphere(transform.position, senseRadius, enemyMask);
   bool visible = false;
   foreach (var h in hits)
   {
      Vector3 dir = h.transform.position - eye.position;
      if (!Physics.Raycast(eye.position, dir, dir.magnitude, obstacleMask))
      {
         visible = true;
         lastKnownEnemyPos = h.transform.position;
         break;
```

```
            }
        }
        state.Set("enemyVisible", visible);
        state.Set("hasAmmo", ammoInMagazine > 0);
        state.Set("magazineEmpty", ammoInMagazine == 0);
        state.Set("healthCritical", healthPercent < 30);
    }
```

Окрім зовнішнього сприйняття, сенсорний модуль відповідає за моніторинг внутрішніх параметрів агента: рівня здоров'я, залишку боєприпасів та стану перезарядки і автоматичне відображення їх змін у відповідних ключах стану світу. Це створює замкнений зворотний зв'язок між фізичною симуляцією та абстрактним поданням стану, на основі якого планувальник приймає рішення. Результат роботи сенсорного модуля можна спостерігати безпосередньо у вікні сцени завдяки засобам візуалізації: радіус сприйняття та промені перевірки видимості відображаються у вигляді допоміжних ліній (рис. 3.4).

Рисунок 3.4 – Візуалізація радіуса сприйняття та перевірки лінії видимості

3.5 Реалізація планувальника на основі алгоритму *A**

Модуль планування реалізовано у класі GOAPPlanner, який виконує зворотний пошук плану у просторі станів за алгоритмом *A**. Планувальник є повністю незалежним від ігрового рушія: на вхід він отримує поточний стан світу, цільовий стан і набір доступних дій, а повертає впорядковану послідовність дій (план) або ознаку неможливості досягнення цілі.

Кожен вузол графа пошуку представлено структурою Node, що зберігає стан світу, накопичену вартість шляху *g*, евристичну оцінку *h*, посилання на батьківський вузол та дію, що привела до цього вузла. Така структура дозволяє після завершення пошуку відновити повний план шляхом проходження від цільового вузла до кореневого за батьківськими посиланнями. Оголошення структури вузла наведено у лістингу 3.3.

Лістинг 3.3 Структура вузла графа пошуку:

```
private sealed class Node
{
    public WorldState State;
    public float G;          // вартість шляху від цілі до вузла
    public float H;          // евристична оцінка до поточного стану
    public float F => G + H;   // підсумкова оцінка вузла
    public Node Parent;        // батьківський вузол
    public IGOAPAction Via;    // дія, що веде до цього вузла
}
```

Для зберігання межі пошуку (відкритого набору) використано чергу з пріоритетом, упорядковану за зростанням функції оцінки *f(n)*. Закритий набір реалізовано на основі хеш-множини, що забезпечує перевірку повторного відвідування станів за константний у середньому час. На кожній ітерації з відкритого набору вилучається вузол з мінімальним значенням *f(n)*, після чого

перевіряється умова досягнення поточного стану світу; у разі виконання цієї умови план вважається знайденим. Узагальнений псевдокод алгоритму планування наведено у лістингу 2.1, а ключовий фрагмент його реалізації мовою C# – у лістингу 3.4.

Лістинг 3.4 Основний цикл *A**-планувальника:

```
public List<IGOAPAction> Plan(WorldState current, WorldState goal,
                              IList<IGOAPAction> actions)
{
   var open = new PriorityQueue<Node>();
   open.Push(new Node { State = goal, G = 0, H = Heuristic(goal, current)
});
   var closed = new HashSet<WorldState>();
   while (open.Count > 0)
   {
      Node n = open.PopMin();
      if (n.State.Satisfies(current)) return Reconstruct(n);
      closed.Add(n.State);
      foreach (var a in actions)
         if (EffectsMatch(a, n.State))
         {
            var next = ApplyBackward(a, n.State);
            if (closed.Contains(next)) continue;
            open.Push(new Node { State = next, G = n.G + a.GetCost(),
                  H = Heuristic(next, current), Parent = n, Via = a });
         }
   }
   return null;   // план не знайдено
}
```

Як евристичну функцію використано кількість незадоволених літералів цільового стану, що є допустимою (admissible) евристикою і гарантує оптимальність знайденого плану за критерієм сумарної вартості. Для оптимізації обчислювальних витрат реалізовано механізм кешування планів на основі структури типу LRU: за надходження запиту з параметрами, що збігаються з раніше обробленими, повертається збережений план без повторного запуску пошуку. Додатково реалізовано асинхронне планування засобами async/await, що виносить обчислення плану в окремий потік і не блокує ігровий цикл рендерингу. Узагальнену блок-схему роботи планувальника наведено на рисунку 3.5, а основні параметри його налаштування – у таблиці 3.4.

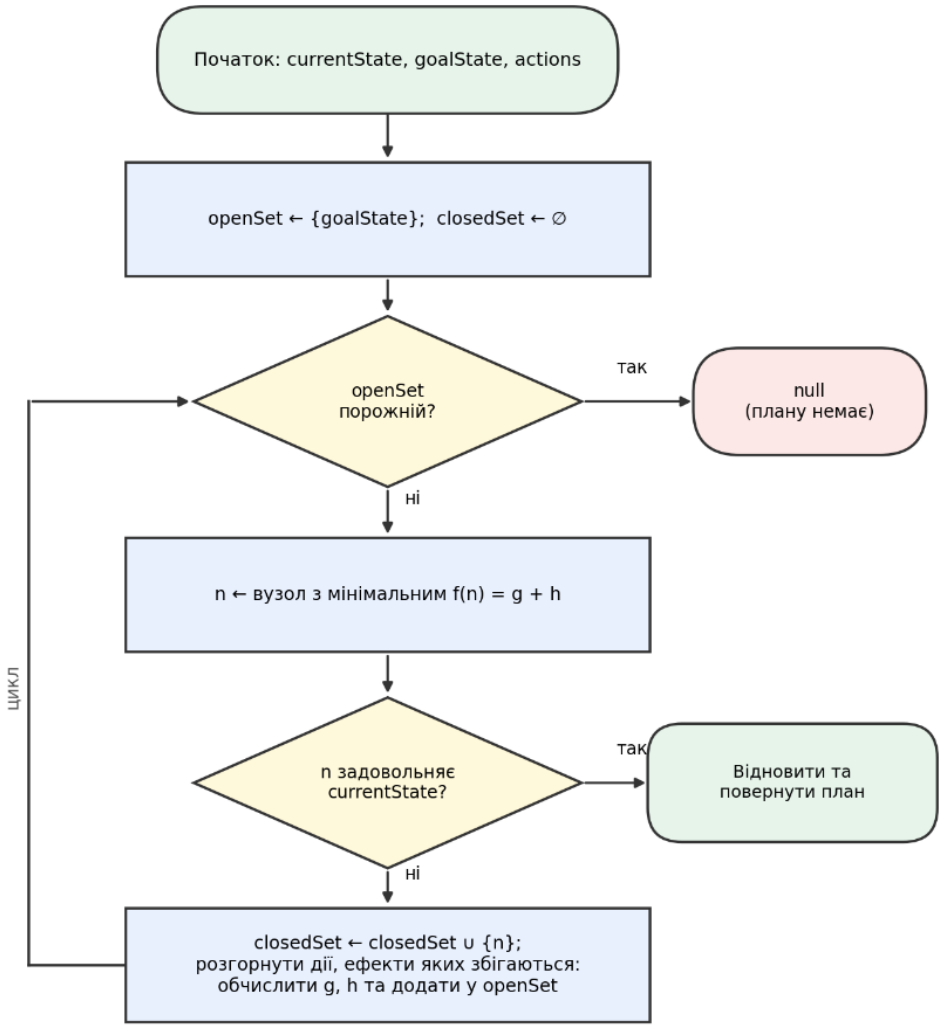

Рисунок 3.5 – Блок-схема роботи планувальника GOAP

Таблиця 3.4 – Параметри налаштування планувальника

Параметр	**Значення за замовчуванням**	**Призначення**
maxIterations	1000	Граничне число ітерацій пошуку
cacheSize	32	Місткість LRU-кешу планів агента
asyncPlanning	true	Виконувати пошук в окремому потоці
replanThreshold	0.2	Поріг зміни стану для перепланування

Окремої уваги потребує забезпечення детермінованості планувальника: за однакових вхідних даних: стану світу, цілі та набору дій він завжди повертає той самий план, що є важливим для відтворюваності поведінки під час тестування й налагодження. Детермінованість досягається стабільним упорядкуванням дій під час розгортання вузлів та узгодженим розв’язанням ситуацій рівної вартості. Асинхронне виконання планувальника при цьому не порушує детермінованості, оскільки звертання до спільних структур даних агента відбувається лише в основному ігровому потоці після завершення обчислень плану.

3.6 Реалізація бібліотеки атомарних дій

Бібліотека атомарних дій побудована навколо інтерфейсу IGOAPAction та абстрактного класу GOAPAction, що реалізує спільну для всіх дій логіку. Інтерфейс визначає контракт взаємодії дії з планувальником і містить п’ять обов’язкових методів: GetPreconditions(), GetEffects(), GetCost(), CheckProceduralPrecondition() та Execute(). Перші три методи надають

планувальнику декларативний опис дії, четвертий дозволяє перевіряти контекстні передумови, які неможливо виразити статично, а п’ятий реалізує безпосереднє виконання дії в ігровому циклі. Призначення методів інтерфейсу узагальнено у таблиці 3.5.

Таблиця 3.5 – Методи інтерфейсу IGOAPAction

Метод	**Повертає**	**Призначення**
GetPreconditions()	словник	Передумови виконання дії
GetEffects()	словник	Зміни стану світу після дії
GetCost()	float	Вартість виконання дії
CheckProceduralPrecondition()	bool	Перевірка можливості дії
Execute()	bool	Виконання дії (true – завершено)

Абстрактний клас GOAPAction інкапсулює словники передумов та ефектів, методи їх формування й типове значення вартості, звільняючи конкретні дії від дублювання коду. Оголошення базового класу наведено у лістингу 3.5.

Лістинг 3.5 Абстрактний базовий клас атомарної дії:

```
public abstract class GOAPAction : IGOAPAction
{
    protected readonly Dictionary<string, bool> pre = new();
    protected readonly Dictionary<string, bool> eff = new();
    protected float cost = 1f;
    public Dictionary<string, bool> GetPreconditions() => pre;
    public Dictionary<string, bool> GetEffects() => eff;
    public virtual float GetCost() => cost;
    public abstract bool CheckProceduralPrecondition(GOAPAgent agent);
    public abstract bool Execute(GOAPAgent agent);
}
```

Як приклад конкретної реалізації у лістингу 3.6 наведено дію зайняття укриття TakeCoverAction, що демонструє типову структуру атомарної дії: у конструкторі задаються передумови та ефекти, метод процедурної перевірки звертається до підсистеми укриттів, а метод виконання здійснює переміщення агента до обраної точки засобами навігації.

Лістинг 3.6 Реалізація атомарної дії зайняття укриття:

```
public class TakeCoverAction : GOAPAction
{
    private CoverPoint _target;
    public TakeCoverAction()
    {
        pre["coverAvailable"] = true;
        eff["inCover"] = true;
        cost = 2f;
    }
    public override bool CheckProceduralPrecondition(GOAPAgent a)
    {
        _target = a.Cover.FindBestCover(a.Position, a.LastKnownEnemyPos);
        return _target != null;
    }
    public override bool Execute(GOAPAgent a)
    {
        a.MoveTo(_target.Position);
        return a.ReachedDestination;
    }
}
```

Ієрархію класів бібліотеки атомарних дій наведено на рисунку 3.6. Завдяки спільному інтерфейсу та базовому класу додавання нової

поведінкової можливості зводиться до створення одного класу-нащадка GOAPAction і не потребує жодних змін у планувальнику чи координуючому агенті, що повністю відповідає принципу відкритості або закритості.

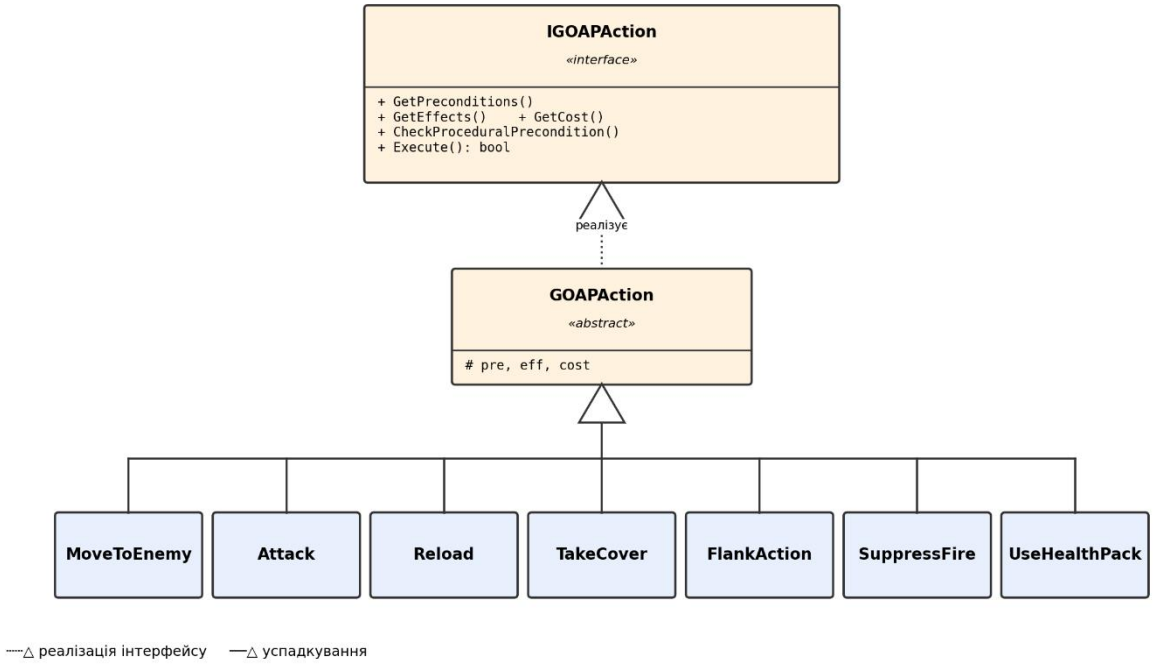

Рисунок 3.6 – Ієрархія класів бібліотеки атомарних дій (UML)

Значення вартостей атомарних дій підібрано емпіричним шляхом так, щоб у типових ситуаціях планувальник віддавав перевагу більш «природним» з тактичної точки зору варіантам наприклад, перезарядці в укритті перед прямим штурмом з порожнім магазином. Найвищі вартості призначено складним маневрам (фланговий обхід, вогонь на придушення), що відображає їх характер як ризикованих дій, тоді як базові реактивні акти (атака, перезарядка) мають мінімальну вартість. Завдяки винесенню вартостей в окремі поля класів дій їх можна коригувати без зміни логіки планувальника, що в перспективі дозволяє автоматизувати їх налаштування методами машинного навчання.

3.7 Реалізація динамічної системи цілей

Кожна ціль реалізована як окремий клас, що успадковує абстрактний клас GOAPGoal і надає реалізацію двох методів: GetDesiredState(), який повертає цільовий стан світу, та CalculatePriority(), що обчислює поточний пріоритет цілі як функцію стану світу. У реалізації передбачено дві основні цілі вищого рівня – EliminateEnemy (ліквідувати противника) та Survive (вижити).

Динамічність пріоритетів забезпечує реалістичне переключення між тактичними намірами агента залежно від ситуації без явного програмування переходів, властивого класичним кінцевим автоматам. Приклад реалізації цілі виживання, пріоритет якої зростає зі зниженням рівня здоров’я, наведено у лістингу 3.7.

Лістинг 3.7 Реалізація цілі виживання агента:

```
public class SurviveGoal : GOAPGoal
{
    public override WorldState GetDesiredState()
    {
        var s = new WorldState();
        s.Set("inCover", true);
        s.Set("healthCritical", false);
        return s;
    }
    public override float CalculatePriority(WorldState ws)
    {
        if (ws.Get("healthCritical")) return 1.0f;   // найвищий пріоритет
        return ws.Get("enemyVisible") ? 0.5f : 0.2f;
    }
}
```

На кожному кроці прийняття рішень менеджер цілей обирає ціль з найвищим обчисленим пріоритетом серед активних і передає її цільовий стан планувальнику. Якщо у процесі виконання поточного плану пріоритети змінюються (наприклад, рівень здоров’я знижується нижче критичного порогу), агент перериває виконання плану та переходить до планування під нову ціль.

3.8 Реалізація підсистеми пошуку укриттів

Підсистему пошуку укриттів реалізовано у класі CoverSystem, який зберігає реєстр зареєстрованих на рівні точок укриття типу CoverPoint і виконує їх фільтрацію за тактичними критеріями. Кожна точка укриття попередньо розставляється дизайнером рівнів через спеціальний компонент і реєструється у єдиному реєстрі у момент активації засобом методу OnEnable().

За запитом агента підсистема виконує багатокритеріальну фільтрацію точок: за відстанню до агента, за забезпеченням фізичного захисту від вогню супротивника (перевіряється викиданням променя від точки укриття до останньої відомої позиції супротивника) та за відсутністю резервування точки іншим союзником. Спрощений фрагмент пошуку найкращої точки укриття наведено у лістингу 3.8.

Лістинг 3.8 Пошук найкращої точки укриття:

```
public CoverPoint FindBestCover(Vector3 from, Vector3 threat)
{
    CoverPoint best = null; float bestScore = float.MinValue;
    foreach (var c in _registered)
    {
        if (c.IsReserved) continue;
        if (Vector3.Distance(from, c.Position) > maxDistance) continue;
```

```
        if (!IsProtected(c, threat)) continue;     // перевірка захищеності
        float score = c.Evaluate(from, threat);
        if (score > bestScore) { bestScore = score; best = c; }
    }
    if (best != null) best.IsReserved = true;       // резервування точки
    return best;
}
```

Для підвищення тактичної грамотності поведінки реалізовано систему оцінки якості укриття: кожна точка отримує метрику CoverScore, що враховує висоту перешкоди, кількість напрямків прострілювання, відстань до найближчого виходу та можливість швидкого переходу до сусідніх позицій. Окремою важливою задачею є динамічне резервування точок: коли агент приймає рішення зайняти певну точку, вона позначається прапором isReserved до моменту фактичного прибуття, що запобігає колізіям руху та неприродному скупченню агентів. Перелік критеріїв оцінки якості укриття наведено у таблиці 3.6.

Таблиця 3.6 – Критерії оцінки якості точки укриття

Критерій	**Вага**	**Опис**
Захищеність	0.40	Блокування лінії вогню супротивника
Відстань	0.25	Близькість до поточної позиції агента
Висота укриття	0.20	Повне або половинне укриття
Шляхи відходу	0.15	Можливість швидкого маневрування

Для прискорення пошуку найближчих точок укриття на рівнях з великою кількістю зареєстрованих позицій реєстр CoverSystem може бути організований у вигляді просторової структури даних – наприклад, k-вимірного дерева (k-d tree). Це знижує складність пошуку кандидатів з лінійної до логарифмічної відносно кількості точок, що стає суттєвим у масштабних

сценаріях оборони будівлі з десятками потенційних укриттів. У поточній реалізації, де кількість точок на рівні є помірною, застосовано простий лінійний перебір з раннім відсіканням за відстанню, що забезпечує цілком достатню швидкодію без ускладнення коду.

3.9 Реалізація підсистеми командної координації

Підсистему командної координації реалізовано у класі SquadCoordinator, що групує агентів у тактичні підрозділи та підтримує спільну дошку даних (Blackboard).

На дошці даних агенти публікують інформацію про свою поточну тактичну позицію, намір зайняти певну точку укриття, намір виконати фланговий маневр та сигнали про критичні події. Координатор також відповідає за розподіл тактичних ролей у підрозділі Leader, Flanker та Support і за їх перерозподіл у разі суттєвих змін ситуації, наприклад автоматичне підвищення одного з агентів типу Support до ролі керівника при загибелі Leader. Інтерфейс доступу до дошки даних наведено у лістингу 3.9.

Лістинг 3.9 Інтерфейс спільної дошки даних підрозділу:

```
public class SquadBlackboard
{
    public Vector3 LastKnownEnemyPos { get; set; }
    public bool SuppressionActive { get; set; }
    private readonly HashSet<CoverPoint> _reserved = new();

    public bool TryReserveCover(CoverPoint c)
    {
        if (_reserved.Contains(c)) return false;
        _reserved.Add(c); return true;
```

```
    }
    public void AssignRole(GOAPAgent a, SquadRole role) => a.Role = role;
}
```

Завдяки спільній дошці даних агенти узгоджують свої дії без прямого обміну повідомленнями: публікація наміру виконати фланговий маневр спонукає інших членів підрозділу перейти до ведення придушувального вогню, а резервування точки укриття запобігає її повторному вибору.

3.10 Реалізація координуючого агента

Центральним координуючим компонентом є клас GOAPAgent, що успадковує MonoBehaviour і реалізує життєвий цикл агента у вигляді кінцевого автомата з трьох станів верхнього рівня: Idle (очікування нової цілі), Planning (запит до планувальника й одержання плану) та Executing (послідовне виконання дій плану). У методі Update() агент опитує сенсорний модуль, за потреби ініціює перепланування та послідовно виконує дії поточного плану, реагуючи на події нижчого рівня – завершення дії, виявлення супротивника або суттєву зміну параметрів агента.

Спрощений фрагмент основного циклу координуючого агента наведено у лістингу 3.10. Він демонструє переходи між станами верхнього рівня та делегування фактичної роботи відповідним модулям системи.

Лістинг 3.10 Основний цикл координуючого агента:

```
private void Update()
{
    perception.Refresh(state);
    switch (fsm)
    {
```

```
            case AgentState.Idle:
                currentGoal = goals.SelectBest(state);
                fsm = AgentState.Planning; break;
            case AgentState.Planning:
                plan = planner.Plan(state, currentGoal.GetDesiredState(), actions);
                fsm = plan != null ? AgentState.Executing : AgentState.Idle; break;
            case AgentState.Executing:
                if (!ExecuteCurrentStep()) fsm = AgentState.Idle; break;
        }
    }
```

Налаштування параметрів агента, радіуса сприйняття, частоти оновлення, посилань на ціль та компоненти здійснюється безпосередньо в інспекторі Unity, що дозволяє швидко конфігурувати поведінку без зміни коду (рис. 3.7). Для забезпечення глибокого контролю за процесом прийняття рішень під час виконання симуляції, стандартних засобів налагодження Unity (таких як механізм Gizmos, що відображає лише радіуси та лінії видимості) виявилося недостатньо. Тому було розроблено спеціалізований внутрішньоігровий інструмент налагодження – GoapAgentDebugUI, реалізований на базі підсистеми Unity IMGUI. Цей інструмент генерує плаваючі інформаційні панелі (статус-картки) безпосередньо над кожним агентом у радіусі видимості камери.

У реальному часі ці картки відображають ключові параметри NPC: поточну тактичну роль у підрозділі (Leader, Flanker, Support), активну ціль (Goal), виконувану в даний момент атомарну дію (Action), рівень здоров’я та стан інвентарю. Окрім того, для агента, що знаходиться найближче до центру уваги гравця, на екран виводиться розширена панель GOAP-дебаггера.

Рисунок 3.7 – Налаштування компонентів GOAP-агента в інспекторі Unity

Вона містить повний розрахований план із покроковою вартістю виконання дій, а також актуальний набір символів стану світу (World State), якими оперує алгоритм A*. Такий підхід робить «видимими» внутрішні рішення планувальника, які інакше залишалися б прихованими, і критично пришвидшує процес тюнінгу системи цілей (рис. 3.8).

Рисунок 3.8 – Налагоджувальна візуалізація плану та стану агента

Проведення стрес-тестів та аналіз продуктивності системи здійснювалися за допомогою інструменту Unity Profiler. Результати профілювання підтвердили, що відмальовка кастомного інтерфейсу налагодження та розрахунки планувальника створюють мінімальне навантаження на систему, не викликаючи відчутних сплесків збирача сміття (Garbage Collector) навіть при одночасній роботі десятків агентів.

3.11 Інтеграція алгоритмічного ядра GOAP зі збройовою підсистемою та інвентарем агента

У тактичному симуляторі поведінка неігрових персонажів нерозривно пов’язана зі станом їхнього спорядження. Оскільки алгоритмічне ядро GOAP оперує абстрактними логічними станами (наприклад, hasAmmo, magazineEmpty або healthCritical), виникає необхідність безперервної синхронізації цих станів із фактичними фізичними компонентами агента в ігровому середовищі. Для вирішення цього завдання в архітектурі системи застосовано принцип слабкої зв’язності (loose coupling). Замість прямого звернення до компонентів зброї з алгоритму планування, відповідальність за оновлення стану делеговано сенсорному модулю.

У процесі роботи PerceptionModule отримує посилання на компоненти інвентарю та поточної екіпірованої зброї. На кожній ітерації сенсорного циклу система перевіряє кількість боєприпасів у поточному магазині та загальний запас аптечок. Якщо магазин порожній, відповідному ключу у словнику WorldState присвоюється значення true, що автоматично підвищує пріоритет дії перезаряджання (ReloadAction). Під час безпосереднього виконання цієї дії координуючий агент викликає метод збройової підсистеми, ініціюючи логіку поповнення боєкомплекту.

Такий підхід гарантує, що планувальник завжди працює з актуальними даними про ресурси агента. Крім того, архітектурне розмежування дозволяє

легко масштабувати проєкт: додавання нових типів спорядження (наприклад, гранат) не вимагає втручання в код алгоритму *A**, обмежуючись лише додаванням нових ключів у стан світу та відповідних класів-нащадків у бібліотеку атомарних дій.

3.12 Управління просторовим переміщенням агентів на базі навігаційної сітки

Більшість тактичних маневрів, таких як зайняття укриття (TakeCover), фланговий обхід (FlankAction) або наближення до супротивника (MoveToEnemy), вимагають коректної просторової навігації у тривимірному середовищі. Перетворення логічних рішень планувальника на фізичне переміщення агента реалізовано за допомогою вбудованої підсистеми Unity NavMesh та компонента NavMeshAgent.

При активації просторової атомарної дії цільові координати, отримані від підсистеми пошуку укриттів або з дошки даних підрозділу, передаються до навігаційного компонента. Оскільки розрахунок оптимального шляху в рушії Unity виконується асинхронно, система постійно перевіряє стан обчислення через властивість pathPending. Атомарна дія в методі Execute() повертає значення true (тобто вважається успішно завершеною) лише тоді, коли агент фактично досягає цільової позиції в межах заданої дистанції зупинки (stoppingDistance).

Окремою проблемою у щільних сценаріях ближнього бою є уникнення колізій між союзними агентами під час командних переміщень у вузьких приміщеннях. Для вирішення цієї проблеми параметри локального уникнення перешкод (Local Avoidance) динамічно коригуються залежно від поточної тактичної ролі агента. Лідер підрозділу (Leader) отримує вищий пріоритет навігації, що змушує інших агентів (Support або Flanker) автоматично розступатися та поступатися йому дорогою. Завдяки цій інтеграції з

навігаційною системою рушія переміщення тактичної групи виглядає природно та запобігає виникненню тупикових ситуацій (deadlocks) у дверних отворах чи на сходах.

3.13 Оптимізація обчислювального навантаження та керування пам’яттю в системі ШІ

Розробка систем штучного інтелекту для ігор у реальному часі висуває жорсткі вимоги до продуктивності, оскільки обчислення логіки поведінки неігрових персонажів не повинно призводити до падіння частоти кадрів (FPS). Алгоритм евристичного пошуку *A**, який лежить в основі планувальника GOAP, є обчислювально містким, а його виконання у головному потоці (Main Thread) ігрового рушія Unity для десятків агентів одночасно здатне викликати помітні затримки. Крім того, створення великої кількості вузлів під час розгортання графа створює значне навантаження на систему автоматичного збирання сміття (Garbage Collector) у середовищі .NET. Для вирішення цих проблем у програмній реалізації застосовано комплексний підхід до оптимізації.

Першим ключовим архітектурним рішенням стало рознесення обчислень за допомогою асинхронного програмування. Оскільки розроблений алгоритмічний модуль планувальника повністю ізольований від платформозалежних класів простору імен UnityEngine (таких як Transform чи GameObject), він не підпадає під жорсткі обмеження Unity щодо виконання виключно в головному потоці. Для ініціалізації процесу планування застосовано конструкції async/await на базі об’єктів Task мови C#. Коли координуючому агенту потрібно побудувати новий план, він формує ізольовану копію поточного стану світу і передає її планувальнику, який виконує пошук у фоновому потоці, використовуючи системний пул потоків (ThreadPool). Після знаходження оптимального плану результати безпечно

повертаються у головний потік для подальшого виконання. Такий підхід повністю усуває проблему блокування рендерингу сцени під час прийняття складних тактичних рішень.

Другим важливим аспектом оптимізації є ефективне керування пам’яттю. Під час пошуку шляху у просторі станів алгоритм створює сотні тимчасових об’єктів типу Node для зберігання контексту графа. Виділення пам’яті під них на кожній ітерації неминуче призводить до частих спрацьовувань Garbage Collector і, як наслідок, мікрофризів ігрового процесу. Для усунення цього явища в системі реалізовано шаблон проєктування «Пул об’єктів» (Object Pool). На етапі ініціалізації рівня створюється попередньо виділений масив екземплярів вузлів. Під час роботи планувальник замість створення нового вузла запитує вільний об’єкт із пулу, заповнює його новими значеннями (стан, накопичена вартість, посилання на батьківський вузол), а після завершення реконструкції фінального плану масово повертає всі використані вузли назад. Це знижує обсяг динамічних алокацій пам’яті (GC Allocations) практично до нуля.

Оптимізації також зазнав сенсорний модуль PerceptionModule. Хоча обчислення лінії прямої видимості за допомогою трасування променів виконується досить швидко завдяки внутрішній архітектурі фізичного рушія PhysX, одночасний масовий виклик цих методів для тактичної групи агентів в одному кадрі створює небажаний стрибок навантаження на центральний процесор. Для балансування обчислень застосовано підхід просторового та часового розподілу. Оновлення сенсорів виконується не у кожному кадрі ігрового циклу, а через задані інтервали з частотою 10 разів на секунду (10 Гц). При цьому для кожного окремого NPC під час появи на сцені генерується невелика випадкова затримка старту внутрішнього таймера (фазовий зсув). Завдяки цьому ресурсомісткі запити до фізичної підсистеми від різних агентів рівномірно розмазуються по різних кадрах, що робить графік часу виконання кадру максимально рівним.

Нарешті, для зменшення загальної кількості ресурсомістких запитів до планувальника розширено впроваджений раніше механізм кешування станів. Якщо декілька агентів з однаковими тактичними цілями потрапляють у схожі умови навколишнього середовища, планувальник не перераховує граф простору станів з нуля. Система генерує унікальний цілочисельний хеш на основі поточних параметрів світу та шукає його в LRU-кеші (Least Recently Used). У разі збігу агенту миттєво повертається посилання на готову послідовність дій. У синергії перелічені заходи оптимізації дозволили забезпечити стабільну роботу тактичного симулятора без деградації продуктивності навіть при значному збільшенні кількості інтелектуальних NPC на сцені.

3.14 Інтеграція компонентів та цикл прийняття рішень

Описані вище модулі: сенсорний модуль, планувальник, бібліотека атомарних дій, система цілей, підсистеми пошуку укриттів та командної координації, інтегруються в єдиний конвеєр прийняття рішень, координований класом GOAPAgent. Розуміння повного циклу взаємодії компонентів є важливим для цілісного сприйняття роботи системи, оскільки емерджентна поведінка агентів виникає саме як результат узгодженої взаємодії окремих модулів, а не як властивість будь-якого з них окремо.

Повний цикл прийняття рішень агентом складається з п’яти послідовних етапів. На першому етапі сенсорний модуль оновлює стан світу WorldState на основі сприйняття середовища та внутрішніх параметрів агента. На другому етапі менеджер цілей обчислює пріоритети активних цілей і обирає ціль з найвищим пріоритетом. На третьому етапі планувальник будує оптимальний план досягнення цільового стану з урахуванням доступних дій та поточного стану світу. На четвертому етапі координуючий агент послідовно виконує дії плану, делегуючи фактичну роботу відповідним модулям: навігації,

підсистемі укриттів, системі ведення вогню. На п’ятому етапі, у разі суттєвої зміни стану світу або зміни пріоритетів цілей, поточний план переривається й цикл повторюється від етапу планування.

Така організація забезпечує постійну адаптацію поведінки агента до змін у середовищі: на відміну від статичних сценаріїв, агент перебудовує свій план щоразу, коли цього вимагає ситуація, що й породжує враження «розумної» поведінки. Саме завдяки чіткому розмежуванню відповідальностей між модулями та їх інтеграції через єдиний координуючий компонент досягається поєднання гнучкості, розширюваності й обчислювальної ефективності системи.

Важливою властивістю інтегрованої системи є її стійкість до часткових збоїв: якщо планувальник не може побудувати план для поточної цілі (наприклад, через відсутність доступних укриттів), агент не «застигає», а повертається до стану очікування та повторює спробу планування на наступному циклі вже з оновленим станом світу. Це забезпечує надійність роботи системи протягом тривалого ігрового сеансу, що було підтверджено під час тестування.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано актуальну задачу розроблення адаптивної системи штучного інтелекту неігрових персонажів для тактичних симуляторів ближнього бою на основі методу планування дій, орієнтованого на досягнення цілей (GOAP). У результаті виконання роботи отримано такі основні результати.

– Проведено аналіз сучасного стану систем штучного інтелекту для ігрових NPC, у ході якого розглянуто класичні підходи (кінцеві автомати та дерева поведінки) і виявлено їх принципові обмеження – комбінаторний вибух станів та неможливість адаптації до непередбачених ситуацій; обґрунтовано вибір методу GOAP як архітектурної основи системи;

– розроблено модульну архітектуру системи, що складається з сенсорного модуля, планувальника та бібліотеки атомарних дій із чітким розмежуванням відповідальностей, що забезпечує гнучкість, розширюваність і можливість незалежного тестування компонентів;

– реалізовано алгоритм *A**-планування у просторі станів, що виконує зворотний пошук оптимальної послідовності дій від цільового стану до поточного; застосування механізмів кешування планів та асинхронного планування забезпечило прийнятні показники продуктивності у реальному часі;

– розроблено підсистему пошуку та керування тактичними позиціями (точками укриття), що враховує лінії прямої видимості, відстані й зайнятість позицій, а також підсистему командної координації агентів на основі спільної дошки даних та динамічного розподілу ролей;

– реалізовано динамічну систему цілей агента, у якій пріоритети цілей залежать від поточного стану світу, що забезпечує реалістичне переключення між тактичними намірами агента без явного програмування переходів;

– створено працездатний прототип тактичного симулятора CQB у середовищі Unity, який демонструє адаптивну тактичну поведінку агентів:

флангові обходи, використання укриттів, вогонь на придушення та координовані командні дії;

– проведено функціональне тестування системи та порівняльний аналіз поведінки агентів GOAP і FSM, який підтвердив істотно вищу адаптивність та емерджентність поведінки агентів GOAP за прийнятних обчислювальних витрат.

Практична цінність роботи полягає у створенні відкритої, модульної та добре задокументованої архітектурної основи для розробки інтелектуальних NPC, яка може бути використана у комерційних ігрових проєктах, навчальних симуляторах для силових структур, а також як платформа для подальших досліджень у галузі мультиагентних систем планування. Напрямами подальшого розвитку системи є інтеграція методів машинного навчання для автоматичного коригування вартостей дій, розширення моделі стану світу до нечислових і нечітких значень, а також перенесення алгоритмічного ядра на інші ігрові платформи.

Результати роботи апробовано у вигляді тез доповіді на 30-му Міжнародному молодіжному форумі «Радіоелектроніка і молодь у XXI столітті» (Харків, 2026) [31].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Newzoo. (2024). Global Games Market Report 2024. Amsterdam: Newzoo International B.V. URL: https://newzoo.com/resources/trend-reports (дата звернення 15.04.2026).

2. Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 2(3–4), 189–208.

3. Orkin, J. (2006). Three states and a plan: The AI of F.E.A.R. In Game Developers Conference Proceedings. San Francisco, CA, USA. URL: https://www.gamedevs.org/uploads/three-states-plan-ai-of-fear.pdf (дата звернення 15.04.2026).

4. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107.

5. Colledanchise, M., & Ögren, P. (2018). Behavior Trees in Robotics and AI: An Introduction. Boca Raton, FL: CRC Press, 120–131.

6. Millington, I. (2019). AI for Games. Boca Raton, FL: CRC Press, 33–38.

7. Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2023). Monte Carlo Tree Search: a review of recent modifications and applications. Artificial Intelligence Review, 56(3), 2497–2562.

8. Tomei, L. (2017). ReGoap: Generic GOAP (Goal Oriented Action Planning) library for Unity. GitHub. URL: https://github.com/luxkun/ReGoap (дата звернення 15.04.2026).

9. Unity Technologies. (2023). Unity User Manual (version 2022 LTS). URL: https://docs.unity3d.com (дата звернення 15.04.2026).

10. Jacopin, E. (2014). Game AI planning analytics: The case of three first-person shooters. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 10(1), 119–124.

11. Microsoft. (2023). C# language documentation. Microsoft Learn. URL: https://learn.microsoft.com/dotnet/csharp (дата звернення 15.04.2026).

12. JetBrains. (2023). JetBrains Rider documentation. URL: https://www.jetbrains.com/rider/documentation (дата звернення 15.04.2026).

13. Gorokhovatskyi, V., Tvoroshenko, I., & Yakovleva, O. (2024). Transforming image descriptions as a set of descriptors to construct classification features. Indonesian Journal of Electrical Engineering and Computer Science, 33(1), 113–125.

14. Gorokhovatskyi, V., Tvoroshenko, I., Yakovleva, O., & Hudáková, M. (2026). Feature space quantization and application of metrics in structural methods of image recognition. IEEE Access, 14, 17812–17824.

15. Suprun, A., Tvoroshenko, I., Gorokhovatskyi, V., & Yakovleva, O. (2025). Development and research of a method for the combined use of large language models for text generation. International Journal of Academic and Applied Research, 9(10), 249–263.

16. Larin, I., Tvoroshenko, I., Gorokhovatskyi, V., & Liubchenko, V. (2025). Research and comparison of facial color normalization methods relative to an etalon image. International Journal of Academic and Applied Research, 9(11), 36–47.

17. Bohdan, N., Tvoroshenko, I., Gorokhovatskyi, V., & Kobylin, O. (2025). Development of a hybrid method to enhance context memory for a chatbot application based on large language models. International Journal of Academic Information Systems Research, 9(10), 7–18.

18. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., Gadetska, S., & Al-Dhaifallah, M. (2023). Statistical data analysis models for determining the relevance of structural image descriptions. IEEE Access, 11, 126938–126949.

19. Daradkeh, Y. I., Tvoroshenko, I., Gorokhovatskyi, V., Latiff, L. A., & Ahmad, N. (2021). Development of effective methods for structural image recognition using the principles of data granulation and apparatus of fuzzy logic. IEEE Access, 9, 13417–13428.

20. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., Gadetska, S., & Al-Dhaifallah, M. (2021). Methods of classification of images on the basis of the values of statistical distributions for the composition of structural description components. IEEE Access, 9, 92964–92973.

21. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., & Zeghid, M. (2022). Cluster representation of the structural description of images for effective classification. Computers, Materials & Continua, 73(3), 6069–6084.

22. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., & Al-Dhaifallah, M. (2022). Classification of images based on a system of hierarchical features. Computers, Materials & Continua, 72(1), 1785–1797.

23. Tvoroshenko, I., & Gorokhovatskyi, V. (2022). The application of hybrid intelligence systems for dynamic data analysis. International Journal of Engineering and Information Systems, 6(2), 40–48.

24. Gorokhovatskyi, V., Tvoroshenko, I., & Chmutov, Yu. (2022). Application of systems of orthogonal functions for feature-space formation in image classification methods. Suchasni informatsiini systemy, 6(3), 5–12.

25. Gorokhovatskyi, V., Peredrii, O., Tvoroshenko, I., & Markov, T. (2023). Distance matrix for a set of structural description components as a tool for image classifier creation. Suchasni informatsiini systemy, 7(1), 5–13.

26. Pomazan, V., Tvoroshenko, I., & Gorokhovatskyi, V. (2023). Development of an application for recognizing emotions using convolutional neural networks. International Journal of Academic Information Systems Research, 7(7), 25–36.

27. Gorokhovatskyi, V., Tvoroshenko, I., Kobylin, O., & Vlasenko, N. (2023). Search for visual objects by request in the form of a cluster representation for the structural image description. Advances in Electrical and Electronic Engineering, 21(1), 19–27.

28. Tvoroshenko, I., Gorokhovatskyi, V., Kobylin, O., & Tvoroshenko, A. (2023). Application of deep learning methods for recognizing and classifying

culinary dishes in images. International Journal of Academic and Applied Research, 7(9), 57–70.

29. Творошенко, І. С. (2021). Технології прийняття рішень в інформаційних системах: навч. посібник. Харків: ХНУРЕ.

30. Гороховатський, В. О., & Творошенко, І. С. (2022). Аналіз багатовимірних даних за описом у формі множини компонент: монографія. Харків: ХНУРЕ, 124 с.

31. Веліканов, О. Р. (2026). Розробка адаптивного штучного інтелекту NPC для тактичних симуляторів з використанням методу GOAP. У Матеріали 30-го Міжнародного молодіжного форуму «Радіоелектроніка і молодь у XXI столітті» (Т. 7, с. 38). Харків: ХНУРЕ.