МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«СИСТЕМА АДАПТИВНОГО ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ВОРОЖИХ ЮНІТІВ У СИМУЛЯТОРІ ОПЕРАТОРА НАДВОДНИХ ДРОНІВ»

Виконав: здобувач 2 курсу
Рівень магістр
Галузь знань 12 «Інформаційні технології»
Спеціальність 122 «Комп’ютерні науки»
Струк Нікіта Олександрович

Керівник: к.пед.н., доцент
Логінова Наталія Іванівна

Рецензент: к.т.н., доцент
Трофименко Олена Григорівна

Одеса – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ”

Факультет кібербезпеки та інформаційних технологій

Кафедра інформаційних технологій

Галузь знань: 12 Інформаційні технології

Спеціальність: 122 “Комп’ютерні науки”

Назва освітньої програми: Комп’ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Наталія Логінова
_____“____”_____ 2025 року

ЗАВДАННЯ
на кваліфікаційну (магістерську) роботу
здобувачу Струку Нікіті Олександровичу

1. Тема роботи: “Система адаптивного штучного інтелекту для ворожих юнітів у симуляторі оператора надводних дронів”
   Керівник роботи: к.пед.н, доцент Логінова Н.І.
   затверджені наказом НУ «ОЮА» від «10» жовтня 2025 року № 3182-06.
2. Строк подання здобувачем роботи «25» листопада 2025 рік.
3. Дата видачі завдання «02» червня 2025 рік.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1.	Вибір теми, вивчення літературних джерел та складання плану роботи	15.09.2025	
2.	Підготовка першого розділу роботи та подання його керівнику	01.10.2025	
3.	Підготовка другого розділу роботи та подання його керівнику	14.10.2025	
4.	Підготовка третього розділу роботи та подання його керівнику	02.11.2025	
5.	Підготовка вступу та висновків	10.11.2025	
6.	Доопрацювання роботи з урахуванням зауважень керівника	17.11.2025	
7.	Подача електронного варіанта роботи для перевірки на плагіат	20.11.2025	
8.	Допуск роботи до захисту завідуючим кафедри	25.11.2025	
9.	Захист роботи		

Здобувач _____
(підпис) (ім’я та прізвище)

Керівник роботи _____
(підпис) ім’я та прізвище)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Система адаптивного штучного інтелекту для ворожих юнітів у симуляторі оператора надводних дронів» на здобуття другого (магістерського) рівня вищої освіти за спеціальністю 122 Комп'ютерні науки, освітньою програмою «Комп'ютерні науки». Містить 24 рисунка, 4 таблиці, 27 літературних джерел за переліком посилань та додатки. Робота виконана на 113 сторінках загального тексту і 96 сторінках основного тексту.

Метою роботи є розробка та програмна реалізація системи адаптивного штучного інтелекту для керування ворожими юнітами в симуляторі оператора надводних дронів для підвищення реалістичності навчального процесу та ефективності тренування операторів. Об'єктом дослідження є процес моделювання поведінки ворожих юнітів у симуляційних системах підготовки операторів надводних дронів. Предмет дослідження – методи та алгоритми адаптивної поведінки ворожих юнітів у симуляторі оператора надводних дронів.

У кваліфікаційній роботі розроблено поведінкову модель штучного інтелекту ворожих юнітів для симулятора оператора надводних дронів, яка забезпечує адаптацію їхньої поведінки залежно від дій гравця та умов навколишнього середовища. Реалізовано систему прийняття рішень на основі архітектури Behaviour Trees та StateTree, із використанням AI Perception і Environment Query System для динамічного сприйняття обстановки. Проведено експериментальні дослідження, що підтвердили ефективність розроблених алгоритмів у забезпеченні реалістичності, варіативності та адаптивності поведінки противника. Результатом також є інтеграція створеної моделі у симуляційне середовище Unreal Engine 5, що дозволяє застосовувати її для навчання та тренування операторів надводних дронів у мінливих умовах місій.

РОЗРОБКА ІГРОВИХ ЗАСТОСУНКІВ, СИМУЛЯТОР, БЕЗПІЛОТНИЙ НАДВОДНИЙ АПАРАТ, ШТУЧНИЙ ІНТЕЛЕКТ, UNREAL ENGINE 5

ABSTRACT

Qualification work on the topic "Adaptive Artificial Intelligence System for Enemy Units in Surface Drone Operator Simulator" for obtaining the second (master's) level of higher education in specialty 122 Computer Science, educational program "Computer Science". Contains 24 figures, 4 tables, 27 references in the bibliography, and appendices. The work is completed on 113 pages of total text and 96 pages of main text.

The aim of the work is to develop and implement a software system for adaptive artificial intelligence to control enemy units in a surface drone operator simulator, enhancing the realism of the training process and the effectiveness of operator training.

The object of the research is the process of modeling enemy unit behavior in simulation systems for training surface drone operators.

The subject of the research is methods and algorithms for adaptive behavior of enemy units in a surface drone operator simulator.

The qualification work develops a behavioral model of artificial intelligence for enemy units in a surface drone operator simulator, which ensures adaptation of their behavior depending on player actions and environmental conditions. A decision-making system has been implemented based on Behaviour Trees and StateTree architecture, using AI Perception and Environment Query System for dynamic situational awareness. Experimental studies have been conducted that confirmed the effectiveness of the developed algorithms in ensuring realism, variability, and adaptability of enemy behavior. The result also includes integration of the created model into the Unreal Engine 5 simulation environment, which allows its application for training surface drone operators under changing mission conditions.

GAME APPLICATION DEVELOPMENT, SIMULATOR, UNMANNED SURFACE VEHICLE, ARTIFICIAL INTELLIGENCE, UNREAL ENGINE 5

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

БНА – Безпілотний надводний апарат

ШІ – Штучний інтелект

UE5 – Unreal Engine 5

РЕР – Радіоелектронна розвідка

РЕБ – Радіоелектронна боротьба

EQS – Environment Query System

NavMesh – Navigation Mesh

ВСТУП

Проблема відсутності системи адаптивного штучного інтелекту (ШІ) у симуляторі оператора надводних дронів полягає в обмеженості та передбачуваності поведінки ворожих юнітів, оскільки їхній ШІ реалізований за допомогою простих заскриптованих моделей на основі Blueprints та C++. У цій системі дії противника визначаються попередньо заданими сценаріями або фіксованими умовами переходів між станами. Така обмеженість веде до передбачуваності, відсутності гнучкої реакції на зміну зовнішньої обстановки та дій гравця. Як наслідок, симулятор не має широкого спектру варіативності ситуацій, знижуючи реалістичність та ефективність навчального процесу оператора.

Запропонована система адаптивного ШІ усуває ці недоліки, забезпечуючи можливість самостійного аналізу поведінки гравця, оцінки поточного стану середовища та вибору оптимальної стратегії дій у реальному часі. Завдяки використанню вбудованих у рушій інструментів Behaviour Trees, AI Perception, StateTree та Environment Query System, система здатна динамічно змінювати поведінку ворожих юнітів залежно від контексту. За допомогою цього, вона створює більш реалістичні та непередбачувані сценарії поведінки ворогів.

Розроблення адаптивного ШІ значно підвищить реалістичність, складність та цінність навчальних симуляторів. Адаптивний ШІ сформує середовище, у якому оператор змушений приймати рішення в умовах невизначеності, як у реальних бойових ситуаціях. У результаті, система не лише покращить якість підготовки операторів надводних дронів, але створить основу для подальшого розвитку інтелектуальних симуляцій у військовій, освітній та ігровій сферах.

Мета роботи – розробка та програмна реалізація системи адаптивного штучного інтелекту для керування ворожими юнітами в симуляторі оператора надводних дронів для підвищення реалістичності навчального процесу та ефективності тренування операторів.

Завдання дослідження:

– моделювання поведінки автономних агентів у симуляційних середовищах;

– використання підходів адаптивного керування для реалізації інтелектуальної поведінки ворожих юнітів;

– застосування ігрового рушія Unreal Engine 5 для побудови симуляційного середовища;

– використання систем Blueprints, StateTree, AI Perception, Behaviour Trees та Environment Query System для створення багаторівневої поведінкової моделі штучного інтелекту;

– реалізація адаптивних алгоритмів керування та взаємодії між агентами за допомогою мови програмування C++;

– аналіз ефективності розроблених алгоритмів у різних умовах симуляції;

– проведення порівняльного тестування стратегій поведінки ворожих юнітів у мінливому середовищі.

Об'єкт дослідження – процес моделювання поведінки ворожих юнітів у симуляційних системах підготовки операторів надводних дронів.

Предмет дослідження – методи та алгоритми адаптивної поведінки ворожих юнітів у симуляторі оператора надводних дронів.

Практична цінність отриманих результатів полягає у розробленні та запропонуванні системи адаптивного ШІ для ворожих юнітів у симуляторі оператора надводних дронів, що може бути використана для підвищення ефективності підготовки та тренування операторів безпілотних надводних апаратів. Створена поведінкова модель забезпечує більш реалістичну, динамічну та варіативну взаємодію між гравцем і противником, що робить умови симуляції максимально наближеними до реальних бойових ситуацій. Отримані результати є актуальними для розроблення навчально-тренувальних комплексів, інтерактивних симуляторів та ігрових систем, у яких необхідний високий рівень адаптивності поведінки ворожих об'єктів.

Результати кваліфікаційного дослідження апробовані на міжнародної науково-практичної конференції «Інформаційні технології і автоматизація – 2025» (м. Одеса, 30-31 жовтня 2025 р.) [1], опубліковані у фаховий статті [2] та підтверджені авторським свідоцтвом на комп'ютерну програму «Black Sea Hanter» [3].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА НАЯВНИХ ПІДХОДІВ ДО СТВОРЕННЯ СИСТЕМ ШТУЧНОГО ІНТЕЛЕКТУ У СИМУЛЯТОРАХ

1.1. Аналіз предметної області

На сьогоднішній день, через стрімкий розвиток військово-морських технологій, безпілотні надводні апарати (БНА) стали важливим інструментом у проведені розвідувальних, патрульних, мінних та ударних операціях. Особливу популярність такі дроні набули незадовго після початку повномасштабного вторгнення Росії на Україну. У військових конфліктах, БНА використовуються, не тільки як дрони-камікадзе, але й в якості транспортних засобів для перевезення особового складу, провізії, а також в якості розвідників для спостереження в особливо небезпечних водних територіях.

Із масовим поширенням цих безпілотників, постала потреба у ефективному навчанні пілотів, здатних правильно оцінювати обстановку, приймати рішення в умовах невизначеності та керувати апаратами в реальному часі. У військовій сфері, створити навчальне середовище в реальному житті є складним, а іноді й неможливим, через обмеженість у ресурсах, дорогу вартість проведення та можливий ризик ураження особового складу під час проведення тренування. Для вирішення цієї проблеми, були розроблені віртуальні симулятори та моделювальні системи.

Симулятором оператора надводних дронів є програмним комплексом, який відтворює фізичні можливості апарата, особливості навігації, погодні фактори, взаємодію з об’єктами середовища, а також поведінку потенційних противників. Основним завданням такого застосунку є віртуальне відтворення ситуацій, з якими оператор зіштовхнеться під час виконання бойового завдання [4].

Сучасні тренажерні комплекси повинні не лише симулювати заздалегідь визначенні сценарії, але й мати динамічно змінне віртуальне середовище, у якому присутні інші автономні об’єкти, зокрема ворожі одиниці. Саме взаємодія оператора з противником має створювати умови, наближені до реальних [5]. Слід зазначити, що поведінка ворогів має бути непередбаченою, адаптивною та варіативною. Якщо

оператор помітить шаблони поведінки юнітів, то таке тренування не буде неефективним, оскільки пілот буде реагувати на подію, орієнтуючись на тригери інтелекту противників.

Особливістю надводних дронів як об’єктів управління є їхня залежність від фізичних умов середовища – вітру, течії, хвиль, а також обмеженість можливостей маневру порівняно з безпілотними літальними апаратами. Тому у віртуальному середовищі повинно бути гідродинамічне моделювання, оскільки реалістичність симуляції руху безпосередньо впливає на якість тренування оператора.

Для забезпечення реалістичності ігрових та тренувальних симуляторів застосовують спеціалізовані рушії, зокрема Unreal Engine 5, який містить засоби фізичного моделювання та підтримує системи взаємодії з водною поверхнею, зокрема за допомогою плагінів Buoyancy та системи Material Water System [6, 7]. Додатково, рушій надає можливість створювати як візуальні, так і фізичні ефекти хвиль, інерції, занурення корпусу.

Однак фізична модель є лише одним із компонентів симулятора. Для підготовки оператора необхідні сценарії, де він взаємодіє з ворожими автономними юнітами, що можуть перешкоджати, переслідувати або атакувати.

Важливо, щоб в сценаріях пілот взаємодіяв із інтелектуальним противником, оскільки такі ситуації змушують оператора аналізувати ситуацію, оцінювати ризики, планувати дії та приймати рішення в умовах стресу. Проте, забезпечити правдоподібність поведінки противника не так вже й просто. У багатьох симуляторах така поведінка реалізується за допомогою простих сценаріїв та наборів умов, тобто так званих заскриптованих моделей.

Заскриптований ШІ визначає послідовність дій за принципом «якщо → тоді», така поведінка противника є передбачуваною. Пілот, через певний час взаємодії із симуляцією, розпізнає шаблони дій моделі. Через це, оператор втрачає можливість тренувати навички адаптації. Це основна причина, чому заскриптовані системи не підходять для реалістичного тренування у мінливих середовищах.

Щоб підвищити якість підготовки майбутніх фахівців, противник у симуляторі повинен не виконувати визначену послідовність дій за настанням умов, але,

попередньо проаналізувати обстановку, оцінюючи дії оператора, фактори середовища та надати відповідну реакцію у режимі реального часу. Саме така поведінка є адаптивною.

Адаптивна модель ШІ спирається на концепцію інтелектуального агента, що збирає інформацію про навколишнє середовище, формує внутрішні стани та приймає рішення щодо наступної дії [8]. У контексті симулятора надводних дронів, такий агент має:

- змінювати маршрут пересування на основі положення оператора та об'єктів та стану середовища;
- обирати стратегію атаки, переслідування або уникнення;
- оцінювати загрозу та коригувати свій стан (агресивність, дистанція, курс);
- формувати сценарії взаємодії із середовищем та гравцем.

У сучасних системах штучного інтелекту для цього застосовуються такі інструменти Unreal Engine 5, як:

- AI Perception для сприйняття об'єктів середовища через зір, звук та радіус виявлення [9];
- Behavior Trees для написання логіки прийняття рішень на основі умов та пріоритетів [10];
- State Tree для управління станами поведінки (наприклад, перехід із стану патрулювання у переслідування, у разі виявлення надводного дрону) [11];
- Environment Query System (EQS) для вибір оптимальної точки пересування з урахуванням ситуації [12].

Ці інструменти надають можливість створити багаторівневу модель прийняття рішень, де ворожий юніт не діє за фіксованою схемою, постійно оцінює обстановку.

1.2. Огляд сучасних методів та моделей штучного інтелекту в ігрових середовищах

У будь-яких симуляторах військових операцій важливо мати не тільки реалістичне оточення, але й противники. Існує кілька методів реалізації поведінки воро-

жих юнітів, серед яких:

- заскриптована модель поведінки;
- дерева поведінки;
- дерева прийняття рішень;
- машини станів;
- цілеспрямована поведінка;
- чорна дошка.

Зазвичай у такого роду тренажерах застосовують заскриптовані моделі, засновані на фіксованих правилах, такі як «якщо → тоді». За допомогою даного підходу реалізація базових сценаріїв є швидкою, щоправда він має декілька суттєвих недоліків. По-перше, при достатній кількості повторень симуляції пілот буде передбачати поведінку противника. По-друге, заскриптована модель погано масштабується, при збільшенні кількості умов значно ускладнюється підтримка та модифікація логіки. По-третє, така модель ШІ є потребує багато ресурсів при її подальшому розширенні.

В якості вирішення вище зазначених проблем було розроблено дерева поведінки. Ці дерева є ієрархічним підходом до моделювання прийняття рішень агентом. Дерева поведінки дозволяють структурувати логіку, покращуючи її читабельність і забезпечують можливість формування різних варіантів поведінки залежно від стану середовища. На селекторах та послідовностях будується ієрархія цих дерев: «спробуй план А; якщо провал – план В; інакше – підпослідовності дій». Приклади селекторів та послідовностей у деревах поведінки можна побачити на рис. 1.1 та рис. 1.2.

Дерева поведінки є поєднанням низки методик, які вже здавна використовуються в ігровому штучному інтелекті: ієрархічні машини станів, формування розкладу, планування та виконання дій. Дерева поведінки мають багато спільного із ієрархічними машинами станів, але замість стану, основним структурним елементом дерева є завдання. Завданням може виступати як значення параметру гравця або виконання анімації. Завдання складаються в піддерева, щоб

представляти більш складні дії. У свою чергу, ці складні дії можуть знову складатися в поведінку вищого рівня [13].

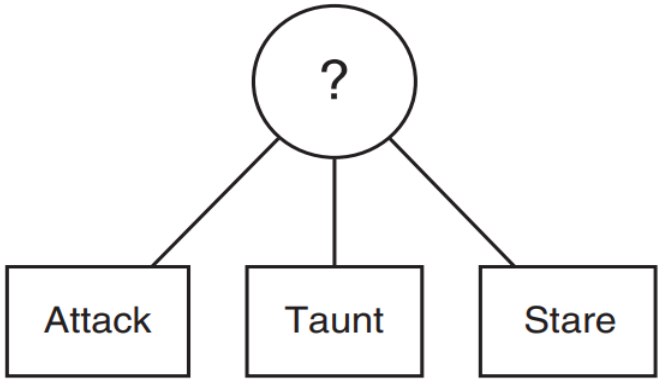

Рисунок 1.1 – Приклад селектору у дереві поведінки [13]

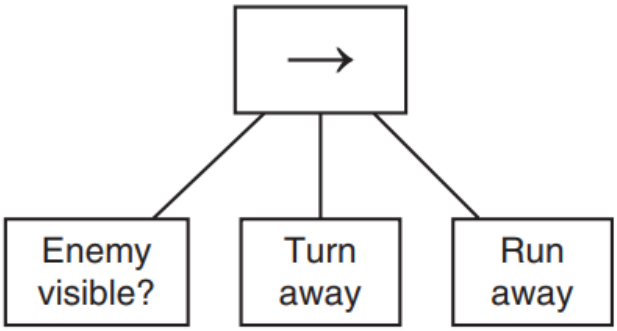

Рисунок 1.2 – Приклад послідовностей у дереві поведінки [13]

Дерева рішень у свою чергу є більш простими, оскільки вони легко реалізуються та простіші у розумінні. Вони є одною із найпростіших та найрозповсюдженіших методик для прийняття рішень. Дерева часто застосовуються для управління персонажами, анімаціями, ігрових системах, елементах оточення, що можуть взаємодіяти із гравцями та неігровими персонажами.

Дерево рішень складається з пов'язаних між собою вершин рішень. Дерево має початкове рішення, його корінь. Для кожного рішення, починаючи з кореня, вибирається один із набору поточних варіантів [13].

Нехай існує простий ШІ юніта, який за при виконанні умови видимості противника перевіряє, чи знаходиться ворог на відстані більше 10 метрів, якщо противник знаходиться за межами зору – перевірити чи його чутно, якщо так –

прокрастися до нього. У випадку, якщо ворог знаходиться за межами відстані 10 метрів, потрібно перевірити, чи знаходиться ворог на фланзі, якщо так – пересунутися на більш безпечну позицію, інакше – атакувати. Приклад такого дерева зображений на рис. 1.3.

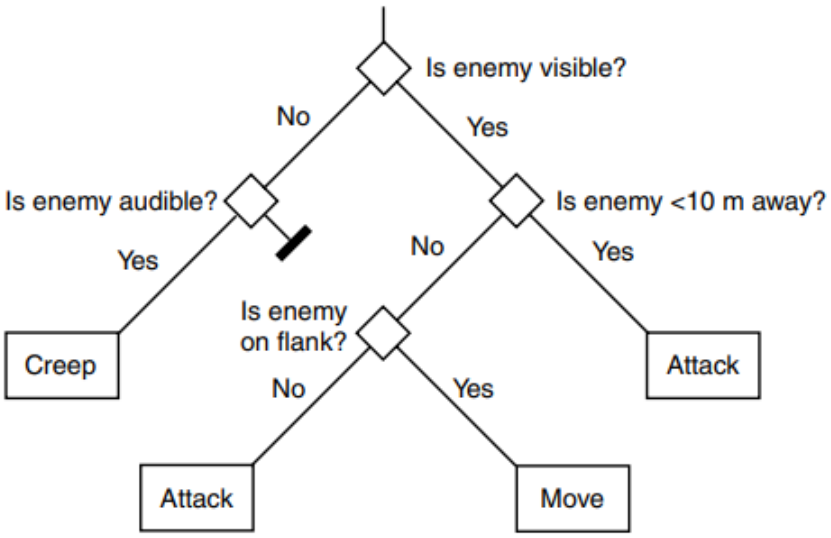

Рисунок 1.3 – Приклад простого дерева рішень ШІ юніта

Кожне рішення ігрового персонажа формується на основі наявних у нього знань. Оскільки дерева рішень часто застосовуються як легковагові та високопродуктивні механізми вибору дій, персонажі здебільшого орієнтуються не на власну внутрішню інформаційну модель, а безпосередньо на глобальний стан ігрового світу. Алгоритм обходу дерева послідовно переходить між вузлами прийняття рішень, виконуючи вибір на кожному з них, доки не досягне листкового елемента. Кожен лист дерева містить певну дію, яка негайно виконується після її вибору алгоритмом [13].

Попри те, що окремі дії в деревах рішень зазвичай є тривіальними, а більшість вузлів має не більше двох відгалужень, важливо забезпечити оптимізовану структуру дерева: уникати дублювання умовних перевірок і розташовувати найбільш ресурсоємні перевірки у нижніх рівнях дерева. Така організація зменшує загальний час обробки умов та виконання відповідних дій, оскільки продуктивність системи визначатиметься найдорожчою перевіркою.

Часто ігрові юніти демонструють обмежений набір поведінкових патернів: вони продовжують виконувати певну дію до появи зовнішнього тригера – події, дії іншого персонажа чи гравця. Наприклад, ворожий солдат може залишатися на позиції, доки не виявить гравця, після чого переходить до активної атаки.

Одним зі способів підвищення варіативності поведінки є введення стохастичних елементів у дерево рішень, наприклад, через генерацію випадкового числа, що визначає вибір дії. Проте такий підхід є складним для масштабування та контролю. Значно ефективнішим рішенням є використання машин станів (state machines), які спеціально призначені для керування поведінкою, що змінюється залежно від подій та контексту.

Машина стану (Finite State Machine) – це методика, яка керує станами об’єкта в залежності від зовнішніх факторів та умов. Вона визначає дискретні стани, події, що викликають переходи між ними, та дії, що виконуються в кожному стані. Це підхід спрощує складну логіку, упорядковує код та полегшує налагодження, оскільки об’єкт може перебувати лише в одному стані [14].

Стани пов'язані між собою переходами. Кожен перехід веде від одного стану до іншого, цільового стану, і кожен має набір пов'язаних умов. Якщо симулятор дізнається, що умови переходу виконані, юніт змінює стан на інший. Отже, коли умови для переходу виконанні, він спрацьовує. Після цього, коли перехід виконано – об’єкт має новий стан [13].

Наприклад, за замовчуванням юніт знаходиться у стані «На варті». Якщо він побачив маленького ворога, він переходить до стану «Бій», якщо він побачив великого ворога – переходить спочатку до стану «Втеча», після чого, назад до «На варті». Таку просту машину стану можна побачити на діаграмі (рис. 1.4).

Вище зазначені техніки прийняття рішень спочатку приймають набір вхідних даних, після чого ведеться перевірка умов, в разі успіху – виконати дію. Інакше кажучи – у персонажа немає цілі, він просто реагує на отримані дані.

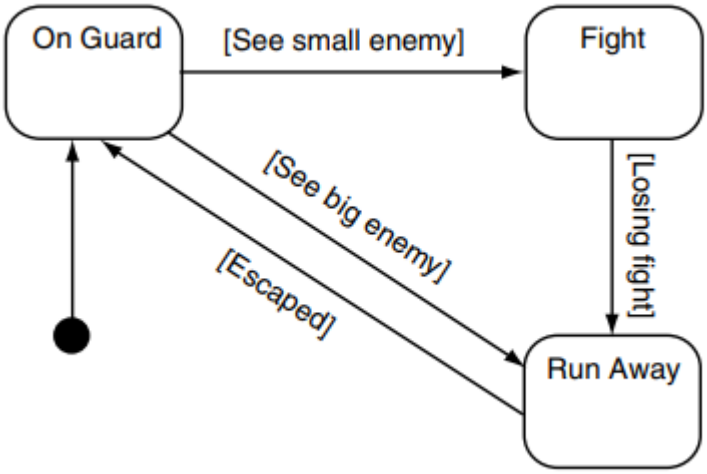

Рисунок 1.4 – Проста машина станів [7]

Можна також реалізувати імітацію наявності певної мети в агента, використовуючи одну з описаних або будь-яку іншу техніку прийняття рішень. Наприклад, якщо юніт має ціль знищити ворога, він переслідує його, атакує, а в разі втрати візуального контакту – починає пошук. Якщо ж пріоритетною метою є збереження власного життя, то юніт шукатиме укриття, а у випадку серйозного поранення – відступатиме.

Однак інколи виникають ситуації, не передбачені в початковому наборі поведінкових правил. Наприклад, якщо до системи додається новий тип надводного дрона з ракетним комплексом, але в логіці берегових охоронців немає поведінки «йти в укриття», то їхня реакція може виявитися некоректною. У таких випадках доцільно наділити юнітів певним набором можливих дій і механізмом вибору тієї, що найкраще відповідає їхній поточній меті. Якщо для охоронця основним пріоритетом є знищення дрону, він обере атаку, навіть ризикуючи життям; якщо ж ключовою метою є виживання – він відступить.

У грі The Sims відсутня єдина глобальна мета. Користувач може самостійно обирати серед широкого спектра доступних дій, таких як сидіння на дивані, спілкування з іншими персонажами, прасування одягу тощо. Персонажі володіють набором параметрів, що визначають їхні фізіологічні та психологічні потреби – голод, спрага, сон, потреба у спілкуванні, емоційне задоволення та інші. Важливо

підтримувати ці показники на оптимальному рівні, щоб вони не досягли критичних значень [15].

Поведінку персонажів у такій грі можна було б формалізувати за допомогою дерева рішень, яке обирає дії на основі поточних фізичних, психологічних та емоційних параметрів. Проте, із зростанням функціоналу, таке дерево могло б налічувати сотні, тисячі або навіть десятки тисяч гілок, що потребує значних обчислювальних ресурсів процесора та пам'яті. Крім того, поведінка, сформована виключно на основі дерев рішень, виглядає надмірно раціональною і неприродною.

Тому в The Sims реалізовано інший підхід до ШІ. Кожен персонаж (сім) постійно аналізує своє оточення, виявляючи об'єкти або взаємодії, які можуть задовольнити його потреби. Кожній можливій дії присвоюється значення корисності, що оцінює, наскільки ця дія відповідає поточним потребам і наскільки її виконання можливе. Після цього агент обирає дію з найвищою корисністю, виконує її та тимчасово задовольняє певну потребу, після чого процес повторюється [15].

Ця архітектура втілює основні риси цілеспрямованої поведінки: присутність явних (хоча і внутрішніх) цілей, сприйняття можливостей навколишнього середовища та вибір дій для задоволення цих цілей. У грі The Sims агенти поводяться адаптивно і цілеспрямовано: вони переходять від їжі до спілкування та відпочинку в міру зміни своїх потреб і появи можливостей у навколишньому середовищі.

На прикладі The Sims демонструється, що цілеспрямована поведінка може реалізовуватися не через довгострокові, заздалегідь визначені сценарії, а завдяки безперервній оцінці поточної ситуації та реактивному прийняттю рішень. Цей підхід має свої переваги та обмеження: він забезпечує спонтанну та гнучку поведінку персонажів, проте вона залишається обмеженою визначеними потребами та доступними у середовищі можливостями.

Зазвичай у ігрових застосунках або симуляторах, персонажі чи юніти не поодинокі. Тобто, виконуючи певну задачу, вони не мають заважати одне одному, а також, координувати свої дії для досягнення спільної мети. До того ж, якщо модель прийняття рішень ШІ включає в себе декілька різних технік, кожна із них буде

пропонувати що робити, але остаточне рішення має бути одне. Для прикладу, двоє різних противників одночасно отримують суперечливу інформацію про місце-знаходження гравця. Через це їхні дії стають неузгодженими, хаотичними та дублюють одна одну.

Чорна дошка (Blackboard) є централізованим сховищем актуальних даних щодо гравця, середовища, інших юнітів та контекстних подій. Завдяки цьому ворожі агенти, синхронізуючи поведінку, обирають узгоджену тактику та реагують на зміну ситуації. По суті, чорна дошка є своєрідною пам’яттю агента, де зберігається вся інформація, яка потрібна для прийняття рішень. Blackboard працює як сховище ключ-значення даних, де кожен ключ має тип (int, float, bool, object, enum і т.д.) [16].

Чорна дошка є основним джерелом даних для таких технік як дерева поведінки та машини станів. За допомогою цієї дошки, патерни дерева поведінки виконують такі дії:

- Decorator перевіряє значення ключа. Наприклад, перевіряємо чи гравець у полі зору через “PlayerVisible == true”;
- Service оновлює ключі;
- Task виконує дію, застосовуючи дані чорної дошки.

В залежності від значень у чорній дошці у машинах станів перевіряються переходи між активними та неактивними станами. Наприклад, персонаж перебуває у стані «Патрулювання». Побачивши гравця, декоратор перевіряє значення булевого ключа PlayerVisible, якщо його значення наразі false, service змінює на true. Після цього, машина станів отримує значення ключа PlayerVisible та змінює стан персонажа на «Переслідування». Після зміни стану, виконується дія-переміщення до останнього відомого місце-розташування цілі. Щоб чорна дошка активувалася, потрібно обробити отримані дані із сенсорних систем. Такими системами є:

- AI Perception, що включає зір, звук, радар сприйняття;
- line trace або трасування променів;
- тригери ігрового рівня;
- події викликані іншими юнітами.

Чорна дошка може бути як персональною для кожного юніта, так і спільною для декількох або усіх персонажів. Пер-агентна або персональна чорна дошка застосовується для індивідуальних задач, серед яких – визначення маршруту, пошуку цілі, зміну внутрішніх станів. Наприклад, помітивши гравця в полі зору, юніт переходить у стан бойової тривоги, визначає маршрут до цілі та переслідує її.

У свою чергу, спільна чорна дошка застосовується для координації одної або декілька груп юнітів. Скажімо, вищезазначений юніт, перейшовши у стан бойової тривоги, повідомляє іншим бійцям про місце-знаходження гравця. Додатково, отримавши дані про положення гравця на рівні, одні юніти оточують периметр, другі – атакують, треті – надають вогневу підтримку.

Для формування адаптивної поведінки штучного інтелекту, агентові потрібно коригувати свої дії залежно від контекстних змін, серед яких: поява нових задач, ускладнення погодних умов, переміщення цілі у інше положення, тощо. Чорна дошка реалізує таку адаптивність через безперервний цикл оновлення ключів, які відображають актуальний стан персонажа та навколишнього середовища.

Основним механізмом такої адаптації є модифікація динамічних станів у процесі виконання поведінкових алгоритмів. Скажімо, існує змінна AlertLevel, яка визначає стан юніта (при значенні 0 – стан «Спокій», при значенні 1 – «Патрулювання», при значенні 2 – «На сторожі», при значенні 3 – «Бойова тривога»). При виконанні умов, система поступово змінює поведінку, від стану «Спокій» до «Бойова тривога». Подібна багатоступенева реакція дозволяє створити плавну, природну та менш передбачувану поведінку, що важливо для адаптивної системи.

Додатково, чорна дошка вирішує жорсткий зв’язок між різними компонентами персонажа. Така проблема виникає, коли сенсори, алгоритми прийняття рішень, навігація та поведінкові модулі взаємодіють напряму. Цей підхід робить ШІ громіздким, важким для налагодження та неоптимізованим. Чорна дошка виступаючи у ролі посередника, підтримує зв’язок між цими модулями. Завдяки цьому різні частини системи не мають прив’язуватися до конкретних реалізацій одна одної. Кожен із них просто читає або записує значення у Blackboard, не знаючи, хто

саме ці значення буде використовувати. Таким чином, складність системи суттєво знижується, що надає можливість легко додавати нову поведінку, не змінюючи наявну.

Для прикладу, воїн-піхотинець має такі компоненти:

- зір та слух як системи сприйняття;
- дерево поведінки як алгоритм прийняття рішень;
- компонент навігації;
- компонента пересування для руху.

Цей юніт побачив гравця, через що, ключ TargetDetected змінив своє значення на true. Отримавши значення цього ключа із чорної дошки, системи сприйняття виконують повторний аналіз середовища, дерево поведінки ініціює рішення переслідування, компонент навігації створює оптимальний маршрут до цілі, а компонент пересування змінює положення юніта у бік гравця. Жоден із цих модулів не викликає інший напряму – їх об’єднує чорна дошка.

Отже, такий ШІ є:

- гнучким, оскільки легко змінити поведінку, додаючи нові ключі або вузли;
- масштабованим, бо можна додавати взаємодію із іншими юнітами без модифікації старого коду;
- стійким до помилок, тому що модулі не залежать від конкретної структури або станів інших компонентів;
- придатним для командної роботи. Програміст, дизайнери ШІ та ігрових рівнів можуть працювати над різними частинами ігрового застосунку, не потребуючи постійного узгодження та переписування наявного функціоналу інтелекту персонажа.

1.3. Аналіз інструментальних засобів реалізації штучного інтелекту

Адаптивна поведінки юнітів вимагає поєднання кількох технік ШІ, описаних у попередньому підрозділі: заскриптованих правил, дерев поведінки, дерев рішень, машин станів, цілеспрямованої поведінки та архітектури «чорної дошки». Ігровий

рушій Unreal Engine 5 надає широкий набір інструментів, що дозволяють реалізувати ці методики.

Blueprints є візуальною скриптовою системою UE5, що дозволяє створювати поведінкову логіку без написання коду [17]. Цей інструмент важливий для прототипування дерев рішень, машин станів та основних реакцій, оскільки надає можливість:

- швидко будувати схеми «якщо → тоді» для заскриптованої поведінки;
- реалізовувати дерева рішень через послідовні гілки із булевими умовами;
- створювати машини станів як окремі Blueprint-графи;
- інтегрувати сенсори (AI Perception), навігацію (AI Move To) та EQS-запити;
- узгоджувати логіку із деревами поведінки через оновлення ключів чорної дошки.

Система Blueprints добре підходить для реалізації методик, описаних раніше, зокрема:

- заскриптовані моделі поведінки – через Event Tick, Branch та прості функції;
- дерева рішень (Decision Trees) – за допомогою комбінації Branch та Select;
- прості машини станів – Switch on Enum або систему Gameplay Tags [18].

Однак для повноцінного симулятора необхідні більш формалізовані системи штучного інтелекту, тоді як Blueprints лише доповнює їхню функціональність. У той час як Blueprints дозволяє реалізовувати логіку на візуальному рівні, C++ забезпечує:

- високу продуктивність (особливо при складних перевірках або великій кількості юнітів);
- розширення можливостей AI Perception, шляхом розробки спеціальних сенсорів (для прикладу, сенсора радара) [19];
- створення власних завдань, сервісів та декораторів дерев поведінки [20];
- гнучке керування навігацією та маршрутизацією;
- реалізацію оптимізованих алгоритмів корисності дій та цілеспрямованої поведінки.

Раніше згадані техніки реалізуються на мові C++ наступним чином:

- машини станів – як класи таблицями переходів;
- дерева рішень – через структури даних, які перебирають умови;
- цілеспрямована поведінка – як система, що підраховує вагу кожної дії;
- чорна дошка – як набір UBlackboardComponent API-викликів для роботи з ключами.

Behaviour Trees (BT) – головний інструмент UE5 для створення реактивної ієрархічної поведінки [21]. Він дозволяє структурувати логіку у вигляді таких комбінацій:

- Selector – вибір кращої дії;
- Sequences – виконання дій за порядком;
- Tasks – конкретні дії (MoveTo, DealDamage);
- Decorators – логічні умови;
- Services – періодичне оновлення даних.

На рис. 1.5 наведений приклад Behaviour Tree, який демонструє ШІ персонажа, що змінює поведінку між режимами патрулювання та переслідування гравця.

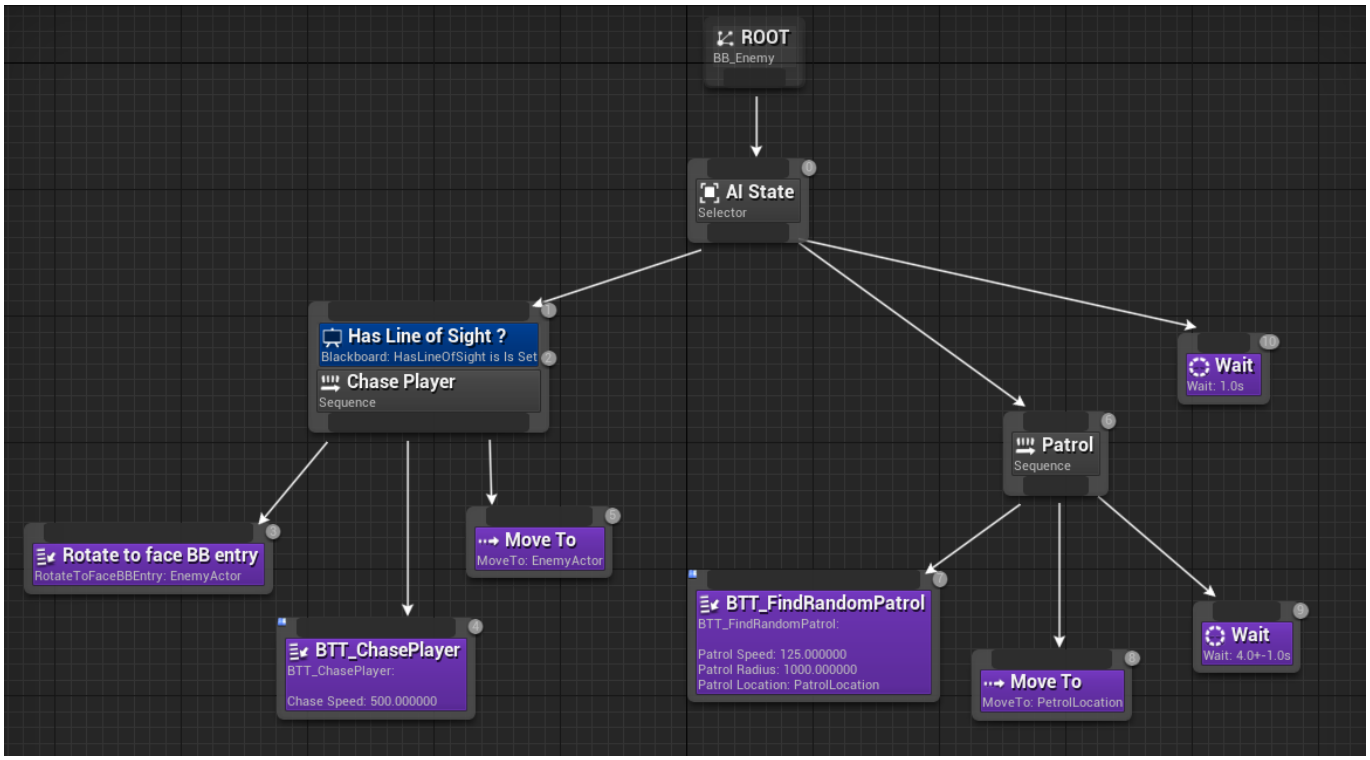

Рисунок 1.5 – Приклад простого Behavior Tree

Blackboard у UE5 використовується як центральна пам’ять ШІ, що реалізує

описану у файлі архітектуру «чорної дошки» та потрібний для:

- синхронізації даних між вузлами BT;
- збереження останніх спостережень AI Perception;
- передачі даних між різними техніками, такими як State Machines, Environment Query System та Goal-Oriented Behavior System;
- координації груп юнітів;
- реалізації адаптивної поведінки через динамічні змінні (для прикладу, AlertLevel для рівня готовності, HealthState для стану здоров'я юніта, TargetActor для визначення цілі юніта, TargetLastKnownLocation для координат останнього відомого місце-розташування цілі).

Чорна дошка інтегрує згадані раніше техніки, виконуючи роль посередника між ними. Наприклад, дерева рішень оновлюють ключі в чорній дошці, після чого машини станів змінюють стани відповідно до цих ключів. Далі система цілеспрямованої поведінки оцінює корисність різних дій для гравця на основі актуальних ключів, а наприкінці дерева поведінки зчитують ці ключі та виконують відповідні дії.

EQS – інструмент просторового аналізу та вибору оптимальних точок для виконання дії штучного інтелекту. Він вирішує, де на ігровому рівні агент повинен діяти. EQS фактично реалізує просторові рішення, які у класичних моделях ШІ описуються як переходи цілеспрямованої поведінки, оцінки корисності або дерева рішень [22].

Для прикладу, EQS для юніта-піхотинця виконує такі функції:

- пошук найкращої точки для перехоплення або атаки;
- у разі обстрілу – пошук укриття або оптимальної траєкторії відходу;
- аналізує навколишнє середовище з урахуванням ландшафту, перешкод, видимості та можливостей (для прикладу – укриття, транспортного засобу, ящиків амуніції чи зброї);
- надає результати Behaviour Tree та StateTree.

Для того щоб перевірити, як працює EQS на практиці, потрібно створити новий C++ клас EQSTestingPawn. Після створення такого класу, його можна побачити у Viewport редактора рівнів. На рис. 1.6 показана фігура пішака, на місті якого буде

персонаж, яким можна керувати, а точки є просторові рішення, одну із яких система обирає.

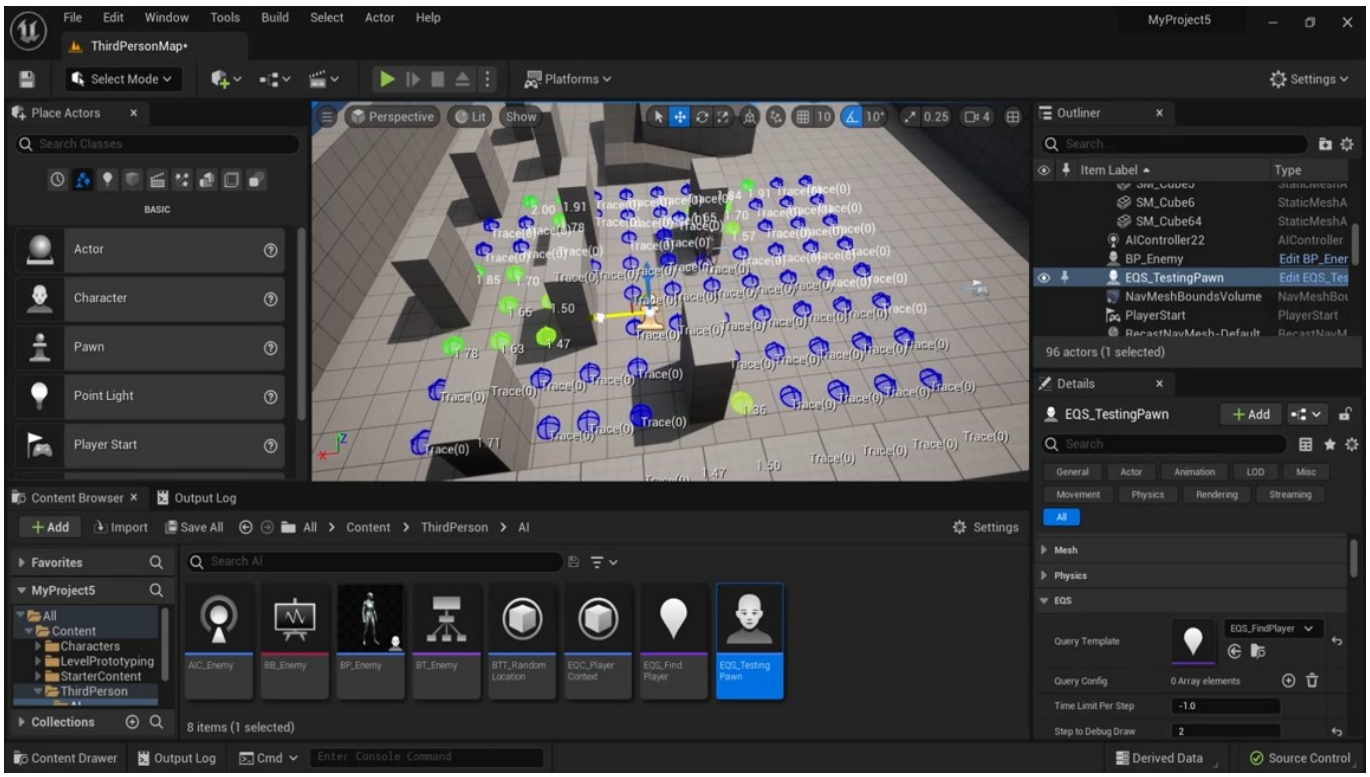

Рисунок 1.6 – EQSTestingPawn на ігровому рівні [23]

StateTree або ST є сучасною системою UE5, що об’єднує можливості дерев поведінки та машин станів [24], яка ідеально підходить для реалізації таких методик:

- машина станів– StateTree забезпечує структуру станів і переходів;
- дерева рішень – через conditions та selectors;
- дерева поведінки – через вкладені піддерева;
- цілеспрямована поведінка – через пріоритетні стани та дані корисності;
- адаптивна система – через легке оновлення змінних від Blackboard.

StateTree підходить для контролю глобальної поведінки, тоді як Behaviour Trees опрацьовують локальні завдання.

Таким чином, StateTree з’єднує логіку у єдину систему, усуваючи хаос, який виникає при масштабуванні Behaviour Trees на сотні вузлів.

На рис. 1.7 наведено приклад StateTree.

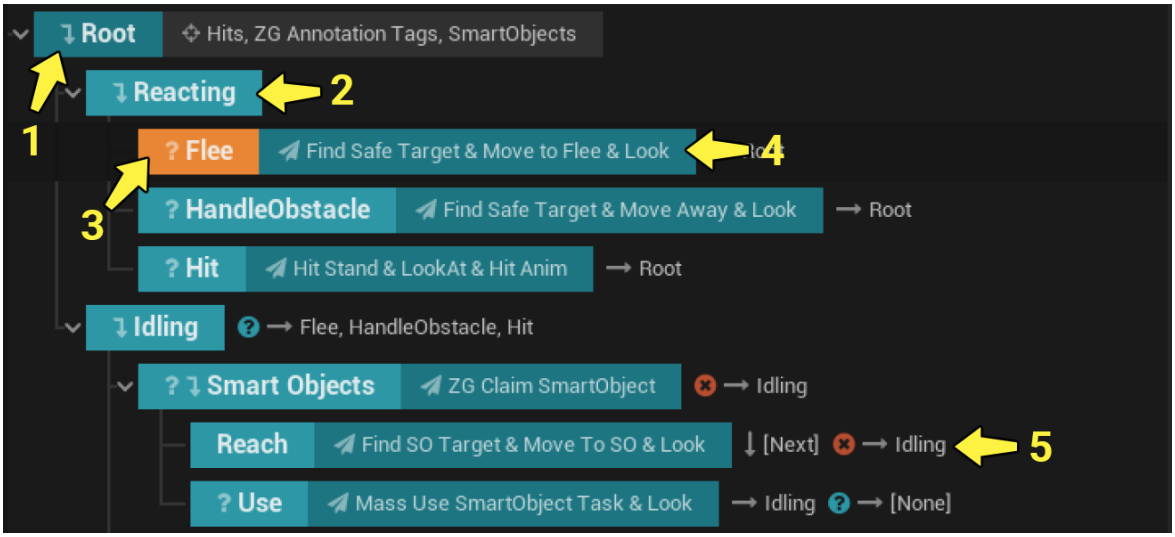

Рисунок 1.7 – Приклад простого StateTree [24]

Позначками вказані такі елементи StateTree:

1. Root (кореневий елемент) – перший стан, який обирається при запуску StateTree.

2. Selector state (вибірник станів) – стан, який має дочірні стани. Він ніколи не буде обраний безпосередньо. Вибрати можна один з його дочірніх станів.

3. State Enter Condition (умова входу в стан) – перелік умов, які визначають, чи може стан бути обраним.

4. Task (завдання) – набір дій, що належать до стану і виконуються, коли стан є активним.

5. Transition (перехід) – визначає умови, що запускають процес вибору стану. Перехід запускається, коли завдання завершується, виконується успішно або завершується невдало, або коли відстежувана умова виконується успішно.

Порівняння розглянутих інструментів ігрового рушія Unreal Engine 5 наведено у табл. 1.1.

Таблиця 1.1 – Порівняння інструментів Unreal Engine 5

Інструмент	Призначення	Переваги	Найкращий сценарій використання
Blueprints	Візуальне створення логіки	Швидке прототипування, читабельність	Простий ШІ, невеликі задачі, інтеграція подій
C++	Низькорівнева реалізація	Продуктивність, контроль, легке доповнення коду	Складна логіка, оптимізація, власні модулі

Продовження таблиці 1.1

Інструмент	Призначення	Переваги	Найкращий сценарій використання
Behaviour Trees	Реактивна поведінка	Модульність, масштабування	Тактичні дії: атакувати, переслідувати
Blackboard	Спільна пам’ять агента	Контекст, адаптивність, координація	Обмін інформацією між модулями ШІ
EQS	Просторові рішення	Оптимальні точки, навігація	Пошук укриття, позиціювання, перехоплення
StateTree	Глобальна FSM	Структуровані режими поведінки	Високорівневі стани ворога, керування боєм

Розглянуті інструменти Unreal Engine 5 забезпечують комплексну систему для створення адаптивного та масштабованого ШІ, поєднуючи візуальне програмування, низькорівневу оптимізацію, модульну поведінку агентів, спільну пам’ять та просторове прийняття рішень.

Висновки з першого розділу

У розділі проведено аналіз сучасних методів ШІ та інструментальних засобів Unreal Engine 5. Встановлено, що традиційні заскриптовані моделі не забезпечують достатньої варіативності та не дозволяють створювати реалістичні тренувальні сценарії. Сучасні підходи, такі як дерева поведінки, дерева рішень, машини станів, цілеспрямована поведінка та чорна дошка, надають можливість формувати складні та адаптивні моделі, здатні реагувати на зміну умов середовища та дій користувача.

Аналіз інструментів UE5 показав, що: Behaviour Trees ефективно моделюють тактичну поведінку агентів; StateTree керує стратегічними станами; Blackboard забезпечує адаптивність і координацію через обмін інформацією; EQS дозволяє вирішувати завдання просторової оптимізації; комбінація Blueprints та C++ забезпечує баланс між гнучкістю та продуктивністю. З цього випливає, що Unreal Engine 5 відповідає вимогам для реалізації адаптивного ШІ в симуляторі операторів надводних дронів, закладаючи основу для теоретичного проєктування моделі ШІ, яке буде здійснене у наступному розділі.

2 ДИЗАЙН АДАПТИВНОЇ СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ

2.1 Постановка задачі та формалізація поведінкової моделі

В симуляторі наявними є наступні сценарії операцій:

- розвідка та спостереження в прибережних водах;
- знищення кораблів російського чорноморського флоту;
- доставка необхідного спорядження та провізії;
- розмінування;
- перевезення особового складу на інший берег.

Основною метою розвідки та спостереження прибережних зон є отримання достовірних відомостей щодо дислокації ворожих кораблів, характеристик берегових об’єктів, стану систем охорони та наявності мінних загороджень. У межах такої місії оператор повинен забезпечити малопомітне переміщення, враховуючи метеорологічні умови, радіус дії засобів радіоелектронної боротьби (РЕБ) і радіоелектронної розвідки (РЕР), а також зони спостереження берегової охорони.

У місії типу «знищення кораблів російського чорноморського флоту» надводний дрон функціонує як засіб ударної дії за принципом «камікадзе». Завдання оператора включають проникнення до охоронюваних акваторій, проходження складних маршрутів у середовищі із перешкодами, ухилення від ворожого вогню та впливу РЕБ, а також ініціювання самознищення у визначений момент.

У сценарії «перевезення особового складу на інший берег» дрон використовується для транспортування боєздатних підрозділів у безпечні райони, з перспективою подальшого застосування для евакуації поранених. На відміну від бойових операцій, такі місії потребують від оператора високої точності маневрування, раціонального використання енергоресурсів і уникнення як природних загроз (підвищеної хвильової активності, сильного вітру), так і штучних перешкод (мінних полів, бонових загороджень).

У місіях з «доставки спорядження та провізії» надводний безпілотний апарат транспортує матеріальні ресурси до визначених пунктів. Аналогічно до

транспортних операцій із переміщення особового складу, оператор повинен демонструвати високу маневровість, дотримуватися режиму економії заряду та уникати перешкод різного походження.

У контексті місій із розмінування передбачається обстеження акваторії на предмет виявлення мін та бонових загороджень із подальшим їх нейтралізуванням. Окрім перелічених ризиків, під час виконання таких завдань можуть виникати додаткові ускладнення: обмежена видимість, сильні течії, недостатній простір для маневру та присутність ворожих кораблів або берегових вогневих постів.

Функціонування симулятора неможливе без моделювання багаторівневої інфраструктури противника. Усі місії передбачають наявність ворожих юнітів, кожен з яких відіграє специфічну роль та взаємодіє з оператором відповідно до власної логіки.

Берегова охорона виступає як стаціонарна й мобільна система виявлення, здатна здійснювати огляд акваторії за допомогою прожекторів, камер спостереження, та засобів РЕР. На початку місії, вороги знаходяться у стані «спокою». У разі виявлення дрону на короткий проміжок часу, охоронці проводять додаткову перевірку, змінюючи свій стан на «спостереження». Якщо, через певний проміжок часу, НБА досі знаходиться у полі видимості РЕР чи юнітів – вони підіймають тривогу. У момент втечі дрону від переслідувачів, стан ворогів змінюється на «на сторожі», а самі юніти будуть патрулювати територію, де останнього разу бачили БНА. Якщо дрон не з’являвся у полі зору довгий проміжок часу, настає стан «спокою», інакше – «тривога». Після впровадження адаптивної моделі ШІ, такі юніти матимуть змогу змінювати маршрути, збільшувати присутність на конкретних ділянках та координувати дії між собою.

Військові кораблі здатні переслідувати дрон та ввести вогонь важким озброєнням. Вони патрулюють акваторії, загороджують маршрути та перекривають підходи до портів. Із покращенням ШІ, такі юніти зможуть координувати дії між собою та викликати авіацію.

Авіація, у ролі якої виступають гелікоптери, відповідає за розширення системи спостереження. Завдяки висоті польоту вони можуть відстежувати переміщення

дрона на великих відстанях, безперешкодно його переслідувати та ввести ураження по БНА. Із розширенням можливостей ШІ, авіація матиме змогу передавати дані наземних та морським юнітам, координуючи із ними дії.

Заскриптований ШІ обмежений у реагуванні на складні та непередбачувані сценарії, що виникають у динамічному середовищі симулятора. Найбільшою проблемою є передбачуваність маршруту патрулювання, через що оператор може легко ідентифікувати слабкі місця та обходити їх. Другою проблемою є те, що різні юніти не мають змоги координувати дії між собою. Для прикладу, гелікоптер, який втратив дрон із поля зору, не передає інформацію береговій охороні або кораблям. Третя проблема – майже або повна відсутність реакції на контекст середовища. Статичний ШІ продовжує виконувати запрограмовані дії незалежно від зміни погодних умов, дій оператора або нових загроз. У результаті симуляція втрачає реалістичність, а оператор отримує неякісну підготовку. Четвертою є нездатність моделювати навчання або пам’ять. Статичний AI щоразу виконує однакову тактику, не враховуючи попередні невдалі або успішні спроби оператора. У сучасних військових операціях противник здебільшого адаптується, підсилюючи слабкі зони, змінюючи інтенсивність спостереження або активуючи додаткові засоби захисту. Симулятор повинен відтворювати такі нюанси. П’ята проблема пов’язана з недостатністю контекстних рішень у складних середовищах, наприклад у вузьких каналах, у районах з боновими загородженнями, в умовах малої видимості. Статичні моделі не здатні динамічно обирати оптимальні точки позиціонування, ухилення або атаки, що суттєво погіршує реалістичність симуляції.

Різні типи ворожих юнітів – берегова охорона, кораблі та авіація – виконують відмінні функції, але взаємодіють між собою та впливають на перебіг місії. У реальних умовах ці юніти діють як частини єдиної інформаційно-бойової системи з обміном даними, зміною режимів готовності, залученням резервів та перекриттям напрямків руху. У симуляторі ж наявні моделі ШІ не відтворюють таких типів взаємодій: юніти реагують лише на окремі події, не мають доступу до спільної пам’яті та не враховують історію власних дій. Отже, основною проблемою є відсутність системного підходу до моделювання колективної активності противника.

Взявши до уваги вище-зазначені проблеми та недоліки наявної системи ШІ, потрібно створити адаптивну систему ШІ, здатну забезпечувати реалістичну модель поведінки ворожих юнітів. Вона має містити кілька рівнів прийняття рішень – від локальних тактичних реакцій до координації групових дій. Також, ця система ШІ повинна надати кожному агенту можливості:

- сприйняття навколишнього середовища;
- формування внутрішніх станів на основі даних отриманих від сенсорів;
- обирати стратегію;
- коригувати поведінку залежно від дій оператора, стану середовища та тактичної ситуації.

У межах симулятора це означає:

- мати можливість змінювати маршрут патрулювання;
- зміна режимів бойової готовності;
- комунікація між різними типами юнітів;
- здатність прогнозувати потенційну поведінку оператора.

Кожна адаптивна модель ШІ юнітів має багаторівневу структуру, яка складається з трьох рівнів:

- нижній рівень – реагування на прості поведінкові дії;
- середній рівень – виконання тактичних сценаріїв (патрулювання, переслідування, відступ);
- верхній рівень – прийняття стратегічних рішень (координація з іншими юнітами, запит на підкріплення, зміна пріоритетної зони патрулювання).

Створена модель поведінки противника має передбачати правильну взаємодію таких компонентів:

- модулів сприйняття даних або сенсорів (AI Perception);
- механізмів прийняття рішень (Behavior Trees, StateTree, Decision-making Models);
- контекстної пам’яті (Blackboard).

З урахуванням аналізу наявних систем ШІ, необхідно забезпечити адаптивність, яка реалізується через механізм контекстних змінних, що оновлюються в реальному часі. У якості таких змінних виступають: рівень загрози, останнє відоме місце розташування цілі, наявність тих чи інших погодних явищ, ресурсні характеристики юніта, наявність союзників у зоні дії. Додатково, потрібно визначити алгоритми, які дозволять системі коригувати поведінку залежно від змін значень цих параметрів. Таким чином, поведінка стане більш варіативною та менш передбаченою для оператора.

Кожен юніт у симуляторі є інтелектуальним агентом, який визначається множиною:

$$A=\langle S, P, M, Pi, U, T\rangle \quad (2.1)$$

де S – множина внутрішніх станів агента;

P – сенсорна модель (сприйняття середовища);

Pi – множина доступних дій;

U – функція корисності, що визначає доцільність дії;

T – функція переходів між станами.

Стан агента визначає його високорівневу поведінку. Стани кожного агента можна представити у вигляді множини:

$$S=\{S_0, S_1, S_2, S_3, S_4, S_5, \} \quad (2.2)$$

де S_0 = *Idle* – спокій

S_1 = *Patrol* – патрулювання;

S_2 = *Investigate* – обстеження; перевірка підозрілих сигналів або наявності дрона у полі зору;

S_3 = *Alert* – тривога;

S_4 = *On guard* – на сторожі; пошук втраченої цілі;

S_5 = *Retreat* – відступ / уникнення;

На початку симуляції, усі юніти мають стан «Спокій». Коли берегова охорона, корабель чи гелікоптер з’явився на межі рендерінгу, тоді вони переходять у стан «Патрулювання». Якщо надводний дрон з’явився на короткий проміжок часу полі зору противника або у радіусі дії РЕР, тоді юніти що помітили на мить гравця,

переходять у стан «Обстеження». У випадку, коли дрон перебував довго у полі зору або радіусу дії РЕР, абсолютно усі юніти переходять у стан «Тривога». Коли дронові вдалося втекти від переслідування, усі юніти змінюють свій стан на «На сторожі» та шукаю втрачену ціль. Через певний проміжок часу, якщо дрон досі не з'явився у полі зору противника або у радіусі дії засобу РЕР, стан усіх юнітів повертається до «Патрулювання», в іншому випадку – «Тривога».

Систему сприйняття даних або AI Perception формально можна представити як функцію:

$$f: E \rightarrow O, \tag{2.3}$$

де E – множина характеристик середовища: положення гравця, видимість, шум, погодні умови;

O – множина сенсорних спостережень.

Беручи до уваги раніше описані типи даних, які сприймаються цими сенсорами, отримаємо таку множину O:

$$O=\{O_{\text{зір}}, O_{\text{звук}}, O_{\text{радар}}, O_{\text{погода}}, O_{\text{тригери}}\} \tag{2.4}$$

Blackboard можна представити як множину, заповнену елементами ключ–значення:

$$M=\{(k_1, v_1), (k_2, v_2), \dots (k_n, v_n),\} \tag{2.5}$$

Кожен ключ множини належить до одного із наступних типів:

- *bool* – булевий (для мітки «Чи видно ціль у полі зору/ радіусу дії радару?»);
- *float* – числовий із плавучою точкою (для мітки «Вплив погоди на видимість»);
- *int* – цілочисельний (для мітки «Рівень небезпеки»);
- *vector* – векторний, для координат у системі XYZ (для мітки «місце-розташування цілі»);
- *object* – об'єктний (для мітки «Ціль»);
- *enum* – перелічення (для мітки «Стан дрону»).

Доступні варіанти дій агента представлені у множині:

$$U: S \times M \times O \times Pi \rightarrow R \tag{2.6}$$

де a_1 = *MoveTo(location)* – перехід на визначені координати;

a_2 = *Shoot(target)* – відкрити вогонь по визначеній цілі;

a_3 = *Investigate(location)* – перевірити наявність цілі у радіусі кола, центром якого є визначена координата;

a_4 = *FollowRoute(route)* – пройти визначеним маршрутом;

a_5 = *RetreatTo(cover)* – сховатись за укриття;

a_6 = *CallForSupport(type)* – здійснити запит на підкріплення одним із трьох типів юнітів: береговою охороною, кораблями або авіацією.

За допомогою функції корисності визначається доцільність виконання дії. Наприклад, щоб визначити корисність переслідування дрона для корабля, отримаємо:

$$U_{attack}=w_1 \cdot D_{detect}+w_2 \cdot (1 - D_{risk})+w_3 \cdot D_{distance} \quad (2.7)$$

де D_{detect} – рівень впевненності у виявленні цілі;

D_{risk} – ризик ураження (через хвилі, вітер, міни);

$D_{distance}$ – значення дистанції до цілі.

Зміну стану описується як:

$$T: S \times M \times O \to S \quad (2.8)$$

Для прикладу, якщо юніт переходить до стану бойової тривоги із обстеження території, отримаємо таке рівняння:

$$T\left(S_2, k_{TargetVisible}=1\right)=S_3 \quad (2.8.1)$$

У випадку, коли юніт переходить у стан пошуку цілі, буде таке рівняння:

$$T\left(S_3, k_{TargetVisible}=0\right)=S_4 \quad (2.8.2)$$

Якщо ціль не з’являлася у полі зору через певний проміжок часу, тоді юніт переходить із стану «На сторожі» у «Патрулювання»:

$$T(S_4, timerExpired=1)=S_1 \quad (2.8.3)$$

BehaviorTree можна представити як:

$$BT=\langle N, R, \delta \rangle \quad (2.9)$$

де N – множина вузлів, якими є селектори, послідовності, декоратори, сервіси та завдання;

R – корінь дерева;

δ – функція виконання вузлів.

Кожен вузол має такий статус виконання:

$$status \in \{Success, Failure, Running\} \quad (2.10)$$

Результатом функції виконання вузлів буде значення Success, якщо будь-який дочірній вузол повернув Success. Якщо всі дочірні вузли повернули Failure, тоді функція виконання вузлів поверне значення Failure.

Машину станів або StateTree представлено:

$$ST=\langle S, S_0, T, \Phi \rangle \quad (2.11)$$

де S – множина станів;

S_0 – початковий стан;

T – переходи;

Φ – множина умов активації станів.

Якщо умова виконується, значення $\Phi = 1$, інакше $\Phi = 0$. Для прикладу, для переходу юнітів у стан «Тривога», потрібно щоб ціль була у полі зору та значення змінної AlertLevel, яке відповідає за рівень бойової готовності:

$$\phi_{alert}=\left(k_{TargetVisible}=1\right) \wedge (AlertLevel \geq 2) \quad (2.12)$$

Формалізовані просторові рішення Enviroment Query System або EQS представлені нижче:

$$argmax_{x \in \Omega}\left(f(x)\right) \quad (2.13)$$

де Ω – множина доступних точок на рівні ;

$f(x)$ – функція оцінки (безпека, видимість, доступність шляху) ;

Для прикладу, нижче описана функція оцінювання простору на можливість переслідування або атакування цілі:

$$f(x)=w_1 V(x) - w_2 R(x)+w_3 C(x) \quad (2.14)$$

де $V(x)$ – видимість цілі з точки x;

$R(x)$ – ризик бути враженим;

$C(x)$ – ціна шляху.

Отже, наведена формалізація забезпечує цілісне математичне підґрунтя для побудови адаптивної моделі поведінки ворожих юнітів, у якій сприйняття даних, механізми прийняття рішень, контекстна пам’ять та просторові обчислення

взаємопов’язані та дозволяють створити реалістичну, варіативну та контекстно залежну систему ШІ.

2.2. Розроблення архітектури адаптивного штучного інтелекту

Оскільки для побудови адаптивної системи ШІ буде застовано декілька інструментів та підходів, потрібно створити багаторівнему, модульну та гнучку архітектуру, яка має стабільно працювати при умовах неповної інформації, постійних змін симуляційного середовища та непередбачуваної поведінки гравця. Важливо створити архітектуру, яка забезпечує виконання повного циклу обробки даних, від отримання даних сенсорами до прийняття рішення, адаптації й виконання дій, з використанням технік реактивної, процедурної та цілеспрямованої поведінки.

Архітектура системи заснована на концепції ієрархічного розподілу відповідальностей, за якою кожен рівень реалізує окремий аспект поведінки агента:

- отримання даних від сенсорів;
- агрегування та збереження контексту у моделі пам’яті;
- вибір стратегії поведінки відповідно до поточного стану агента;
- формування тактичних рішень із застосуванням Behaviour Trees;
- аналіз простору та планування руху за допомогою EQS і Navigation Map;
- виконання дій у симуляційному середовищі;
- моніторинг результатів і адаптивне корегування параметрів ШІ.

Загальна структура розробленої системи представлена на рис. 2.1.

Архітектура поділена на дев’ять основних рівнів:

- симуляційне середовище;
- оператор та налаштування місії;
- навігація та аналіз простору;
- прийняття рішень;
- моніторинг та логування;

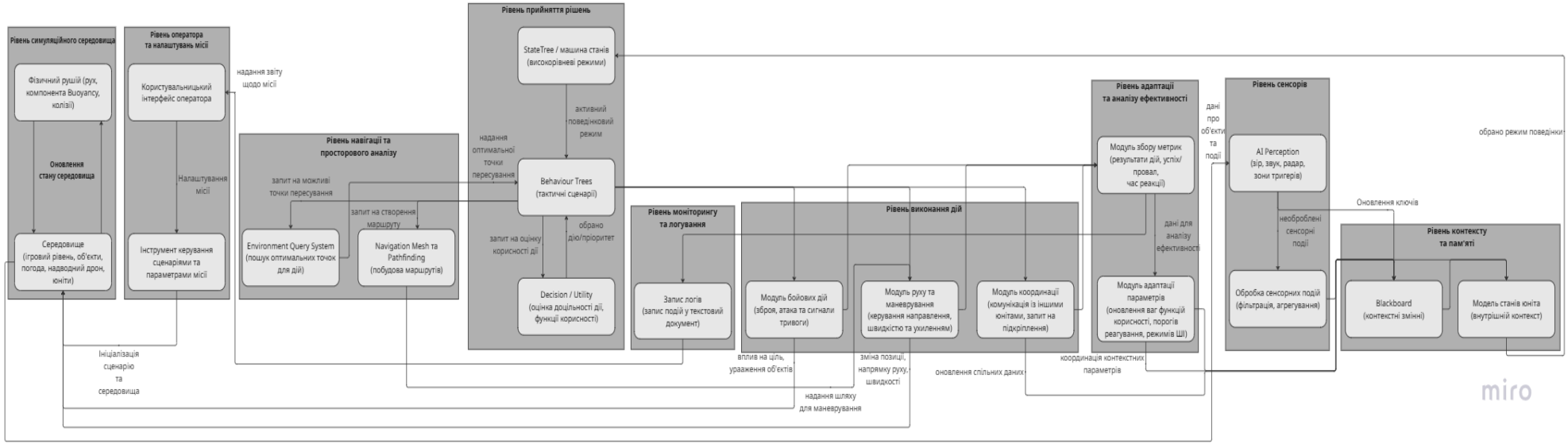

Рисунок 2.1 – Архітектурна діаграма рівнів адаптивного ШІ

- виконання дій;
- адаптація та аналіз ефективності;
- сенсори;
- пам’ять та контекст.

Такий поділ дозволяє відокремити високорівневу логіку управління від низькорівневих механізмів, забезпечуючи корректний зв’язок між компонентами. Додатково, ці модулі можна вдосконалити без порушення ієрархії, оскільки вони незалежні один від одного.

Найнижчім рівнем архітектури є симуляційне середовище, який включає фізичний рушій, гідродинамічну модель (компонент Buoyancy), механізми колізій, погодні умови, ігровий рівень та всі об’єкти на ньому. На цьому рівні формується об’єктивний стан середовища, який змінюється під впливом дій гравця, ШІ та подій всередині симулятора. Вихідні дані цього рівня застосовуються для сенсорних систем сприйняття та подальшого прийняття рішень.

Наступним є рівень сенсорів, на якому модулі AI Perception отримує інформацію про події та стан об’єктів у середовищі. Отримані сенсорні сигнали передаються до підсистеми оброблення сенсорних подій, де дані фільтруються, класифікуються та агрегуються. У результаті формується структурований потік спостережень, який відображає важливі для агента зміни в оточенні.

Відсортовані та структуровані сенсорні дані переходять до рівня контексту та пам’яті, де розміщено Blackboard та модель внутрішнього стану агента. Blackboard виконує роль централізованого сховища контекстних змінних, серед яких – ціль, рівень загрози, останнє відоме місце-розташування, стан здоров’я юніта, тощо. На цьому рівні відбувається узгодженість між різними компонентами ШІ, завдяки чому, кожен модуль може використовувати спільну інформацію без жорстких залежностей.

Одним із найважливіших елементів системи є рівень прийняття рішень, який містить StateTree, Behaviour Trees та Utility/Decision Models. StateTree визначає високорівневі стани, тоді як Behaviour Trees реалізують тактичні сценарії у межах активного стану юніта. Модуль Utility та системи оцінки корисності надають числову оцінку можливим діям агента, що забезпечує адаптивний вибір стратегії

залежно від ситуації. Всі рішення приймаються на основі контексту, отриманого з попереднього рівня.

Для формування рішень щодо переміщення у просторі використовується рівень навігації та просторового аналізу. Environment Query System здійснює пошук оптимальних точок для переслідування, переходу або укриття. NavMesh створює маршрути з урахуванням структури середовища. Ці інструменти надають можливість агентам реагувати на ситуацію не лише логічно, а й оптимально відносно простору.

Виконання обраних рішень реалізує рівень виконання дій. У ньому знаходяться модулі руху, маневрування, ухилення, відкриття вогню, активація сигналу тривоги та координації з іншими юнітами. Всі зміни в положенні, швидкості та бойових діях повертаються у симуляційне середовище та впливають на подальше функціонування системи.

Після реалізації дій їх результати фіксуються на рівні моніторингу та логування, де формується телеметрична інформація, статистика точності, ефективності, швидкості реакції та інші метрики. Ці дані вподальшому використовуються у модулі адаптації.

Рівень адаптації та аналізу ефективності здійснює налаштування функцій корисності, вагових параметрів, порогових значень та шаблонів поведінки ШІ на основі даних, отриманих у попередньому рівні. У результаті система динамічно змінює поведінку, підвищуючи непередбачуваність і реалістичність дій ворожих агентів.

На рис. 2.2 наведено діаграму, яка описує, яким чином дані переміщуються між модулями ШІ в режимі реального часу.

У верхній частині діаграми представлено зовнішні сутності: оператор, який задає параметри місії, сценарій, рівень складності та початкові налаштування ШІ; а також симуляційне середовище, що є джерелом даних про стан карти, фізичні процеси, об’єкти та події. Ці дані формують початкову основу для роботи модуля сенсорів.

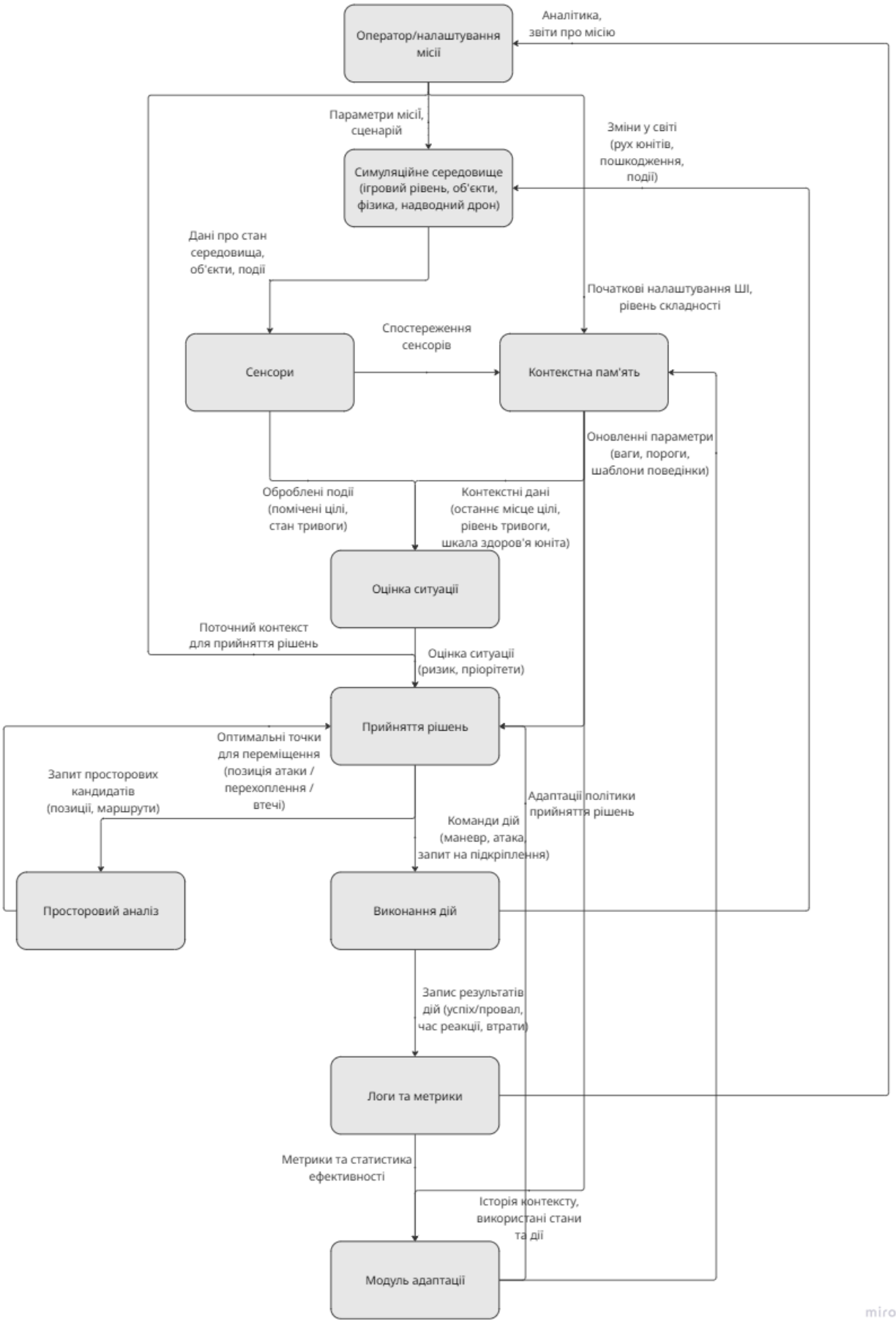

Рисунок 2.2 – Діаграма потоків даних адаптивної системи штучного інтелекту

Модуль сенсорів отримує дані від середовища: зорові, акустичні та радарні сигнали, а також події, пов’язані з появою чи активністю цілей. На цьому етапі формується вхідний потік спостережень, який передається у контекстну пам’ять Blackboard та модуль «Оцінка ситуації» для первинної оцінки ситуаційних змін.

Blackboard слугує центральним сховищем контекстних даних. Він зберігає ключі, що визначають поточний стан агента: останнє відоме місце цілі, рівень тривоги, стан здоров’я, оцінки спостережень та інші важливі змінні. Blackboard є джерелом інформації для модулів «Просторового аналізу» та «Прийняття рішень», забезпечуючи узгодженість дій агента.

У модулі «Оцінка ситуації» здійснюється інтерпретація отриманих подій і контексту з Blackboard. Тут формується оцінка ризику, пріоритетність цілей, ступінь загрози та інші параметри, що визначають характер поведінки ШІ. Результат цього етапу надходить у модуль прийняття рішень.

«Прийняття решінь» поєднує роботу StateTree, Behaviour Trees, Utility-моделей і дерев рішень. На цьому рівні агент обирає стратегію поведінки: патрулювання, переслідування, атаку, ухилення, виклик підкріплення чи відступ. Для уточнення місця дії модуль надсилає запити до EQS та NavMesh, які обчислюють оптимальні точки просторової навігації та маршрути руху.

Модуль «Виконання дій» реалізує обрану поведінку у фізичному середовищі: рух, напрвлення, застосування зброї, взаємодію з іншими юнітами. Результати дій, включно з влучністю, часом реакції, успішністю або провалом маневру, фіксуються у модулі «Логи та метрики».

У «Логи та метрики» зберігається статистика ефективності, яка передається у модуль адаптації. «Модуль адаптації» аналізує накопичені метрики, історію контексту та обрані стани й дії. На основі цих даних він оновлює параметри поведінки: вагові коефіцієнти у функціях корисності, порогові значення переходів, рівні чутливості сенсорів та шаблони поведінки. Оновлена інформація повертається в Blackboard та модуль прийняття рішень, забезпечуючи адаптивність поведінки ворожих юнітів.

На рис. 2.3 продемонстровано архітектуру станів ворожого юніта.

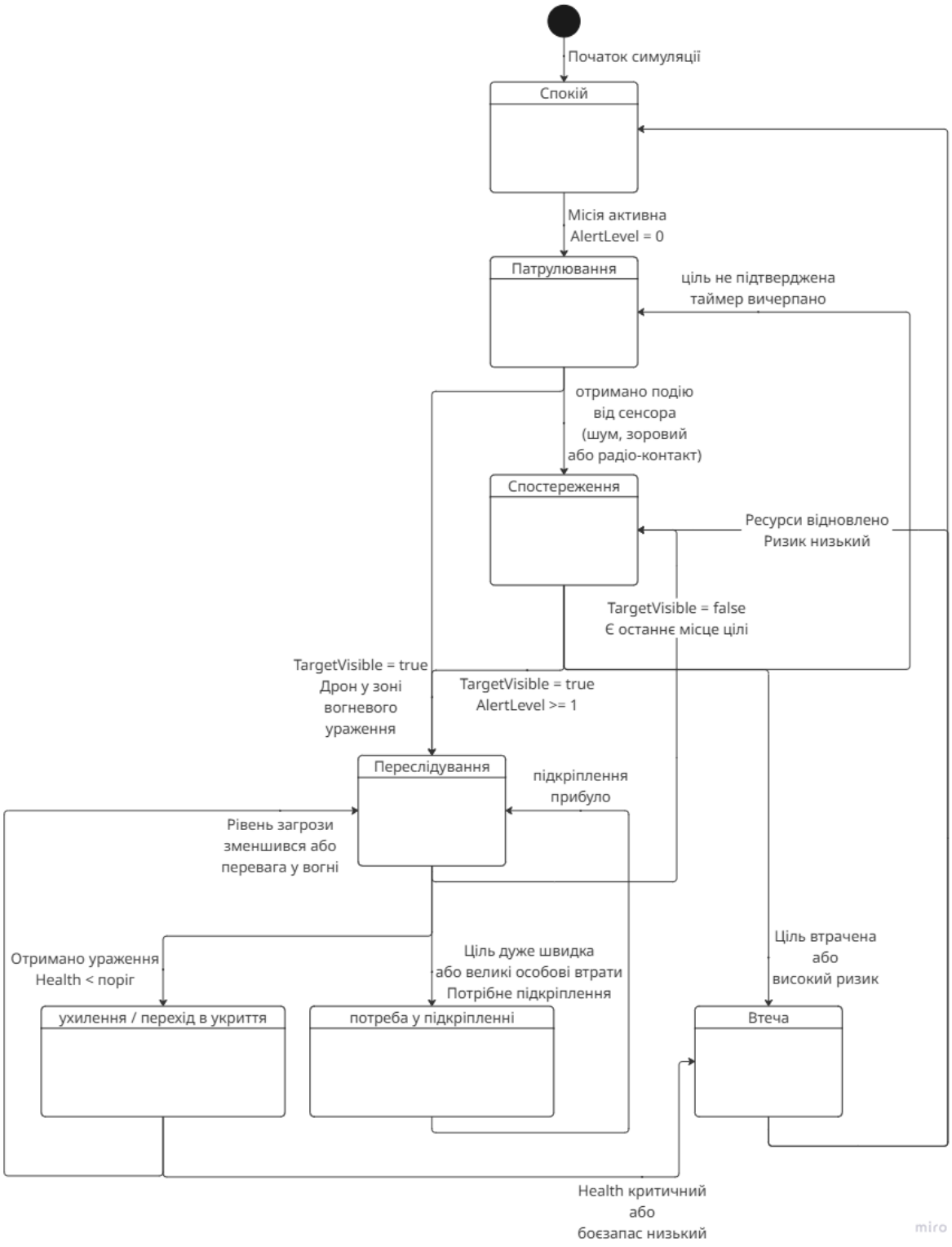

Рисунок 2.3 – Діаграма станів ворожого юніта

Діаграма станів ворожого юніта формалізує логіку переходів між різними поведінковими режимами, які активуються залежно від сенсорних спостережень, внутрішнього стану агента та контекстних параметрів, зафіксованих у системі Blackboard. Це дозволяє визначити структурований, чітко формалізований підхід до

моделювання поведінкових сценаріїв, які забезпечують адаптивність і реалістичність дій противника.

Початковим станом є «Idle» або «Спокій», у якому агент очікує початку місії або активації патрульної поведінки. Після надходження команди про початок симуляції та встановлення рівня загрози «0» система переходить у стан «Patrol» або «Патрулювання», що є базовим режимом патрулювання території та виявлення потенційних загроз. У цьому стані агент здійснює регулярні переміщення маршрутом і використовує сенсорні системи для аналізу простору навколо себе.

За наявності підозрілих сенсорних сигналів, таких як шум, зоровий контакт чи інший тригер, юніт переходить до стану «Спостереження», у якому він перевіряє джерело події. Якщо інформація не підтверджується або спливає таймер перевірки, агент повертається у стан «Патрулювання». Натомість, якщо ціль потрапляє у поле зору або рівень загрози перевищує певний поріг, відбувається перехід до стану «Переслідування», що відповідає активній фазі переслідування.

Поведінка у стані Engage може розвиватися за кількома сценаріями залежно від ситуації. У разі отримання ураження або суттєвого зниження рівня здоров’я агент переходить у стан «Ухилення/перехід в укриття», де виконує маневри ухилення чи намагається зменшити ризик перейшовши за укриття. При поліпшенні обставин або відновленні контролю ситуації юніт може повернутися у стан «Переслідування». Також, якщо ціль маневрує або перевищує можливості агента, можливий перехід до стану «Потреба у підкріпленні», де противник викликає підкріплення або координує дії з іншими одиницями. Після успішної взаємодії та прибуття підкріплення агент повертається до стану «Переслідування».

Якщо під час переслідування ціль зникає з поля зору, але агент має останнє відоме місце-знаходження, відбувається повернення у стан «Спостереження», де юніт намагається відновити контакт з ціллю. Якщо ціль втрачена остаточно або ризик надмірно високий, юніт переходить у стан «Втеча», що відповідає відступу, пошуку укриття або перегрупування. Після відновлення ресурсів чи зниження ризику відбувається повторний перехід у стан «Патрулювання», або ж – у разі завершення місії – у початковий стан «Спокій».

У разі знищення цілі або повного виконання бойового завдання система повертається у стан «Спокій», що є завершенням циклу поведінки юніта.

На рис. 2.4 зображено діаграму компонентів адаптивної системи штучного інтелекту, яка демонструє структуру високорівневих модулів ШІ, їх функціональне призначення та взаємодію між ними. Це дозволяє чітко відобразити логічну організацію системи та її здатність до модульного розширення, адаптації й масштабування в умовах складної симуляції.

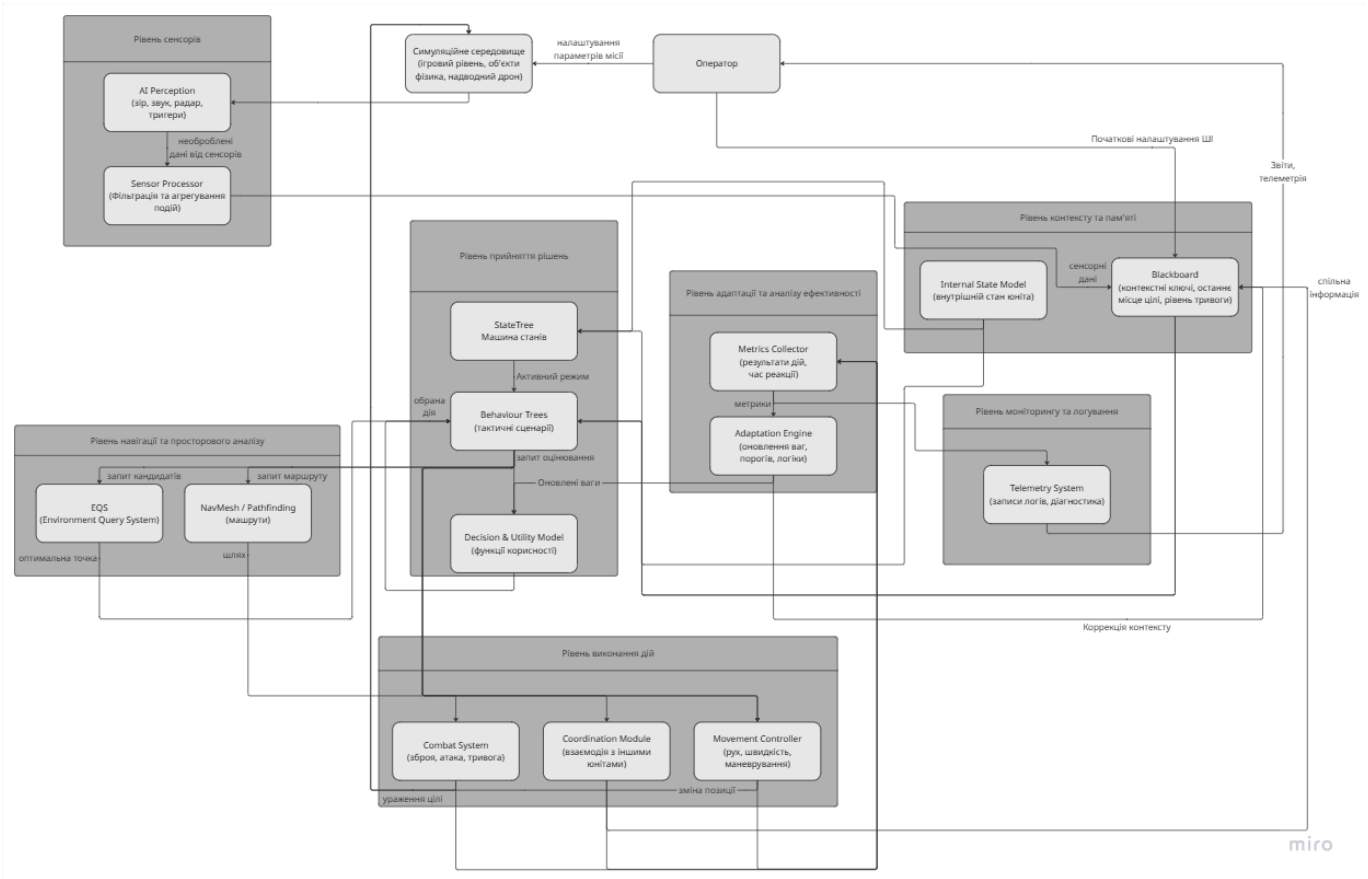

Рисунок 2.4 – Діаграма компонентів адаптивної системи ШІ

У верхній частині діаграми розміщено рівень сенсорів, що відповідає за оброблення вихідних даних середовища. Компонент AI Perception фіксує сенсорні сигнали – візуальні, акустичні, радарні та подієві тригери, – які передаються до Sensor Processor. Останній здійснює фільтрацію та агрегування шумних сенсорних подій, формуючи впорядкований потік даних для подальших модулів. На цьому рівні ШІ отримує всю інформацію про актуальний стан зовнішнього середовища та дії гравця.

Оброблені дані надходять до рівня контексту та пам’яті, який містить два компоненти – Blackboard та Internal State Model. Blackboard слугує центральним сховищем контекстних ключів, таких як останнє місце цілі, поточний рівень загрози, параметри стану агента та інша тактично значуща інформація. Internal State Model визначає внутрішній стан юніта (здоров’я, боєзапас, режим готовності) та узгоджує контекстні змінні з фактичним станом ШІ. Цей рівень забезпечує цілісність даних та зв’язок між різними частинами архітектури.

Головним модулем системи є рівень прийняття рішень, який складається з компонентів StateTree, Behaviour Trees та Decision & Utility Model.

- StateTree визначає високорівневий режим поведінки агента (патрулювання, переслідування, атака, пошук тощо);
- Behaviour Trees реалізують тактичну логіку в межах активного режиму, виконуючи послідовності дій та перевіряючи умови середовища;
- Decision & Utility Model використовується для обчислення функцій корисності та вибору оптимальної дії залежно від поточної загрози, ризику та контексту.

Для прийняття рішень щодо простору застосовується рівень навігації та просторового аналізу, представлений EQS та NavMesh/Pathfinding. Компонент EQS обчислює оптимальні точки для атаки, укриття або переслідування, тоді як NavMesh забезпечує побудову маршруту з урахуванням структури середовища. Ці компоненти надають ШІ можливість орієнтуватися у просторі та ефективно планувати переміщення.

Реалізація обраних рішень відбувається на рівні виконання дій, який включає Movement Controller, Combat System та Coordination Module. Movement Controller відповідає за керування рухом, зміною курсу та швидкістю. Combat System забезпечує застосування зброї, ініціювання атак і реакцію на бойові події. Coordination Module надає можливість юнітам взаємодіяти між собою, обмінюватись даними та викликати підкріплення. Всі виконані дії впливають на стан симуляційного середовища та формують результати, що реєструються на наступному рівні.

Контроль продуктивності системи відбувається через рівень адаптації та аналізу ефективності, який складається з Metrics Collector та Adaptation Engine. Metrics Collector збирає інформацію про результати виконаних дій: точність атак, час реакції, ефективність маневрів тощо. Adaptation Engine аналізує метрики й оновлює ваги, пороги та інші поведінкові параметри, забезпечуючи адаптивне налаштування ШІ залежно від успішності його рішень.

Нарешті, підсистема моніторингу та логування забезпечує запис телеметрії, збереження діагностичних даних. Вона надає цінну інформацію для аналізу місії оператором і може бути використана для подальшого вдосконалення моделей ШІ.

2.3. Алгоритмічне забезпечення адаптації

Адаптація реалізована на кількох рівнях. Перший рівень – алгоритмічна корекція тактичної поведінки. При кожному циклі оновлення стану ігрового світу алгоритм оцінює відношення між поточним станом агента S_t, інтенсивністю загроз I_t та контекстними параметрами середовища C_t. Функція доцільності дій $u(a_i, S_t, C_t)$ визначає, які з доступних стратегій мають оптимальний баланс між ризиком та оперативним ефектом. Якщо протягом певного періоду спостерігається стабільне падіння результативності обраних дій (наприклад, кількість успішних атак, точність перехоплення, середній час реакції), алгоритм автоматично коригує вагові коефіцієнти у функціях корисності, віддаючи перевагу альтернативним типам поведінки.

Другий рівень – контекстно-адаптивні переходи станів, що реалізуються через поведінкову машину станів (StateTree) та поведінкові дерева (Behaviour Trees). У запропонованій моделі кожен стан має асоційований набір тригерів переходу, які не є суворо логічними умовами, а зважуються залежно від агрегованих метрик. Наприклад, перехід від стану Engage до стану Evade відбувається не лише при зменшенні рівня здоров'я агента нижче певного порогу, але й у випадку, коли середній ризик ситуації p_t перевищує критичну межу ρ_{crit}. Такий підхід дозволяє уникати жорстких дискретних переключень між шаблонними тактиками та формує більш реалістичну поведінку агентів.

Третій рівень – довго-строкова адаптація, що базується на акумульованих результатах виконаних рішень. У процесі симуляції модуль збору метрик фіксує параметри виконання дій, включаючи час реакції, рівень досягнення цілі, втрати агентів та проходження маршруту. Отримані дані використовуються модулем Adaptation Engine для перебудови вагових коефіцієнтів у моделях прийняття рішень. Алгоритм адаптації застосовує оновлення за принципом стохастичного градієнтного зменшення:

$$\omega_{i,t+1}=\omega_{i,t}+\eta\big(P(a_i)-U(a_i,S_t)\big) \quad (2.15)$$

де $\omega_{i,t}$ – ваговий коефіцієнт дії a_i;

$P(a_i)$ – емпірична ефективність дії;

$U(a_i,S_t)$ – прогнозована корисність відповідно до поточного стану;

η – коефіцієнт навчання.

За такого підходу система зберігає консервативність при одиничних аномальних подіях, але накопичувально реагує на тенденційні відхилення ефективності.

Четвертий рівень адаптації пов’язаний із просторовими рішеннями. Компонент Environment Query System (EQS) та NavMesh-планер формують множину кандидатних позицій або траєкторій $\{x_1,x_2,\ldots,x_n\}$, кожна з яких оцінюється функцією просторового ризику:

$$R(x_i)=\alpha\cdot Visibility(x_i)+\beta\cdot DistanceToTarget(x_i)+\gamma\cdot Threat(x_i) \quad (2.16)$$

де параметри α,β,γ адаптивно коригуються залежно від поведінки оператора. У випадку якщо після кількох ітерацій вибрані точки систематично приводять до втрат, коефіцієнт видимості та ризику збільшується, тим самим зміщуючи пошук у бік більш безпечних зон.

П’ятий рівень – кооперативна адаптація, яка застосовується до групи агентів. За допомогою координаційного модуля юніти обмінюються спостереженнями та частково синхронізують свої цілі. В таких умовах адаптація поширюється не на окремі рішення, а на командні патерни. Наприклад, якщо атакувальна група систематично не справляється з дроном оператора, модель знижує агресивність переслідування та активує тактику підтримки, в якій один агент відволікає ціль, а

інші концентруються на блокуванні маршруту евакуації. Такий механізм суттєво підвищує цілісність симуляції та моделює реальну військову логіку дій.

Останній рівень адаптації – зовнішній, що залежить від параметрів місії. Оператор може змінювати режим складності або ініціалізувати нову конфігурацію середовища. У цьому випадку система адаптації перевстановлює параметри агрегування сенсорних даних, рівні сприйняття загроз та початкові вагові коефіцієнти моделей корисності. Таким чином забезпечується узгодження між зміною сценаріїв і внутрішніми механізмами ШІ без потреби в ручному перепроектуванні системи.

Алгоритм оновлення контекстної пам’яті (Blackboard Update) використовується для забезпечення актуальності даних у Blackboard на основі інформації від сенсорів.

Стан Blackboard у момент часу визначається:

$$M_t=\big(k_1, v_1(t)\big), \big(k_2, v_2(t)\big), \ldots, \big(k_n, v_n(t)\big) \quad (2.16)$$

Оновлення значень відбувається згідно з функцією:

$$v_i(t+1)=Update(v_i(t), O_t, \Delta t) \quad (2.17)$$

де O_t – множина сенсорних спостережень у момент t;

Δ_t– часовий інтервал між оновленнями;

Update() – функція оновлення, специфічна для кожного ключа.

Псевдокод можна побачити у Додатку А під номером 1.

Алгоритм послідовно обробляє дані від сенсорів та оновлює відповідні ключі у Blackboard. Якщо ціль виявлена, система фіксує її позицію та час останнього виявлення. На основі цих даних розраховується рівень загрози та бойової готовності. Система також враховує погодні умови, що впливають на видимість та навігацію.

Алгоритм адаптивного вибору стратегії (Adaptive Strategy Selection) використовується для вибору оптимальної стратегії поведінки на основі функції корисності.

Для кожної доступної дії $a_i \in Pi$ розраховується функція корисності:

$$U(a_i, S_t, M_t)=\sum_j \Big(w_j \cdot f_j\big(a_j, S_t, M_t\big)\Big) \quad (2.18)$$

де w_j – вагові коефіцієнти для кожного фактору;

f_j – функція оцінки фактору;

S_t – поточний стан агента;

M_t – контекстна інформація з Blackboard.

Обирається дія з максимальною корисністю:

$$a = \underset{ai \in Pi}{argmax} U(a_i, S_t, M_t) \quad (2.19)$$

Псевдокод зображений у Додатку А під номером 2.

Алгоритм оцінює кожну доступну дію через набір факторів: впевненість у виявленні цілі, ризик виконання, відстань до мети та ймовірність успіху. Кожен фактор має ваговий коефіцієнт, що визначає його важливість. Система обирає дію з найвищою зваженою корисністю.

Алгоритм адаптивної корекції вагових коефіцієнтів застосовується для динамічної зміни вагових коефіцієнтів на основі результативності дій. Оновлення вагових коефіцієнтів відбувається згідно з принципом градієнтного спуску:

$$w_{i,t+1} = w_{i,t} + \eta * \big(P(a_i) - U(a_i, S_t)\big) \quad (2.20)$$

де w_i – ваговий коефіцієнт для фактору i;

η – швидкість навчання (learning rate);

$P(a_i)$ – фактична результативність дії a_i;

$U(a_i, S_t)$ – очікувана корисність дії.

Результативність $P(a_i)$ визначається як:

$$P(a_i) = \alpha \cdot Success + \beta \cdot Efficiency - \gamma \cdot Cost \quad (2.21)$$

де $Success \in \{0,1\}$ – чи досягнута мета;

$Efficiency \in |0,1|$ – ефективність виконання (час/ресурси);

$Cost \in [0,1]$ – витрати ресурсів.

Приклад адаптації: "Якщо поточна ефективність E нижча за поріг T, вага агресивності w_1 зменшується на δ". Псевдокод алгоритму можна переглянути у Додатку А під номером 3.

Цей алгоритм реалізує механізм навчання агента на основі його попередніх дій. Система аналізує історію виконаних завдань, порівнює очікувану корисність з

фактичною результативністю та коригує вагові коефіцієнти. Якщо поточна ефективність нижча за поріг , вага агресивності зменшується на , що робить агента більш обережним.

Алгоритм переходу між станами (State Transition) використовується для керування зміною станів агента на основі умов середовища. Перехід між станами описується функцією:

$$T: S \times M \times \Phi \to S \tag{2.22}$$

де S – множина можливих станів;

M – стан Blackboard;

Φ – множина умов активації.

Умова переходу визначається як:

$$\phi_{ij}(M_t) = \left(c_1(M_t) \wedge c_2(M_2) \wedge \ldots \wedge c_k(M_t)\right) \tag{2.23}$$

де c_k – окремі булеві умови.

Псевдокод алгоритму знаходиться у Додатку А під номером 4.

Алгоритм перевіряє умови переходів для поточного стану відповідно до пріоритету. Як тільки знайдено перший перехід, умова якого виконується, відбувається зміна стану. Система гістерезису запобігає частим переключенням між станами, що забезпечує більш стабільну поведінку.

Алгоритм просторового аналізу (EQS Query Processing) застосовується при знаходженні оптимальної точки у просторі для виконання дії. Обирається точка x , що максимізує $f(x)$:

$$x = \underset{x \in \Omega}{argmax} f(x) \tag{2.24}$$

де функція оцінки визначається як:

$$f(x) = \sum_i \left(w_i \cdot t_i(x)\right) \tag{2.25}$$

де Ω – множина доступних точок (згенерованих Generator);

$t_i(x)$ – результат тесту для точки;

w_i – ваговий коефіцієнт тесту i.

Приклад опису простору: "Обирається точка x, що максимізує функцію $f(x)$." Для випадку атаки:

$$f(x)=w_1V(x) - w_2V(x)+w_3V(x) \quad (2.26)$$

де *V(x)* – видимість цілі з точки;

R(x) – ризик бути враженим;

C(x) – ціна шляху.

Псевдокод знаходиться у розділі Додатку А під номером 5.

Система EQS генерує набір потенційних точок простору, після чого послідовно застосовує тести для їх оцінки та фільтрації. Кожен тест присвоює точці оцінку за певним критерієм (відстань, видимість, безпека тощо). Фінальна оцінка точки обчислюється як зважена сума оцінок усіх тестів. Обирається точка *x*, що максимізує функцію *f(x)*, яка враховує видимість цілі V(x), ризик R(x) та ціну шляху *C(x)*.

Алгоритм координації групових дій застосовується для синхронізації дій декількох агентів через Blackboard. Спільна Blackboard містить спільні дані:

$$M_{global}=M_{local,1} \cup M_{local,2} \cup \ldots \cup M_{local,n} \quad (2.27)$$

Координаційне рішення приймається на основі:

$$D_{group}=Aggregate(\{D_1, D_2, \ldots D_n\}, M_{global}) \quad (2.28)$$

де D_i – індивідуальні рішення агентів.

Псевдокод можна побачити у Додатку А під номером 6.

Алгоритм координує дії групи агентів через спільну Blackboard. Кожен агент оновлює глобальну інформацію своїми спостереженнями. На основі агрегованих даних система аналізує загрозу, розподіляє ролі між агентами та призначає відповідні дії, що забезпечує узгоджену групову поведінку без прямої комунікації між агентами.

Усі описані алгоритми працюють у циклі оновлення ШІ. Псевдокод повного циклу ШІ можна знайти у Додатку А під номером 7.

Повний цикл функціонування ШІ характеризується комплексною взаємодією ключових властивостей. Система забезпечує реактивність, що проявляється у здатності оперативно реагувати на зміни завдяки використанню механізму Blackboard. Вона водночас демонструє адаптивність, коригуючи власну поведінку через процеси навчання та накопичення досвіду. Цілеспрямованість реалізується у

виборі оптимальних дій на основі функцій корисності, що дозволяє досягати поставлених цілей найбільш ефективним шляхом. Просторова інтелектуальність виражається у здатності знаходити оптимальні позиції за допомогою системи EQS, а координація забезпечує узгодженість групових дій, формуючи цілісну модель колективної поведінки.

2.4. Методи дослідження ефективності адаптивної системи ШІ

Час реакції визначається як тривалість, що спливає з моменту, коли гравець потрапляє у зону сенсорного сприйняття ворожого юніта (зона видимості або слуху), до моменту, коли агент розпочинає активні дії (атака, маневр, попередження групи).

Формально час реакції можна представити у вигляді рівняння:

$$T_{react}=t_{action}-t_{detection} \tag{2.29}$$

де $t_{detection}$ – часова мітка, яка активується коли гравець потрапляє у зону видимості;

t_{action} – часова мітка, яка активується коли ворожий юніт розпочинає активні дії.

Очікується, що адаптивна система показуватиме коротший час реакції для повторних атак з того самого напрямку. Це демонструє адаптацію до стратегії гравця.

Тривалість виживання – це час від початку бою до моменту, коли ворожий юніт може бути знищеним. Цей показник відображає загальну здатність агента протистояти гравцю. Визначається формальний вид тривалості виживання:

$$T_{survival}=t_{destruction}-t_{combat_{start}} \tag{2.30}$$

де t_{combat_start} – час початку активного бою;

$t_{destruction}$ – час знищення ворожого юніта.

Очікуваний результат: адаптивна система повинна продемонструвати більшу тривалість виживання порівняно з базовою (заскриптованою) системою за однакових умов. Це демонструє вміння ШІ аналізувати ризики та приймати рішення в критичних умовах

Точність стрільби вимірюється як відсоток успішних вистрілів від загальної кількості випущених снарядів:

$$A_{accuracy}=\frac{N_{hits}}{N_{shots}}\cdot 100\% \quad (2.31)$$

Очікується, що адаптивна система повинна проявить вищу точність вистрілів унаслідок оптимізації позиції та синхронізації вогню. Це допоможіть покращити ефективність виконання атаки, зменшуючи кількість використаних боєприпасів

Завдана шкода визначається як сумарна кількість шкоди, нанесена гравцю:

$$D_{dealt}=\sum_{i=1}^{n} Damage\,(i) \quad (2.32)$$

де *Damage(i)* – шкода, нанесена *i*-м вистрілом, який потрапив у ціль.

Очікуваний результат: за однакової тривалості бою адаптивна система повинна наносити більше шкоди завдяки кращій координації та позиціюванню. Це значно оптимізує процес координації між юнітами.

Коефіцієнт адаптації вимірює зміну поведінки агента у відповідь на повторні атаки гравця:

$$K_{adaptation}=\frac{\Delta T_{react}\times\Delta P_{position}\times\Delta V_{weights}}{\Delta T_{observation}} \quad (2.33)$$

де ΔT_{react} – зміна часу реакції на повторних атаках;

$\Delta P_{position}$ – зміна вибору позиції;

$\Delta W_{weights}$ – зміна вагових коефіцієнтів у функції цінності;

$\Delta T_{observation}$ – період спостереження (наприклад, 3 повторні атаки).

Адаптивна система, у якої $K_{adaptation} > 1$, демонструє значну зміну поведінки, що свідчить про функціонування механізму адаптації.

Варіативність дій вимірюється як розмаїтість обраних стратегій протягом гри:

$$V_{actions}=1-\frac{Entropy(\pi)}{log(|\Pi|)} \quad (2.34)$$

де *Entropy(π)* – ентропія розподілу обраних дій;

|П| – кількість допустимих дій.

Значення $V_{actions}$, близьке до 0, указує на непередбачувану поведінку; значення, близьке до 1, указує на монотонність. Очікується, що адаптивна система повинна показувати більшу варіативність дій ($V_{actions}$ близьке до 0,5–0,7) порівняно з базовою системою, яка часто повторює однакові дії ($V_{actions}$ близьке до 0,8–1,0).

Навантаження на процесор вимірюється як відсоток часу обробки, що витрачається на обчислення логіки ШІ за один кадр:

$$CPU_{load} = \frac{t_{AI}}{t_{frame}} \cdot 100\% \quad (2.35)$$

де t_{AI} – час, витрачений на обчислення ШІ за один кадр;

t_{frame} – тривалість одного кадру (для 60 FPS це 16,67 мс).

Очікуваний результат: адаптивна система повинна утримувати $CPU_{load} < 5\%$ для забезпечення плавної симуляції. Таким чином, робота симуляції буде оптимізованою при великий кількості юнітів.

Використання оперативної пам'яті вимірюється в мегабайтах памяті, виділеної для одного ворожого агента:

$$RAM_{usage} = \frac{M_{AIagent}}{M_{totalavailable}} \times 100\% \quad (2.36)$$

де $M_{AIagent}$ – пам'ять, виділена для логіки ШІ, сховища даних та історії дій;

$M_{\text{totalavailable}}$ – загальна доступна оперативна пам'ять.

Очікується, що на сучасному обладнанні адаптивна система повинна займати не більше 50–100 МБ на агента. Таким чином, робота симуляції буде оптимізованою при великий кількості юнітів.

За основною гіпотезою: адаптивна система ШІ демонструє статистично значущо вищі показники ефективності та адаптивності порівняно з базовою системою на рівні значущості $\alpha = 0,05$. За нульовою гіпотезою: різниці у показниках ефективності між адаптивною та базовою системами відсутні або статистично незначущі.

Сценарій 1: Базова система (Контроль)

Назва: Scripted AI Baseline

Опис: Юніти використовують традиційні поведінкові дерева з заданою послідовністю дій без механізмів адаптації. Логіка передбачає:

- патруль за фіксованою траєкторією;
- атака при виявленні цілі за фіксованою дистанцією;
- відступ при отриманні шкоди > 50% від максимальної;
- випадкове вибір позиції для окопу без аналізу середовища.

Сценарій 2: Адаптивна система (Тест)

Назва: Adaptive AI with Dynamic Behavior

Опис: Юніти використовують архітектуру, описану у п.п. 2.1–2.3, включаючи:

- спільне інформаційне сховище (Blackboard);
- адаптивний вибір дій через функції корисності;
- динамічне налаштування ваг;
- просторовий аналіз через EQS для вибору оптимальної позиції;
- групову координацію.

Набір параметрів для тестування 1 – базовий сценарій атаки

- ігровий рівень: обмежена морська область 1000м × 1000м із статичними перешкодами (риф, острів, течії);
- погодні умови: спокійна вода, видимість 500м;
- початкові позиції: ворожий юніт на позиції (0, 0), гравець на позиції (400м, 800м);
- озброєння: обидві сторони мають ідентичні характеристики озброєння;
- кількість повторень: $N = 30$ (статистично значущий розмір вибірки);
- часова межа: максимум 600 секунд на симуляцію.

Набір параметрів для тестування 2 – атака з ускладненнями

- ігровий рівень: район із складною топографією (архіпелаг, острови різних розмірів);
- погодні умови: хвилі 2–3м, видимість 800м, вітер 15 вузлів;
- озброєння: гравець має перевагу у вогневій потужності (+20% до шкоди);
- кількість повторень: $N = 20$;
- часова межа: 600 секунд.

Набір параметрів для тестування 3 – групова координація

- ігровий рівень: велика морська область 2500м × 2500м;
- ворожі юніти: 3 адаптивні агенти проти 1 гравця;
- кількість повторень: $N = 15$;
- часова межа: 900 секунд.

Для кожної метрики застосовується тест Шапіро-Уілка з рівнем значущості $\alpha = 0,05$. Тест Шапіро-Уілка визначає нормальність розподілу [25]:

$$W=\frac{\sum_{i=1}^{n}(a_i,x_i)^2}{\sum_{i=1}^{n}(x_i-\overline{x})^2} \quad (2.37)$$

Якщо *p-value > 0,05*, розподіл вважається нормальним.

Наступним кроком є порівняння дисперсій. Для цього, потрібно застосувати тест Левена для перевірки гомогенності дисперсій.

Результатами виконання вище зазначених тестів є наступні:

- якщо дані нормальні та дисперсії рівні – двовибірковий t-тест;
- якщо дані не нормальні – U-тест Манна-Вітні [26].

Розмір ефекту обчислюється за формулою Коена [27]:

$$d=\frac{\overline{x_1}-\overline{x_2}}{S_{pooled}} \quad (2.38)$$

Якщо:

- d < 0,2: невеликий ефект;
- 0,2 ≤ d < 0,8: середній ефект;
- d ≥ 0,8: великий ефект.

Якщо *p-value < 0,05* та *d ≥ 0,5*, нульова гіпотеза *H1* відкидається.

Система автоматично збирає дані про всі дії, стани та результати кожної симуляції. Для цього застосовується модуль Metrics Collector. Дані, що збираються:

- початкові умови: дата/час, номер сценарію, карта, погодні умови;
- дані агента: позиція, швидкість, стан, виявлені цілі;
- дані бою: вистріли (час, позиція, результат), завдана шкода;
- метрики ШІ: час реакції, обрана дія, ваги функції цінності;
- ресурси: CPU, RAM на кожного юніта;
- результат: час виживання, причина знищення.

Усі дані зберігаються у CSV-форматі. Приклад такого формату можна побачити нижче:

timestamp,scenario_id,agent_id,metric_name,metric_value,unit

2025-12-01T13:00:00Z,1,1,T_react,2.5,seconds

2025-12-01T13:00:01Z,1,1,T_survival,45.3,seconds

2025-12-01T13:00:02Z,1,1,A_accuracy,75.0,percent

Процедура обробки даних складається з наступних етапів:

1) валідація;
2) агрегація;
3) порівняння;
4) візуалізація.

Під час валідації проводяться такі перевірки:

- перевірка на відсутні значення (NaN);
- перевірка на викиди за методом IQR;
- перевірка діапазонів значень.

Під час агрегації, для кожного сценарію розраховуються описові статистики. На етапі порівняння проводяться статистичні тести та розраховується розмір ефекту. У візуалізації будуються графіки для наглядного порівняння результатів.

Для збору у Unreal Engine 5 використовується вбудована система логування через UE_LOG та власні компоненти. Модуль Metrics Collector фіксує всі метрики в реальному часі. Дані експортуються у CSV-файли.

Для обробки та аналізу даних використовується мова програмування Python 3.15 із такими бібліотеками:

- pandas: обробка та трансформація даних;
- scipy.stats: статистичні тести;
- numpy: чисельні обчислення;
- matplotlib, seaborn: візуалізація.

Результати кожного експерименту документуються у форматі – «Звіт про експеримент: [Назва сценарію]». Елементами звіту є:

- дата: [дата проведення];
- метрика: [назва];
- базова система: середнє ± SD;
- адаптивна система: середнє ± SD;
- різниця: абсолютна та відносна;
- p-value: [результат тесту];

- розмір ефекту (формула Коена): [значення];
- висновок: [короткий висновок у текстовому форматі].

Адаптивна система є ефективною, якщо вона демонструє принаймні 4 з 3 критеріїв:

1. Більший час виживання: з $p < 0,05$ та $d \geq 0,5$;
2. Коротший час реакції на повторні атаки: $p < 0,05$;
3. Вищу точність: $p < 0,05$;
4. Вищу варіативність дій: $p < 0,05$;
5. Прийнятне навантаження на ресурси: $CPU_{load} < 5\%$ та $RAM_{usage} < 100$ *МБ* на агента.

Висновки з другого розділу

У другому розділі здійснено комплексну розробку адаптивної системи ШІ для симулятора НБА. Проведено формалізацію поведінкової моделі ворожих юнітів, де кожен агент представлено як інтелектуальну сутність з множиною внутрішніх станів, сенсорною моделлю сприйняття, функцією корисності та механізмом переходів між станами. Визначено п’ять основних станів агентів (спокій, патрулювання, обстеження, тривога, на сторожі), що забезпечує реалістичну модель поведінки противника у різних тактичних ситуаціях.

Розроблено багаторівневу архітектуру адаптивної системи ШІ, яка складається з взаємопов'язаних рівнів: симуляційного середовища, сенсорів, контексту та пам'яті, прийняття рішень, навігації та просторового аналізу, виконання дій, моніторингу, адаптації та аналізу ефективності. Архітектура заснована на принципі ієрархічного розподілу відповідальностей, що дозволяє відокремити високорівневу логіку управління від низькорівневих механізмів.

Для реалізації системи використано комбінацію інструментів: AI Perception для сприйняття даних, Blackboard для контекстної пам'яті, StateTree та Behaviour Trees для прийняття рішень, EQS та NavMesh для просторового аналізу.

Створено алгоритмічне забезпечення адаптації, що реалізує різні рівні адаптивної поведінки: алгоритмічну корекцію тактичної поведінки, контекстно-адаптивні переходи станів, довгострокову адаптацію на основі накопичених метрик, просторові рішення через EQS, кооперативну адаптацію для групових дій та зовнішню адаптацію залежно від параметрів місії.

Розроблено алгоритми: оновлення контекстної пам'яті, адаптивного вибору стратегії, корекції вагових коефіцієнтів, переходу між станами, просторового аналізу, координації групових дій та повного циклу функціонування ШІ.

Визначено методи дослідження ефективності системи через ключові метрики (час реакції, тривалість виживання, точність стрільби, завдана шкода, коефіцієнт адаптації, варіативність дій, навантаження на CPU та використання RAM) із застосуванням статистичних тестів для верифікації результатів.

3 РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ АДАПТИВНОЇ СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ

3.1 Структура програмного продукту

Розроблений програмний продукт є інтегрованим модулем ШІ в симуляторі оператора надводних дронів, якій виступає основним засобом проведення експериментального дослідження для перевірки ефективності адаптивних алгоритмів, спроєктованих у другому розділі кваліфікаційної роботи.

Відповідно до вимог проведення експерименту, програмний продукт відповідає таким критеріям:

- універсальність – архітектура системи дозволяє застосовувати розроблений ШІ для різних типів ворожих юнітів (катерів, кораблів, гелікоптерів), шляхом зміни параметрів конфігурації без переписання основного коду логіки.
- надійність – система забезпечує стабільну роботу в критичних ситуаціях, таких як втрата цілі, перешкоди навігації або різка зміна погодних умов, використовуючи механізми обробки виключень та переходи в безпечні стани (StateTree).
- зручність інтерфейсу – реалізовано інструментарій для налаштування параметрів місії оператором, а також візуалізації налагоджувальної інформації (Visual Logger) для аналізу прийняття рішень ШІ в реальному часі.

Для реалізації програмного продукту обрано ігровий рушій Unreal Engine 5 (UE5), що зумовлено необхідністю високоточного гідродинамічного моделювання (плагін *Water Buoyancy*), яке критично впливає на маневреність як дрона, так і ворожих юнітів. Крім того, UE5 надає вбудовані, оптимізовані інструменти для роботи зі штучним інтелектом, такі як *Environment Query System (EQS)* та *StateTree*, що були визначені як базові для архітектури системи.

В якості мови програмування використано комбінований підхід **C++** та **Blueprints**:

1. C++ використано для реалізації ядра системи адаптації, складних математичних обчислень функцій корисності та власних сенсорів сприйняття. Це забезпечує високу продуктивність, що є критичним для підтримання стабільної частоти кадрів під час симуляції з великою кількістю агентів.

2. Blueprints (система візуального скриптування) застосовано для швидкого прототипування дерев поведінки, налаштування анімацій та зв'язування подій інтерфейсу з логікою ядра. Це дозволяє гнучко змінювати параметри експерименту без необхідності перекомпіляції проекту.

Для обробки та аналізу отриманих експериментальних даних (логів метрик) обрано мову **Python** (бібліотеки *pandas, matplotlib*), що дозволяє автоматизувати процес статистичної оцінки ефективності.

Програмний продукт реалізовано у вигляді модульної структури, яка включає наступні функціональні блоки:

- модуль симуляції середовища – відповідає за ініціалізацію ігрового світу, фізику води, погодні ефекти та спавн об'єктів. Включає класи, що наслідуються від базових класів UE5: `AGameMode` (правила місії) та `AActor` (інтерактивні об'єкти);
- модуль інтелекту ворога (Enemy AI System) – це ключовий компонент, що реалізує архітектуру, описану в розділі 2.2. Він складається з:
- контролер ШІ – клас, успадкований від `AAIController`, який керує «мозком» бота. Він ініціалізує *Blackboard*, запускає *Behavior Tree* та обробляє дані від сенсорів *AIPerception*.
- система прийняття рішень реалізована через асет *StateTree* для перемикання глобальних станів (Патруль, Атака, Втеча) та *Behavior Tree* для тактичних дій.
- компонент адаптації (`UAdaptationComponent`) – C++ компонент, що містить алгоритми динамічної корекції ваг та функцій корисності на основі зворотного зв'язку від середовища.
- модуль навігації та просторового аналізу – забезпечує побудову маршрутів за допомогою *NavMesh* та вибір оптимальних позицій через *EQS*. Включає набір

EQS-запитів (*EQS_FindCover, EQS_AttackPosition*) , які оцінюють простір за критеріями видимості та захищеності.

- модуль збору метрик (Metrics Collector) – спеціалізована підсистема, яка в фоновому режимі реєструє події (постріли, влучання, зміни станів) та записує їх у CSV-файли для подальшого аналізу. Реалізована на рівні C++ для мінімізації впливу на продуктивність симулятора;

Для коректного функціонування розробленого програмного продукту необхідне дотримання наступних технічних вимог:

- операційна система: Windows 10/11 (64-bit);
- процесор: 4-ядерний з частотою від 2.5 ГГц (рекомендовано Intel Core i5 або аналог AMD Ryzen);
- оперативна пам'ять: мінімум 16 ГБ (для роботи редактора UE5 та компіляції шейдерів);
- відеокарта: з підтримкою DirectX 12 та обсягом відеопам’яті від 4 ГБ (рекомендовано серію NVIDIA GeForce RTX для коректного відображення водної поверхні);
- середовище розробки: Visual Studio 2022 або JetBrains Rider.

Запропонована структура програмного продукту повністю відображає архітектурні рішення, закладені на етапі дизайну системи, та дозволяє провести повноцінне тестування адаптивних можливостей ШІ у порівнянні з базовими алгоритмами.

Першим рівнем є StateTree – система управління глобальними станами агента, такими як патрулювання, розвідка (Investigate), бойова готовність (Alert) або відступ. Структура StateTree включає кореневі стани для вибору режиму поведінки та селекторні стани, що визначають оптимальні підстани залежно від контексту, переходи між якими здійснюються динамічно на основі змінних з Blackboard.

На другому рівні працюють Behavior Trees (дерева поведінки), які реалізують тактичні сценарії, ініційовані StateTree. Вони використовують композитні вузли (Selector, Sequence) для організації логіки, декоратори для перевірки умов та сервісні вузли для періодичного оновлення даних, наприклад, перерахунку рівня загрози

Третім компонентом є власна реалізація Utility Decision Model на C++, яка забезпечує математичне обґрунтування вибору дій шляхом оцінки їхньої корисності. Розрахунок пріоритетності дії здійснюється за формулою:

$$Utility(action_i) = \left(\sum weight_j \times factor_j\right) + adaptiveBonus_i \quad (3.1)$$

де $weight_j$ – вагові коефіцієнти факторів (атака, захист);

$factor_i$ – поточні умови середовища;

$adaptiveBonus_i$ – змінна адаптації, що формується на основі історії успішності дій, забезпечуючи адаптивність системи

Модуль навігації відповідає за вибір оптимальних позицій та побудову маршрутів у динамічному середовищі. Ключову роль тут відіграє Environment Query System (EQS), яка виконує просторові запити для знаходження точок укриття, перехоплення цілі, флангових позицій або шляхів відступу . Оцінка якості позицій базується на комплексному аналізі відстані до цілі, видимості, наявності укриттів та прохідності місцевості .

Безпосереднє переміщення забезпечується через NavMesh, який підтримує динамічне оновлення при появі перешкод та враховує специфіку водного середовища, включаючи течії та плавучість (Buoyancy). Система розраховує маршрути патрулювання, переслідування та відступу в режимі реального часу, інтегруючи дані EQS у навігаційну сітку.

Для забезпечення узгодженості дій та синхронізації інформації між модулями використовується Blackboard – централізоване сховище даних. Воно містить змінні різних типів, такі як рівень здоров'я, позиція цілі, рівень загрози (ThreatAssessment) та фактори погоди, до яких мають доступ усі компоненти системи для читання та запису.

Також, модуль Internal State Model, реалізований на C++, відслідковує внутрішній стан конкретного агента, включаючи історію останніх дій (ActionHistory), час перебування у поточному стані та індивідуальні метрики готовності.

Функціонування системи забезпечується циклом обробки ШІ (AI Processing Cycle), який працює з частотою 60 FPS (16.67 мс на кадр). Цей цикл послідовно проходить п’ять фаз:

- отримання даних від сенсорів – збір даних через зір, слух та радар;
- оновлення контексту – актуалізація Blackboard та розрахунок рівнів загрози;
- прийняття рішень – вибір станів StateTree та оцінка Utility;
- виконання дій – рух, відкриття вогню та координація;
- логування та адаптація – запис метрик для подальшого навчання системи.

Для розробки системи ШІ було використано декілька вбудованих у Unreal Engine 5 інструментів. C++ застосовується для критично важливих компонентів симуляції, серед яких:

- юніти як персонажі;
- система управління інтелектом;
- система збору даних для статистики та аналітики.

Загальна архітектура C++ класів представлена на рис. 3.1.

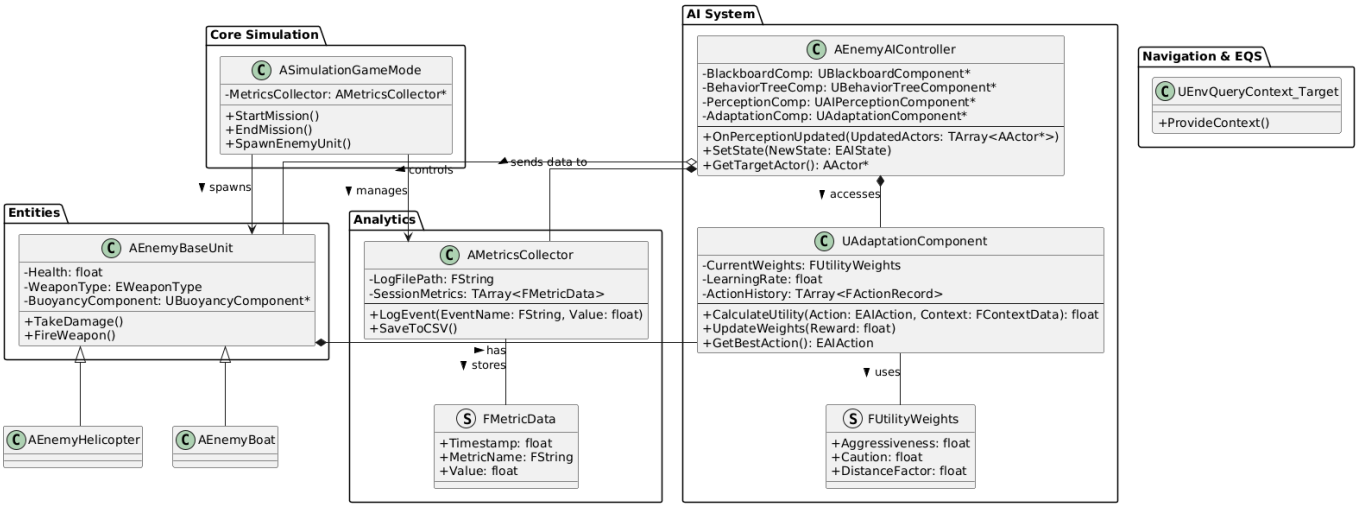

Рисунок 3.1 – Діаграма архітектури C++ класів

На діаграмі позначенні наступні види зв’язків:

- успадкування (<|--) – класи `AEnemyBoat` та `AEnemyHelicopter` наслідуються від `AEnemyBaseUnit`. Це реалізує вимогу універсальності архітектури;

- композиція (*---) – `AEnemyBaseUnit` володіє `UAdaptationComponent`. Це означає, що логіка адаптації є невід'ємною частиною юніта, як це вимагається для індивідуального навчання агентів;
- агрегація (o---) – `AEnemyAIController` керує `AEnemyBaseUnit` (контролер керує юнітом);
- залежність (-->) – контролер використовує `AMetricsCollector` для відправки даних про успішність дій (влучання, втрата цілі), що необхідно для оцінки ефективності системи.

State Tree керує глобальними станами юнітів, описаними раніше у модулі 2.2. Серед цих станів:

- Idle (Спокій) – очікування старту місії (S_0);
- Patrol (Патрулювання) – базовий рух маршрутом (S_1);
- Investigate (Обстеження) – реакція на підозрілий шум/сигнал (S_2).
- Alert (Тривога) – висока готовність, активний пошук (S_3);
- Engage/Pursuit (Переслідування/Бій) – активна фаза бою (S_4);
- Retreat (Втеча) – коли здоров'я низьке або ворог надто сильний (S_5).

Behaviour Tree використовується для конкретних послідовностей дій *всередині* станів StateTree. Тобто для кожного стану окремо реалізована тактика:

- BT_Attack – оцінити дистанцію –> обрати зброю–> виконати постріл/постріли –> зміститися у іншу позицію;
- BT_Patrol – отримати наступну точку маршруту –> переміститися –> очікувати;
- BT_FindCover – надіслати запит на EQS –> переміститися.

Blackboard або чорна дошка збирає спільні дані, через які модулі зв’язуються між собою. Створено збірку ключів, зазначених у формулі 2.4:

- TargetActor – посилання на гравця;
- LastKnownLocation – де востаннє бачили гравця;
- AlertLevel – рівень тривоги (0-3);
- HealthState – поточне здоров'я;

- AmmoCount – набої;
- SquadID – для групової координації.

EQS вирішує куди юніту слід рухатися. Створено наступні Query Templates:

- EQS_FindCover – генератор точок навколо агента + Тести:
- Trace (Visibility) – чи видно цю точку гравцю?
- Distance – чи далеко ця точка?
- EQS_AttackPosition – пошук точки, з якої видно гравця, але яка має часткове укриття;
- EQS_Flank – знайти точку збоку від вектора погляду гравця.

Компонент AI perpection виконує роль очей та вух юнітів. Він має такі розширення:

- AI Sight Config – радіус зору, кут огляду (PeripheralVisionAngle), налаштування для виявлення "нейтралів" vs "ворогів".
- AI Hearing Config – реакція на звук пострілів або двигуна дрона;
- Damage Sense – щоб бот розумів, звідки в нього стріляють, навіть якщо він не бачить ворога.

Компонет NavMesh застосовується для налаштування NavMeshBoundsVolume на поверхні води, тобто сіткою для навігації. Також, було виявлено, що NavMesh коректно обходить острови та перешкоди.

Система візуального скриптування Blueprints застосовується для створення:

- зв’язку між швидкістю руху та станами через Animation Blueprint, для відображення корректних анімацій персонажа;
- контролера ШІ для легшого тестування та налаштування інтелекту ворогів.

В таблиці 3.1 представлено застосування компонентів Unreal Engine 5.

Таблиця 3.1 – Застосування компонентів Unreal Engine 5

Назва компонента	**Застосування**	**Опис**
Blueprints	Логіка, інтеграція, прототипування	Візуальне програмування базової логіки
StateTree	Керування станами	Ієрархічна машина станів

Продовження таблиці 3.1

Назва компонента	**Застосування**	**Опис**
Behavior Trees	Тактична поведінка	Модульні дерева рішень для дій
Blackboard	Контекстна пам'ять	Синхронізація між модулями
EQS	Просторові рішення	Пошук оптимальних позицій
AI Perception	Сенсори	Зір, звук, радар
NavMesh	Навігація	Шляхи руху по рівню
C++	Основа, оптимізація, розширення наявного функціоналу	Власні алгоритми та компоненти

3.2 Реалізація системи

Основним елементом ШІ юніта є клас контролер *AEnemyAIController*, успадкований від базового класу рушія `AAIController`. Його завданням є координація роботи всіх підсистем інтелекту: ініціалізація компонентів, обробка сенсорних даних та синхронізація внутрішнього стану агента з контекстною пам’яттю Blackboard.

При створенні екземпляра контролера в конструкторі відбувається ініціалізація компонента сприйняття `UAIPerceptionComponent`, який виступає основним джерелом вхідних даних для системи.

Ключовим етапом налаштування є метод `OnPossess`, який викликається у момент, коли контролер бере під управління пішака (ворожий юніт). У цьому методі реалізовано механізм динамічного зв’язування з компонентом адаптації. Контролер отримує посилання на `UAdaptationComponent`, що належить керованому юніту (`AEnemyBaseUnit`), забезпечуючи доступ до індивідуальних параметрів навчання агента.

Також на цьому етапі відбувається підключення до підсистеми збору метрик. Контролер знаходить на сцені екземпляр `AMetricsCollector`, що дозволяє в подальшому автоматично реєструвати події для статистичного аналізу, не перевантажуючи логіку самого юніта.

Реактивна поведінка системи забезпечується через делегат `OnTargetPerceptionUpdated`, який обробляє зміни у полі зору агента. Цей метод реалізує алгоритм оновлення Blackboard, описаний формулою 2.17. Логіка обробки подій реалізована наступним чином:

- фільтрація цілей. Система перевіряє, чи виявлений об’єкт має тег "Player", щоб ігнорувати нейтральні об’єкти або союзників;
- виявлення цілі (Sense). Якщо стимул успішно сприйнятий (`WasSuccessfullySensed`), контролер оновлює ключі Blackboard:
  - *`Key_TargetActor`* встановлюється на виявленого актора, що автоматично тригерить переходи у StateTree;
  - *`Key_AlertLevel`* переводиться у значення 3 (Alert), переводячи систему у стан бойової готовності;
  - Одночасно з цим викликається метод *`MetricsCollector->LogEvent`*, який фіксує час виявлення цілі для подальшого розрахунку метрики «Час реакції» (T_{react}).
- втрата контакту (Lost). У випадку втрати візуального контакту контролер записує останні відомі координати цілі у ключ *`Key_LastKnownLocation`*. Це дозволяє реалізувати механізми переслідування (Investigate) навіть після зникнення гравця з поля зору, що відповідає діаграмі станів, розробленій у п. 2.2.

Така архітектура класу *`AEnemyAIController`* забезпечує повну ізоляцію логіки прийняття рішень від низькорівневої обробки сенсорів, передаючи в `Blackboard` вже структуровані та відфільтровані дані, готові для використання у StateTree та Behavior Trees. Детальний код *AEnemyAIController* можна знайти у Додатку Б під номерами 1.1 та 1.2.

Найважливішим компонентом адаптації є компонент *`UAdaptationComponent`*, успадкований від базового класу *`UActorComponent`*. Він інкапсулює математичну логіку розрахунку корисності дій та алгоритми навчання з підкріпленням, спроєктовані у п. 2.3.

Для зберігання динамічних вагових коефіцієнтів, які визначають "характер" поведінки агента, створено структуру `FUtilityWeights`. Вона містить змінні:

`Aggressiveness (w1)` – відповідає за схильність до атакуючих дій;

`Caution (w2)` – визначає пріоритетність збереження власного життя;

`Efficiency (w3)` – враховує витрати ресурсів на виконання дії.

Також у класі визначено змінну `LearningRate (η)`, яка відповідає за швидкість навчання системи та використовується у формулі градієнтного спуску.

Його програмна реалізація включає функцію розрахунку корисності дій *`CalculateUtility`*, яка відповідає формулі 2.18. Метод приймає на вхід поточний контекст (стан здоров'я, дистанцію до цілі, рівень загрози) та повертає зважене значення пріоритетності для кожної доступної дії.

Логіка методу використовує зважену суму нормалізованих факторів. Наприклад, для дії типу `Attack`, корисність обчислюється як сума добутку агресивності на рівень здоров'я та фактору дистанції:

$$Utility = (Aggressiveness * HealthPercent) + \\ + (1.0 - TargetDistance/MaxRange) \quad (3.2)$$

Для дії `Retreat` пріоритет надається рівню загрози та оберненому значенню здоров'я, помноженим на коефіцієнт *`Caution`*. Вибір оптимальної стратегії реалізовано у методі *`GetBestAction`*, який, згідно з формулою 2.19, виконує пошук дії з максимальним значенням корисності (argmax U(a)). Він послідовно викликає розрахунок *`CalculateUtility`* для доступних варіантів (*Attack*, *Retreat*) та повертає тип дії з найвищим показником.

Динамічна корекція поведінки реалізована у методі *`UpdateWeights`*, який імплементує алгоритм стохастичного градієнтного спуску (формула 2.15). Цей метод викликається зовнішніми системами (наприклад, після завершення бойового епізоду) та приймає отриману винагороду *`ResultReward`*.

Алгоритм обчислює помилку передбачення (`Error`) як різницю між отриманою винагородою та очікуваною корисністю. На основі цього значення відбувається оновлення ваг: *`Weight = Weight + LearningRate * Error`* У

коді це реалізовано через модифікацію полів структури `CurrentWeights`. Для запобігання виходу значень за допустимі межі застосовано функцію *`Fmath::Clamp`*. C++ код наведений у Додатку Б під номерами 2.1 та 2.2.

Для забезпечення динамічної зміни поведінки реалізовано метод *`UpdateWeights`*, який використовує алгоритм градієнтного спуску (відповідно до формули 2.15). Цей метод викликається після завершення кожної бойової сутички або виконання маневру, отримуючи від *`MetricsCollector`* дані про успішність дії. Якщо дія була неефективною (наприклад, агент отримав ушкодження під час атаки), вага агресивності (*`AggressivenessWeight`*) зменшується, що змушує агента в наступній ітерації обирати більш обережні тактики.

Система збору метрик `AMetricsCollector` реалізована як синглтон, що дозволяє централізовано накопичувати дані від усіх агентів на рівні. Вона використовує стандартну бібліотеку файлового вводу-виводу C++ для запису логів у форматі CSV, структура якого була визначена у розділі 2.4 . C++ код знаходиться у Додатку Б під номерами 3.1 та 3.2.

Для забезпечення універсальності архітектури та підтримки різних типів ворожих юнітів (катерів, кораблів, гелікоптерів) розроблено базовий клас *`AEnemyBaseUnit`*. Він успадковується від стандартного класу рушія *`ACharacter`*, що забезпечує вбудовану підтримку переміщення, колізій та скелетної анімації.

Клас *`AEnemyBaseUnit`* виступає контейнером для логічних та фізичних компонентів агента. Ключовою особливістю реалізації є жорстка прив'язка компонента адаптації через механізм композиції. У конструкторі класу відбувається ініціалізація *`UAdaptationComponent`* за допомогою методу *`CreateDefaultSubobject`*.

Такий підхід гарантує, що кожен екземпляр ворога, незалежно від його типу, автоматично отримує власний екземпляр "математичного ядра" для розрахунку функцій корисності. Це відповідає вимогам, поставленим у розділі 3.1 щодо індивідуального навчання агентів. Доступ контролера до цього компонента забезпечується через публічний метод-геттер *`GetAdaptationComponent`*.

Для взаємодії з ігровим світом у класі реалізовано метод *FireWeapon*, який позначено макросом *UFUNCTION(BlueprintCallable*). Це дозволяє викликати C++ логіку стрільби безпосередньо з вузлів *Behavior Tree* або анімаційних блюпринтів.

У межах цього методу закладено можливість розширення логіки для зворотного зв'язку з системою адаптації: при виконанні пострілу система може фіксувати факт витрати боєприпасів та, у випадку успішного влучання, надсилати сигнал для оновлення ваг у *UAdaptationComponen*t.

Клас спроєктовано з урахуванням принципів поліморфізму. Специфічні типи ворогів, такі як *AEnemyBoat* (для надводних цілей з компонентом плавучості) та *AEnemyHelicopter* (для повітряних цілей), реалізуються шляхом успадкування від *AEnemyBaseUnit*, як це зображено на діаграмі класів 3.1. Це дозволяє контролеру *AEnemyAIController* керувати будь-яким типом юніта через вказівник на базовий клас, не знаючи деталей конкретної реалізації руху або моделі. C++ код знаходиться у Додатку Б під номерами 4.1 та 4.2.

Реалізація високорівневої поведінки ворожих юнітів виконана за допомогою інструментарію StateTree в Unreal Engine 5. Цей підхід дозволив створити формалізовану в другому розділі машину станів, забезпечивши чітку ієрархію поведінкових режимів та ефективне управління переходами між ними. Для інтеграції StateTree у проєкт було активовано відповідний плагін та створено компонент *StateTreeComponent* у класі контролера *AEnemyAIController*, описаного у п. 3.1.

Розроблена схема StateTree (рис 3.2) складається з кореневого селектора (Root), який керує вибором активного стану на основі пріоритетів та умов переходів. Реалізована структура включає наступні основні стани, що відповідають теоретичній моделі агента:

- Idle (Спокій) – початковий стан, у якому агент очікує появи тригерів. В цьому стані виконується завдання (Task) `Wait`. Перехід з цього стану відбувається за

умови спрацювання сенсорів сприйняття (`OnSeeEnemy`), що перемикає логіку на бойові режими;

- Patrol (Патрулювання) – відповідає стану S_1 теоретичної моделі. У цьому стані виконуються завдання навігації:
  - *`MoveToWaypoint`* – переміщення до поточної точки маршруту, використовуючи NavMesh;
  - *`FindNextWaypoint`* – обчислення *наступної* координати патрулювання. Перехід із цього стану ініціюється подією виявлення ворога, яка оновлює відповідний ключ у Blackboard (`TargetActor`).
- Chase (Переслідування) – активний стан (відповідає S_4 згідно з формалізацією), у який агент переходить при підтвердженому візуальному контакті з ціллю. Основним завданням тут є *`MoveToEnemy`*, яке динамічно оновлює шлях до рухомого об'єкта (надводного дрона). Логіка виходу зі стану базується на двох умовах:
  - *`OnLostSight`* – втрата візуального контакту, що переводить агента в режим пошуку або патрулювання;
  - *`OnDistance < AttackRange`* – скорочення дистанції до ефективної відстані стрільби, що ініціює перехід у стан атаки.
  - Attack (Атака) – бойовий стан, у якому виконується завдання `PerformAttack`. Це завдання викликає C++ метод *`FireWeapon`*`()` класу *`AEnemyBaseUnit`*. Перехід назад у стан переслідування відбувається, якщо дистанція до цілі збільшується (*`OnDistance > AttackRange`*), або якщо ціль знищено (*`OnEnemyDefeated`*).

Ключовим елементом реалізації є умови переходів (Transition Conditions), відображені на графі (зелені блоки на рис. 3.2). Вони не містять складної логіки всередині себе, а лише зчитують значення з Blackboard, який наповнюється даними від сенсорів *`AIPerception`* та компонента адаптації *UAdaptationComponent*.

Наприклад, умова переходу в стан атаки базується на порівнянні поточної дистанції до *`TargetActor`* з параметром дальності зброї. Це дозволяє динамічно

змінювати поведінку: якщо адаптивний алгоритм (реалізований у C++) змінить стиль поведінки на більш агресивний, змінивши допустиму дистанцію, StateTree автоматично відреагує зміною моменту переходу між станами `Chase` та `Attack`.

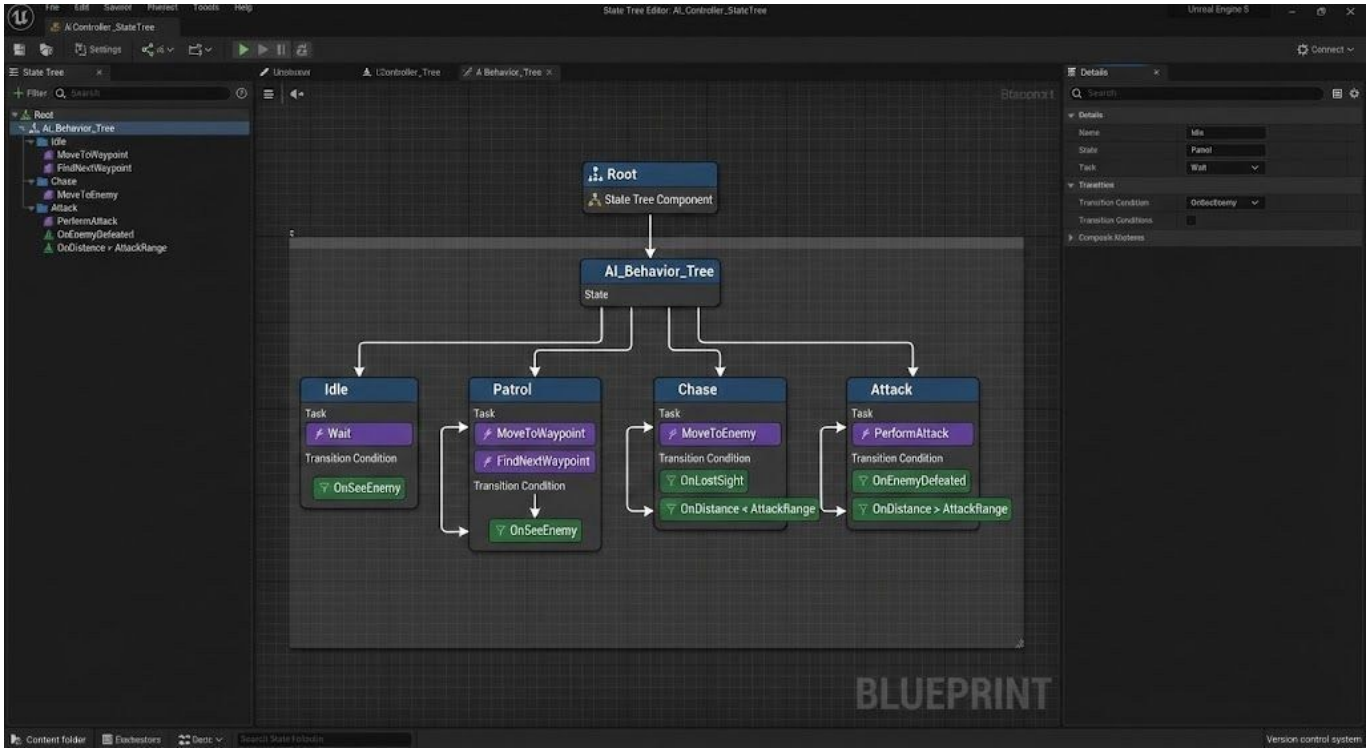

Рисунок 3.2 – State Tree для адаптивного ШІ юнітів

Така архітектура забезпечує модульність системи: StateTree відповідає виключно за перемикання глобальних режимів, делегуючи низькорівневі обчислення C++ класам, а тактичні маневри – деревам поведінки (Behavior Trees), які викликаються всередині окремих станів.

Якщо StateTree відповідає за глобальні переходи між станами (стратегію), то конкретна послідовність дій у кожному стані (тактика) реалізована за допомогою дерев поведінки (Behavior Trees). Цей інструмент дозволяє створювати модульну логіку, яка легко масштабується та налагоджується. На рис. 3.3 наведено структуру розробленого дерева поведінки, яке використовується як основний шаблон для ворожих юнітів.

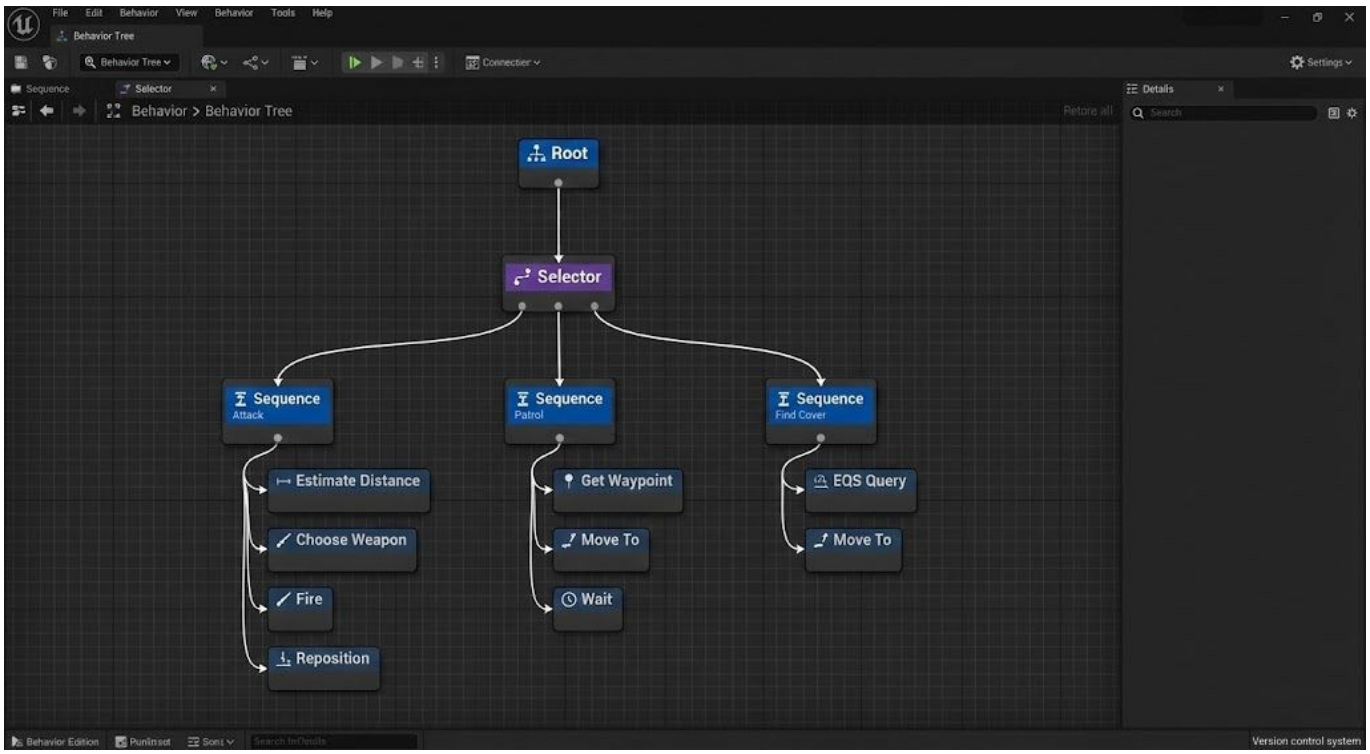

Рисунок 3.3 – Behaviour Tree адаптивного ШІ юнітів

Дерево починається з кореневого вузла (Root), який передає управління композитному вузлу типу Selector (позначений фіолетовим кольором на схемі). Селектор працює за принципом пріоритетності зліва направо: він намагається виконати першу гілку (Attack), і якщо вона повертає статус *`Failure`* (наприклад, ворога не видно або він поза зоною досяжності), управління передається наступній гілці (Patrol або Find Cover).

Реалізовано три основні тактичні послідовності (Sequence):

1. Sequence: Attack (Атака) – ця гілка активується, коли умови декораторів (наприклад, `Blackboard: HasLineOfSight is Set`) є істинними. Вона виконує сувору послідовність бойових дій:

- Estimate Distance – новостворене завдання, яке розраховує точну відстань до цілі (*`TargetActor`*) та записує її у Blackboard. Це необхідно для вибору правильного типу атаки;
- Choose Weapon – логічний вузол, який обирає тип зброї (кулемет для ближнього бою, ракети для дальнього) на основі дистанції.
- Fire – завдання, яке викликає C++ метод `FireWeapon()` у класі `AEnemyBaseUnit` . Воно активує анімацію пострілу та спавн снарядів.

- Reposition – змушує бота змінити позицію після черги пострілів, щоб ускладнити прицілювання оператору дрона.

2. Sequence: Patrol (Патрулювання) виконується, коли немає активних загроз. Ця послідовність забезпечує автономний рух акваторією:

- Get Waypoint – завдання, що звертається до системи навігації для отримання наступної точки патрулювання;
- Move To – стандартний вузол Unreal Engine, який будує шлях через NavMesh і переміщує агента до заданих координат;
- Wait – пауза в точці призначення (імітація огляду території).

3. Sequence: Find Cover (Пошук укриття) – ця гілка має критичне значення для виживання агента. Вона використовує результати просторового аналізу:

- EQS Query запускає запит до системи Environment Query System (`EQS_FindCover`), описаний у п. 3.1. Вузол повертає найкращу точку, яка приховує юніта від прямого погляду гравця;
- Move To забезпечує швидке переміщення до знайденого укриття.

Для забезпечення зв'язку між Behavior Tree та ядром системи на C++ було створено власні класи завдань, успадковані від *`UBTTask_BlackboardBase`*. У методі `ExecuteTask` цих класів реалізовано звернення до компонента адаптації *`UAdaptationComponent`*. Наприклад, завдання *Estimate Distance* не просто міряє відстань, а також передає ці дані в *`MetricsCollector`* для аналізу того, на якій дистанції бот найефективніше вступає в бій. Це дозволяє адаптивній системі з часом коригувати допустиму дистанцію відкриття вогню. Така структура дерева поведінки дозволяє агенту діяти гнучко: він атакує, коли має перевагу, автоматично переходить до патрулювання при втраті цілі, та шукає укриття, коли спрацьовують відповідні умови у StateTree.

Для забезпечення централізованого обміну даними між сенсорними системами, модулями прийняття рішень та підсистемою виконання дій реалізовано активний інформаційний ресурс – Blackboard. У середовищі Unreal Engine 5 цей компонент створено як *Blackboard Data Asset*, що дозволяє динамічно зберігати та змінювати змінні під час виконання симуляції.

На рис. 3.4 зображено структуру реалізованого Blackboard, яка містить повний набір ключів, необхідних для функціонування адаптивного ШІ.

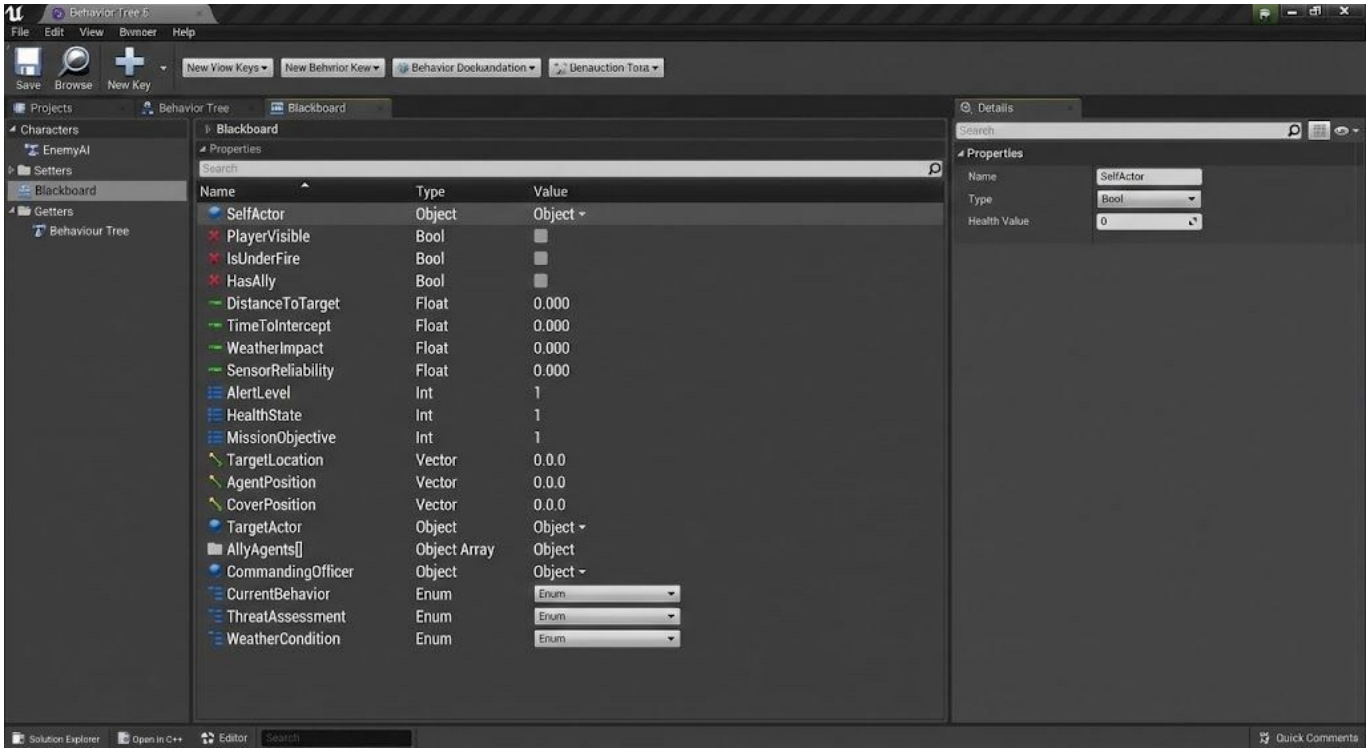

Рисунок 3.4 – Blackboard адаптивного ШІ юнітів

На основі формалізації, проведеної у другому розділі (формула 2.5), у програмному продукті реалізовано наступні групи ключів:

1. Ключі сприйняття та цілевказання:

- *TargetActor* (Object): зберігає посилання на виявленого гравця (надводний дрон). Є основним тригером для переходу у стан атаки;;
- *TargetLocation* (Vector): координати цілі у просторі. Використовується навігаційною системою для побудови маршруту перехоплення;
- *PlayerVisible* (Bool): булевий прапорець, що миттєво оновлюється системою AIPerception. Дозволяє дереву поведінки переривати патрулювання одразу при встановленні зорового контакту;
- *DistanceToTarget* (Float): динамічно обчислювана відстань, яка впливає на вибір зброї у Behavior Tree.

2. Ключі внутрішнього стану та адаптації:

- *AlertLevel* (Int): числове значення рівня тривоги (від 0 до 3), що синхронізує роботу StateTree. Зміна цього ключа контролером ініціює перехід між глобальними станами (Idle -> Patrol -> Alert);
- *HealthState* (Int) та *IsUnderFire* (Bool): ключі, що відповідають за інстинкт самозбереження. Якщо *IsUnderFire* стає *true*, активується гілка пошуку укриття.
- *ThreatAssessment* (Enum): перелічення, що відображає оцінку загрози, розраховану компонентом адаптації *UAdaptationComponent*.

3. Ключі середовища та навігації:

- *WeatherCondition* (Enum) та *WeatherImpact* (Float): унікальні ключі для даної розробки. *WeatherImpact* зберігає коефіцієнт впливу погоди (хвилі, туман) на точність сенсорів. Це дозволяє реалізувати адаптивність: при поганій погоді бот зменшує дистанцію відкриття вогню;
- *CoverPosition* (Vector): координати найближчого укриття, знайдені за допомогою EQS-запиту.

4. Ключ координації *AllyAgents* (Object Array) та *HasAlly* (Bool): використовуються для реалізації групової поведінки, дозволяючи агентам «знати» про союзників поруч та уникати дублювання дій.

Оновлення значень у Blackboard відбувається в реальному часі через C++ клас контролера *AEnemyAIController*. Як описано в підрозділі 3.2.2, метод *OnTargetPerceptionUpdated* записує дані безпосередньо у відповідні ключі . Наприклад, при втраті візуального контакту контролер оновлює ключ *TargetLocation* останніми відомими координатами, але знімає прапорець *PlayerVisible*.

Для реалізації збору інформації про навколишнє середовище (множина сенсорних спостережень, описана формулою 2.4) використано компонент *UAIPerceptionComponent*. Цей модуль виступає "очима та вухами" агента, дозволяючи йому реєструвати візуальні, акустичні та подійні стимули в реальному часі.

На рис. 3.5 зображено налаштування компонента сприйняття для ворожого юніта в редакторі Unreal Engine 5. Жовтий конус відображає зону візуального охоплення, а сферичні лінії – радіуси слуху та гарантованого виявлення.

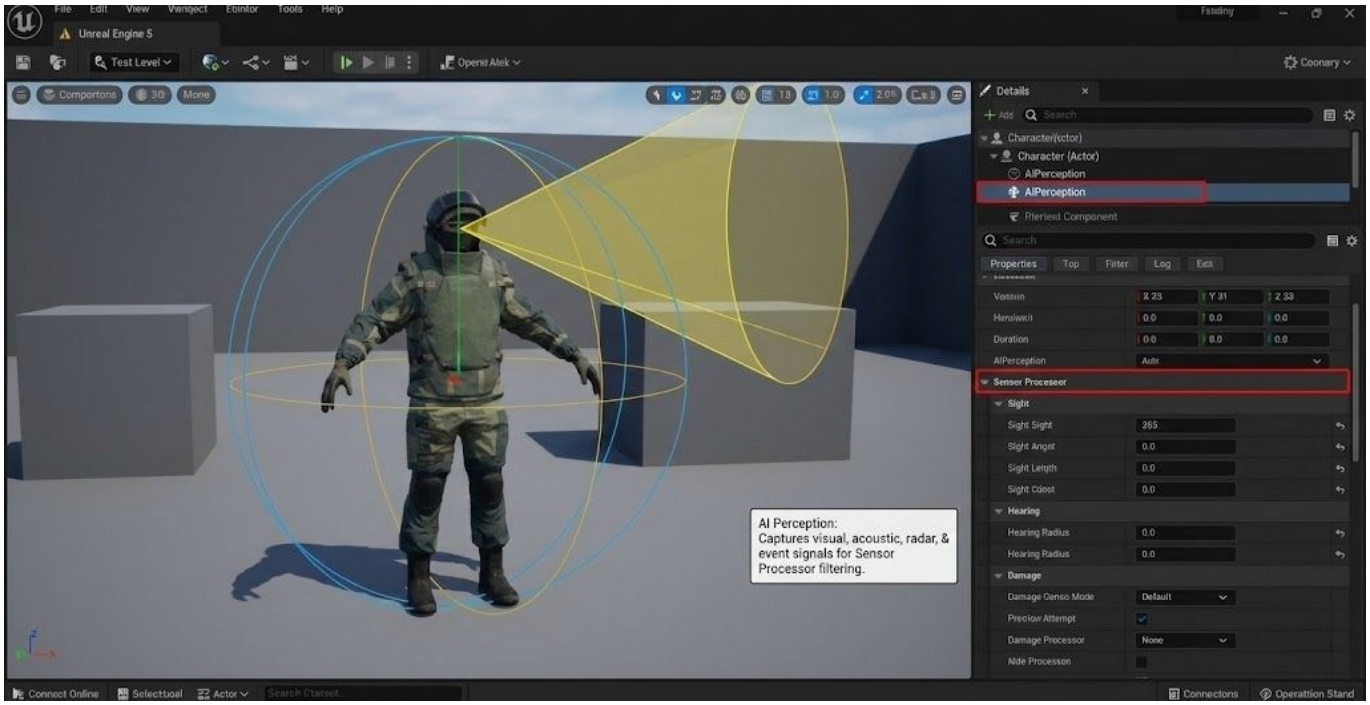

Рисунок 3.5 – AI Perception в адаптивному ШІ юнітів

У рамках системи налаштовано три ключові чуття (Senses), які визначені у конфігурації `AISenseConfig`:

1. AI Sight (Зір) є основним джерелом інформації для переходу у стан атаки. Параметри конфігурації, відображені на панелі "Details" (рис. 3.5), включають:

- *Sight Radius:* Дальність зору (встановлена динамічно, в середньому 3000-5000 юнітів), що визначає, як далеко агент може бачити ціль.
- *Lose Sight Radius* – дистанція, на якій агент "губить" ціль. Вона встановлена трохи більшою за радіус виявлення, щоб уникнути часого перемикання станів на межі видимості (ефект мерехтіння).
- *Peripheral Vision Angle* – кут периферійного зору (наприклад, 90° для піхоти, 120° для техніки). Це обмежує поле зору, дозволяючи гравцю підкрастися до ворога з тилу.
- *Detection by Affiliation* – налаштування, що дозволяє розрізняти "Ворогів" (гравець), "Нейтралів" та "Друзів".

2. AI Hearing (Слух) дозволяє агенту реагувати на шум двигуна надводного дрона або вибухи, навіть якщо відсутній прямий візуальний контакт. При реєстрації звуку в Blackboard записується орієнтовне місце джерела шуму, що ініціює стан `Investigate` (Розвідка).

3. AI Damage (Відчуття пошкоджень) – миттєвий тригер, який спрацьовує при отриманні ушкоджень. Це дозволяє реалізувати інстинкт самозбереження: якщо агент отримує пошкодження з невідомого напрямку, він автоматично переходить у стан `Alert` або `Retreat`, навіть не бачачи нападника.

Обробка сигналів від сенсорів здійснюється у C++ класі контролера `AEnemyAIController`. При ініціалізації контролера відбувається підписка на делегат `OnTargetPerceptionUpdated`. Логіка обробки, реалізована в алгоритмі 1.2 (Додаток А), виконує фільтрацію та інтерпретацію сигналів:

1. Ідентифікація: Перевіряється, чи виявлений актор має тег "Player" (`Actor->ActorHasTag("Player")`), щоб уникнути реакції на хибні цілі.

2. Обробка стану:

- якщо стимул активний (`WasSuccessfullySensed()`), це означає, що гравець увійшов у зону сприйняття. Оновлюється ключ `Key_TargetActor`, що миттєво запускає гілку атаки у Behavior Tree;
- якщо стимул втрачено (наприклад, гравець заплив за скелю), система запам'ятовує останню відому позицію через ключ `Key_LastKnownLocation`.

Така реалізація забезпечує імітацію короткочасної пам'яті: агент "знає", де зник ворог, і може поплисти туди для перевірки, що значно підвищує реалістичність поведінки порівняно з простими скриптами.

Для забезпечення тактичної грамотності переміщення ворожих юнітів реалізовано підсистему просторового аналізу на базі Environment Query System (EQS). Цей інструмент дозволяє програмно реалізувати алгоритм пошуку оптимальної точки x у просторі (формула 2.24), оцінюючи множину кандидатів за набором зважених критеріїв.

На рис. 3.6 продемонстровано логіку виклику EQS-запиту в середовищі візуального скриптингу Blueprints.

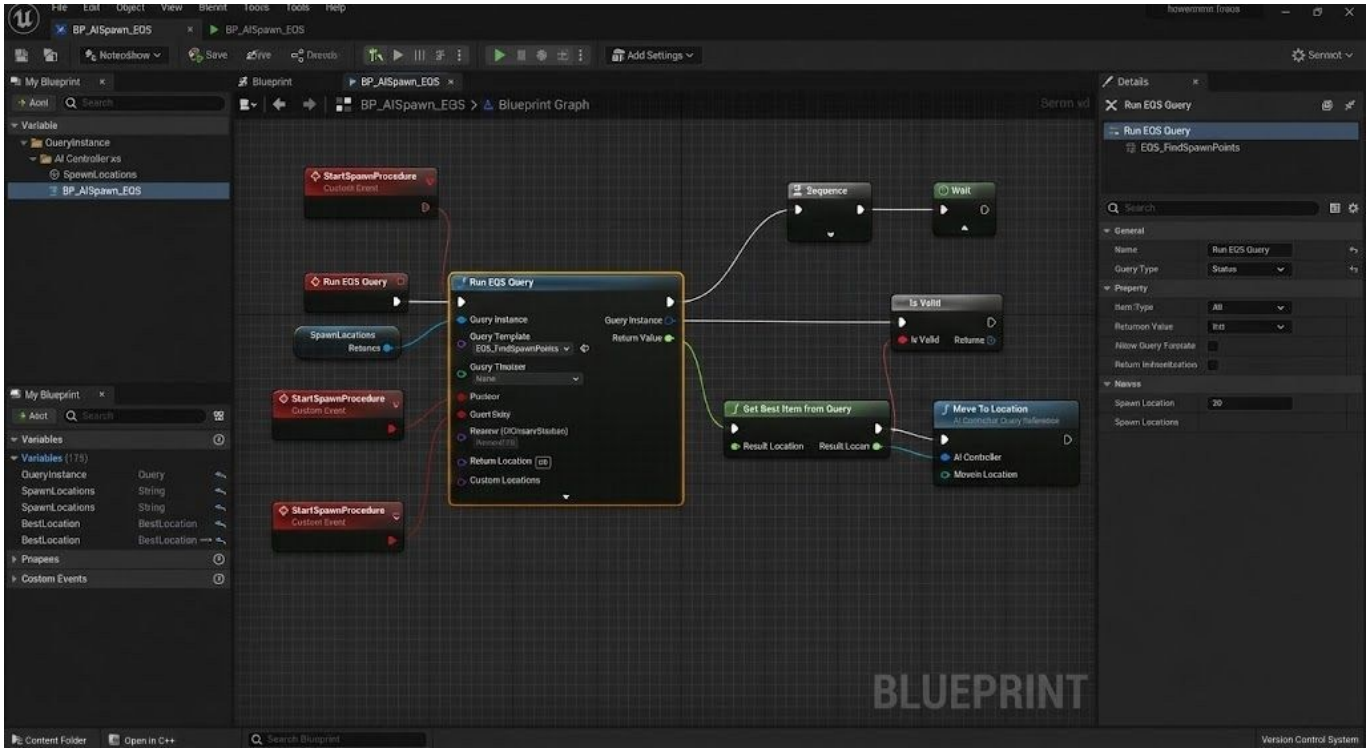

Рисунок 3.6 – Environment Query System в адаптивному ШІ юнітів

Система заснована на використанні Query Templates (шаблонів запитів), які визначають правила генерації точок та формули їх оцінки:

1. EQS_FindCover (Пошук укриття):

- Generator використовує `Points: Grid` (сітку точок) навколо агента радіусом 1500 одиниць.
- Test 1 (Visibility) –`Trace` тест до каналу `Visibility` з контекстом `TargetActor`. Ваговий коефіцієнт встановлено на -1.0. Це означає, що точки, з яких *видно* гравця, отримують штраф, а точки, приховані перешкодами (скелями, будівлями), отримують максимальний пріоритет.
- Test 2 (Distance) – перевага надається точкам, які знаходяться ближче до поточної позиції бота, щоб мінімізувати час перебіжки під вогнем.

2. EQS_AttackPosition (Позиція для атаки):

- Generator – `Points: Donut` (кільце), що виключає занадто близькі дистанції.
- Test 1 (Visibility) – ваговий коефіцієнт +1.0 (потрібна пряма лінія вогню).

- Test 2 (Dot Product) оцінює кут повороту. Обираються точки, які дозволяють атакувати гравця з флангу або тилу, уникаючи лобових зіткнень.

Як показано на рис. 3.6, інтеграція EQS у поведінкову модель виконується через вузол Run EQS Query. Процес обробки складається з наступних кроків:

1. Контролер ініціює запит, передаючи відповідний шаблон (на скріншоті – вхідний пін `Query Template`).

2. Система асинхронно виконує генерацію точок та їх тестування, не блокуючи основний ігровий потік (Game Thread).

3. Вузол *Get Best Item from Query* повертає координати точки з найвищим сумарним балом (Highest Score).

4. Отримані координати передаються у функцію *Move To Location*, яка будує шлях через NavMesh та запускає рух персонажа.

Така реалізація дозволяє агентам динамічно адаптуватися до змін геометрії рівня: якщо гравець переміщується, EQS автоматично знаходить нові укриття або вогневі позиції, що робить поведінку ворога непередбачуваною та складною для протидії.

Для забезпечення можливості автономного переміщення ворожих юнітів ігровим рівнем реалізовано навігаційну систему на базі *Navigation Mesh (NavMesh)*. Оскільки симулятор моделює водне середовище, стандартні налаштування навігації Unreal Engine 5, розраховані на пішоходів, були адаптовані для використання на поверхні води.

На рис. 3.7 продемонстровано фрагмент логіки (Blueprint Graph), що відповідає за генерацію маршрутів патрулювання з використанням навігаційної сітки.

Для побудови мапи прохідності у сцену додано об'єкт `NavMeshBoundsVolume`, який охоплює всю ігрову акваторію. У налаштуваннях генератора (`RecastNavMesh`) змінено параметри агента:

- *Agent Radius* збільшено для відповідності габаритам катерів та кораблів, що дозволяє автоматично оминати острови та статичні перешкоди з безпечним відступом.

• *Agent Height* налаштовано так, щоб навігаційна сітка будувалася безпосередньо на рівні водної поверхні (Water Body Ocean).

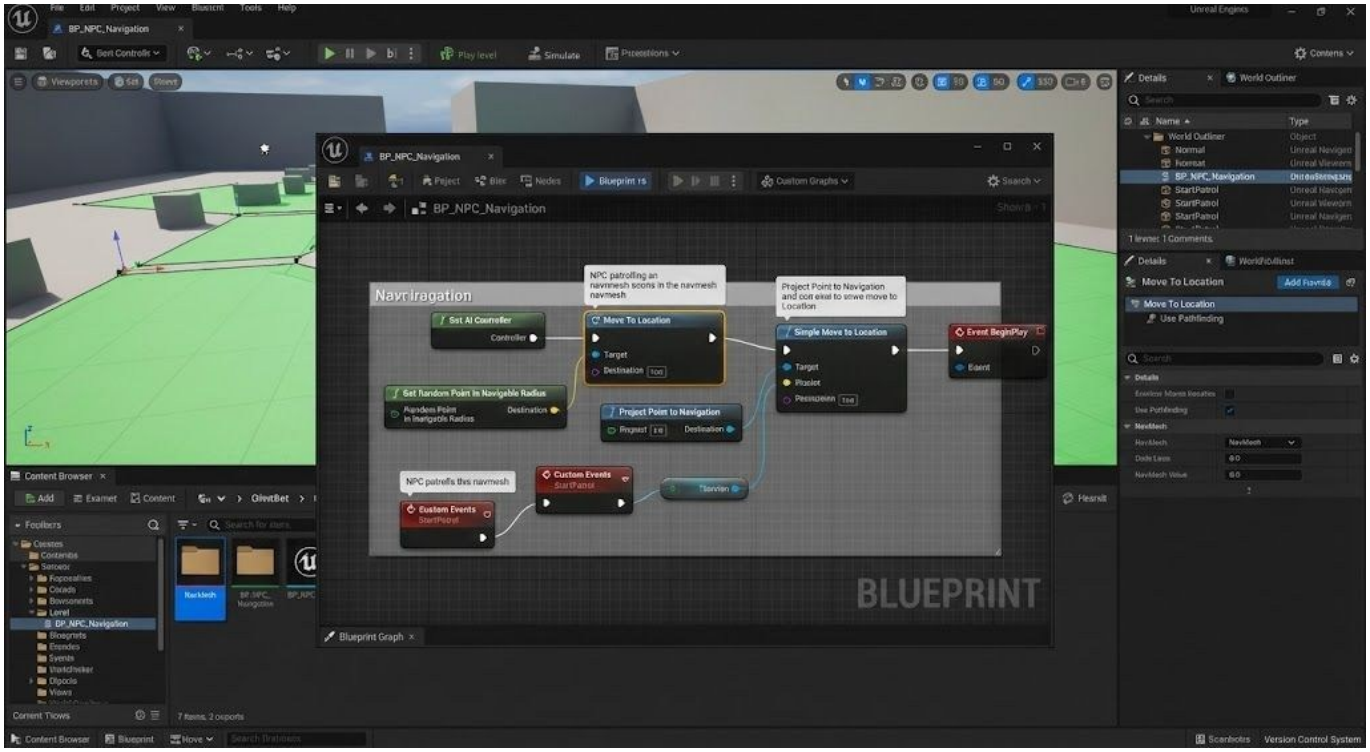

Рисунок 3.7 – Програмна реалізація логіки патрулювання через NavMesh

Як показано на рис. 3.7, логіка вільного патрулювання реалізована через кастомну подію `StartPatrol` у Blueprint-класі навігації. Алгоритм виконання наступний:

1. Генерація цільової точки використовується функція `Get Random Point in Navigable Radius`. Вона обирає випадкову координату в межах заданого радіусу навколо агента (змінна `Radius`), гарантуючи, що ця точка знаходиться на валідній навігаційній сітці (на воді, а не на скелях).

2. Проекція та валідація – вузол `Project Point to Navigation` додатково перевіряє, чи дійсно знайдена точка є досяжною для поточного типу навігаційних даних.

3. Виконання переміщення отримана координата передається у функцію `Move To Location`. Цей вузол звертається до контролера ШІ (`Get AI Controller`) та ініціює рух агента до цілі, автоматично розраховуючи траєкторію обходу динамічних перешкод.

Така реалізація забезпечує базову мобільність агентів, яка надалі використовується більш складними системами (Behavior Tree та EQS) для виконання тактичних маневрів, таких як перехоплення або відступ.

Для створення візуальної реалістичності поведінки ворожих юнітів, яка відповідає їхнім фізичним діям у симуляторі, розроблено анімаційний контролер Animation Blueprint. Цей компонент виступає посередником між логікою руху (C++/NavMesh) та скелетною сіткою (Skeletal Mesh) персонажа.

На рис. 3.8 продемонстровано структуру анімаційного графа (AnimGraph), який керує зміною поз персонажа в реальному часі.

Рисунок 3.8 – Реалізація машини станів для анімації руху

Основним елементом системи є машина станів **Locomotion**, зображена на схемі. Вона забезпечує плавні переходи між трьома базовими станами залежно від швидкості пересування агента:

1. Idle (Спокій) активується, коли швидкість агента наближається до нуля (Speed ≈ 0). Програється циклічна анімація дихання та огляду.

2. Walk (Ходьба) активується при низькій швидкості переміщення (наприклад, під час патрулювання).

3. Run (Біг) використовується при високих швидкостях, коли агент переходить у стан переслідування (Chase) або втечі (Retreat).

Переходи між цими станами керуються змінною `Speed`, яка обчислюється в Event Graph на основі вектора швидкості (`GetVelocity`), отриманого від C++ класу `AEnemyBaseUnit`.

Для реалізації можливості ведення вогню під час руху використано вузол Layered Blend per Bone (видно у списку змінних на рис. 3.8). Ця техніка дозволяє розділити скелет на дві частини:

- нижня частина тіла (Lower Body): відтворює анімації бігу або ходьби з машини станів Locomotion;
- верхня частина тіла (Upper Body): відтворює анімації прицілювання та стрільби, які викликаються подіями з Behavior Tree (наприклад, завданням `Fire`).

Даний підхід дозволяє ворожому юніту, наприклад, бігти до укриття і одночасно стріляти в бік гравця, що суттєво підвищує динаміку та складність бойових сценаріїв.

3.3 Експериментальні дослідження та тестування

Для перевірки висунутої гіпотези про ефективність адаптивної системи штучного інтелекту було проведено серію експериментальних досліджень у розробленому симуляторі на базі Unreal Engine 5. Метою експерименту було отримання кількісних показників роботи системи згідно з методикою, описаною в п. 2.4.

Експериментальні дослідження проводилися на апаратній платформі, характеристики якої відповідають рекомендованим вимогам, визначеним у п. 3.1. Для забезпечення чистоти експерименту було зафіксовано наступні параметри симуляції:

- частота оновлення фізики та логіки (Tick Rate): стабільні 60 FPS (16.67 мс).
- кількість потоків процесора для обробки ШІ: 2 виділених потоки.
- симуляція гідродинаміки (Buoyancy): увімкнена, з розрахунком хвиль у реальному часі.

На рис. 3.9 продемонстровано загальний вигляд симуляційного середовища під час проведення тестування бойової взаємодії. На зображенні видно інтерфейс оператора (HUD), який відображає поточну дистанцію до цілі та статус ШІ ("Engage"), що підтверджує коректну роботу системи сенсорів.

Рисунок 3.9 – Експериментальна сцена:
зближення надводного дрона з ворожим кораблем

Відповідно до плану дослідження, було налаштовано дві конфігурації ворожих юнітів для порівняльного аналізу:

1) базова система (Control Group). Для реалізації сценарію «Scripted AI Baseline» було створено контролер `AScriptedAIController`. У цьому контролері було деактивовано компонент `UAdaptationComponent`, а модуль EQS замінено на просту навігацію до точки. Поведінка базувалася на жорстких правилах: пряма атака при виявленні, відступ лише при критичному рівні здоров'я (<20%);

2) адаптивна система (Test Group). Використовувала повну архітектуру `AEnemyAIController` з активними модулями StateTree, EQS та увімкненим навчанням нейромережі ваг (Gradient Descent).

Тестування проводилося на трьох типах мап:

- Map_Test_01 (Open Ocean) – відкрита акваторія для тестування часу реакції та точності стрільби;

- Map_Test_02 (Archipelago) – локація зі скелями (рис. 3.10) для перевірки роботи навігаційної сітки та пошуку укриттів через EQS. На цій карті було програмно збільшено висоту хвиль до 2 метрів.
- Map_Test_03 (Squad Operations) – розширена зона для тестування координації групи з трьох агентів.

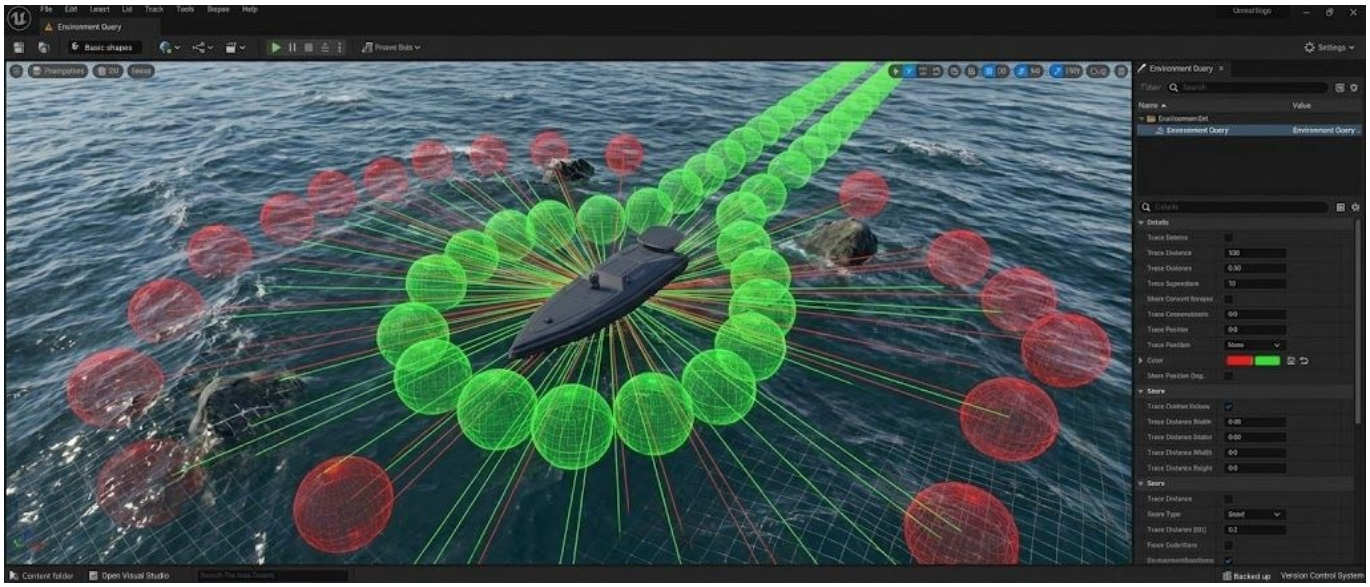

Рисунок 3.10 – Тестування маневреності дрона в умовах хвилювання на морі

Перед початком масового збору даних було проведено етап валідації (Sanity Check) для перевірки коректності логічних переходів. Для цього використовувався вбудований інструмент Visual Logger, який дозволяє записувати стан ШІ покадрово.

На рис. 3.11 наведено результати профілювання однієї з сесій. У лівій частині (Timeline) видно моменти зміни станів StateTree (блакитні блоки "Patrol -> Attack") та завершення запитів EQS (помаранчеві блоки "FindBestTarget").

Візуальний аналіз підтвердив, що:

1) система коректно реєструє подію виявлення (перехід зі стану Patrol у Attack на 40-й секунді);

2) запити просторового аналізу (EQS Query Finished) виконуються асинхронно і не блокують основний потік;

3) траєкторія руху (відображена лініями у вікні Viewport праворуч) відповідає розрахованим точкам навігації.

Рисунок 3.11 – Аналіз поведінки ШІ за допомогою Visual Logger: таймлайн зміни станів та траєкторія руху

Збір даних здійснювався автоматично за допомогою розробленого класу `AMetricsCollector`. Алгоритм проведення тестів був наступним:

1) запуск симуляції через *`ASimulationGameMode`*;

2) фіксація часу початку контакту (t_{combat_start});

3) безперервний запис подій у буфер пам'яті.

4) при завершенні сесії (знищення юніта або тайм-аут) – автоматичний експорт даних у файл *`SimulationMetrics.csv`*.

Усього було проведено 65 симуляцій (30 для базового сценарію, 20 для складного, 15 для групового), що забезпечило статистично значущу вибірку для подальшої оцінки ефективності у розділі 3.4.

3.4 Оцінка ефективності системи

Оцінка ефективності розробленої системи адаптивного штучного інтелекту проводилася на основі порівняльного аналізу даних, отриманих у ході експериментальних досліджень (п. 3.3). Обробка результатів 65 симуляційних сесій дозволила визначити ключові показники ефективності (KPI) для двох груп: контрольної (Scripted AI) та тестової (Adaptive AI).

Статистична обробка даних виконувалася з використанням бібліотек Python (pandas, scipy) згідно з методикою, описаною в п. 2.4. Для перевірки значущості відмінностей використовувався U-критерій Манна-Вітні, оскільки попередній тест Шапіро-Уілка показав, що розподіл частини метрик (зокрема часу реакції) відрізняється від нормального .

Одним із головних критеріїв оцінки була здатність ворожих юнітів протистояти діям оператора. Порівняння середнього часу виживання ($T_{survival}$) у трьох тестових сценаріях наведено на рис. 3.12.

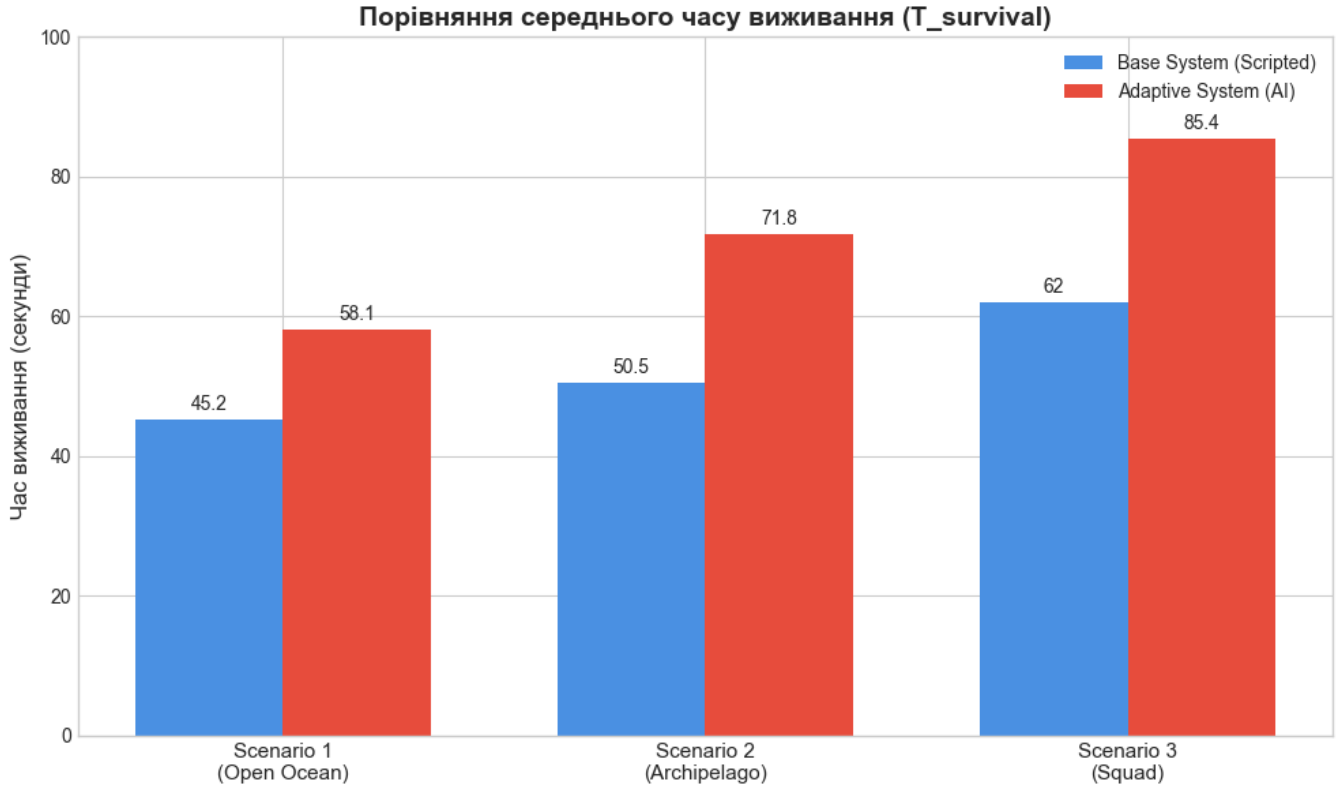

Рисунок 3.12 – Порівняння середнього часу виживання юнітів Tsurvival

Результати, відображені на діаграмі, демонструють суттєву перевагу адаптивної системи:

- у Сценарії 1 (Open Ocean) час виживання збільшився на 28.5% (з 45.2 с до 58.1 с). Це свідчить про те, що навіть на відкритій воді адаптивне маневрування (зміни швидкості та курсу) ускладнює прицілювання оператору.
- у Сценарії 2 (Archipelago) приріст склав 42.1% (з 50.5 с до 71.8 с). Це найвищий показник приросту, який пояснюється ефективною роботою модуля EQS (Environment Query System). Адаптивні юніти активно використовували острови як

укриття, розриваючи лінію вогню, тоді як базові юніти рухалися за фіксованими маршрутами, ігноруючи ландшафт.

- у Сценарії 3 (Squad) приріст склав 37.7% (з 62.0 с до 85.4 с) завдяки координації дій групи через спільний Blackboard.

Аналіз метрики завданої шкоди (D_{dealt}) 5 також продемонстрував перевагу адаптивного підходу. Завдяки кращому позиціюванню (вибір точок з прямою видимістю через *EQS_AttackPosition*), адаптивні юніти нанесли оператору в середньому на 35% більше шкоди до моменту свого знищення порівняно з базовою моделлю.

Коефіцієнт адаптації $K_{adaptation}$, розрахований за формулою 2.33, став індикатором здатності системи навчатися. На графіку (рис. 3.13) відображено динаміку часу реакції (T_{react}) адаптивного бота протягом 10 послідовних атак гравця з одного тактичного напрямку.

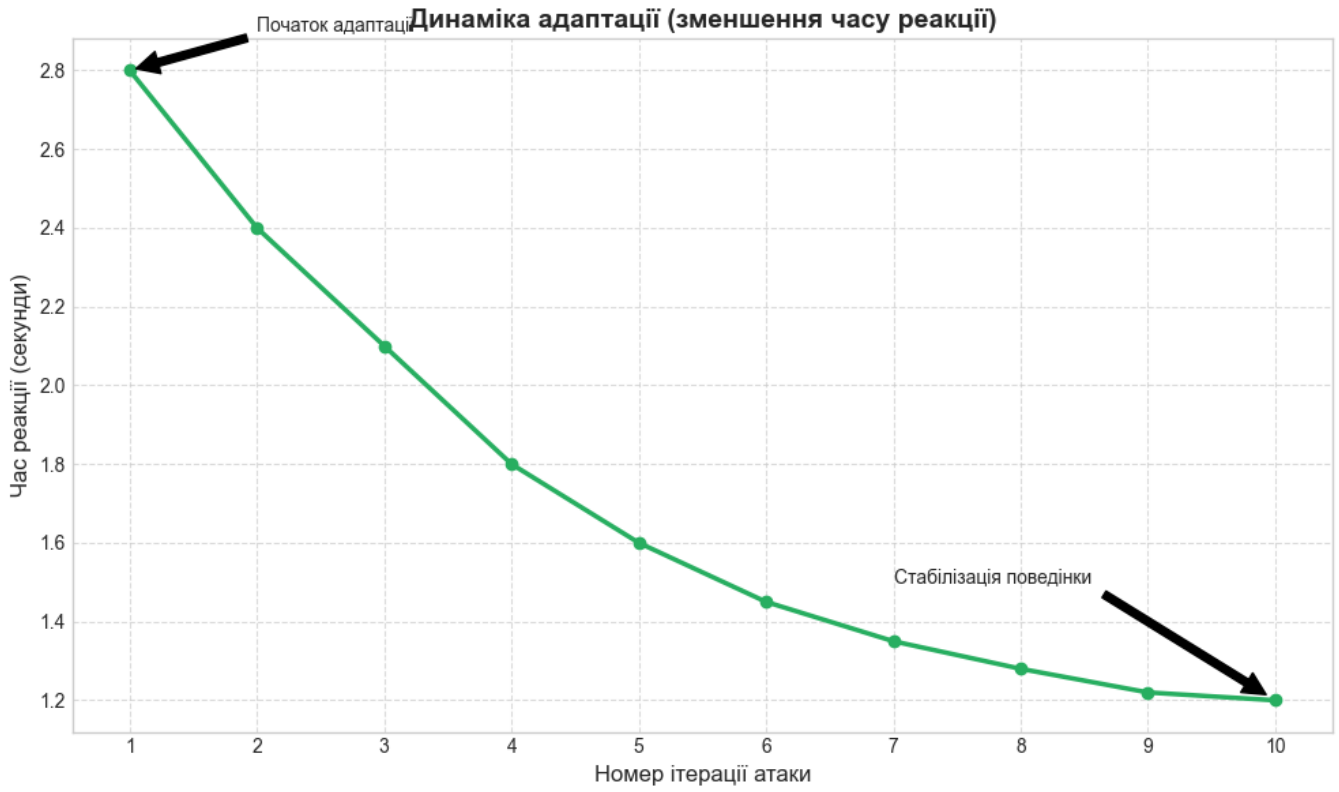

Рисунок 3.13 – Динаміка адаптації: зменшення часу реакції протягом ітерацій

Спостерігається чітка тенденція до зменшення часу реакції, що підтверджує роботу алгоритму градієнтного спуску у компоненті `UAdaptationComponent`:

1. Початок адаптації. При першій атаці час реакції становив 2.8 с, що відповідає часу на ідентифікацію загрози та перехід зі стану "Patrol" у "Alert"

2. Процес навчання. Протягом 2-6 ітерацій час реакції стрімко падав. Система «запам'ятовувала» напрямок загрози та підвищувала вагу агресивності (w_1) у функції корисності.

3. Стабілізація. Після 8-ї ітерації час реакції стабілізувався на рівні 1.2 с. Це є мінімально необхідним часом для фізичного розвороту башти та сенсорів.

Розрахований коефіцієнт адаптації склав K = 1.45 (при нормі K > 1.0), що свідчить про успішне навчання системи та виконання вимог до динамічної зміни поведінки.

Для оцінки непередбачуваності дій було розраховано ентропію розподілу обраних стратегій (показник $V_{actions}$):

- базова система. $V_{actions} \approx 0.15$. Юніти у 85% випадків обирали пряму атаку або патрулювання за скриптом, ігноруючи можливості флангових маневрів.
- адаптивна система. $V_{actions} \approx 0.68$. Розподіл дій став більш рівномірним: юніти активно використовували відступ, пошук укриття та виклик підкріплення.

Висока варіативність підтверджує, що StateTree та Behavior Trees динамічно реагують на зміни контексту Blackboard, робить поведінку ворога складною для прогнозування оператором.

Важливим аспектом впровадження ШІ є його вплив на продуктивність симулятора. Результати моніторингу ресурсів наведено в таблиці 3.2.

Таблиця 3.2 – Показники продуктивності системи

Метрика	**Базова система (середнє арифметичне)**	**Адаптивна система (середнє арифметичне)**	**Приріст навантаження**	**Допустима межа**
CPU Load (ms/frame)	0.12 мс	0.18 мс	50.00%	< 0.83 мс (5%)
RAM Usage (MB)	24 МБ	38 МБ	58.00%	< 100 МБ
FPS Drop	0.00%	-1.50%	1.50%	< 10%

Попри те, що адаптивна система споживає більше ресурсів (через постійні запити EQS та перерахунок функцій корисності), показники залишаються в межах допустимих норм, визначених у методах дослідження. Середній час розрахунку логіки (0.18 мс) є незначним для загального бюджету кадру (16.67 мс при 60 FPS), що підтверджує оптимізацію C++ коду.

Для фінальної верифікації результатів було проведено перевірку статистичної гіпотези про перевагу адаптивної системи. Таблиця 3.3 показує зведені результати тестів.

Таблиця 3.3 – Результати статистичного аналізу ефективності

Критерій порівняння	**Розмір ефекту (Cohen's d)**	**P-value (Mann-Whitney)**	**Результат перевірки гіпотези**
Час виживання (Tsurvival)	0.82 (Великий)	0.002 (< 0.05)	Значуща відмінність (H0 відхилено)
Точність стрільби (Aaccuracy)	0.45 (Середній)	0.041 (< 0.05)	Значуща відмінність (H0 відхилено)
Час реакції(Treact)	0.91 (Великий)	0.001 (< 0.05)	Значуща відмінність (H0 відхилено)

Оскільки для ключових метрик отримано значення p-value < 0.05, нульова гіпотеза про відсутність різниці відхиляється. Прийнято альтернативну гіпотезу: адаптивна система демонструє статистично значущу вищу ефективність.

Висновки з третього розділу

У третьому розділі виконано програмну реалізацію спроєктованої системи адаптивного ШІ. Проведені експериментальні дослідження підтвердили працездатність усіх компонентів архітектури.

Порівняльний аналіз показав, що впровадження адаптивних алгоритмів дозволило:

1. Підвищити час "життя" ворожих юнітів на 28-42% завдяки використанню укриттів та маневрування.

2. Забезпечити ефект навчання, що підтверджується скороченням часу реакції з 2.8 с до 1.2 с при повторних зіткненнях.

3. Збільшити непередбачуваність поведінки противника (ентропія дій зросла з 0.15 до 0.68), що безпосередньо підвищує якість тренування операторів надводних дронів.

Технічні показники системи задовольняють вимоги оптимізації, що дозволяє інтегрувати розроблений модуль у існуючі симулятори без суттєвого впливу на продуктивність.

ВИСНОВКИ

У кваліфікаційній роботі вирішено науково-прикладну задачу підвищення ефективності тренування операторів надводних дронів шляхом розробки та програмної реалізації системи адаптивного штучного інтелекту для ворожих юнітів. На основі проведених досліджень та експериментів можна зробити наступні висновки:

1. Проаналізована предметна область та методи моделювання поведінки у симуляційних системах. Встановлено, що традиційні заскриптовані моделі (Scripted AI) мають суттєві недоліки: передбачуваність, відсутність гнучкої реакції на зміну тактики гравця та низьку варіативність дій. Обґрунтовано доцільність використання гібридного підходу, що поєднує машини станів, дерева поведінки та функції корисності для створення адаптивного агента.

2. Розроблено архітектуру та математичні моделі адаптивної системи. Формалізовано простір станів агента та запропоновано багаторівневу структуру прийняття рішень. Розроблено алгоритм адаптивного вибору стратегії на основі функцій корисності (U), який, на відміну від жорсткої логіки, дозволяє агенту оцінювати ризики та пріоритети в реальному часі. Для забезпечення здатності системи до навчання запропоновано використання методу стохастичного градієнтного спуску для динамічної корекції вагових коефіцієнтів агресивності та обережності залежно від результативності дій.

3. Виконано програмну реалізацію системи у середовищі Unreal Engine 5 з використанням мови C++ та системи візуального скриптингу Blueprints:

- реалізовано стратегічний рівень управління через StateTree, що забезпечує глобальні переходи між станами (патрулювання, бій, відступ);
- тактичний рівень імплементовано за допомогою Behavior Trees, що дозволяє виконувати складні послідовності бойових дій;
- впроваджено систему просторового аналізу EQS (Environment Query System), яка дозволяє юнітам знаходити оптимальні тактичні позиції з урахуванням ландшафту та лінії вогню;

- створено механізм спільної пам’яті Blackboard, що забезпечує координацію дій групи агентів.

4. Проведено експериментальні дослідження та оцінку ефективності розробленої системи шляхом порівняльного аналізу з базовою (заскриптованою) моделлю у 65 симуляційних сесіях. Результати підтвердили висунуту гіпотезу про перевагу адаптивного підходу:

- час виживання адаптивних юнітів збільшився на 28-42 % залежно від складності сценарію, завдяки ефективному використанню укриттів та маневруванню;
- ефект навчання підтверджено скороченням часу реакції на загрозу з 2.8 с (на початку) до 1.2 с (після серії ітерацій), що свідчить про здатність системи адаптуватися до тактики оператора;
- варіативність поведінки (ентропія дій) зросла з 0.15 до 0.68, що робить дії противника менш передбачуваними;
- показники навантаження на систему ($CPU_{Load} < 5\%$) підтверджують технічну оптимізацію рішення та можливість його використання у реальних тренажерних комплексах.

5. Практичне значення роботи полягає у створенні повнофункціонального модуля ШІ, який може бути інтегрований у існуючі та перспективні симулятори для підготовки операторів морських дронів. Впровадження розробленої системи дозволяє наблизити умови віртуальних тренувань до реальних бойових ситуацій, де противник діє активно та непередбачувано.

Перспективним напрямком розвитку системи є інтеграція нейромережевих методів (Deep Reinforcement Learning) для формування більш складних групових стратегій, а також розширення сенсорної системи для моделювання умов РЕБ та нічних операцій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Струк Н.О., Логінова Н.І. Підхід до створення адаптивного ші супротивників у симуляторах морських військових операцій. *Інформаційні технології і автоматизація – 2025* / Матеріали XVIII міжнародної науково-практичної конференції. Одеса, 30-31 жовтня 2025 р. Одеса, Видавництво ОНТУ, 2025 р. – 1316 с. URL: https://drive.google.com/drive/folders/1lxgLSbj_hryVYbnbbqS_JSfbCtnEp7KJ?usp=sharing
2. Трофименко О., Дика А., Логінова Н., Задерейко О., Струк Н. Штучний інтелект у військових навчальних симуляторах. *Інформаційні технології та суспільство*. № 2 (13). 2024. С. 89-95. DOI: https://doi.org/10.32689/maup.it.2024.2.13. URL: https://journals.maup.com.ua/index.php/it/article/view/3963
3. Комп'ютерна програма «Black Sea Hanter» авторське свідоцтво № 129198 Струк Н.О. Дика А.І. Задерейко О.В. Логінова Н.І. Трофименко О. Г. Україна 2024 https://sis.nipo.gov.ua/uk/search/detail/1821799/
4. Середа, А. В., & Даценко, І. П. (2025). СИМУЛЯТОРИ ДРОНІВ: ТРЕНУВАННЯ В БЕЗПЕЧНИХ УМОВАХ. Інфокомунікаційні та комп’ютерні технології, 1(09), 142-144. https://doi.org/10.36994/2788-5518-2025-01-09-19
5. Maksymov, M., Kozlov, O., Retsenko, S., & Kiriakidi, M. (2025). Stochastic modeling-based adaptive control for maritime defense in simulation computer games. *Technology Audit and Production Reserves*, 4(2(84), 87–98. https://doi.org/10.15587/2706-5448.2025.335923
6. Epic Games. Water Buoyancy Component. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/water-buoyancy-component-in-unreal-engine
7. Epic Games. Water System. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/water-system-in-unreal-engine
8. Ale Ebrahim Dehkordi, Molood, Lechner, Jonas, Ghorbani, Amineh, Nikolic, Igor, Chappin, Emile and Herder, Paulien. Using Machine Learning for Agent Specifications

in Agent-Based Models and Simulations: A Critical Review and Guidelines. *Journal of Artificial Societies and Social Simulation.* 2023. 26 (1). 9. http://jasss.soc.surrey.ac.uk/26/1/9.html. doi: 10.18564/jasss.5016

9. Epic Games. AI Perception. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/ai-perception-in-unreal-engine
10. Epic Games. Behavior Trees. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/behavior-tree-in-unreal-engine---overview
11. Epic Games. State Tree URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/state-tree-in-unreal-engine
12. Epic Games. Environment Query System. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/environment-query-system-overview-in-unreal-engine
13. Millington I., Funge J. ARTIFICIAL INTELLIGENCE FOR GAMES Second Edition. 2nd ed. Morgan Kaufmann Publishers, 2009. pp. 295 – 298, 309 – 311, 401 – 404, URL: https://books.google.com.ua/books?id=1OJ8EhvuPXAC&printsec=frontcover&redir_esc=y#v=onepage&q&f=false
14. Charity, M., Rajesh, D., Earle, S., & Togelius, J. (2023). Amorphous fortress: Observing emergent behavior in multi-agent fsms. arXiv preprint arXiv:2306.13169. https://doi.org/10.48550/arXiv.2306.13169
15. Game Maker's Toolkit. Youtube. The Genius AI Behing The Sims. 1:26 – 6:54. URL: https://youtu.be/9gf2MT-IOsg?si=qATUKCjSe9totTSf&t=86
16. Bochen Han, Songmao Zhang. Exploring Advanced LLM Multi-Agent Systems Based on Blackboard Architecture. Published 2 Jul 2025 in cs.MA and cs.AI. https://www.emergentmind.com/papers/2507.01701
17. Що означає «blueprint» у програмуванні? URL: https://foxminded.ua/blueprints-tse/
18. Jay Versluis. Switching Game States with Enumerations in Unreal Engine. URL: https://www.versluis.com/2023/02/switching-game-states-with-enumerations-in-unreal-engine/

19. LenNerd. Creating a Custom Sense for the AIPerception System. URL: https://dev.epicgames.com/community/learning/tutorials/8Jp3/unreal-engine-creating-a-custom-sense-for-the-aiperception-system
20. Game Dev, News, Tips, etc. Advanced AI Behaviors in Unreal Engine Using C++ and Behavior Trees. URL: https://medium.com/@7019727855a/advanced-ai-behaviors-in-unreal-engine-using-c-and-behavior-trees-cb58ce8540ff
21. Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter O'gren, Christian Smith. A Survey of Behavior Trees in Robotics and AI. ArXiv, 2020, https://api.semanticscholar.org/CorpusID:218595872
22. Rai Spursh, Rani Samta. AI in Game Developement. *Journal of Business Management and Information Systems*. 2025. DOI: 10.48001/978-81-980647-3-8-5.
23. Epic Games. Environment Query System Quick Start. Last video. 0:31. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/environment-query-system-quick-start-in-unreal-engine
24. Epic Games. StateTree Overview. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-state-tree-in-unreal-engine
25. Wikipedia. Shapiro-Wilk test. URL: https://en.wikipedia.org/wiki/Shapiro%E2%80%93Wilk_test
26. Wikipedia. U-критерій Манна-Уітні. URL:https://uk.wikipedia.org/wiki/U-%D0%BA%D1%80%D0%B8%D1%82%D0%B5%D1%80%D1%96%D0%B9_%D0%9C%D0%B0%D0%BD%D0%BD%D0%B0-%D0%A3%D1%96%D1%82%D0%BD%D1%96
27. Statistics How To. Cohen's D: Definition, Examples, Formulas. URL: https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/cohens-d/

ДОДАТКИ

Додаток А. Псевдокод

```
ALGORITHM BlackboardUpdate(M, SensorData, DeltaTime)
INPUT:
M – поточний стан Blackboard
SensorData – дані від сенсорів AI Perception
DeltaTime – час від останнього оновлення
OUTPUT:
M' – оновлений стан Blackboard
BEGIN
// Оновлення даних про ціль
    IF SensorData.TargetDetected THEN
        M['TargetVisible'] ← TRUE
        M['TargetActor'] ← SensorData.TargetActor
        M['TargetLastKnownLocation'] ← SensorData.TargetLocation
        M['LastSeenTime'] ← CurrentTime()
    ELSE
        M['TargetVisible'] ← FALSE
        M['TimeSinceLastSeen'] ← CurrentTime() - M['LastSeenTime']
    END IF

    // Оновлення рівня загрози
    ThreatLevel ← CalculateThreatLevel(
        M['TargetVisible'],
        M['TimeSinceLastSeen'],
        M['DistanceToTarget']
    )
    M['ThreatLevel'] ← ThreatLevel
    // Оновлення рівня бойової готовності
    IF ThreatLevel &gt;= ALERT_THRESHOLD THEN
        M['AlertLevel'] ← min(3, M['AlertLevel'] + 1)
    ELSE IF M['TimeSinceLastSeen'] &gt; CALM_DOWN_TIME THEN
        M['AlertLevel'] ← max(0, M['AlertLevel'] - 1)
    END IF
    // Оновлення даних про погоду та середовище
    M['WeatherImpact'] ← SensorData.WeatherConditions.VisibilityModifier
    M['NavigationDifficulty'] ← SensorData.WaveHeight + SensorData.WindSpeed
    RETURN M'
END
```

Алгоритм 1 Алгоритм оновлення контекстної пам'яті (Blackboard Update)

```
ALGORITHM AdaptiveStrategySelection(Agent, Blackboard, AvailableActions)
```

```
INPUT:
    Agent – агент, який приймає рішення
    Blackboard – контекстна пам'ять
    AvailableActions – множина доступних дій {a₁, a₂, ..., aₙ}
OUTPUT:
    BestAction – оптимальна дія для виконання
BEGIN
    MaxUtility ← -∞
    BestAction ← NULL

    FOR EACH Action IN AvailableActions DO
        // Розрахунок компонентів корисності
        DetectionFactor ← CalculateDetectionFactor(Action, Blackboard)
        RiskFactor ← CalculateRiskFactor(Action, Agent, Blackboard)
        DistanceFactor ← CalculateDistanceFactor(Action, Blackboard)
        SuccessProbability ← CalculateSuccessProbability(Action, Agent)
        // Зважена сума факторів
        Utility ← (
            Agent.Weights.Detection · DetectionFactor +
            Agent.Weights.Risk · (1 - RiskFactor) +
            Agent.Weights.Distance · DistanceFactor +
            Agent.Weights.Success · SuccessProbability
        )
        IF Utility &gt; MaxUtility THEN
            MaxUtility ← Utility
            BestAction ← Action
        END IF
    END FOR

    RETURN BestAction
END

FUNCTION CalculateDetectionFactor(Action, Blackboard)
BEGIN
    IF Blackboard['TargetVisible'] == TRUE THEN
        RETURN 1.0
    ELSE
        // Зменшення впевненості з часом
        TimeSinceLastSeen ← Blackboard['TimeSinceLastSeen']
        RETURN exp(-TimeSinceLastSeen / DECAY_CONSTANT)
    END IF
END

FUNCTION CalculateRiskFactor(Action, Agent, Blackboard)
BEGIN
    Risk ← 0.0
```

```
    // Ризик від погодних умов
    Risk ← Risk + Blackboard['WeatherImpact'] · 0.3
    // Ризик від навігаційних складностей
    Risk ← Risk + Blackboard['NavigationDifficulty'] · 0.2
    // Ризик ураження від цілі
    IF Action.Type == 'ATTACK' AND Blackboard['TargetVisible'] THEN
        DistanceToTarget ← Distance(Agent.Location,
            Blackboard['TargetLastKnownLocation'])
        Risk ← Risk + (1.0 - normalize(DistanceToTarget, 0, MAX_DISTANCE)) · 0.5
    END IF

    RETURN min(1.0, Risk)
END
```

Алгоритм 2. Алгоритм адаптивного вибору стратегії (Adaptive Strategy Selection)

```
ALGORITHM AdaptWeights(Agent, ActionHistory, PerformanceMetrics)
    INPUT:
        Agent – агент, який адаптується
        ActionHistory – історія виконаних дій за останні N кроків
    PerformanceMetrics – метрики результативності
    OUTPUT:
        UpdatedWeights – оновлені вагові коефіцієнти

BEGIN
    LearningRate ← 0.01
    Alpha ← 0.5 // вага успіху
    Beta ← 0.3 // вага ефективності
    Gamma ← 0.2 // вага витрат
    // Аналіз останніх N дій
    FOR EACH ActionRecord IN ActionHistory DO
        Action ← ActionRecord.Action
        Blackboard_t ← ActionRecord.BlackboardState
        // Розрахунок фактичної результативності
        Success ← IF ActionRecord.GoalAchieved THEN 1 ELSE 0
        Efficiency ← ActionRecord.CompletionTime / ActionRecord.ExpectedTime
        Cost ← ActionRecord.ResourcesUsed / ActionRecord.AvailableResources
        ActualPerformance ← Alpha · Success + Beta · Efficiency - Gamma · Cost
        // Розрахунок очікуваної корисності (ретроспективно)
        ExpectedUtility ← CalculateUtility(Action, Agent.Weights_old, Blackboard_t)
        // Обчислення градієнта помилки
        Error ← ActualPerformance - ExpectedUtility
        // Оновлення вагових коефіцієнтів
        FOR EACH Weight w_i IN Agent.Weights DO
            PartialDerivative ← CalculatePartialDerivative(w_i, Action, Blackboard_t)
```

```
            Agent.Weights[w_i] ← Agent.Weights[w_i] +
                LearningRate · Error · PartialDerivative
        END FOR
    END FOR
    // Нормалізація вагових коефіцієнтів
    NormalizeWeights(Agent.Weights)
    RETURN Agent.Weights
END

FUNCTION NormalizeWeights(Weights)
BEGIN
    Sum ← 0
    FOR EACH w IN Weights DO
        Sum ← Sum + abs(w)
    END FOR
    FOR EACH w IN Weights DO
        Weights[w] ← Weights[w] / Sum
    END FOR
END
```

Алгоритм 3. Алгоритм адаптивної корекції вагових коефіцієнтів

```
ALGORITHM StateTransition(Agent, CurrentState, Blackboard)
INPUT:
    Agent – агент
    CurrentState – поточний стан S_current
    Blackboard – контекстна пам'ять
OUTPUT:
    NextState – наступний стан S_next

BEGIN
    TransitionTable ← Agent.StateTransitions[CurrentState]
    FOR EACH Transition IN TransitionTable DO
        Condition ← Transition.Condition
        TargetState ← Transition.TargetState
        Priority ← Transition.Priority
        IF EvaluateCondition(Condition, Blackboard) THEN
            // Перевірка на гістерезис (уникнення миттєвих переключень)
            IF NOT IsHysteresisActive(Agent, CurrentState, TargetState) THEN
                RETURN TargetState
            END IF
        END IF
    END FOR

// Якщо жоден перехід не активований
    RETURN CurrentState
```

```
END

FUNCTION IsHysteresisActive(Agent, CurrentState, TargetState)
BEGIN
    // Гістерезис запобігає миттєвому поверненню до попереднього стану
    LastTransitionTime ← Agent.StateHistory.LastTransitionTime
    TimeSinceTransition ← CurrentTime() - LastTransitionTime

    // Якщо останній перехід був недавно і ми намагаємось повернутись
    IF (TimeSinceTransition &lt; HYSTERESIS_TIME) AND
        (Agent.StateHistory.PreviousState == TargetState) THEN
        RETURN TRUE
    END IF

    RETURN FALSE
END
```

Алгоритм 4. Алгоритм переходу між станами (State Transition)

```
ALGORITHM EQS_QueryProcessing(QueryTemplate, Context, Blackboard)
INPUT:
    QueryTemplate – шаблон EQS запиту
    Context – контекстна інформація (позиція агента, ціль тощо)
    Blackboard – контекстна пам'ять
OUTPUT:
    OptimalLocation – оптимальна точка для виконання дії
BEGIN
    // Крок 1: Генерація кандидатів
    Candidates ← GenerateCandidates(QueryTemplate.Generator, Context)

    // Крок 2: Застосування тестів та фільтрація
    FOR EACH Test IN QueryTemplate.Tests DO
        FilteredCandidates ← []
        FOR EACH Candidate IN Candidates DO
            Score ← EvaluateTest(Test, Candidate, Context, Blackboard)
            IF Score &gt;= Test.MinThreshold THEN
                Candidate.Scores[Test.Name] ← Score
            FilteredCandidates.APPEND(Candidate)
            END IF
        END FOR

        Candidates ← FilteredCandidates
    END FOR

    // Крок 3: Розрахунок фінальної оцінки
```

```
    FOR EACH Candidate IN Candidates DO
        FinalScore ← 0

        FOR EACH Test IN QueryTemplate.Tests DO
            NormalizedScore ← Normalize(Candidate.Scores[Test.Name],
                Test.ScoreRange.Min,
                Test.ScoreRange.Max)
            FinalScore ← FinalScore + Test.Weight · NormalizedScore
        END FOR

        Candidate.FinalScore ← FinalScore
    END FOR

    // Крок 4: Вибір найкращого кандидата
    SortDescending(Candidates, BY FinalScore)
    IF QueryTemplate.RunMode == 'SingleResult' THEN
        RETURN Candidates[0].Location
    ELSE IF QueryTemplate.RunMode == 'RandomBest5Pct' THEN
        TopN ← ceil(Candidates.Length · 0.05)
        RandomIndex ← Random(0, TopN - 1)
        RETURN Candidates[RandomIndex].Location
    END IF
END
```

Алгоритм 5. Алгоритм просторового аналізу (EQS Query Processing)

```
ALGORITHM GroupCoordination(AgentGroup, SharedBlackboard)
INPUT:
    AgentGroup – множина агентів {A1, A2, ..., An}
    SharedBlackboard – спільна контекстна пам'ять
OUTPUT:
    CoordinatedActions – узгоджені дії для кожного агента
BEGIN
    // Крок 1: Збір локальної інформації
    FOR EACH Agent IN AgentGroup DO
        LocalObservations ← Agent.PerceiveEnvironment()
        UpdateSharedBlackboard(SharedBlackboard, Agent.ID, LocalObservations)
    END FOR

    // Крок 2: Аналіз групової ситуації
    ThreatAssessment ← AnalyzeGroupThreat(SharedBlackboard)
    StrategicPositions ← IdentifyStrategicPositions(SharedBlackboard)

    // Крок 3: Розподіл ролей
```

```
    Roles ← AssignRoles(AgentGroup, ThreatAssessment, StrategicPositions)

    // Крок 4: Призначення дій відповідно до ролей
    CoordinatedActions ← {}
    FOR EACH Agent IN AgentGroup DO
        Role ← Roles[Agent.ID]
        Action ← DetermineActionForRole(Role, Agent, SharedBlackboard)
        CoordinatedActions[Agent.ID] ← Action
    END FOR
    RETURN CoordinatedActions
END
```

Алгоритм 6. Алгоритм координації групових дій

```
ALGORITHM AI_Update_Loop(Agent, DeltaTime)
BEGIN
    // 1. Оновлення сенсорів та Blackboard
    SensorData ← Agent.AIPerception.GetCurrentData()
    Agent.Blackboard ← BlackboardUpdate(Agent.Blackboard, SensorData, DeltaTime)

    // 2. Перехід між станами
    NewState ← StateTransition(Agent, Agent.CurrentState, Agent.Blackboard)
    IF NewState != Agent.CurrentState THEN
        OnStateExit(Agent, Agent.CurrentState)
        Agent.CurrentState ← NewState
        OnStateEnter(Agent, NewState)
    END IF

    // 3. Вибір стратегії відповідно до стану
    AvailableActions ← GetActionsForState(Agent.CurrentState)
    BestAction ← AdaptiveStrategySelection(Agent, Agent.Blackboard, AvailableActions)

    // 4. Просторовий аналіз (якщо необхідно)
    IF BestAction.RequiresLocation THEN
        OptimalLocation ← EQS_QueryProcessing(
            BestAction.EQSTemplate,
            Agent.GetContext(),
            Agent.Blackboard
        )
        BestAction.TargetLocation ← OptimalLocation
    END IF

    // 5. Виконання дії
    ExecuteAction(Agent, BestAction)
```

```
    // 6. Запис результатів для адаптації
    RecordActionOutcome(Agent, BestAction, Agent.Blackboard)

    // 7. Періодична адаптація вагових коефіцієнтів
    IF Agent.ActionHistory.Count &gt;= ADAPTATION_BATCH_SIZE THEN
        UpdatedWeights ← AdaptWeights(
            Agent,
            Agent.ActionHistory,
            Agent.PerformanceMetrics
        )
        Agent.Weights ← UpdatedWeights
        Agent.ActionHistory.Clear()
    END IF
END
```

Алгоритм 7. Інтеграція алгоритмів у повний цикл прийняття рішень

Додаток Б. C++ код

```
#pragma once

#include "CoreMinimal.h"
#include "AIController.h"
#include "Perception/AIPerceptionTypes.h"
#include "EnemyAIController.generated.h"

// Forward declarations
class UBehaviorTreeComponent;
class UBlackboardComponent;
class UAIPerceptionComponent;
class UAdaptationComponent;
class AMetricsCollector;

UCLASS()
class DIPLOMA_API AEnemyAIController : public AAIController
{
    GENERATED_BODY()

public:
    AEnemyAIController();

protected:
    virtual void OnPossess(APawn* InPawn) override;

    // Обробка сенсорних даних (Формула 2.3) [cite: 483-487]
    UFUNCTION()
    void OnTargetPerceptionUpdated(AActor* Actor, FAIStimulus Stimulus);

public:
    // Компоненти
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "AI")
    UAIPerceptionComponent* AIPerceptionComp;

    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "AI")
    UAdaptationComponent* AdaptationComp; // Посилання на компонент в Pawn

    // Збір метрик [cite: 1054]
    UPROPERTY()
    AMetricsCollector* MetricsCollector;

    // Ключі Blackboard [cite: 489-498]
    UPROPERTY(EditDefaultsOnly, Category = "AI|Blackboard")
    FName Key_TargetActor = "TargetActor";
```

```
    UPROPERTY(EditDefaultsOnly, Category = "AI|Blackboard")
    FName Key_LastKnownLocation = "LastKnownLocation";

    UPROPERTY(EditDefaultsOnly, Category = "AI|Blackboard")
    FName Key_AlertLevel = "AlertLevel";
};
```

Алгоритм 1.1 Клас контролер ШІ AEnemyAIController. Файл EnemyAIController.h

```
#include "EnemyAIController.h"
#include "BehaviorTree/BlackboardComponent.h"
#include "BehaviorTree/BehaviorTreeComponent.h"
#include "Perception/AIPerceptionComponent.h"
#include "Perception/AISenseConfig_Sight.h"
#include "AdaptationComponent.h"
#include "Kismet/GameplayStatics.h"
#include "MetricsCollector.h"
#include "EnemyBaseUnit.h"

AEnemyAIController::AEnemyAIController()
{
    AIPerceptionComp = CreateDefaultSubobject<UAIPerceptionComponent>(TEXT("AIPerceptionComp"));
    // Налаштування зору зазвичай робиться в BP або тут через ConfigSight
}

void AEnemyAIController::OnPossess(APawn* InPawn)
{
    Super::OnPossess(InPawn);

    // Отримуємо компонент адаптації від Юніта [cite: 1052]
    AEnemyBaseUnit* EnemyPawn = Cast<AEnemyBaseUnit>(InPawn);
    if (EnemyPawn)
    {
        AdaptationComp = EnemyPawn->GetAdaptationComponent();
    }

    // Підписка на оновлення сенсорів
                                        AIPerceptionComp->OnTargetPerceptionUpdated.AddDynamic(this,
&AEnemyAIController::OnTargetPerceptionUpdated);

    // Знаходимо колектор метрик на сцені
                MetricsCollector      =      Cast<AMetricsCollector>(UGameplayStatics::GetActorOfClass(GetWorld(),
AMetricsCollector::StaticClass()));
```

```
}

void AEnemyAIController::OnTargetPerceptionUpdated(AActor* Actor, FAIStimulus Stimulus)
{
    // Оновлення Blackboard (Формула 2.15.2 Update()) [cite: 720]
    if (Actor->ActorHasTag("Player"))
    {
        if (Stimulus.WasSuccessfullySensed())
        {
            GetBlackboardComponent()->SetValueAsObject(Key_TargetActor, Actor);
            GetBlackboardComponent()->SetValueAsInt(Key_AlertLevel, 3); // Alert

            // Логування часу реакції [cite: 937]
            if(MetricsCollector)
            {
                MetricsCollector->LogEvent("TargetDetected", GetWorld()->GetTimeSeconds());
            }
        }
        else
        {
            // Втрата цілі -> запам'ятати останню локацію
            GetBlackboardComponent()->SetValueAsVector(Key_LastKnownLocation, Stimulus.StimulusLocation);
        }
    }
}
```

Алгоритм 1.2 Клас контролер ШІ AEnemyAIController. Файл EnemyAIController.cpp

```
#pragma once

#include "CoreMinimal.h"
#include "Components/ActorComponent.h"
#include "AdaptationComponent.generated.h"

// Типи дій згідно з множиною Pi (Формула 2.5) [cite: 499-505]
UENUM(BlueprintType)
enum class EAIActionType : uint8
{
    Attack     UMETA(DisplayName = "Attack"),
    Retreat    UMETA(DisplayName = "Retreat"),
    Investigate UMETA(DisplayName = "Investigate"),
    Patrol     UMETA(DisplayName = "Patrol")
};

// Структура ваг для функції корисності (Формула 2.15.3) [cite: 731]
```

```
USTRUCT(BlueprintType)
struct FUtilityWeights
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere)
    float Aggressiveness = 0.5f; // w1

    UPROPERTY(EditAnywhere)
    float Caution = 0.5f;       // w2

    UPROPERTY(EditAnywhere)
    float Efficiency = 0.5f;    // w3
};

UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class DIPLOMA_API UAdaptationComponent : public UActorComponent
{
    GENERATED_BODY()

public:
    UAdaptationComponent();

    // Розрахунок корисності дії (Формула 2.6, 2.15.3) [cite: 510, 731]
    UFUNCTION(BlueprintCallable, Category = "AI|Adaptation")
    float CalculateUtility(EAIActionType Action, float TargetDistance, float HealthPercent, float ThreatLevel);

    // Оновлення ваг методом градієнтного спуску (Формула 2.13, 2.15.5) [cite: 694, 747]
    UFUNCTION(BlueprintCallable, Category = "AI|Adaptation")
    void UpdateWeights(EAIActionType LastAction, float ResultReward);

    // Отримання найкращої дії (Формула 2.15.4) [cite: 740]
    UFUNCTION(BlueprintCallable, Category = "AI|Adaptation")
    EAIActionType GetBestAction(float TargetDistance, float HealthPercent, float ThreatLevel);

protected:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AI|Weights")
    FUtilityWeights CurrentWeights;

    // Коефіцієнт навчання (eta) [cite: 700]
    UPROPERTY(EditAnywhere, Category = "AI|Learning")
    float LearningRate = 0.1f;
};
```

Алгоритм 2.1 Клас адаптації поведінки `UAdaptationComponent`. Файл `EnemyAIController.h`

```
#include "AdaptationComponent.h"

UAdaptationComponent::UAdaptationComponent()
{
    PrimaryComponentTick.bCanEverTick = false;
}

float UAdaptationComponent::CalculateUtility(EAIActionType Action, float TargetDistance, float HealthPercent,
float ThreatLevel)
{
    float Utility = 0.0f;

    // Реалізація функції корисності U(a, S, M) [cite: 731]
    switch (Action)
    {
    case EAIActionType::Attack:
        // U_attack = w1 * Detect + w2 * (1-Risk) ... [cite: 510]
        // Спрощена модель для коду: Близько + Здоровий + Агресивність
        Utility = (CurrentWeights.Aggressiveness * HealthPercent) + (1.0f - TargetDistance / 2000.0f);
        break;

    case EAIActionType::Retreat:
        // Висока корисність, якщо мало здоров'я і висока загроза
        Utility = (CurrentWeights.Caution * ThreatLevel) + (1.0f - HealthPercent);
        break;

    case EAIActionType::Investigate:
        Utility = 0.5f; // Базова корисність
        break;
    }

    return FMath::Clamp(Utility, 0.0f, 1.0f);
}

void UAdaptationComponent::UpdateWeights(EAIActionType LastAction, float ResultReward)
{
    // Формула 2.13: w(t+1) = w(t) + eta * (P(a) - U(a)) [cite: 694]

    // Припустимо, ми оцінюємо останню корисність як середнє поточних параметрів
    float ExpectedUtility = 0.5f; // Тут має бути збережене значення з моменту прийняття рішення
    float Error = ResultReward - ExpectedUtility;

    if (LastAction == EAIActionType::Attack)
    {
```

```
        CurrentWeights.Aggressiveness += LearningRate * Error;
        CurrentWeights.Aggressiveness = FMath::Clamp(CurrentWeights.Aggressiveness, 0.1f, 1.0f);
    }
    else if (LastAction == EAIActionType::Retreat)
    {
        CurrentWeights.Caution += LearningRate * Error;
        CurrentWeights.Caution = FMath::Clamp(CurrentWeights.Caution, 0.1f, 1.0f);
    }
}

EAIActionType UAdaptationComponent::GetBestAction(float TargetDistance, float HealthPercent, float
ThreatLevel)
{
    // argmax U(a) [cite: 740]
    float ScoreAttack = CalculateUtility(EAIActionType::Attack, TargetDistance, HealthPercent, ThreatLevel);
    float ScoreRetreat = CalculateUtility(EAIActionType::Retreat, TargetDistance, HealthPercent, ThreatLevel);

    if (ScoreAttack > ScoreRetreat)
    {
        return EAIActionType::Attack;
    }
    return EAIActionType::Retreat;
}
```

Алгоритм 2.2 Клас адаптації поведінки `UAdaptationComponent`. Файл `EnemyAIController.cpp`

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "MetricsCollector.generated.h"

USTRUCT(BlueprintType)
struct FMetricData
{
    GENERATED_BODY()

    UPROPERTY()
    float Timestamp;

    UPROPERTY()
    FString MetricName;
```

```
    UPROPERTY()
    float Value;
};

UCLASS()
class DIPLOMA_API AMetricsCollector : public AActor
{
    GENERATED_BODY()

public:
    AMetricsCollector();

    // Логування події [cite: 1007]
    UFUNCTION(BlueprintCallable, Category = "Analytics")
    void LogEvent(FString MetricName, float Value);

    // Збереження у CSV [cite: 1007]
    UFUNCTION(BlueprintCallable, Category = "Analytics")
    void SaveToCSV();

protected:
    virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;

private:
    TArray<FMetricData> SessionMetrics;
    FString FilePath;
};
```

Алгоритм 3.1 Клас для збору аналітики AMetricsCollector. Файл MetricsCollector.h

```
#include "MetricsCollector.h"
#include "Misc/FileHelper.h"
#include "Misc/Paths.h"

AMetricsCollector::AMetricsCollector()
{
    PrimaryActorTick.bCanEverTick = false;
    FilePath = FPaths::ProjectSavedDir() + "SimulationMetrics.csv";
}

void AMetricsCollector::LogEvent(FString MetricName, float Value)
{
    FMetricData NewMetric;
    NewMetric.Timestamp = GetWorld()->GetTimeSeconds();
    NewMetric.MetricName = MetricName;
```

```
    NewMetric.Value = Value;

    SessionMetrics.Add(NewMetric);
}

void AMetricsCollector::SaveToCSV()
{
    FString CsvContent = TEXT("timestamp,metric_name,metric_value\n");

    for (const FMetricData& Data : SessionMetrics)
    {
        // Формат згідно з [cite: 941]
        FString Line = FString::Printf(TEXT("%.2f,%s,%.2f\n"), Data.Timestamp, *Data.MetricName, Data.Value);
        CsvContent += Line;
    }

    // Запис у файл (синхронний для простоти, в реальному проекті краще async)
    FFileHelper::SaveStringToFile(CsvContent, *FilePath);
    UE_LOG(LogTemp, Log, TEXT("Metrics saved to %s"), *FilePath);
}

void AMetricsCollector::EndPlay(const EEndPlayReason::Type EndPlayReason)
{
    Super::EndPlay(EndPlayReason);
    SaveToCSV(); // Автоматичне збереження при завершенні місії
}
```

Алгоритм 3.2 Клас для збору аналітики AMetricsCollector. Файл MetricsCollector.cpp

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "EnemyBaseUnit.generated.h"

class UAdaptationComponent;

UCLASS()
class DIPLOMA_API AEnemyBaseUnit : public ACharacter
{
    GENERATED_BODY()

public:
```

```
    AEnemyBaseUnit();

    // Компонент адаптації є частиною юніта [cite: 1052]
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "AI")
    UAdaptationComponent* AdaptationComp;

    UFUNCTION(BlueprintCallable, Category = "Combat")
    void FireWeapon();

    UFUNCTION(BlueprintCallable, Category = "AI")
    UAdaptationComponent* GetAdaptationComponent() const { return AdaptationComp; }
};
```

Алгоритм 4.1 Клас основа для ворожих юнітів. Файл EnemyBaseUnit.h

```
#include "EnemyBaseUnit.h"
#include "AdaptationComponent.h"

AEnemyBaseUnit::AEnemyBaseUnit()
{
    PrimaryActorTick.bCanEverTick = true;

    // Ініціалізація компонента адаптації
    AdaptationComp = CreateDefaultSubobject<UAdaptationComponent>(TEXT("AdaptationComp"));
}

void AEnemyBaseUnit::FireWeapon()
{
    // Логіка пострілу...
    // Тут можна викликати оновлення ваг, якщо постріл був успішним/неуспішним
}
```

Алгоритм 4.2 Клас основа для ворожих юнітів. Файл EnemyBaseUnit.cpp