Національний університет «Полтавська політехніка імені Юрія Кондратюка»
(повне найменування вищого навчального закладу)
Навчально-науковий інститут інформаційних технологій та робототехніки
(повна назва факультету)
Кафедра комп'ютерних та інформаційних технологій і систем
(повна назва кафедри)

Пояснювальна записка
до дипломного проекту (роботи)
магістра
(освітньо-кваліфікаційний рівень)
на тему
Розробка та інтеграція системи адаптивного інтелекту у відеоігри через плагін для Unreal Engine

Виконав: студент 6 курсу, групи 601-ТН
спеціальності
122 Комп'ютерні науки
(шифр і назва напряму)
Яковенко А.В.
(прізвище та ініціали)
Керівник Двірна О.А
(прізвище та ініціали)
Рецензент _____
(прізвище та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ПОЛТАВСЬКА ПОЛІТЕХНІКА ІМЕНІ ЮРІЯ КОНДРАТЮКА»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА РОБОТОТЕХНІКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
Спеціальність 122 «Комп'ютерні науки»
на тему

«Розробка та інтеграція системи адаптивного інтелекту у відеоігри через плагін для Unreal Engine»

Студента групи 601-ТН Яковенка Артема Володимировича

Керівник роботи
Кандидат фізико-математичних наук,
доцент
Двірна О.А.
Завідувач кафедри
Кандидат фізико-математичних наук,
доцент
Двірна О.А.

Полтава 2024

РЕФЕРАТ

Кваліфікаційна робота магістра: 76 с., 34 малюнків, 1 додаток, 22 джерела.

Об’єкт дослідження: аналіз та порівняння систем прийняття рішенн в ігровому штучному інтелекті.

Мета роботи: Розробка та інтеграція системи адаптивного інтелекту у відеоігри через плагін для Unreal Engine

Методи: проектування, розробка та інтеграція системи прийняття рішенн за у вигляді плагіну для рушія Unreal Engine 5, розробка демонстраційної сцени.

Ключові слова: Unreal Engine, плагін, штучний інтелект, прийняття рішенн, Blueprints, asset, агент, контроллер, менеджер.

ABSTRACT

Master's qualification work: 76 pages, 34 drawings, 1 appendice, 22 sources.

Object of investigation: analysis and alignment of systems for decision-making in game artificial intelligence.

Purpose of work: Development and integration of the adaptive intelligence system for video games through a plugin for Unreal Engine

Methods: design, development and integration of the system are decided upon in the form of a plugin for Unreal Engine 5, development of a demo scene .

Key words: Unreal Engine, plugin, artificial intelligence, decision making, Blueprints, asset, agent, controller, manager.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

Геймплей – ігровий процес.

NPC (Non-playable character) – неігровий пересонаж, агент, який керується комп’ютером.

Плагін – додатковий компонент або розширення, надбудова до ПЗ, для покращення її функціональності або надання нових можливостей.

Юніт – неігровий персонаж, бойова одиниця.

STRIPS (Stanford Research Institute Problem Solver**) –** автоматичний планувальник.

GOAP (Goal Oriented Action Planing**) –** архітектура системи прийняття рішеннь, основою якої є робота планувальника

Реіграбельність – можливість повторного проходження гри.

SOLID – парадигма об’єктно-орієнтованого програмування.

CDO – (Class Default Object), базовий дочірній об’єкт певного класу.

Actor – об’єкт, який може бути розташовано на сцені.

Асет – будь-який ресурс, що використовується під час розробки гри (моделі, анімації, текстури, звуки, тощо.)

ВСТУП

Однією з найперших відеоігор вважається гра “Tennis for Two” (1958), розроблену на базі осцилографа. Через декілька років, на її зміну прийшла гра “Pong”, геймплей якої був майже ідентичним. На відміну від “Tennis for Two”, “Pong” вважається однією з найперших власне комп’ютерних ігор. Проте, в обох випадках, гравці змагалися один проти одного на одному пристрої, а тому необхідністі в додаткових алгоритмах не було.

З розвитком ігрової індустрії великі компанії почали все частіше випускати нові проекти, що ознаменувало початок ери аркадних та одиночних ігор. У цих іграх гравці могли змагатися не лише один з одним, але й проти комп’ютера. Для створення відчуття протистояння з віртуальним суперником використовувалися прості правила та алгоритми на основі псевдовипадкових чисел:

- **«Space Invaders»** (1978): Вороги представлені космічними кораблями, які поступово опускаються до гравця та з певною частотою випускають снаряди. Ця проста поведінка теж не містить справжньої адаптивності, однак створює ефект загрози з боку комп'ютерних опонентів;
- **«Pac-Man»** (1980): У цій грі противниками виступають "привиди", які рухаються за певними траєкторіями та випадково змінюють напрямок для перехоплення гравця. Хоча така механіка може розглядатися як рання форма ігрового штучного інтелекту, вона базується на жорстко запрограмованих правилах, без адаптивності.

Розвиток локальних та глобальних мереж, як Internet, дав розробникам нову нішу для розробки та розвитку ігрової індустрії. Так з’явилися перші мережеві ігри, де головною механікою була взаємодія гравця з іншими гравцями в мережі, тому в іграх такого жанру ігровий штучний інтелект майже не використовувався. Такі системи в мережевих іграх доволі примітивні та використовуються переважно для тренуваннь чи наповнення ігрового світу,

що підтримує ігрову реалістичність або допомагає гравцям опанувати базові механіки гри. Проте, на відміну від одиночних ігор, де NPC є ключовими для створення цікавого ігрового процесу, в онлайн-іграх акцент робиться на взаємодії між гравцями, тому потреба у розвиненому ігровому ШІ для ігор такого жанру стає меншою.

Сьогодні, з ростом геймерської культури, розробники пропонують дедалі більше ігрових механік, кількість яких збільшується майже в геометричній прогресії. Сучасні ігри, особливо з відкритими світами, тактичними боями або складними взаємодіями між персонажами, вже не можуть обмежуватися простими алгоритмами та випадковими діями. Такі проекти вимагають динамічної зміни поведінки та адаптивності персонажів, які мають реагувати на зміни в ігровому оточенні та події, а також на дії гравця чи інших персонажів.

Це дозволяє створювати ворогів із більш складною поведінкою, які не лише реагують на гравця, але й будують стратегії, кооперуються один з одним та демонструють поведінку, подібну до людської. Високорозвинені системи штучного інтелекту значно підвищують реіграбельність, тобто можливість повертатися до гри та отримувати новий, унікальний досвід під час кожного проходження. Штучний інтелект забезпечує непередбачуваність, адаптацію до стилю гравця і зміну тактики ворогів у кожній новій грі.

Таким чином, завдяки динамічній поведінці та адаптивності ШІ, кожне проходження стає унікальним, адже гравець не може повністю передбачити дії ворогів чи NPC, що робить ігровий досвід більш захопливим та багатогранним.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД СИСТЕМ ШТУЧНОГО ІНТЕЛЕКТУ У ВІДЕОІГРАХ

1.1 Ігровий процес

Ігровий процес у відеоіграх пройшов значний шлях розвитку від простих аркадних ігор з взаємодією між гравцями, до сучасних складних проектів із відкритими світами та адаптивним штучним інтелектом. Системи ШІ стали однією з ключових частин ігрового процесу, дозволяючи створювати інтерактивні, динамічні та захоплюючі ігрові світи.

У перших відеоіграх роль ігрового ШІ виконували прості алгоритми, які здебільшого просто збільшували складність гри відповідно до ігрової прогресії, збільшуючи швидкість чи кількість ворогів. Дії таких алгоритмів часто були передбачуваними, а виклик для гравця полягав здебільшого в швидкості його реакції. (Рис. 1.1)

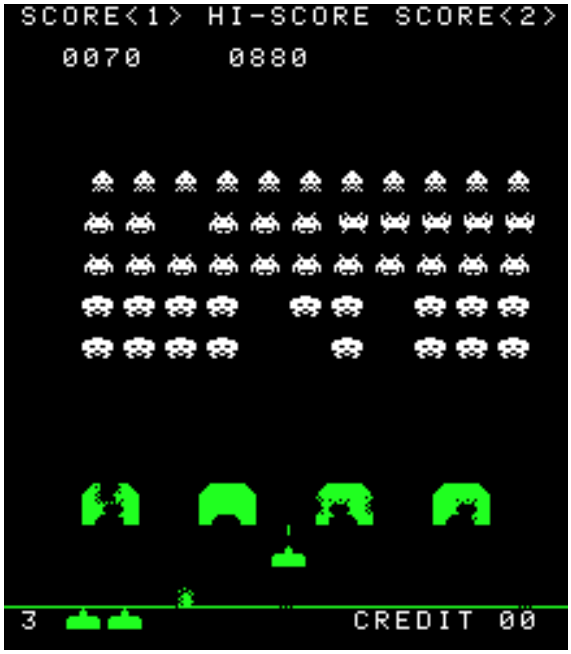

Рисунок 1.1 – Кадр з гри “Space Invaders”, де після знищення кожного ворожого персонажа збільшується швидкість руху інших.

У 90-х роках ігровий процес почав ускладнюватися, а разом із ним і системи ШІ. Яскравим прикладом прориву в розвитку систем ігрового ШІ являється виникнення жанру стратегій в реальному часі (RTS). Ігри, такі як «Dune II» чи серія «Warcraft» (рис. 1.2), вимагали від персонажів виконання більш складних завдань, таких як управління ресурсами, побудова баз і координація військ. Для цього використовувалися тактичні алгоритми, що імітували прийняття рішень людиною на полі бою.

Рисунок 1.2 – Кадр з гри “Warcraft III: Frozen Throne”, на якому зображена атака ворожих юнітів на позицію гравця.

Сучасні відеоігри, особливо ігри з відкритим світом чи певною свободою дій гравця, потребують використання значно складніших систем ШІ. Ігри, накшталт “The Elder Scrolls V: Scyrim”, “The Wither 3: Wild Hunt”, “S.T.A.L.K.E.R.” чи “Red Dead Redemption 2” моделюють поведінку персонажів не лише на основі дій гравця, але й на основі власних потреб чи розпорядку дня (рис. 1.3). В таких іграх, персонажі створюють вигляд

повноціноого життя, виконують рутинні задачі, взаємодіють одне з одним та з оточенням так само, як взаємодіють з гравцем.

Рисунок 1.3 – Кадр з гри “Red Dead Redemption 2”, на якому зображено, як персонажі в вільний час займаються власними справами

У 2007 році було представлено першу частину культової серії ігор «S.T.A.L.K.E.R.», у якій використовувалася пропрієтарна та ексклюзивна система штучного інтелекту під назвою «A-Life». Систему було представлено не просто у вигляді моделі керування поведінкою агентів, а як «симуляцію справжнього життя» в ігровому світі. На відміну від інших ігор, де персонажі існують лише на одному рівні з гравцем або в його полі зору, A-Life дозволяла агентам діяти незалежно від місцезнаходження чи дій гравця. A-Life поділяється на локальну та глобальну. Локальна діє приблизно в радіусі 150 метрів від гравця, — це складна система штучного інтелекту: на головного героя нападають монстри і неігрові персонажі, відбуваються напади на табори угруповань за участю гравця: загалом все те, що можна спостерігати на власні очі, а не припускати. Глобальна A-Life — це спрощена система штучного інтелекту, що діє скрізь і завжди. Всі події, які не можна бачити, але потім дізнатися про них, в результаті яких, наприклад, можна знайти мертвих NPC. Всі переміщення «населення» Зони також контролюються A-Life: якщо

сталкер переходить з однієї локації на іншу, він насправді туди переходить, а не переміщується, телепортуючись з місця на місце [1].

У 2020 році було анонсовано продовження серії у вигляді гри «S.T.A.L.K.E.R. 2: Heart of Chornobyl», у якій розробники суттєво доопрацювали та представили оновлену версію системи штучного інтелекту під назвою «A-Life 2.0». Окрім основних технічних рішень, реалізованих у попередній версії, було додано механіку «ідеології» кожного агента, яка значною мірою впливає не лише на проходження гри, а й суттєво впливає на зміни та підії ігрового світу. Під час одного з інтерв'ю, розробники згадали дану систему та дали невеликий опис того, як вона працюватиме у грі: «Все залежить від фракції або типу монстра, якого ти зустрів. Деякі з них захищають свою територію, бо наша система A-Life 2.0 побудована на декількох стовпах. Є оффлайн режим, де якісь події відбуваються на фоні. І під час вашого проходження ви можете побачити результати або перестрілок або битв з мутантами тощо. У деяких фракцій або мутантів є своє середовище проживання, а деякі з них можуть просто бродити по карті. Тому життя в Зоні відбувається як перед очима гравця так і поза його полем зору. [2]»

В контексті розвитку ігрового штучного інтелекту, неможливо не згадати про системи, що використовуються для реалізації суперника в шахах. Перше дослідження на тему шахового програмування зробив американський математик Клод Шеннон в 1950 році, він успішно передбачив два основні можливі методи пошуку, які можна використовувати при грі і назвав їх «Тип А» і «Тип В». Першу успішну шахову програму створили 1952 року в лабораторії Лос-Аламоса на комп'ютері MANIAC I з тактовою частотою 11 кГц, комп'ютер грав на дошці розміром 6x6 і без участі слонів, всього зіграно було дві партії проти сильного шахіста і новачка. Першу партію комп'ютер програв, а другу зміг виграти на 23-му ходу. [3] В 1995 році компанією IBM було завершено побудову шахматної машини DeepBlue II, яка в 1997 році отримала перемогу в матчі з 6 партій над тодішнім чемпіоном світу Гаррі Каспаровим [4].

1.1.1. Ігровий досвід. Основною метою будь-якої відеоігри є створення ігрового досвіду, який досягається через поєднання різних елементів: сюжету, ігрових механік, варіативності, інтенсивності процесу, а також графіки, звукових ефектів і візуальних елементів. У аркадних та казуальних іграх, таких як «Tetris» або «Pac-Man», основний ігровий досвід зосереджений на наборі очок і встановленні рекордів. Вони досягають цього через поступове підвищення складності, що виявляється у збільшенні швидкості руху елементів гри або посиленні ворогів, що стимулює гравця до досягнення нових результатів.

Із стрімким розвитком ігрової індустрії та розширенням ігрових механік, зростають і масштаби проектів, що, відповідно, впливає на їх вартість. Якою б не захоплюючою була б гра, гравці зазвичай не готові платити великі суми лише за одне проходження, тому однією з задач для розробників є збільшення кількості можливих проходжень гри.

Для досягнення реіграбельності сучасні ігри використовують різноманітні стилі гри, як прямий бій або стелс у шутерах, різні класи персонажів у рольових іграх, або гілки розвитку головного героя, що дозволяє гравцям обирати різні підходи до виконання завдань. Крім того, варіативність сюжетних що спонукає гравців пройти гру кілька разів, вибираючи різні варіанти розвитку подій.

Одним із популярних способів збільшення реіграбельності є підвищення рівня складності в процесі проходження гри або впровадження механізму "гри+" (New Game Plus), що дозволяє гравцям знову пройти гру, зберігаючи деякі елементи прогресу, наприклад, спорядження або здібності персонажа. Ці методи сприяють підтримці інтересу гравців до гри після її першого проходження та забезпечують можливість відкривати нові аспекти ігрового процесу.

Проте більшість із цих підходів не впливають суттєво на сам ігровий світ та його наповнення, що може призводити до відчуття повторюваності після кількох проходжень, та, як наслідок, втрати інтересу до гри.

В більш загальному розумінні, ігровий досвід є сукупністю подій чи ситуацій, що виникають у процесі гри. Оскільки кожен такий набір згенерований випадковим чином, для кожного гравця під час кожного проходження ігровий досвід є унікальним, незважаючи на те, що всі механіки, ефекти та сюжет залишаються незмінними.

Як розробники, ми не створюємо події безпосередньо, займаючись лише проектуванням та розробкою ігрових механік. Події генеруються в результаті дій гравця, або ж персонажів, що населюють ігровий світ.

Ігровий ШІ має значний вплив на якість і занурення у ігровий досвід. Завдяки можливості динамічно адаптуватися до оточення та змінювати поведінку залежно від конкретної ситуації, система ШІ може забезпечувати унікальний досвід під час кожного проходження гри, а в комбінації з ігровими механіками створювати унікальні ситуації, які раніше не мали місця. Це дозволяє уникнути передбачуваних сценаріїв і сприяє збереженню інтересу гравців, адже кожне проходження може кардинально відрізнятися від попереднього.

Неігрові персонажі та їх поведінка. У відеоіграх, неігрові персонажі (NPC, non-playable characters), це персонажі, якими не керує людина. Як правило, ними керує комп’ютер, за допомогою планової (скриптованої) чи реактивної (динамічної) поведінки.

В будь якому випадку, ігровий штучний інтелект здебільшого являє собою набір алгоритмів. Всі можливі дії, сценарії поведінки, а також умови та ситуації, за яких ці дії здійснюються, заздалегідь визначаються та прописуються розробником. Задачею агента являється процес прийняття рішенн, на основі певних встановлених розробником факторів, коли та яку саме із запропонованих дій потрібно виконати.

Незалежно від використовуваної архітектури системи ігрового штучного інтелекту, будь-яка сучасна модель спирається на три основні алгоритми:

- **Pathfinding** (пошук шляху) – алгоритм пошуку найоптимальнішого та найкоротшого шляху з точки А в точку В. Існує певна кількість можливих алгоритмів, серед яких найбільш застосовуваним в сфері розробки ігор являється алгоритм A* (а-стар), який знаходить найкоротший шлях з найменшою вартістю між двома точками графу [5];
- **Data Mining –** процес збору даних з ігрового оточення, а також інформації про стан агента для подальшої обробки та використання під час прийняття рішень;
- **Decission making –** процесс прийняття рішеннь агентом який є критично важливим аспектом штучного інтелекту в іграх, що дозволяє NPC приймати обгрунтовані рішення на основі оточення, їх цілей та мотивацій.

В залежності від проекту, процес збору та збереження даних ігровими персонажами ґрунтується на імітації людського сприйняття та пам’яті. Завдяки системам, які моделюють зір, слух чи дотик, персонажі отримують інформацію про події та об’єкти в ігровому світі. Короткочасна пам’ять використовується для збереження миттєвих даних, як-от останнє місце, де був помічений ворог, або звуки, що привернули увагу. Довготривала пам’ять дозволяє персонажам накопичувати глобальні дані, наприклад, інформацію про союзників, вивчені маршрути, стан здоров’я агента, його спорядження, що впливає на їх поведінку впродовж всієї гри.

Процес прийняття рішень NPC зазвичай заздалегідь моделюється розробником. Як правило, система будується на основі моделі "Якщо-То" та складається з можливих дій і станів, зв'язків між ними та умов переходів. Задача агента полягає у зборі даних та на їх основі перевірки всіх можливих умов переходу між встановленими станами. Якщо хоч одна з них виконується, агент автоматично переходить до виконання поставленої задачі.

Основна відмінність ігрового ШІ від традиційних моделей полягає в тому, що його метою є не створення справжньої інтелектуальної поведінки, а лише імітація її реалістичності для гравця. Такі системи зазвичай не

використовують методи машинного навчання або еволюційні алгоритми, оскільки вони створюються під специфічні вимоги гри. Використання таких технологій може значно ускладнити процес розробки та оптимізації гри.

1.2. Основні стратегії розробки систем прийняття рішенн

Найпростішою формою ігрового штучного інтелекту є скриптована поведінка. У таких системах агенти реагують на взаємодію з гравцем або іншими персонажами за допомогою тригерів, які ініціюють виконання певних дій у фіксовані моменти часу. Агенти не застосовують складних алгоритмів для аналізу ситуації чи прийняття рішень, а лише слідують чітко визначеному сценарію, заздалегідь написаному розробником.

Одним з варіантів розробки більш продвинутих систем є реалізація за допомогою теорії графів. В якості найбільш розповсюджених прикладів такого підходу можна привести кінцеві автомати станів (Finite State Machines, FSM) та дерева поведінки (Behavior Trees). Обидві архітектури є ієрархічними структурами, що дозволяють розбивати поведінку персонажів на менші, керовані частини. Основною перевагою таких систем є їх модульність і відносна простота реалізації. FSM та дерева поведінки часто використовуються для створення структурованої і передбачуваної поведінки, але можуть бути недостатньо гнучкими для складніших сценаріїв, що вимагають адаптивності та варіативності в діях агентів. Окрім того, ці підходи можуть виявлятися нестабільними при реалізації складної логіки, оскільки збільшення кількості можливих станів або умов ускладнює управління системою, підвищуючи ризик виникнення конфліктів чи непередбачуваної поведінки.

Кінцевий автомат (або машина станів) – один з найстаріших способів моделювання поведінки персонажів. В основу FSM покладено принцип того, що в кожний момент часу NPC знаходиться в певному чітко визначеному

стані. Таких станів є кінцеве число, і всі вони відомі заздалегідь. Так, наприклад, станами можуть бути: нічого не робіння, ходіння по маршруту, програвання звуку або анімації. Особливим станом є знаходження NPC під контролем гри. NPC може перейти з одного стану до іншого після виконання деякої умови переходу. При заданні умови переходу зі стану A стан B ми фактично визначаємо, за яких умов буде здійснено перехід з одного стану в інший. При цьому перехід з B A вимагає визначення своєї власної умови переходу. Якщо умови переходу, між якими або двома станами не задано, перехід вважається неможливим. [7]

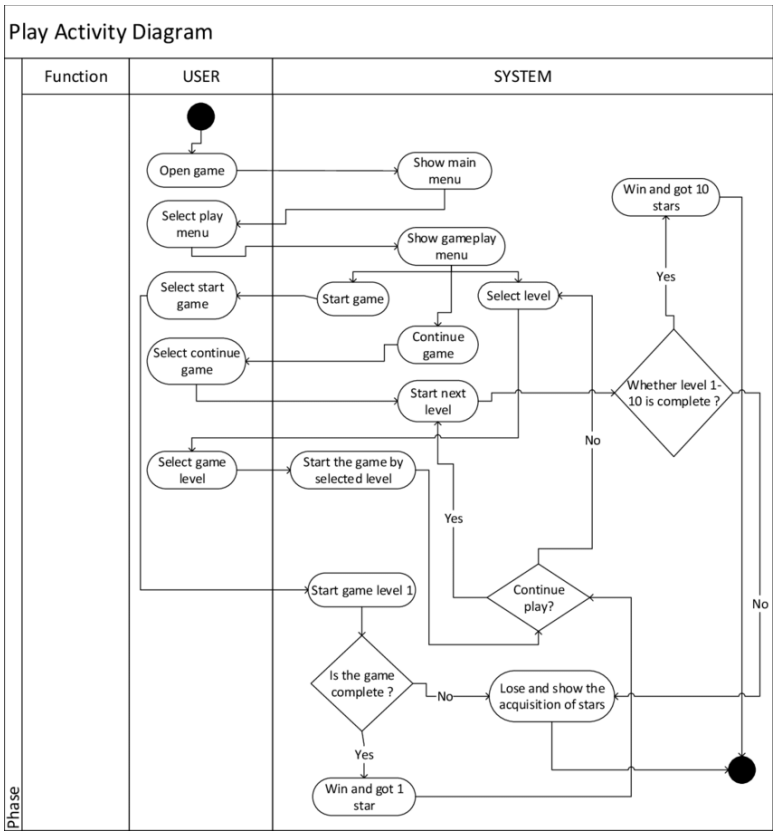

Рисунок 1.4 – Приклад графу кінцевого автомату

Основним недоліком цієї системи є її масштабованість: чим більше станів та переходів містить кінцевий автомат, тим складніше його налаштовувати та відстежувати результат роботи. При зміні геймплею гри потрібно змінювати і сам граф станів, але з ростом кількості станів зростає ймовірність появи помилок під час його адаптації до актуальних потреб. Ця

ймовінність зростає майже в геометричній прогресії, що суттєво впливає на гнучкість і стійкість системи.

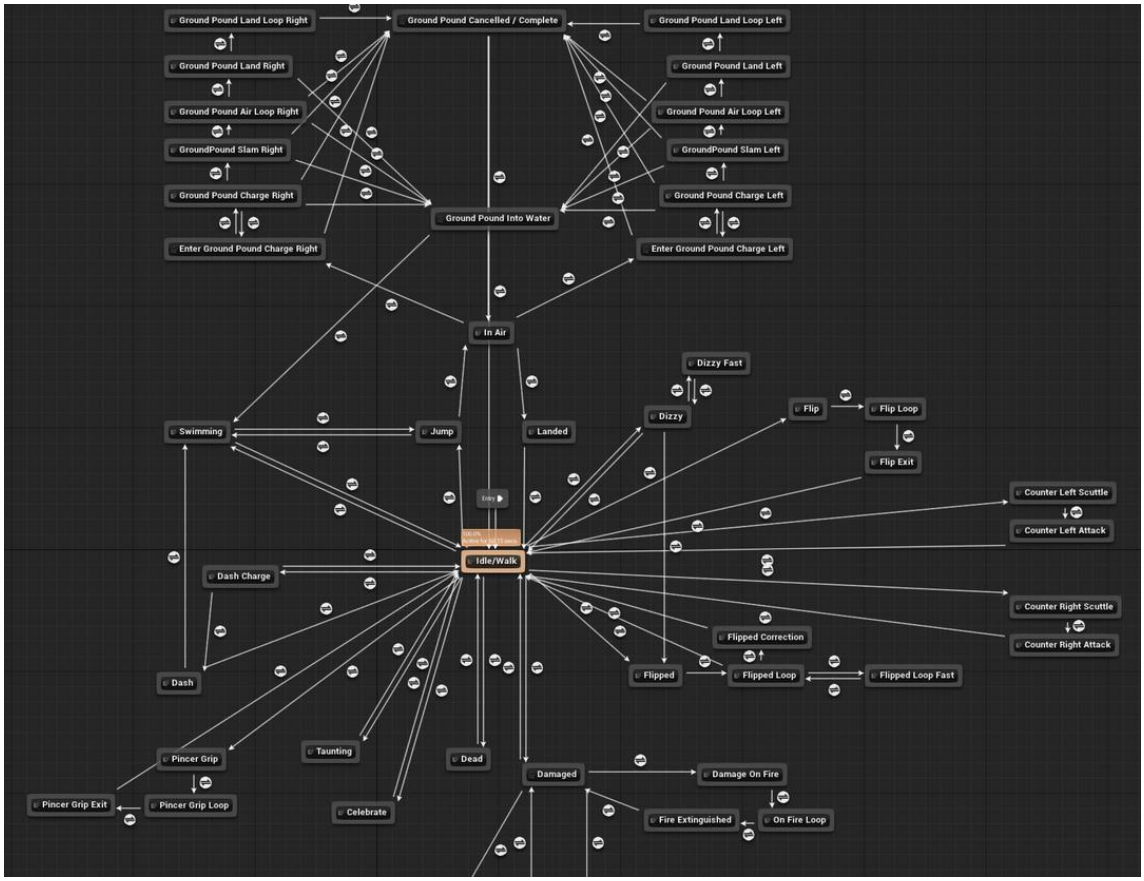

Рисунок 1.5 – Комплексний граф машини станів.

Частково дану проблему вирішує використання ієрархічних FSM, однак при збільшенні кількості станів система все ще залишається занадто ненадійною.

Найбільшої популярності система набрала з виходом гри “HalfLife” (1998). На даний час вважається достатньо застарілою та незовсім зручною для моделювання більш продвинутих систем прийняттня рішенню, проте часто використовується в розробці для моделювання систем анімацій персонажів.

Дерева поведінки (Behavior Tree’s). Дерева поведінки – ієрархічна система прийняття рішеннь у вигляді направленого ациклічного графу. Структура даних в якій задається набір правил, поведінок, їхні умови та порядок їх виконання. Поведінкові дерева це дуже розповсюджений для сучасний

комп’ютерних ігор дизайн паттерн, за допомогою якого можна визначити поведінку та логіку прийняття рішень для ігрової сутності[8]. Граф складається зі скінченного набору об’єктів (вузлів), кожен з яких в класичній реалізації повертає значення Success (успіх), Failure (невдача), Running (виконання).

Кожен з таких елементів відповідає певній поведінці або режимам роботи в системі, наприклад, «рухатися вперед», «закрити захоплення», «блимнути попереджувальними лампами», «перейти до зарядної станції». Кожен клас моделі має певний набір правил, які описують, коли агент повинен виконувати кожну з цих дій, і, що більш важливо, як агент повинен перемикатися між ними[9]. Такий спосіб достатньо зручний для моделювання та маштабування поведінки персонажів, а також доволі прозорий, що спрощує відслідковування його роботи.

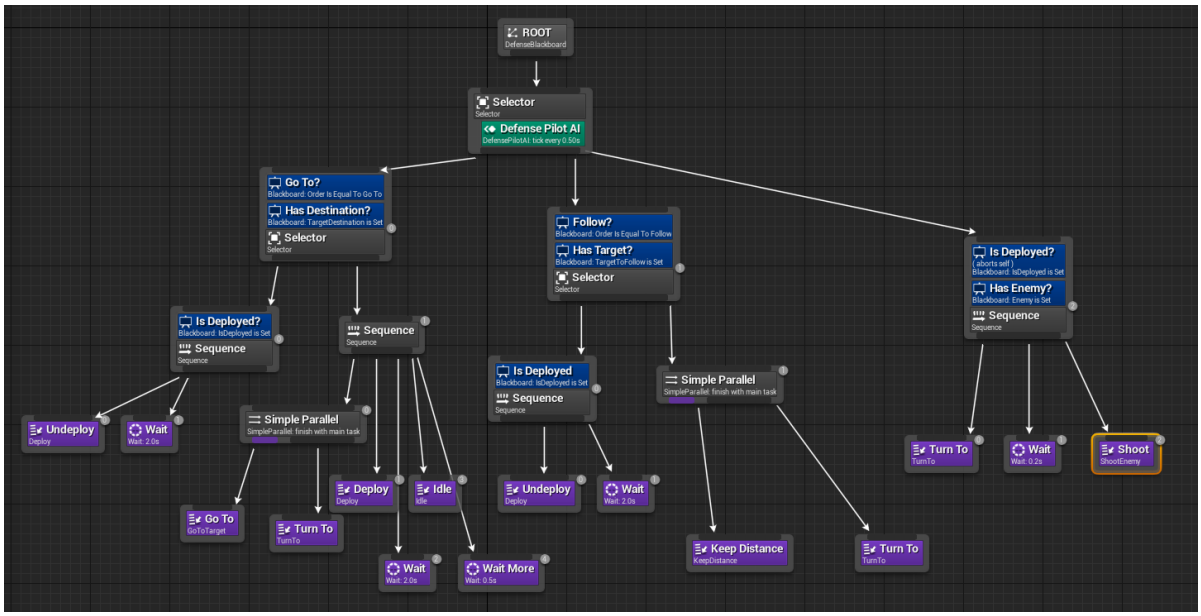

Рисунок 1.6 – Дерево поведінки

Хоч, в порівнянні з Finite State Machine дерева поведінки є більш структурованими моделями, проте при реалізації більш комплексної логіки дерево може обрости значною кількістю гілок, що збільшує ймовірність виникнення помилок, особливо при зміні загальноігрової логіки (будь то створення нових механік чи видалення або зміна інших).

GOAP — архітектура на основі **STRIPS**, яка надає NPC більше автономії, ніж просто реагувати на гравця. Замість цього вони вибирають мету зі списку варіантів і складають найкращий план для досягнення цієї мети. Для цього система використовує два стандартних компоненти штучного інтелекту – A* і кінцевий автомат (**FSM**), але вони використовуються нетрадиційно. Зазвичай FSM контролює всю поведінку NPC за допомогою списку можливих станів, причому A* планує шляхи. Однак FSM має лише три стани ("GoTo", "Animate" і "UseSmartObject"), а A* використовується для планування послідовності дій, а також для планування шляхів. По суті, це означає, що A* здійснює навігацію в FSM, вибирає стан, вибирає, коли ініціювати перехід стану, і вибирає, які параметри виконувати в кожному стані. [10]

В загальному, архітектура планувальника виглядає наступним чином: кожен план складається з цілі, якої необхідно досягти агенту, та послідовностей дій необхідних для її досягнення. Кожна дія в свою чергу складається з умови, необхідної для виконання даної дії, ефекту – результату, який досягається по завершенню виконання, та вартості виконання дії, значення якої може бути як статичним, так змінюватися динамічно відповідно до контексту задачі та поточної ситуації.

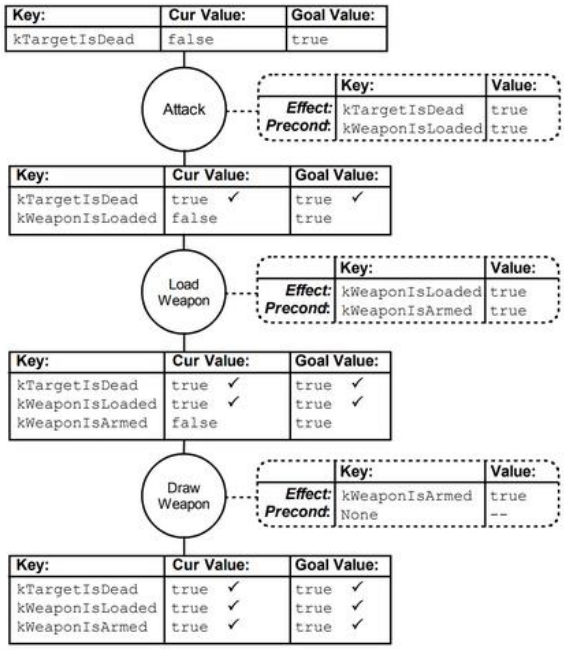

Рисунок 1.7 – Схема плану для виконання задачі ліквідації противника.

По завершенню палнувальник обирає найбільш дешевий план, на основі вартості кожної дії в ньому. Проте, якщо в ході виконання плану агент зазнає невдачі, планувальник виконує наступний по собівартості розроблений раніше план, що дозволяє агенту швидко адаптуватися до змін та зменшує імовірність виникнення так званого «простою».

Одним з найперших та найуспішніших прикладів застосування архітектури GOAP є гра F.E.A.R (2005). В свій час гра здобула визнання через одних з найкращих та найрозумніших противників, у порівнянні з іншими конкурентами. Персонажі демонструють тактичну поведінку, адаптуючись до дій гравця, шукаючи укриття, обходячи з флангів, що створює враження високого інтелекту NPC.

Utility AI – архітектура проектування системи прийняття рішенн агентом ігрового ШІ. Основна ідея архітектури в тому, що кожна можлива дія чи стан у рамках даної моделі може бути описана з єдиним значенням. Це значення, яке зазвичай називають корисністю, описує доцільність використання конкретної дії в певному контексті.

Оцюнювання корисності дій відбувається за допомогою оцінювача (**evaluator**) на основі певних критеріїв, або «міркуваннь» (**considerations**), які повертають значення в певному числовому діапазоні (зазвичай [0, 1]). Під час кожної ітерації оцінювач обирає задачу з найбільшим показником доцільності та запускає її виконання. Розрахунок оцінки виконується різними способами, за допомогою формул, з фикористанням кривих чи за допомогою поєднання цих способів.

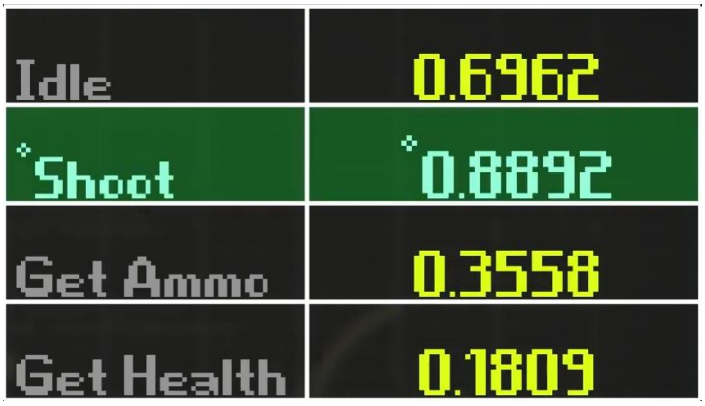

Рисунок 1.8 – Оцінювання та вибір доступних дій

Перевага використання подібної архітектури полягає у відсутності зв'язків між станами. Система складається з масиву задач, з якого в кожен момент часу обирається задача з найбільшою корисністю для поточного контексту. Розробнику потрібно лише додати нову задачу або видалити існуючу за необхідності, що спрощує підтримку та розширення системи.

У The Sims (2000), в якій архітектура виступає основною механікою гри, поточна «потреба» агента (наприклад, відпочинок, їжа, соціальна активність) була об’єднана з оцінкою об’єкта чи діяльності, які могли б задовольнити ту саму потребу. Комбінації цих значень давали оцінку дії, яка вказувала агенту, що він повинен робити. У The Sims 3 (2009) Річард Еванс використовував модифіковану версію розподілу Больцмана, щоб вибрати дію для агента, використовуючи низьку температуру, коли персонаж щасливий, і високу, коли персонаж працює погано, щоб обрати дію з низькою корисністю. Він також включив «особистості» в Sims. Це створило свого роду 3-осьову модель — розширення числових «потреб» і «значень задоволеності», щоб включити вподобання, щоб різні агенти могли реагувати по-різному від інших за тих самих обставин на основі їхніх внутрішніх бажань і прагнень. [9]

Така архітектура є більш гнучкою і пропонує значно більші можливості для масштабування в порівнянні з деревами поведінки або автоматами станів. Це обумовлено тим, що розробнику достатньо лише налаштувати нову дію та додати її до існуючого списку, без необхідності кардинальних змін у всій

системі. Проте такий підхід також має свої недоліки: процес прийняття рішень агентом стає менш прозорим та важчим для контролю. Відстеження і аналіз поведінки агента у такій системі може ускладнитися через велику кількість факторів, які одночасно впливають на вибір дій.

. Комбіновані архітектури. Незважаючи на наявну варіативність підходів до розробки систем ігрового ШІ, часто розробники використовують комбіновані рішення, адаптуючи їх під специфіку конкретного проекту. Такий підхід дозволяє поєднувати різні алгоритми та методи, а також комбінувати сильни сторони різних архітектур що сприяє створенню більш складних і гнучких моделей поведінки для NPC.

Гра «Alien: Isolation» вирізняється серед інших своєю унікальною та, на даний час, однією з найкращих систем ігрового штучного інтелекту. Ця система поєднує дерева поведінки (**Behavior Trees**) з утилітарним ШІ (**Utility AI**) і використовує двошарову архітектуру, що дозволяє противнику не лише реагувати на гравця та оточення, а й вивчати, запам’ятовувати та адаптуватися до його поведінки.

Серії ігор «Tom Clancy's Rainbow Six», «HALO», а також значна кількість ігор в жанрі стратегії активно використовують поєднання кількох архітектур для створення так званого тактичного ігрового ШІ (**Tactical AI**). Це дозволяє агенту в реальному часі обирати найдоцільніші завдання для конкретного контексту, створювати план їх виконання, а також передбачати та планувати подальші дії.

Моделі на основі машинного навчання. Втім, розробники постійно шукають шляхи для вдосконалення існуючих рішеннь та підвищення ігрового досвіду. Окрім основних описаних стратегій, одним з найочевидніших варіантів є використання алгоритмів машинного навчання та нейронних мереж. Така стратегія розвитку дозволяє створювати майже повністю автономних

персонажів, здатних адаптуватися до різних умов і виявляти поведінку, яка не була чітко запрограмована розробниками.

Навідміну від інших стратегій, використання алгоритмів машинного навчання не передбачає створення конкретних правил розробником зазадлегідь для керування поведінкою агентів. Навпаки, використання таких алгоритмів, як **навчання з підкріпленням** (Reinforcement learning) дозволяє агентам вчитися на власному досвіді методом проб та помилок безпосередньо під час гри, адаптуючи та моделюючи стратегії поведінки. Окрім того, відеоігри є практично ідеальним середовищем для навчання та тестування таких алгоритмів, оскільки основним суперником виступає велика кількість гравців, кожен з яких створює унікальні виклики для системи.

За останні роки прогрес у розвитку машинного навчання дозволив розробникам ігор створювати більш захоплюючий і складний ігровий досвід, де сама гра адаптується до дій і рішень гравця. Використання навчання з підкріпленням у стратегічних іграх, таких як Total War та інших рольових іграх, стало дуже перспективним напрямком досліджень із великим потенціалом для розвитку штучного інтелекту і машинного навчання. Унікальні виклики, пов’язані з цими іграми, включно з багатовимірним станом простору і та необхідністю контролювати мільйони віртуальних одиниць за допомогою штучного інтелекту, зробили їх переконливим випробувальним майданчиком для алгоритмів машинного навчання. [10]

Як інший приклад можна навести метод **імітаційного навчання** (Imitation Learning), у якому агенти націлені на наслідування поведінки людини. На відміну від інших алгоритмів, під час імітаційного навчання агенти вчаться на основі набору даних, наданих експертом. Мета полягає у відтворенні поведінки експерта в подібних або навіть ідентичних ситуаціях. У контексті відеоігор агент навчається не лише з наданого набору даних, але й шляхом безпосереднього наслідування поведінки гравців у реальному часі.

Проте підхід із використанням алгоритмів машинного навчання має низку суттєвих недоліків, які ускладнюють або навіть унеможливлюють їхнє

застосування у реальних проєктах. По-перше, використання таких алгоритмів значно підвищує вимоги до обчислювальних ресурсів пристроїв гравців, що особливо критично в умовах і без того ресурсомістких проєктів. По-друге, реалізація поведінки агентів на основі машинного навчання суттєво обмежує можливості розробників контролювати цю поведінку, яка часто має відповідати певним вимогам, зокрема сюжетним. По-третє, поведінка агентів може стати абсолютно непередбачуваною, що ускладнює як процес розробки гри, так і її проходження гравцем.

Через це системи на основі нейронних мереж та машинного навчання найчастіше застосовуються для симуляцій, навчання й тестування алгоритмів, академічних досліджень або використовуються в інших сферах геймдеву, наприклад під час створення анімацій чи дизану рівнів.

1.3 Виклики для сучасних систем ігрового ШІ

Розвиток відеоігор вимагає постійного вдосконалення систем ігрового штучного інтелекту. Сучасні системи ШІ повинні відповідати високим стандартам як з точки зору ігрового дизайну, так і з урахуванням апаратних можливостей.

З одного боку, від ігрових агентів очікується динамічна, адаптивна та варіативна поведінка, що дозволяє їм реагувати на непередбачувані дії гравців та змінні умови ігрового середовища. Це підвищує вимоги до розробки більш складних і гнучких архітектур ШІ, здатних підтримувати нелінійні сценарії та забезпечувати більш глибоке занурення в ігровий процес. Окремою важливою складовою при розробці ігрових ШІ є контроль поведінки агентів. З точки зору розробки, система ШІ повинна бути достатньо прозорою і керованою, щоб дозволити дизайнерам гнучко налаштовувати поведінку персонажів у межах визначених сценаріїв. Система ШІ повинна бути простою в проектуванні, проте водночас здатною на виконання складної поведінки. Це забезпечує можливість точного налаштування реакцій агентів на різноманітні ігрові події

та забезпечує баланс між непередбачуваністю та контрольованістю їх дій, які необов’язково можуть бути передбічені дизайнером. Відсутність контролю може призвести до небажаних або нелогічних дій персонажів, що негативно вплине на якість гри.

У разі потреби обробки значного обсягу вхідних даних, які визначають одну з численних можливих моделей поведінки, існуючі технології можуть виявитися недостатньо ефективними для виконання поставлених завдань. З розширенням можливостей гри та збільшенням кількості ігрових механік дизайнери стикаються з проблемою неможливості врахувати всі можливі ситуації та винести їх обробку у рамках окремого автомату або дерева поведінки. Це ускладнює моделювання всіх потенційних сценаріїв, які можуть виникати під час ігрового процесу. Таким чином, стає очевидною потреба у впровадженні більш сучасних і гнучких технологій для виконнання процесу прийняття рішень, здатних адаптуватися до динамічних умов і забезпечувати високу ефективність у складних системах.

З іншого боку, такі системи мають бути оптимізованими з точки зору обчислювальних ресурсів, особливо враховуючи сучасні вимоги до графіки та продуктивності ігор на різних платформах. Баланс між складністю поведінкових моделей та ефективністю їх виконання є ключовим викликом для розробників, оскільки невірно спроектовані системи ШІ можуть критично выдобразитися на продуктивності гри. Тому створення ефективних алгоритмів, здатних до масштабування та адаптації, є важливим завданням для розробки сучасних ігрових ШІ систем.

РОЗДІЛ 2
ПРОЕКТУВАННЯ СИСТЕМИ ПРИЙНЯТТЯ РІШЕННЬ

2.1 Вимоги до системи

Першочерговим етапом проектування та розробки системи є визначення її вимог. По-перше, система має відповідати сучасним стандартам ігрового штучного інтелекту, включаючи вимоги до динамічності, адаптивності та реактивності. Це передбачає здатність агентів адаптувати свою поведінку залежно від змін та подій у ігровому середовищі.

Основна ідея та головна вимога полягає в тому, щоб агенти поводилися так, ніби вони контролюються реальними людьми, забезпечуючи при цьому максимально природну та реалістичну поведінку. Для гравця це має створювати відчуття, що він протистоїть справжнім суперникам.

Реалізація поведінки персонажів не повинна базуватися на традиційних методах, таких як жорсткі зв’язки між станами, перевірка умов переходу між ними або використання таймерів для симуляції поведінки. Натомість, агенти мають аналізувати навколишнє середовище, дії супротивників та власні потреби для вибору найбільш релевантної дії в кожен конкретний момент часу. Система має бути ефективною та оптимізованою до використання у проектах, які передбачають значну кількість агентів, кожен із яких виконує унікальні задачі в режимі реального часу. Важливо забезпечити продуктивність та оптимізацію, щоб система могла функціонувати без втрати швидкодії навіть в умовах підвищеного навантаження.

Важливим аспектом при розробці системи є її гнучкість і масштабованість. Вона повинна надавати розробникам можливість використовувати її в різних типах проєктів, незалежно від жанру, а також забезпечувати легку адаптацію під різні типи ігрових світів, механік та умов. Така універсальність дозволяє інтегрувати систему як у шутери, так і в

рольові, стратегічні ігри або симулятори без необхідності внесення значних змін у її архітектуру.

З огляду на те, що розробка ігор зазвичай триває 3-4 роки, повторне використання існуючих рішень та модулів є класичною моделлю розробки, що дозволяє зменшити витрати часу та ресурсів. Система повинна мати архітектуру, яка забезпечує можливість подальшого розширення та розвитку функціональності, щоб її можна було доповнювати новими модулями та адаптувати до зростаючих вимог індустрії.

Крім того, для підвищення ефективності та зручності розробки, система має бути інтуїтивно зрозумілою та доступною як для розробників, так і для ігрових дизайнерів. Це особливо важливо, оскільки ігрові дизайнери зазвичай мають лише поверхневий досвід у програмуванні й займаються переважно створенням та тестуванням прототипів ігрових механік. Такий підхід дозволить їм легко налаштовувати поведінку агентів без необхідності глибоких знань у кодуванні.

Важливим фактором під час реалізації системи є забезпечення модульності архітектури. Беручи до уваги такі рішення, як дерева поведінки чи кінцеві автомати, збільшення кількості зв’язків між елементами системи призводить до пропорційного росту помилок під час розробки проекту, які часто стають критичними. Зменшення кількості зв’язків та залежностей між елементами системи значно спростить розвиток проекту, адже будь які зміни потребуватимуть редагування лише окремих модулів, без необхідності перебудови чи редагування всієї системи.

Хоча поведінка агентів і має бути динамічною та реактивною, водночас вона повинна залишатися передбачуваною та керованою. Для досягнення цього система повинна мати функціонал для ручної активації та деактивації, та має бути обладнана зручним та потужним відладником, який дозволить розробникам і дизайнерам швидко виявляти й коригувати можливі помилки чи невідповідності в поведінці агентів. Відладник забезпечує прозорість у процесі налаштування, надаючи зворотний зв'язок у реальному часі, що дає

змогу ретельно перевіряти й адаптувати дії агентів відповідно до вимог геймдизайну. Це допоможе забезпечити контроль над поведінкою агентів, підтримуючи баланс між адаптивністю системи та її відповідністю чітким критеріям ігрового дизайну.

2.2 Проектування архітектури

Проектування архітектури системи прийняття рішенн полягає в розробці гнучкої та універсальної структури, яка дозволить розробникам використовувати її в проектах різних жанрів та маштабів. Сама система повинна мати мінімальні або повністю відсунті зв’язки та переходи між задачами, що гарантувало б запобігання виникнення помилок (багів) та спрощення маштабування всього проекту, з мінімальнимизмінами в архітерктури поведінки персонажів.

Основними та ключовими компонентами системи є **задача**, **фактори оцінювання** та **оцінювач**. Кожна задача реалізовує логіку певної дії агента, яка має бути виконана, а також перелік критеріїв, які формують оцінку її корисності. В свою чергу, в кожен момент часу з певною частотою, оцінювач в режимі реального часу на основі всіх факторів кожної задачі зі списку, визначає та запускає виконання найдоцільнішої з них.

Оскільки система повинна бути керованою, необхідно передбачити функціонал для активації та деактивації всіх її компонентів, а також можливість контролю виконання поточних завдань. Це дасть змогу не лише забезпечити контроль за станом системи, а й дозволити розробникам та дизайнерам вчасно коригувати поведінку агентів залежно від зміни умов гри. Це важлива умова, коли потрібно виконати конкретну задачу в певний момент часу, обмежуючи агента в самостійному прийнятті рішень. Таке обмеження може бути необхідним коли поведінка агента повинна відповідати встановленому сюжету або після активації певних тригерів, що вимагають синхронізації з іншими подіями чи сценарієм гри.

2.2.1. Задача. Кожна задача повинна мати чітко визначені точки входу та виходу. Реалізація логіки кожної задачі покладається безпосередньо на розробника гри, який задає умови її виконання та визначає момент завершення. Це забезпечує універсальність кожної задачі незалежно від типу проекту.

Корисність задачі може визначатися двома основними способами: шляхом встановлення константного значення, яке залишається незмінним незалежно від контексту, або ж через список лічильників, кожен з яких повертає оцінку в межах від 0 до 1, враховуючи певний критерій.

Для обчислення загального показника корисності завдання у випадку використання списку лічильників слід визначити метод агрегування значень з кожного лічильника. Агрегування даних забезпечує отримання узагальненого значення шляхом комбінування значень кожного атрибута. До основних методів агрегування належать сума, середнє значення, мінімальне та максимальне значення.

Однак, з метою підвищення логічної гнучкості кожної дії, значення критеріїв якої представлені числовим виразом, а не чіткими булевими значеннями, замість єдиного методу агрегування для всіх лічильників одночасно було прийнято рішення про застосовування операторів нечіткої логіки, зокрема **ТА**, **АБО** та **НЕ**, для кожного лічильника окремо.

Оператори нечіткої логіки опрацьовують значення параметрів належності подібно до операторів булевої алгебри. Класична інтерпретація операторів нечіткої логіки подається у формі нечіткої імплікації Заде, де **ТА** = min(x, y), **АБО** = max(x,y) та **НЕ =** (1-x). Проте такий підхід є лише одним з можливих варіантів. Іншим варіантом інтерпретації операторів нечіткої логіки може буди поданий у вигляді, де **ТА** = x*y, **АБО** = (x+y)-x*y. Такий підхід, у порівнянні з класичною реалізацією, дає можливість для більш плавної зміни результуючих значеннь.

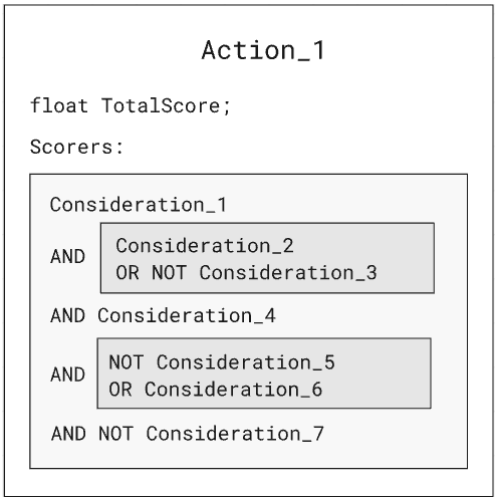

Рисунок 2.1 – Приклад структурної схеми задачі

Однак постійний вибір агентом лише найкращої задачі може надати його поведінці занадто роботизованого характеру, оскільки лише одна певна дія завжди обиратиметься у визначеній ситуації. Частково цю проблему можна вирішити збільшенням кількості параметрів для задачі, що охоплюватимуть ширший спектр факторів контексту для прийняття рішення. Проте для вдосконалення створимо додаткові рішення.

По-перше, варто запровадити можливість **паралельного** виконання кількох задач одночасно, що додасть динаміки до дій агента. Проте, необхідно врахувати певні обмеження, наприклад, чи дозволено виконувати конкретну задачу паралельно з іншими, а також визначити винятки — тобто, вказати, з якими саме задачами паралельне виконання не допускається. Такий підхід дозволить збалансувати гнучкість і реалістичність поведінки агента, запобігаючи конфліктам між взаємовиключними діями.

Іншим рішенням може бути введення **тайм-ауту** після кожного завершення виконання задачі, що забезпечує затримку її доступності упродовж певного часу для повторного вибору з загального списку. Таке рішення гарантує, що агент не зможе відразу повторно обрати ту ж дію, збільшуючи можливість вибору інших, менш очевидних задач.

Окремо слід звернути увагу на проблему інерції, яка може виникнути, коли **коефіцієнти корисності** кількох різних задач мають однакові значення.

У випадку прийняття рішеннь з певною встановленою частотою, це може призвести до нестабільності поведінки агента через циклічне перемикання виконуваних задач.

Частково це питання можна вирішити запровадженням **пріорітету** або **ваги** кожної з задач. Подібне рішення визначатиме важливість кожної дії в загальному механізмі вибору, дозволяючи агенту враховувати не тільки корисність дій, але й їхню відносну важливість у поточному контексті. Агент буде утримувати обрану дію активною до моменту, поки не з'явиться більш краща задача. Іншим рішенням може стати блокування обрахунку корисності всіх задач на певний період, або до моменту. доки активна дія не завершить своє виконання.

Окрім розглянутих варіантів додатково можна впровадити систему модифікації загального показника корисності задачі за допомогою використання спеціальних **кривих**. Такий метод забезпечить вибір оптимальної задачі при схожому наборі вхідних даних, гарантуючи вибір найбільш доцільної дії, а також виконає роль вагів, вказуючи при якому показнику корисності дія стає найбільш важливою, а при якому задачу можна проігнорувати. Цей підхід допоможе не лише уникнути неоднозначності під час прийняття рішення, а й дозволить розбити одну універсальну задачу на дрібніші та конкретніші дії.

Використання різних підходів до модифікації оцінки дозволить не лише конкретизувати важливість кожної з задач, а й надасть змогу моделювати характер поведінки окремих агентів. Зокрема, за їх допомогою можна налаштувати більш агресивний і ризикований стиль дій для агентів, які виконують роль професійних солдатів, або, навпаки, сформувати обережну і стриману поведінку для цивільних персонажів чи агентів з нижчим рівнем тактичної підготовки.

$$U = \max\left(\min\left(\left(1 - \frac{\text{health}-\text{minDmg}}{\text{maxDmg}-\text{minDmg}}\right) * (1 - \text{a}) + \text{a}, 1\right), 0\right) \qquad (2.1)$$

Рівняння 2.1 показує приклад залежності поточного показника здоров’я агента до шкоди, якої йому може задати противник, формуючи таким чином показник імовірності проведення атаки. Змінна «a» в рівнянні вказує на показник ризикованості, встановленому індивідуально для кожного агента, за допомогою якого формується нижча межа можливості виконання атаки. В даному прикладі показник збільшується лінійно зі зменшенням показника здоров’я агента, що робить його більш агресивним та схильним до ризику під час загрози смерті. Інвертований показник даної залежності навпаки, робить агента більш обережним та схильном до проведення атаки лише у випадку достатньої кількості здоров’я чи низької загрози життю.

2.2.3. Пул задач. Втім, можуть ставатися випадки, коли вибір певного масиву дій може бути недоречним, навіть не зважаючи на те, що в контексті ці дії мають найвищі показники корисності. Наприклад, серед задач агента на вибір є такі, як патрулювання, розмова з друзями, відпочинок біля вогнища чи інші. Однак, якщо агент був атакований, він не повинен навіть проводити розрахунок корисності подібних задач, віддаючи перевагу задачам, що забезпечують захист або контратаку.

Для вирішення цієї проблеми в більш широкому контексті можна застосувати використання **пулу задач**. Усі можливі дії класифікуються відповідно до їхнього характеру за сегментами, кожному з яких присвоюється вага або пріоритет. Пул з найвищим пріоритетом обробляється в першу чергу.

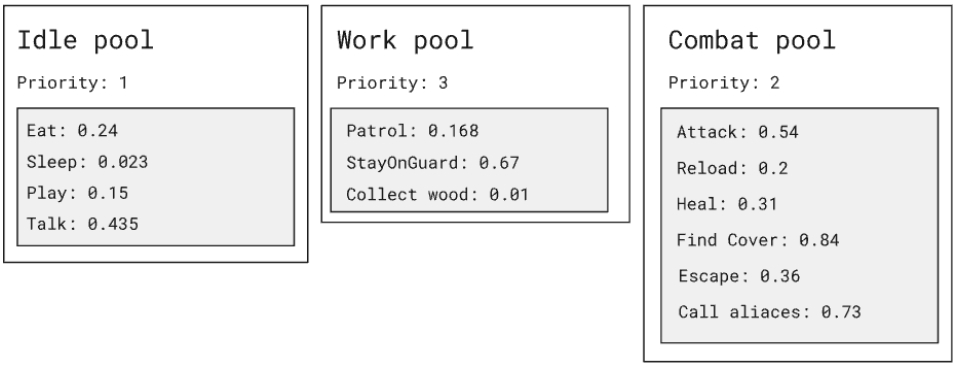

Рисунок 2.2 – Схематичне зображення пулів задач

Якщо в пулі з найвищим пріоритетом є **валідні** задачі (тобто з оцінкою корисності, що перевищує нуль), агент обирає найбільш корисну задачу з цього сегмента та не розглядає інші пули. Лише у випадку, коли в пулі відсутні відповідні задачі, агент переходить до обробки наступного за пріоритетом сегмента.

Окрім того, для збереження адаптивності системи доцільно впровадити можливість динамічної зміни пріоритетів пулів. Такий підхід є більш ефективним порівняно з використанням статичних значень пріоритетів, оскільки дозволяє агенту коригувати їх значення залежно від зміни зовнішніх умов чи внутрішнього стану.

Фактори прийняття рішенн. Рішення щодо вибору певної дії в режимі реального часу приймається на основі оцінки ряду **критеріїв**, які формують загальний показник корисності задачі.

Доцільно реалізувати це у вигляді лічильників, де кожен лічильник аналізує та оброблює певний набір вхідних даних, повертаючи результат у вигляді оцінки окремого фактору. Оновлення значень критеріїв для кожного лічильника здійснюється безпосередньо розробником, який викликає функціонал для оновлення даних відповідно до визначеної логіки системи ШІ конкретного проекту.

Під час обчислення загального показника корисності важливо забезпечити узгодженість, оскільки вхідні дані можуть значно відрізнятися за характером та значенням. Отже, сирі дані для кожного лічильника повинні бути оброблені та приведені до стандартизованого формату, однакового для всіх лічильників, незалежно від їх типу та вхідних значень. Для цього слід здійснити нормалізацію вхідних даних, перевівши їх у вигляд числа з плаваючою точкою в інтервалі від 0 до 1 включно.

Серед методів нормалізації даних найпоширенішими є використання математичних залежностей, які маштабують вхідне значення до діапазону

[0,1]. Найпростішим з них є лінійна залежність, або представлення значення у відсотковій формі:

$$U = \frac{x}{m} \tag{2.2}$$

де x – поточне значення, m – максимальне значення.

Степенева залежність є однією з форм параболічної кривої, яка забезпечує більш згладжене значення нормалізованих даних порівняно з лінійною пропорцією. Вона виражається формулою:

$$U = \left(\frac{x}{m}\right)^k \tag{2.3}$$

де k - ступінь залежності, що визначає рівень згладження результуючої кривої. Зміна значення k дозволяє керувати формою кривої, надаючи більший чи менший вплив вхідним значенням у різних діапазонах.

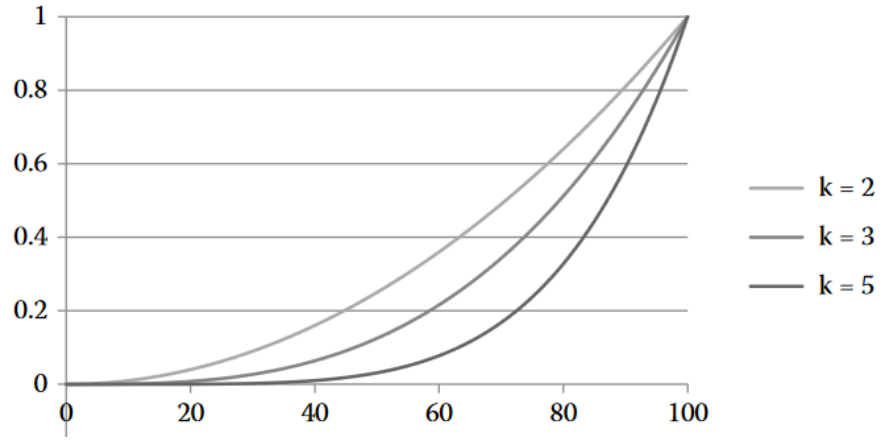

Рисунок 2.3 – Графік степеневої функції

Логістична функція є ще одним варіантом масштабування даних, що забезпечує контрольованіший вплив на форму кривої. Використання цього підходу дозволяє плавніше коригувати результуюче значення, оскільки крива має характерний S-подібний вигляд, що поступово наближається до верхньої та нижньої межі діапазону. Такий підхід особливо корисний у випадках, коли

потрібно зменшити вплив екстремальних значень і зберегти стабільність у центральних діапазонах:

$$U = \frac{1}{1+e^{-x}} \tag{2.3}$$

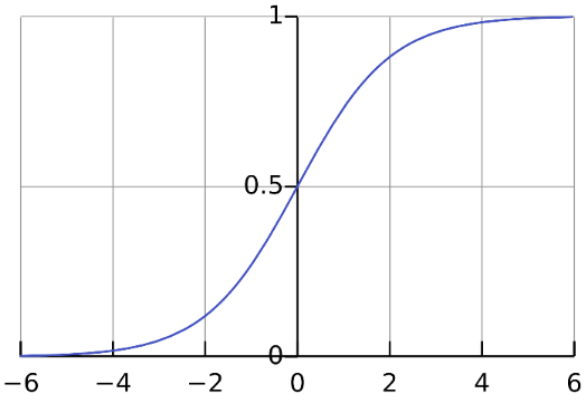

Рисунок 2.4 – Графік логістичної функції

Об’єднуючи нормалізовані значення кожного лічильника, можна будувати практично необмежені комбінації різних оцінок. Єдиною умовою залишається забезпечення однакового формату виведення кожного критерію, що гарантує узгодженість результатів. Такий підхід надає розробникам можливість легко додавати до дії нові лічильники чи видаляти існуючі без необхідності глобальної перебудови всієї системи. Крім того, ізоляція критеріїв один від одного дозволяє повторно використовувати існуючі лічильники між різними задачами, комбінуючи їх для створення унікальних факторів при прийнятті рішень.

Доречно також додати ваги для кожного окремого критерію, що дозволить врахувати їхню відносну важливість у кожній із ситуацій. Таке рішення дозволить точніше налаштувати вплив кожного фактора на підсумкове значення корисності дії, що використовується для прийняття рішень, надаючи додаткову гнучкість у балансуванні значень корисності залежно від контексту.

Оцінювач. Оцінювач є основним компонентом у системі прийняття рішень, інтегруючи всі її елементи в єдину структуру контролю та обробки. Як **центральний вузол**, оцінювач не лише проводить розрахунок корисності можливих задач, а й здійснює комплексне керування як загальною системою, так і її окремими елементами. Це спрощує процес моніторингу та управління процесом прийняття рішень агентом, а також сприяє оптимізації архітектури системи. Така структура дозволяє зберігати унікальність даних для кожного елемента, уникаючи їх дублювання та підвищуючи ефективність використання системних ресурсів.

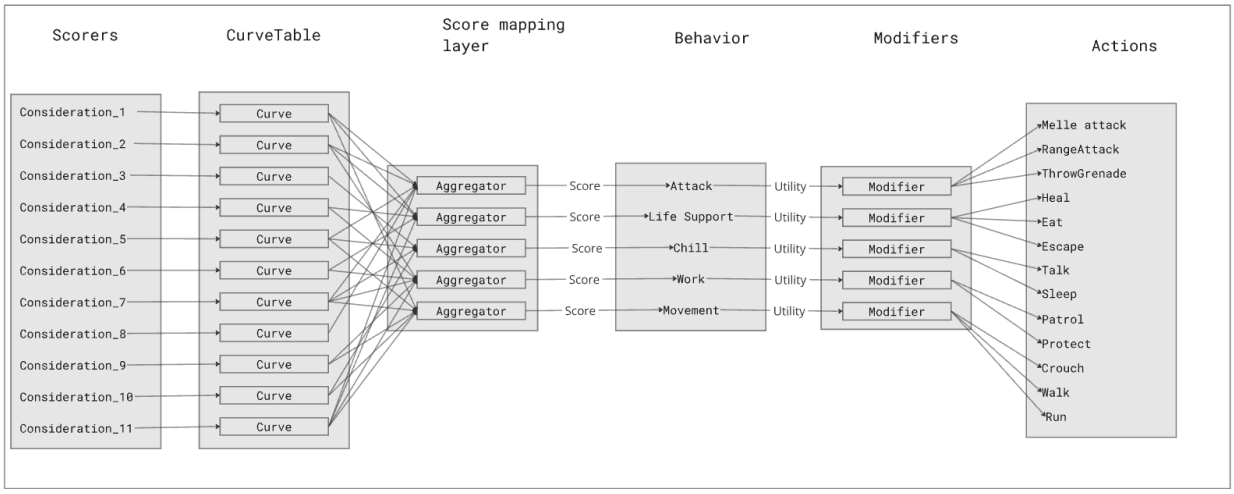

Рисунок 2.5 – Структурна схема процесу прийняття рішенн

Оцінювач, виконуючи роль центрального вузла системи, має повний доступ до всіх її складових, зокрема задач, лічильників та їх відповідних кривих. Це забезпечує можливість як злагодженої внутрішньої взаємодії елементів, так і контроль процесом прийняття рішенн для розробника системи ШІ.

Насамперед важливо передбачити можливість активації та деактивації системи прийняття рішень. Це дозволить оптимізувати використання ресурсів, забезпечуючи активність системи лише тоді, коли вона справді необхідна. Крім того, така функціональність надасть розробнику можливість у керуванні агентом, дозволяючи блокувати самостійне прийняття рішень у випадках,

коли повний контроль над поведінкою персонажа. Це може бути корисним для сценаріїв, де важливо, щоб агент виконував конкретні завдання в певні моменти гри, слідуючи визначеним вказівкам, без можливості ухилення від заданих цілей.

Також доцільно надати розробникам можливість встановлення та налаштування частоти оновлення даних та оцінки корисності задач. Це рішення допоможе встановити оптимальний баланс між ефективністю системи, швидкістю реакції агента та споживанням ресурсів. Завдяки цьому агент може адаптувати свою поведінку під різні умови: від високої інтенсивності та динамічних подій до менш активної та спокійної поведінки, таким чином підвищуючи ефективність роботи системи.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ ПРИЙНЯТТЯ РІШЕННЬ

3.1. Розробка прототипу системи

Перш ніж переходити до розробки системи, важливо створити прототип, який слугуватиме основою для тестування та перевірки концептуальної архітектури проекту. Такий підхід дозволяє на ранньому етапі виявити можливі недоліки у дизайні та функціоналі системи. У нашому випадку прототип буде включати декілька умовних задач і факторів, які впливають на процес прийняття рішень.

Для спрощення тестування прототипу, процес обчислення балів для кожного окремого лічильника буде виключено. Замість цього їм призначатимуться фіксовані значення вручну, оскільки ми знаємо напевне, який результат хочемо отримати при заданих показниках вхідних даних. Це дозволить зосередитися на алгоритмі підрахунку загального показника корисності кожної змодельованої задачі, перевіряючи коректність логіки прийняття рішень у спрощеному вигляді.

З використанням мови програмування Python, для тестового прикладу сформуємо список факторів, що будуть впливати на загальний показник корисності кожної дії, а також складемо список уявних задач, доступних для виконання агентом.

Змінюючи значення цих критеріїв, змоделюємо на їх основі кілька варіацій сценаріїв, які будуть імітувати ситуації, що виникають у ігровому світі (рис. 3.1). Це дозволить порівняти отримані результати, що спростить виявлення можливих недоліків та перевірку коректності роботи системи.

```
scenarios = [
    {"healthProximity": 1, "hungerProximity": 1.0, "seeEnemy": 0,
     "combat": 0, "enemiesProximity": 0.0, "distanceToEnemy": 1.0,
     "distanceToCover": 0.0, "ammoProximity": 1},

    {"healthProximity": 0.73, "hungerProximity": 0.813, "seeEnemy": 1,
     "combat": 1, "enemiesProximity": 0.5, "distanceToEnemy": 0.3,
     "distanceToCover": 0.234, "ammoProximity": 0.5},

    {"healthProximity": 0.15, "hungerProximity": 0.84, "seeEnemy": 0,
     "combat": 1, "enemiesProximity": 0.6, "distanceToEnemy": 1.0,
     "distanceToCover": 0.0, "ammoProximity": 0.78},

    {"healthProximity": 0.9, "hungerProximity": 0.42, "seeEnemy": 1,
     "combat": 1, "enemiesProximity": 0.1, "distanceToEnemy": 0.69,
     "distanceToCover": 0.21, "ammoProximity": 0.9},
]
```

Рисунок 3.1 – Сценарії можливих ситуацій

Кожна із змоделюваних задач повинна отримувати перелік необхідних їй факторів (рис. 3.2). Окрім того, необхідно подбати про правильний процесс підрахунку загального показника корисності. Оскільки для цього процесу було обрано використання методики елементів нечіткої логіки, до кожного параметра оцінки необхідно застосувати власний оператор окремо, який буде впливати на загальний показник. Також необхідно додати показник пріорітетності для кожного завдання, який буде вказувати на валжившість конкретної задачі під час прийняття рішеня

```
actions = [
    Action('Idle', score=0.1),  # constant score
    Action('MeleeAttack',
           # If healthProximity AND seeEnemy AND NOT enemiesProximity AND NOT distanceToEnemy
           score=healthProximity * seeEnemy * (1 - enemiesProximity) * (1 - distanceToEnemy)),
    Action('RangedAttack',
           # If healthProximity AND seeEnemy AND distanceToEnemy AND ammoProximity AND NOT enemiesProximity
           score=healthProximity * seeEnemy * distanceToEnemy * ammoProximity * (1 - enemiesProximity)),
    Action('MoveToCover', 2,
           # If NOT healthProximity OR NOT ammoProximity OR enemiesProximity AND NOT distanceToEnemy AND NOT distanceToCover
           score=((1 - healthProximity) + (1 - ammoProximity) + enemiesProximity) * (1 - distanceToEnemy) * (1 - distanceToCover)),
    Action('Eat',
           #If NOT hungerProximity AND (distanceToEnemy OR NOT combat)
           score=(1 - hungerProximity) * (distanceToEnemy + (1 - combat))),
    Action('Heal', 2,
           # If NOT healthProximity AND (distanceToEnemy OR NOT combat)
           score=(1 - healthProximity) * (distanceToEnemy + (1 - combat))),
    Action('Patrol',
           # If NOT combat AND NOT seeEnemy AND healthProximity AND hungerProximity
           score=(1 - combat) * (1 - seeEnemy) * healthProximity * hungerProximity),
    Action('Reload',
           # If NOT combat OR distanceToEnemy AND NOT ammoProximity
           score=((1 - combat) + distanceToEnemy) * (1 - ammoProximity))
]
```

Рисунок 3.2 – Список тестових задач агента

Бібліотеки мови Python також дозволяють проводити візуалізацію даних, у вигляді графічного представлення, що також спрощує аналіз отриманих даних (рис. 3.3).

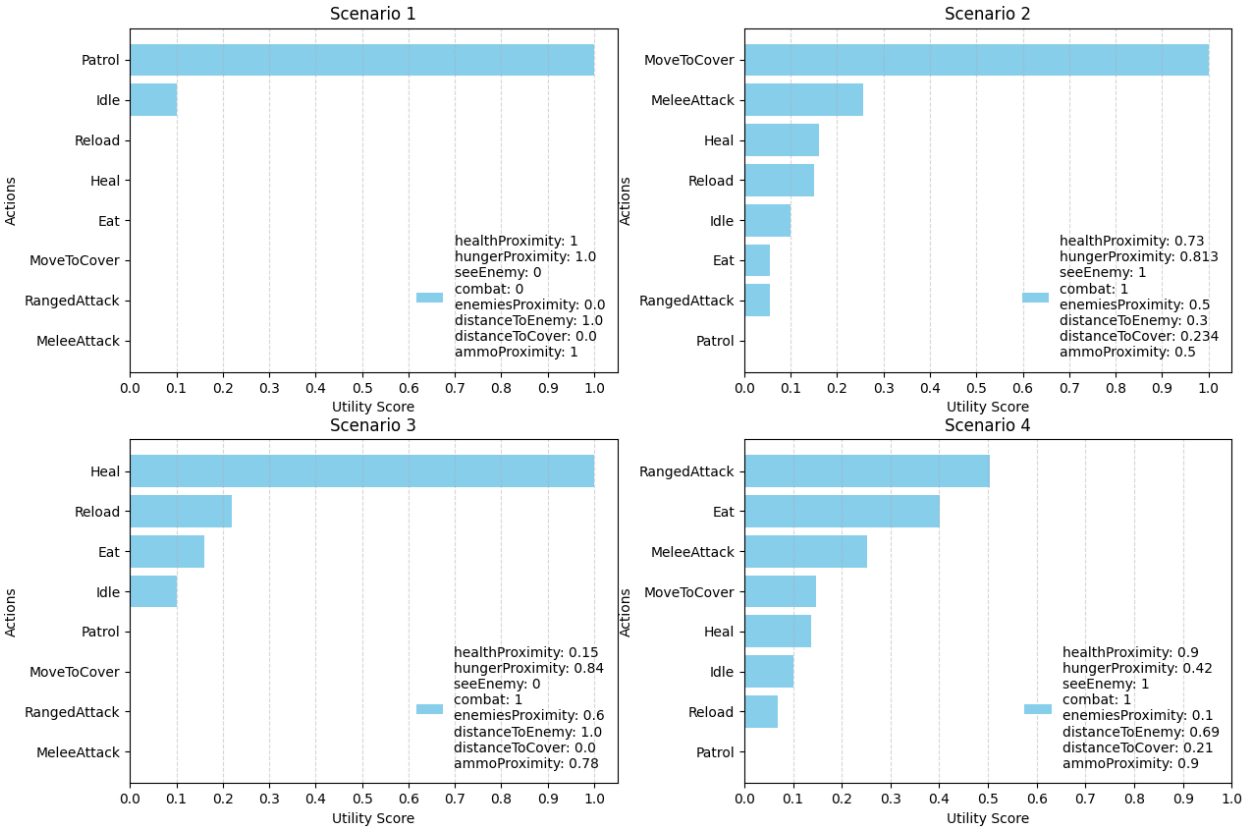

Рисунок 3.3 – Результати проведення тестування прототипу системи

В результаті проведеного тестування отримуємо графіки, які демонструють, що незалежно від обраного сценарію та заданих вхідних параметрів, завжди обирається найбільш оптимальна дія з урахуванням її пріоритету.

Крім того, тестування прототипу показало, що редагування списку доступних дій або окремих задач не впливає на загальний процес прийняття рішень, що робить систему гнучкою та досить простою, мінімізуючи з’вязки між окремими діями. Обраний метод підрахунку загального показника корисності також працює коректно, забезпечуючи логічні результати.

Таким чином, можна зробити висновок, що розроблений дизайн архітектури проекту відповідає основним вимогам, а створений прототип

демонструє задовільні результати, що дозволяє перейти до етапу реалізації проекту.

3.1. Реалізація системи

3.1.1. Створення проекту. Реалізацію системи представлено у формі плагіна для ігрового рушія «Unreal Engine 5». Завдяки модульній архітектурі рушія, забезпечується гнучке налаштування інструментарію відповідно до індивідуальних потреб користувача. Такий підхід мінімізує необхідність у внесенні змін до базового коду рушія, сприяючи збереженню його цілісності. Крім того, модульність дозволяє підключати лише необхідні компоненти для виконання конкретних завдань, залишаючи неактивними всі інші, що підвищує зручність ефективність роботи розробників.

Рушій забезпечує не лише можливість для інтеграції та використання офіційних модулів, але й надає широкий список шаблонів для розробки власних плагінів, що розширює функціональність і адаптивність платформи до специфічних вимог користувача.

Перш за все, за допомогою одного з представлених шаблонів необхідно створити власний модуль та зареєструвати його для використання у власній копії рушія (рис. 3.4).

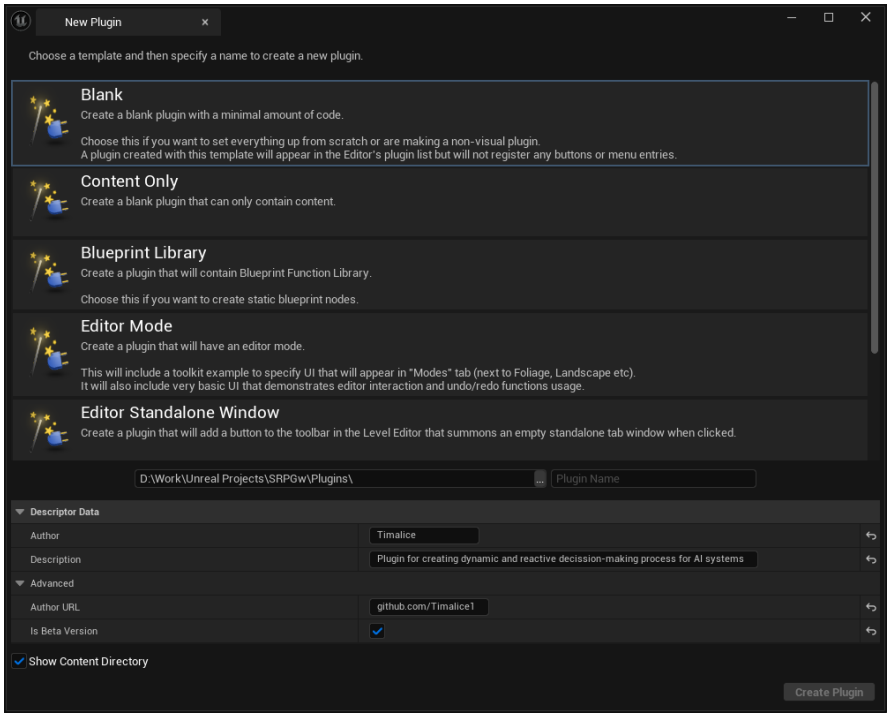

Рисунок 3.4 – Вікно створення нового плагіну

У результаті рушій автоматично генерує новий порожній модуль з необхідними базовими компонентами на основі мов програмування C++ та C#, який за замовчуванням інтегрується у поточний проєкт і готовий до використання (рис. 3.5).

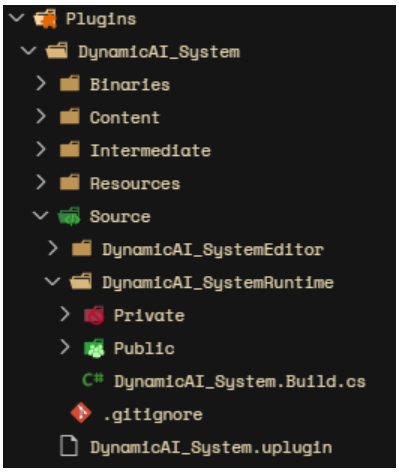

Рисунок 3.5 – Базова структура модуля

Для запобігання випадковому або навмисному редагуванню вихідного коду модуля кінцевими користувачами, система за замовчуванням розділена на публічну та приватну частини. Публічна частина доступна для

використання та перевизначення у дочірніх проєктах, тоді як у приватній частині здійснюється безпосередня реалізація функціоналу системи, забезпечуючи її цілісність і захищеність.

3.1.2. Реалізація базової архітектури проекту. Згідно з розробленим дизайном архітектури проєкту, ключовими компонентами системи є задача, лічильник та оцінювач. У процесі реалізації важливо дотримуватись принципів об’єктно-орієнтованого програмування (ООП), зокрема принципів SOLID, що забезпечують модульність, перевикористання компонентів і гнучкість архітектури а також сприяють зниженню зв’язності між елементами системи та спрощує подальше розширення або модифікацію функціоналу.

Оскільки Unreal Engine підтримує розробку проєктів як на основній мові програмування C++, так і за допомогою візуальної системи скриптингу Blueprints, необхідно забезпечити сумісність системи з обома підходами. Це дозволить зробити використання системи більш доступним як для професійних програмістів, так і спростить процес швидкого прототипування ігрових механік дизайнерами у зрозумілому та зручному середовищі.

Система рефлексії рушія Unreal Engine забезпечує інтеграцію коду з редактором Unreal Editor за допомогою спеціальних макросів, таких як UPROPERTY, UFUNCTION, UCLASS, USTRUCT, тощо (рис. 3.6). Ці макроси не лише відкривають доступ до створених елементів у редакторі, але й забезпечують автоматичну синхронізацію між кодом і візуальним інтерфейсом. Крім того, вони використовують систему метаданих, яка дозволяє гнучко налаштовувати доступність і поведінку параметрів, включаючи їх відображення, редагування та використання у внутрішніх механізмах рушія, зокрема в системі Blueprints.

```
UCLASS(MinimalAPI,
        BlueprintType,
        Blueprintable,
        Abstract,
        DefaultToInstanced,
        EditInlineNew,
        DisplayName = "UtilityAction")
```

Рисунок 3.6 – Макрос UCLASS зі встановленими метаданими

Основою процесу прийняття рішеннь агентом є певні роздуми, чи критерії оцінювання ситуації. Згідно розробленого дизайну архітектури, такі критерії представлені у вигляді певних лічильників, кожен з яких є незалежним та існує окремо від інших. Список таких лічильників у кожній задачі формує основу для розрахунку показника корисності кожної можливої дії.

Кожен лічильник має бути незалежним від інших і здатним до одночасного використання в різних задачах. Згідно з принципами об’єктно-орієнтованого програмування, доцільним є розроблення окремого базового класу, на основі якого можна створювати об'єкти для спільного використання між різними задачами. Однак, для підвищення зручності роботи кінцевого розробника, було прийнято рішення про створення певного проміжного шару, у вигляді окремої структури (рис. 3.7). На основі даних, визначених користувачем у цій структурі, автоматично та непомітно для нього генеруються об’єкти класу лічильника, кожен з яких є унікальним та зберігається в окремому глобальному сховищі, що не тільки відповідає поставленим вимогам, а й спрощує використання та інтеграцію лічильників у системі.

Рисунок 3.7 – Панель налаштуваннь структури даних лічильника

З метою підвищення зручності використання та забезпечення унікальності кожного об’єкта лічильника було вирішено застосувати систему тегів GameplayTags, інтегровану в інструментарій рушія Unreal Engine. Кожен тег є унікальним і зберігається в окремому конфігураційному файлі проєкту, що гарантує їхню ідентичність і контрольованість (рис. 3.8). Крім того, Unreal Engine надає потужний функціонал для редагування та управління тегами, що включає створення, пошук, ієрархічну організацію та валідацію тегів.

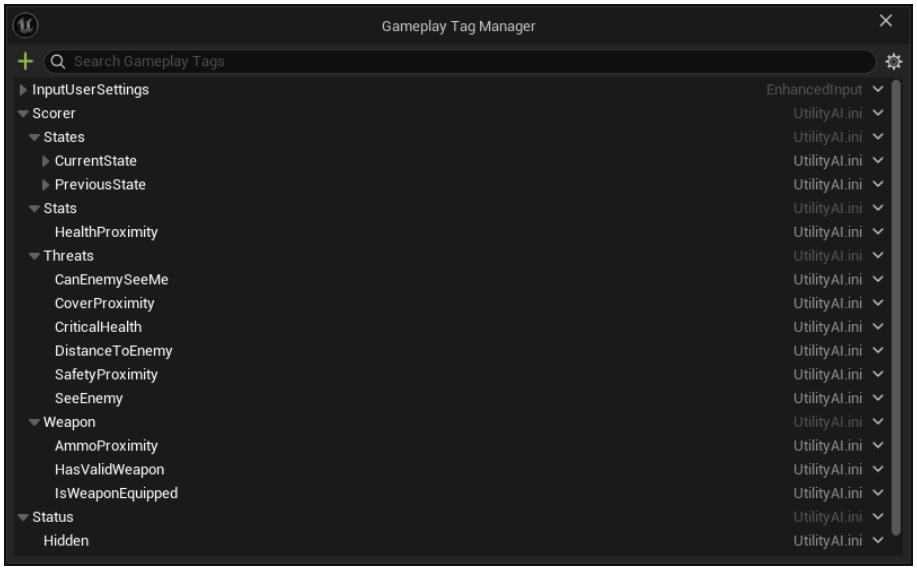

Рисунок 3.8 – Вікно налаштування тегів

Розглянуті під час розробки дизайну проекту приклади методів нормалізації — лише частина можливих підходів. Існує безліч інших способів трансформації даних. Однак, використання лише математичних формул для оцінки корисності задачі на основі певного критерію може бути недостатнім. Часто дизайнери та розробники потребують можливості задавати більш точне й конкретне значення фактору для певного значення вхідного параметру. Наприклад, мати змогу вказати, при якому значені показника здоров’я агента можна вважати високим чи критично низьким, при якому значенні відстані об’єкт вважається далеким чи близьким від агента, чи яку кількість противників можна класифікувати як багато. Оскільки для реалізації системи було прийнято рішення про використання елементів теорії нечітких множин, за допомогою можливостей рушія можна не лише забезпечити нормалізацію

вхідних даних, але й надати розробникам можливість для їх фазифікації, що дозволяє визначати ступінь належності значення вхідного параметра до певних категорій.

Для цього рушій Unreal Engine надає інструменти для створення та редагування власних кривих, а також можливість зберігати їх у вигляді таблиць кривих (Curve Tables) (рис. 3.9). Цей функціонал є ефективним засобом, який дозволяє подолати розрив між даними, визначеними ігровим дизайнером, та алгоритмічним управлінням, реалізованим програмістом. Їх використання дозволяє не лише провести маштабування вхідних даних до спільного вигляду, а й задати конкретні значення показника для різних вхідних параметрів.

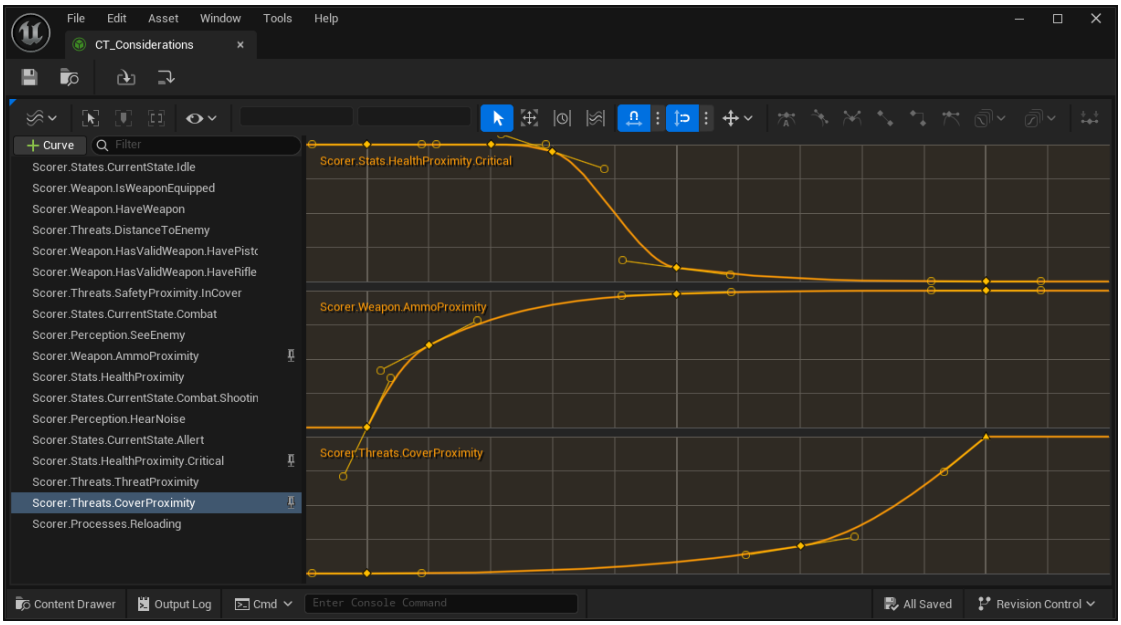

Рисунок 3.9 – Графічний інтерфейс таблиці кривих в Unreal Engine

Оскільки вихідне значення кожного критерію повинно знаходитися у визначених межах, необхідно подбати про те, щоб розробник не перевищив мінімальне та максимальне значення під час налаштування параметрів кривої.

Сукупність створених лічильників утворює набір критеріїв для обчислення показника корисності кожної дії (рис. 3.10). Відповідно до розробленої архітектури, значення цього показника визначається із застосуванням елементів систем нечіткої логіки. Для кожного лічильника

необхідно призначити індивідуальний оператор, який визначатиме спосіб впливу його значення на загальний показник оцінки корисності. Залежно від зазначеного оператора, значення окремого лічильника включається до загального показника шляхом множення або додавання до поточного результату оцінки.

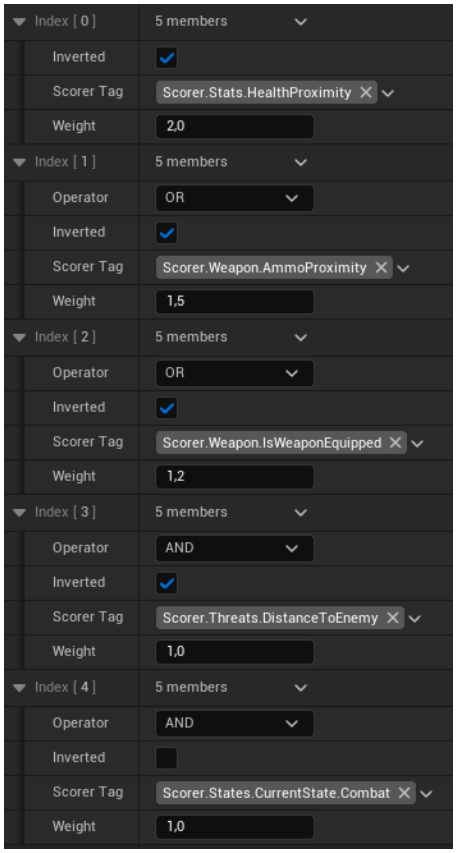

Рисунок 3.10 – Приклад використання списку критеріїв оцінювання

Реалізація логіки виконання кожної задачі покладається безпосередньо на розробника системи ШІ. Наша мета — забезпечити необхідну інфраструктуру: визначити точки входу та виходу (рис. 3.11), надати можливість виконання налаштуваннь задачі, провести її ініціалізацію та запустити процесс виконання задачі.

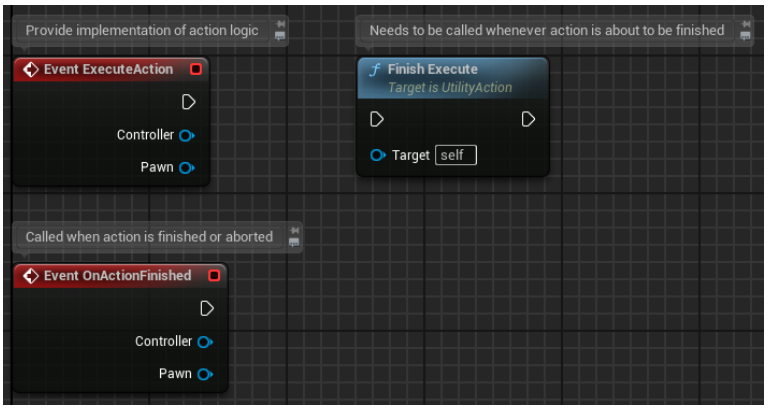

Рисунок 3.11 – Функції початку та завершення виконання логіки задачі

Процес обчислення показника корисності реалізується безпосередньо в класі задачі, базуючись на значеннях лічильників і призначених їм операторах. Для кожної дії передбачена можливість використання модифікаційної кривої (рис. 3.12), що дозволяє уникнути неоднозначності у виборі дій за умов подібних параметрів оцінки. Якщо задачі призначено модифікаційну криву, обчислене базове значення корисності коригується за допомогою зазначеної кривої. Отриманий результат модифікується на основі показника пріоритетності задачі, формуючи остаточну оцінку корисності дії.

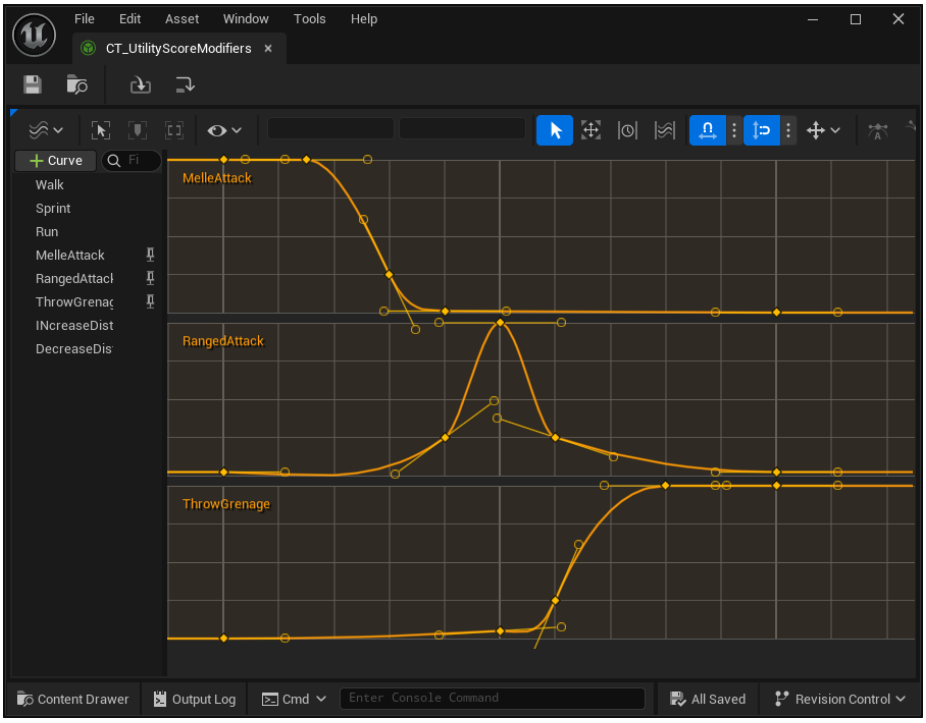

Рисунок 3.12 – Модифікатори оцінки корисності задач

Після завершення виконання процесу задача автоматично переходить у стан блокування, що унеможливлює її повторну оцінку та активацію протягом визначеного розробником періоду. Такий підхід забезпечує, що задача не буде одразу обрана знову, дозволяючи системі звернути увагу на інші, менш ймовірні завдання. Це сприяє різноманітності вибору та, у певній мірі, підвищує реалістичність поведінки агента, створюючи враження продуманих і зважених дій.

Реалізований клас задачі, створений на основі C++, дозволить генерувати об’єкти Class Default Object (CDO) (рис. 3.13) безпосередньо в редакторі Unreal, для проведення подальшого налаштування кожного об’єкту та реалізації логіки за допомогою візуального програмування, а також використання створених об’єктів в поточному проекті.

Рисунок 3.13 – Об’єкти CDO класу Action

Створені об’єкти задач групуються у відповідні пули (рис. 3.14). Кожен пул містить список окремих задач і має власний показник пріоритету. Пули з найвищим значенням пріоритетності обробляються першочергово, і лише в разі відсутності доступних задач у поточному пулі черга переходить до наступного пулу за пріоритетом.

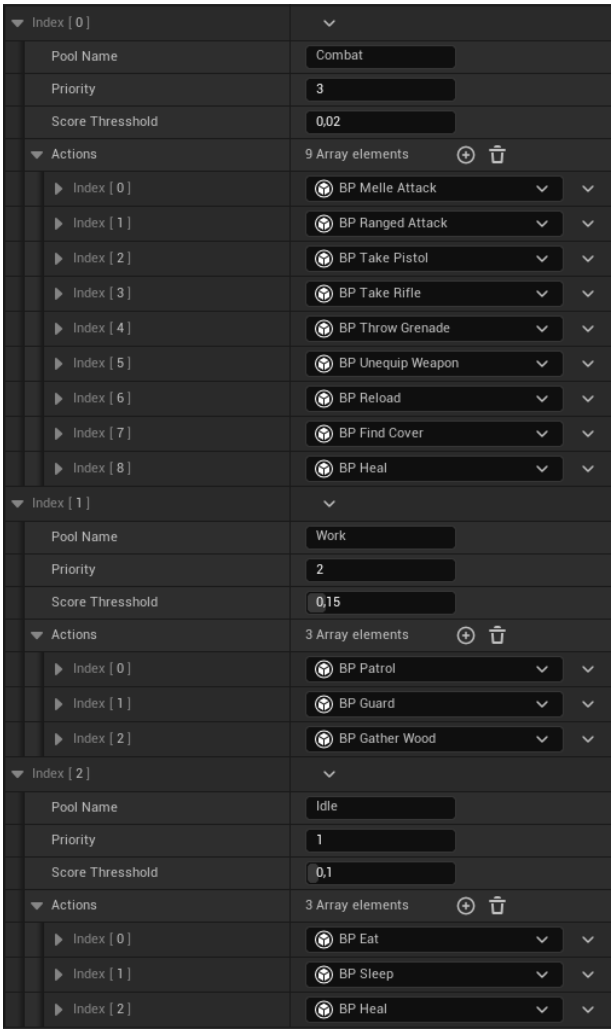

Рисунок 3.14 – Приклад використання пулів задач

Крім того, розробники мають можливість задавати граничні значення корисності для задач у кожному пулі. Це дозволяє автоматично виключати задачі, чия оцінка корисності нижча за вказане порогове значення, забезпечуючи ефективну фільтрацію і оптимізацію вибору дій.

Після створення нового об’єкта класу задачі розробники в окремому редакторі можуть виконати початкове налаштування кожної задачі, задаючи її необхідні параметри за замовчуванням. Однак, у деяких випадках виникає потреба у детальнішому налаштуванні задач з урахуванням їх взаємодії з іншими. Для підвищення зручності процесу надається можливість налаштовувати параметри кожної дії безпосередньо в списку задач пулу, що

забезпечує більш ефективну організацію та спрощує управління поведінкою системи.

Основним компонентом системи є клас оцінювача, який виконує ключові функції: зберігає список можливих завдань, обчислює їхню корисність та слугує центральним вузлом управління системою. З огляду на його значущість, оцінювач реалізовано як дочірній клас від ActorComponent.

Такий підхід, згідно архітектури рушія Unreal, дозволяє використовувати компоненти у вигляді окремих незалежних модулів (рис. 3.15). Це забезпечує можливість інтеграції оцінювача у будь-який тип класу Actor. У даній системі оцінювач закріплено за класом AI Controller, який є основним елементом управління поведінкою агентів.

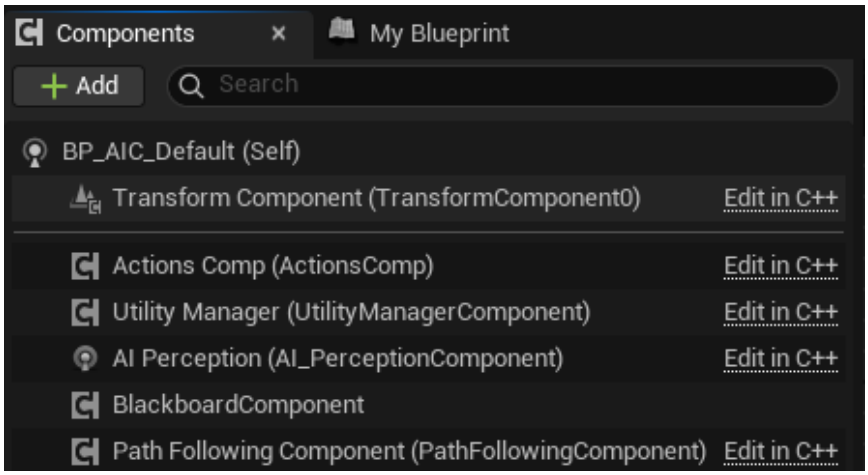

Рисунок 3.15 – Компоненти класу AI Controller

Для кожного агента рушій автоматично створює унікальний CDO класу **Controller**, разом із усіма закріпленими за ним компонентами. Це дозволяє провести налаштування **CDO** одноразово, забезпечуючи при цьому унікальність поведінки кожного агента.

Використання компонентів у вигляді окремих модулів відкриває широкі можливості для проведення менеджменту систем. Перш за все, основною перевагою такого підходу є можливість динамічної активації та деактивації компонентів у будь який момент часу. Зважаючи на відкритість коду рушія,

ми маємо змогу виконати перевизначення функцій активації та деактивації, розширюючи їх базову логіку.

Кожен компонент має можливість реєстрації та відстеження подій, що відбуваються в модулі, за допомогою спеціального механізму — **диспетчера подій** (Event Dispatcher). Цей диспетчер дозволяє у будь-який момент викликати сповіщення про виконання компонентом певної дії.

Окрім цього, користувач може скористатися делегатом (рис. 3.16) для обробки результатів таких сповіщень. Це забезпечує високу гнучкість і дозволяє налаштовувати систему відповідно до потреб, додаючи можливість реагування на події в реальному часі.

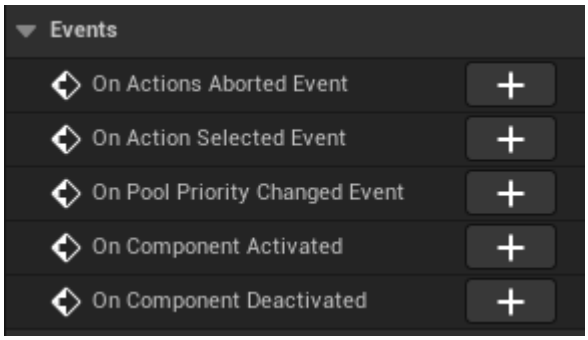

Рисунок 3.16 – Список делегатів компоненту оцінювача

Після активації система оцінювача проходить етап ініціалізації, під час якого створюються та перевіряються всі пули, задачі та їхні критерії оцінювання у вигляді лічильників. Процес ініціалізації включає перевірку кожного нового елемента на унікальність, щоб уникнути дублювання й забезпечити коректну роботу системи. Після завершення цього етапу оцінювач починає працювати з частотою, заданою розробником, оновлюючи значення лічильників і розраховуючи корисність кожної доступної задачі. На основі отриманих результатів система вибирає задачу з найвищим рейтингом, додає її до списку активних процесів і активує процес виконання її внутрішньої логіки.

Перед початком виконання задачі оцінювач здійснює перевірку її сумісності з іншими активними задачами. Зокрема, перевіряється можливість

паралельного виконання з поточними процесами, а також враховуються виключення — процеси, з якими задача не може працювати одночасно. Це забезпечує коректність виконання задач і оптимізацію роботи системи, а також забезпечує підтримання ігрової логіки, реалізованої розробниками та дизайнерами. Після завершення процесу, задача відправляє до оцінювача відповідне сповіщення, і компонент видаляє поточну задачу зі списку активних .

Для реалізації можливості менеджменту системи було розроблено набір функціональних можливостей (рис. 3.17), які дозволяють здійснювати маніпуляції з процесами у реальному часі, включаючи контроль виконання активних задач, управління процесом оновлення даних лічильників, динамічне регулювання пріоритетів як окремих задач, так і пулів.

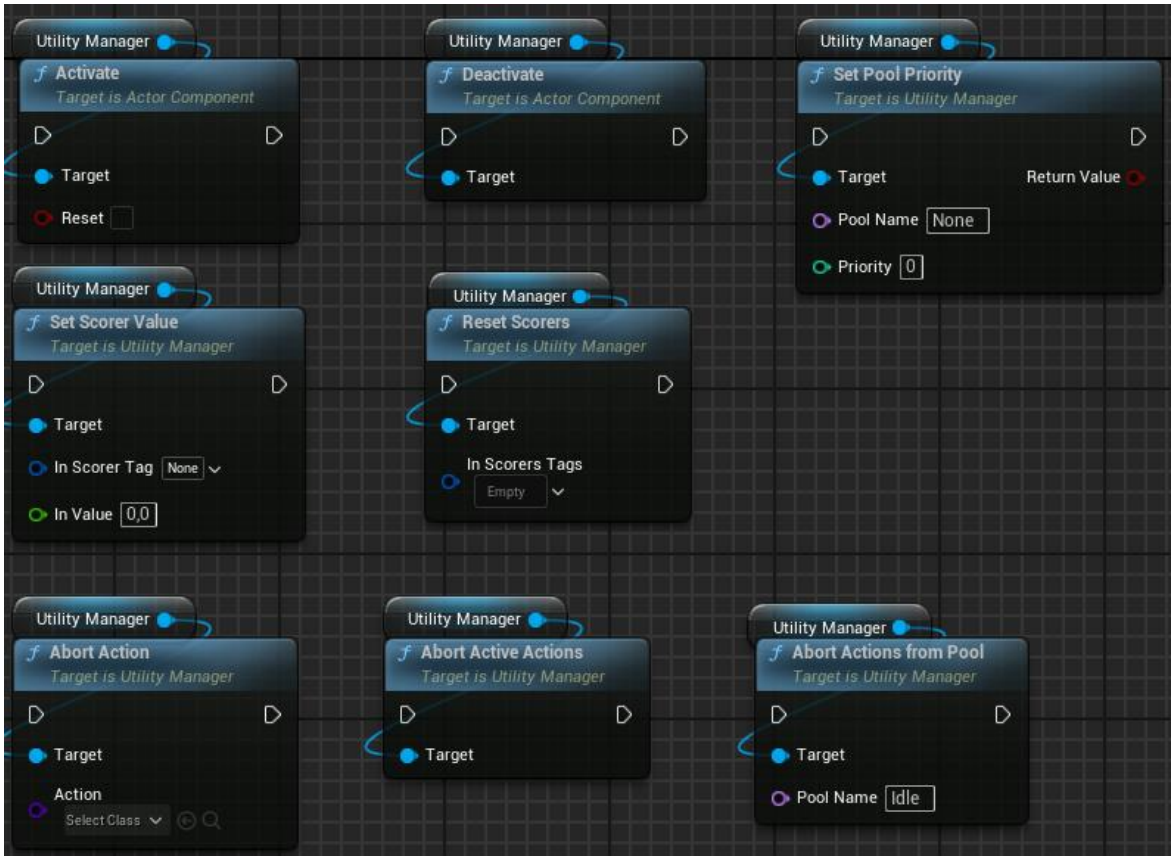

Рисунок 3.17 – Основні функціональні вузли управління менеджером задач

Додатково, у проєкті також розроблено окремий сервісний елемент, який працює незалежно від основних задач та лічильників. Основна мета цього

сервісу — спрощення налаштування системи та виконання певних сервісних процесів у окремому ізольованому середовищі. Сервіс, подібно до менеджера, оновлює дані з частотою, визначеною розробником, але функціонує автономно, розвантажуючи центральний блок керування. Це сприяє гнучкості та зручності використання, полегшуючи розробку й обслуговування системи за рахунок чіткої сегментації функцій.

У результаті роботи отримується блок керування системою прийняття рішень, реалізований як окремий незалежний модуль у складі системи штучного інтелекту (рис. 3.18). Розробник має змогу проведення детального налаштування системи за допомогою наданого списку параметрів, а також функціоналу для управління процессом в реальному часі.

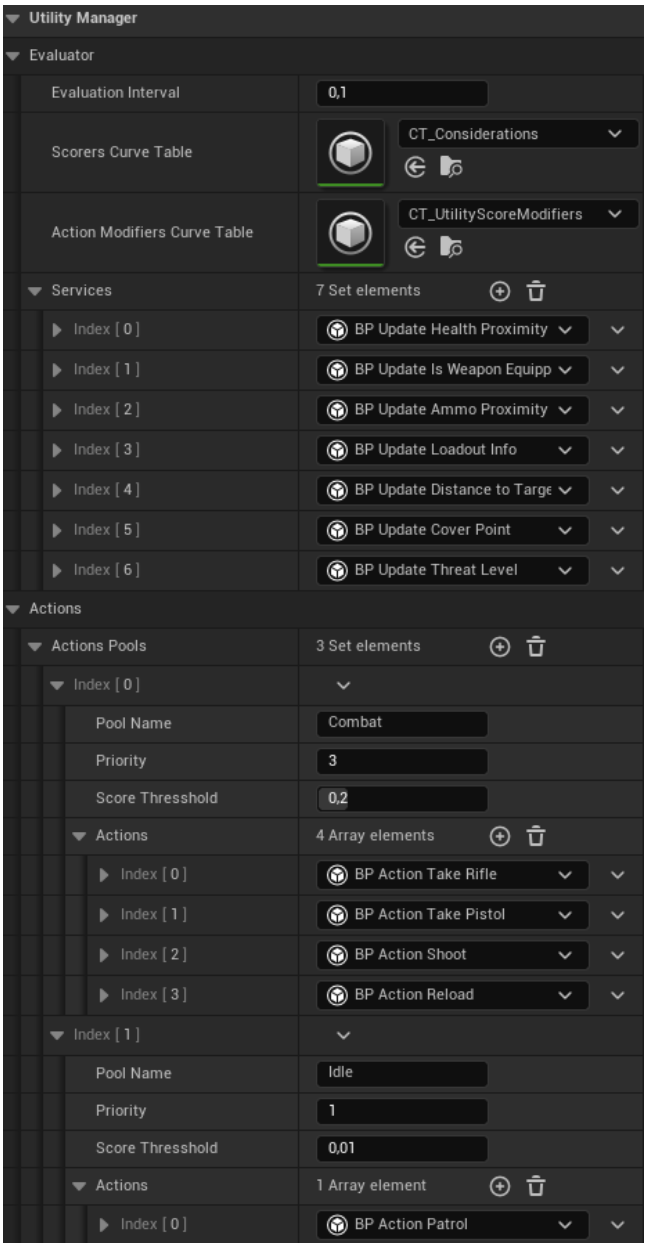

Рисунок 3.18 – Панель налаштуваннь компонента UtilityManager

Відладник. Важливою складовою функціональності будь-якої програми є не лише здатність збирати та аналізувати результати її роботи, але й можливість відслідковувати процеси в режимі реального часу. Сучасні інструменти програмування, як правило, включають інтегровані засоби для налагодження та моніторингу виконання завдань, що значно полегшує роботу розробників.

Через надання агенту можливості самостійного прийняття рішень прозорість роботи системи значно знижується порівняно з більш чітко. визначеною логікою, реалізованою у вигляді поведінкових дерев або кінцевих

автоматів. Це певною мірою ускладнює процес відстеження даних, оскільки поведінка агента розроблена для забезпечення динамічності та миттєвої реакції на змінні умови та вхідні дані.

Проте все ж необхідно надати можливість розробникам відслідковувати та аналізувати як результат роботи системи, так і хід її роботи в реальному часі. Це сприятиме виявленню потенційних проблем, поліпшенню налаштування логіки прийняття рішень та забезпеченню оптимального функціонування системи в умовах реальної гри.

Базовий функціонал рушія Unreal Engine надає розробникам потужний набір інструментів для налагодження, серед яких основне місце займає система текстових логів. Ця система забезпечує ведення детальної документації виконання процесів, дозволяючи збирати інформацію для аналізу у вигляді окремого файлу.

Крім цього, система підтримує відображення даних у режимі реального часу безпосередньо в окремому вікні редактора (рис. 3.19), що робить процес відстеження більш інтерактивним і зручним. Для запису даних розробникам пропонується широкий вибір заготовлених категорій, які поставляються разом із рушієм. Водночас система дозволяє створювати та реєструвати власні категорії, що надає гнучкість у налаштуванні логів під конкретні потреби проєкту.

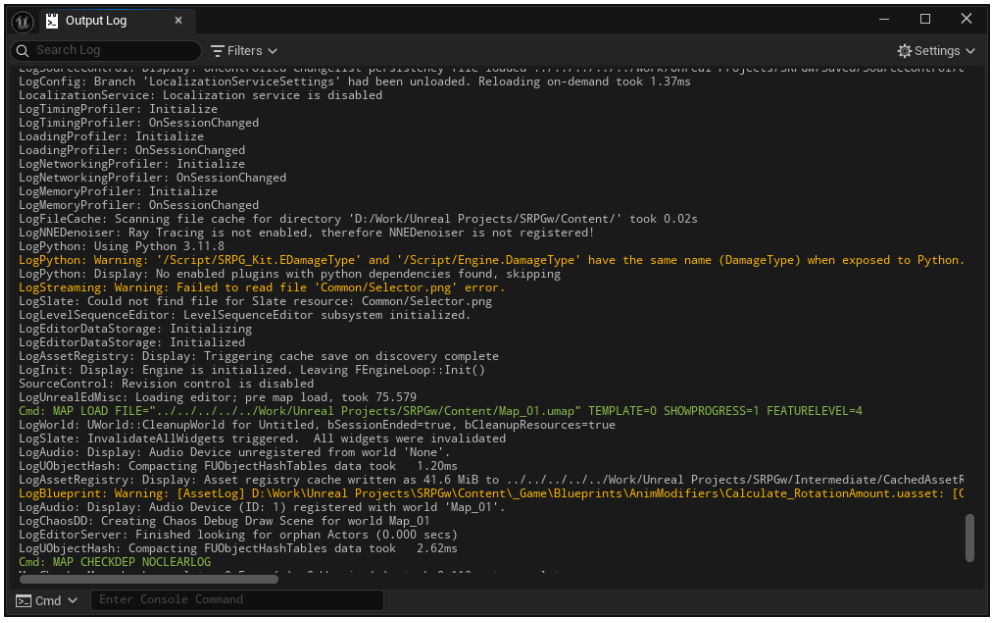

Рисунок 3.19 – Вікно виводу даних логів

Окрім функції логування, Unreal Engine також надає потужний інструмент відладки — **Gameplay Debugger Tool (GDT)**. Цей інструмент створений для ефективної роботи з основними системами ігрового штучного інтелекту, інтегрованими в рушій. Інструмент GDT корисний для перегляду даних у реальному часі під час виконання, навіть на клієнтах у мережевих іграх із використанням реплікації. Він працює в Play In Editor (PIE), Simulate In Editor (SIE) і в окремих ігрових сеансах, і всі дані відображаються як накладення у вікні перегляду гри. [11]

Водночас система Unreal Engine надає гнучку структуру, яку можна розширити для відображення та візуалізації специфічних даних, пов’язаних із грою. Розробники можуть створювати власні модулі відладки, реєструючи окремі категорії для надаючи доступ до необхідних даних.

Розроблена система надає можливість відстеження процесу вибору дій зі списку можливих, процесу виконання активних дій, показника корисності кожної з них, а також оцінки для кожного окремого лічильника (рис. 3.20). Окрім того, для підвищення зручності використання та збільшення ефективності процесу налагодження, розробникам надається можливість керування відображенням даних, як-то відключення відображення окремих задач чи списку критеріїв оцінювання. Це дозволяє залишати на екрані лише необхідні в даний момент параметри, забезпечуючи більш зручний інтерфейс для аналізу та корекції системи.

Рисунок 3.20 – Інтерфейс відладника

3.2. Тестування

Для перевірки ефективності створеної системи було розроблено проєкт на основі безкоштовного шаблону «Advanced Locomotion System (ALS v4.0)», доступного на офіційному маркетплейсі «Unreal Marketplace». Проєкт реалізовано у вигляді гри в жанрі шутера від третьої особи, у якій впроваджено базові ігрові механіки, такі як стрільба, використання спорядження, система здоров’я персонажів та обробка отриманих пошкоджень. Система переміщення персонажів і базові анімації запозичені із готових рішень, наданих розробниками ALS v4.0, що дозволило спрямувати зусилля на інтеграцію та тестування створеної системи прийняття рішень.

Базовий клас персонажа, наданий у шаблоні ALS, було модифіковано відповідно до специфіки проєкту. Для управління поведінкою NPC розроблено дочірній клас на основі інтегрованого в рушій класу **AI Controller**.

Незважаючи на можливість інтеграції системи керування процесом прийняття рішень безпосередньо в клас персонажа, відповідно до архітектури рушія, AI Controller виконує роль центрального блоку керування поведінкою агентів. Тому модуль було інтегровано як частину загального елемента керування.

Для тестування системи було розроблено кілька завдань, які виконують агенти. Для кожного завдання визначено відповідний перелік факторів оцінювання ситуації. Для кожного створеного лічильника система автоматично генерує відповідні криві у таблиці, які розробники можуть модифікувати. За замовчуванням генерується крива у вигляді прямої лінійної регресії.

Усі створені завдання було розподілено за категоріями відповідно до пулів, визначених у менеджері задач, кожному з яких призначено пріоритет виконання. Хоча початкове налаштування параметрів для кожного завдання виконується в об’єкті CDO, для підвищення зручності було реалізовано можливість редагування параметрів безпосередньо в масиві задач. Це дозволяє адаптувати кожне завдання у контексті загальної сукупності задач, пов’язаних із конкретним агентом.

У процесі виконання проєкту розробникам доступний режим відлагодження, який представлений у кількох форматах. Зокрема, це текстові логи, доступні для перегляду як у реальному часі, так і у вигляді окремого файлу для подальшого аналізу роботи системи. Крім того, відлагодження реалізовано у вигляді спеціальної категорії відладника GDT, розробленої в рамках проекту.

Після активації компонента система формує сповіщення про початок його роботи та процес ініціалізації всіх елементів, які для зручності були винесені в окрему категорію. Це дозволяє розробникам, за необхідності, фільтрувати сповіщення та здійснювати детальний аналіз роботи системи. Усі потенційні точки виникнення помилок, що можуть виникати під час ініціалізації або в процесі функціонування, також забезпечуються

відповідними сповіщеннями для оперативного виявлення та усунення проблем.

Однією з ключових вимог до системи є її оптимізація, що передбачає мінімальне споживання обчислювальних ресурсів та здатність забезпечувати коректну роботу зі значною кількістю контрольованих агентів без істотного впливу на продуктивність проєкту. Для аналізу продуктивності роботи системи використовуються інструменти профілювання, вбудовані в рушій Unreal Engine. Аналіз результатів тестування на вибірках з одного, десяти, двадцяти та п’ятидесяти контрольованих агентів показав, що збільшення кількості контрольованих агентів майже не впливає на загальні показники продуктивності всього проекту (рис 3.2).

Рисунок 3.21. – Порівняння результатів тестування системи на різних вибірках

ВИСНОВКИ

В рамках дипломної роботи було проведено дослідження та порівняння існуючих систем ігрового штучного інтелекту, а також розроблено систему динамічного та адаптивного штучного інтелекту у вигляді плагіну для рушія Unreal Engine 5.

Реалізована система прийняття рішенньь представлена у вигляді незалежного ізольованого компоненту, та дозволяє реалізовувати унікальну поведінку як для кожного неігрового персонажа окремо, так і для окремих типів.

Основна концепція системи полягає у відсутності зв’язків між станами й діями агентів та вибору найбільш корисної дії в режимі реального часу на основі поточних потреб агента, його стану та стану ігрового оточення. Оцінювання корисності всіх можливих задач базується на визначеному переліку факторів, які подаються у вигляді логічних виразів і інтерпретуються як числові значення в діапазоні від 0 до 1. Обробка значень критеріїв оцінювання здійснюється з використанням елементів теорії нечітких множин. Система дозволяє розробникам визначати ступінь належності вхідних параметрів через використання кривих належності, агрегувати дані кожного критерію за допомогою операторів нечіткої логіки, а також виконувати дефазифікацію отриманого показника корисності задачі.

Поведінка агентів обмежується лише кількістю можливих задач, стровених розробниками та переліком факторів їх оцінювання. Створена система являється зручною та доступною як для професійних розробників, так і для ігрових дизайнерів, забезпечуючи можливість швидкого прототипування моделі поведінки агентів.

Окрім можливості реалізації поведінки агентів, розробники також мають змогу опрацювання всіх можливих ситуацій та подій, що виникають в процесі роботи системи. Для цього було проведено логування всіх можливих подій, а також реалізовано власний компонент для відладки системи в реальному часі.

Реалізована система дозволяє створювати динамічну та адаптивну поведінку агента, що забезпечує враження небередбачуваності та автономності кожного персонажа, проте водночас залишається повністю контрольованою та прозорою для розробника. Для цього було передбачено та створено ряд функціональних можливостей для проведення керування кожним із елементів системи.

Проект, створений у відповідності до розробленого дизайну повністю відповідає всім поставленим вимогам. Однак, проект залишається відкритим для подальших покращень та розширень. Зокрема, для підвищення зручності роботи є можливості для створення окремого графічного редактору, розширення можливостей відладника, та реалізації додаткових елементів та функціональних можливостей. Окрім того, користувачі також мають можливості для розширення чи модифікації системи у відповідності до власних потреб.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. S.T.A.L.K.E.R.: Тінь Чорнобиля — Вікіпедія. URL: https://uk.wikipedia.org/wiki/S.T.A.L.K.E.R.:_%D0%A2%D1%96%D0%BD%D1%8C_%D0%A7%D0%BE%D1%80%D0%BD%D0%BE%D0%B1%D0%B8%D0%BB%D1%8F#%D0%A8%D1%82%D1%83%D1%87%D0%BD%D0%B8%D0%B9_%D1%96%D0%BD%D1%82%D0%B5%D0%BB%D0%B5%D0%BA%D1%82 (дата звернення: 6.10.2024).

2. S.T.A.L.K.E.R. 2 — розробники розповіли про A-Life 2.0, підтримку модів та інше | Na chasi. URL: https://nachasi.com/videogames/2024/10/21/stalker-2-a-life-2/ (дата звернення: 6.10.2024).

3. Эволюция искусственного интеллекта в шахматах. URL: https://www.chess.com/ru/blog/KirillBatalov15/evoliutsiia-iskusstvennogo-intellekta-v-shakhmatakh (дата звернення: 7.10.2024).

4. Deep Blue — Вікіпедія. URL: https://uk.wikipedia.org/wiki/Deep_Blue (дата звернення: 7.10.2024).

5. Алгоритм пошуку A* — Вікіпедія. URL: https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%BF%D0%BE%D1%88%D1%83%D0%BA%D1%83_A* (дата звернення: 7.10.2024).

6. Выбор игрового ИИ и его «сложность» / Хабр. URL: https://habr.com/ru/articles/790834/ (дата звернення: 10.10.2024)

7. Finite State Machines (конечные автоматы) — xrWiki URL: https://xray-engine.org/index.php?title=Finite_State_Machines_(%D0%BA%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D1%8B%D0%B5_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D1%8B) (дата звернення: 10.10.2024)

8. Daniel Hilburn. Simulating Behaviour Trees URL: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter08_Simulating_Behavior_Trees.pdf (дата звернення: 11.10.2024)

9. Introduction to behavior trees – Robohub. URL: https://robohub.org/introduction-to-behavior-trees/ (дата звернення: 15.10.2024).

10. F.E.A.R. (video game) – Wikipedia. URL: https://en.wikipedia.org/wiki/F.E.A.R._(video_game)#AI (дата звернення: 16.10.2024).

11. Intro to Goal Oriented Action Planning (GOAP). URL: https://www.youtube.com/watch?v=LhnlNKWh7oc&t=109s&ab_channel=ThisIsVini (дата звернення: 16.10.2024).

12. Utility system | Mathematical modeling of behavior – Wikipedia. URL: https://en.wikipedia.org/wiki/Utility_system#Mathematical_modeling_of_behavior (дата звернення: 16.10.2024).

13. An Introduction to Utility Theory. URL: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf (дата звернення: 21.10.2024).

14. Choosing Effective Utility-Based Considerations. URL: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter13_Choosing_Effective_Utility-Based_Considerations.pdf (дата звернення: 25.10.2024)

15. Utility AI - Emergent AI - Smart agents and events for games. URL: https://psichix.github.io/emergent/decision_makers/utility_ai/introduction.html (дата звернення: 25.10.2024)

16. Reinforcement Learning Applications: From Gaming to Real-World. URL: https://www.deepchecks.com/reinforcement-learning-applications-from-gaming-to-real-world/ (дата звернення: 26.10.2024)

17. Fuzzy Logic: Controlling AI | Gamedev Math. URL: https://www.youtube.com/watch?v=P4IVal0r-fA&ab_channel=NatsuGames (дата звернення: 10.11.2024)

18. What is a fuzzy rule? — Klu. URL: https://klu.ai/glossary/fuzzy-rule (дата звернення: 10.11.2024)

19. Utility AI configuration as fuzzy logic rules - Rafał Tyl || QED Games. URL: https://www.youtube.com/watch?v=H6QEpc2SQiY&t=2155s&ab_channel=GameIndustryConference (дата звернення: 12.11.2024)

20. Нечітка логіка — Вікіпедія. URL: https://en.wikipedia.org/wiki/Fuzzy_logic (дата звернення: 14.11.2024)

21. Теория алгоритмов и математическая логика Тема 5 Некласична математична логіка Нечітка логіка. URL: https://elearning.sumdu.edu.ua/free_content/lectured:5de5178bb62ca7a97fe35cba8b92d1b337ee8101/latest/8080/index.html (дата звернення: 16.11.2024)

22. Using the Gameplay Debugger in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community | Epic Developer Community. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/using-the-gameplay-debugger-in-unreal-engine (дата звернення: 21.11.2024)

23. Timalice1/UtilityAI: Plugin for UnrealEngine5, for creating dynamic and reactive decission making process for AI systems, based on Utility arhitecture and fuzzy logic. URL: https://github.com/Timalice1/UtilityAI (дата звернення: 2.12.24).

ДОДАТОК А

Посилання на код проекту:

https://github.com/Timalice1/UtilityAI

Ініціалізація системи:

```
void UUtilityManager::BeginPlay()
{
    if (IsEmpty())
    {
        UE_LOG(UtilityManagerLog, Warning, TEXT("No one action is defined"));
        return Super::BeginPlay();
    }

    for (UService *_service : Services)
    {
        if (_service)
            _service->Init(this);
    }

    for (auto &_action : _actionsScorers)
        _action.Key->Init(this, _action.Value);

    _actionsScorers.Empty();
    Super::BeginPlay();
}

void UUtilityManager::PostReinitProperties()
{
    if (GIsEditor && !GIsPlayInEditorWorld)
        return Super::PostReinitProperties();

    if (IsEmpty())
    {
        UE_LOG(UtilityManagerLog, Warning, TEXT("No one action is defined"));
        return Super::PostReinitProperties();
    }

    /*Init properies before owner begin play */
    InitPools();
    InitScorers();

    Super::PostReinitProperties();
}
```

Активація системи:

```
void UUtilityManager::Activate(bool bReset)
{
    if (GIsEditor && !GIsPlayInEditorWorld)
        return Super::Activate(bReset);

    for (UService *_service : Services)
```

```
        if (_service)
            _service->Activate();

    GetWorld()
        ->GetTimerManager()
        .SetTimer(evaluationTimer, this, &ThisClass::EvaluateActions,
EvaluationInterval, true);

    Super::Activate(bReset);
    UE_LOG(UtilityManagerLog, Log, TEXT("[%s]: UtilityManager activated"),
*GetOwner()->GetName());
}
```

Деактивація системи:

```
void UUtilityManager::Deactivate()
{
    GetWorld()->GetTimerManager().ClearAllTimersForObject(this);
    AbortActiveActions();

    for (UService *_service : Services)
        if (_service)
            _service->Deactivate();

    Super::Deactivate();
    UE_LOG(UtilityManagerLog, Log, TEXT("[%s]: UtilityManager deactivated"),
*GetOwner()->GetName());
}
```

Ініціалізація об’єкту задачі:

```
#if WITH_EDITOR
void UAction::PostEditChangeProperty(FPropertyChangedEvent &PropertyChangedEvent)
{
    FName property = PropertyChangedEvent.GetPropertyName();
    if (property == GET_MEMBER_NAME_CHECKED(UAction, Scorers) && !Scorers.IsEmpty())
    {
        Scorers[0].FirstElement = true;
        for (int i = 1; i < Scorers.Num(); i++)
            Scorers[i].FirstElement = false;
    }
    Modify();
}
#endif

void UAction::Init(TObjectPtr<UUtilityManager> InManager, TMap<FGameplayTag,
TObjectPtr<UConsideration>> InConsiderations)
{
    if (!InManager)
        return;

    _manager = InManager;
    /*Initialize controller and controlled pawn references*/
    if (AAIController *Controller = Cast<AAIController>(_manager->GetOwner()))
    {
        _controller = Controller;
        _pawn = Controller->GetPawn();
    }
    else if (APawn *Pawn = Cast<APawn>(_manager->GetOwner()))
```

```
    {
        _controller = Cast<AAIController>(Pawn->GetInstigatorController());
        _pawn = Pawn;
    }

    if (InConsiderations.IsEmpty())
    {
        UE_LOG(UtilityManagerLog, Warning, TEXT("Can't init action [%s]: no one
consideration is defined"), *GetName());
        return;
    }

    /*Assing considerations to their scorers*/
    for (FScorer &Scorer : Scorers)
    {
        TObjectPtr<UConsideration> _cons = *InConsiderations.Find(Scorer.ScorerTag);
        if (!_cons || Scorer.GetConsiderationInstance() != nullptr)
            continue;

        if (auto CurveTable = _manager->GetCurveTableAsset())
        {
            FRichCurve *Curve = CurveTable-
>FindRichCurve(Scorer.ScorerTag.GetTagName(), FString(), false);
            if (Curve)
            {
                _cons->SetResponseCurve(*Curve);
                Scorer.SetConsiderationInstance(_cons);
            }
        }
    }

    if(auto _modifiersTable = _manager->GetActionModifiersCurveTable())
        ModifierCurve = _modifiersTable->FindRichCurve(FName(GetName().LeftChop(4)),
FString(), false);

    UE_LOG(UtilityManagerLog, Log, TEXT("Action [%s] initialized!"), *GetName());

    Modify();
}
```

Встановлення вхідного параметра лічильника:

```
void UUtilityManager::SetScorerValue(FGameplayTag InScorerTag, float InValue)
{
    if (AController *controller = Cast<AController>(GetOwner()))
    {
        if (!controller->GetPawn())
            return;
    }
    if (_considerations.IsEmpty())
    {
        UE_LOG(UtilityManagerLog, Warning, TEXT("UtilityManager::SetScorerValue(%s) -
no one scorer is defined."),
               *InScorerTag.GetTagName().ToString());
        return;
    }
    if (!InScorerTag.IsValid())
    {
        UE_LOG(UtilityManagerLog, Warning,
               TEXT("UtilityManager::SetScorerValue - tag is invalid"));
```

```
        return;
    }
    if (!_considerations.Contains(InScorerTag))
        return;
    TObjectPtr<UConsideration> _cons = *_considerations.Find(InScorerTag);
    _cons->SetValue(InValue);
}
```

Обчислення оцінки критерію:

```
    float _value;
    FRichCurve _responseCurve;

    virtual float GetRawScore()
    {
        if (!_responseCurve.IsEmpty())
            return _responseCurve.Eval(_value);
        return 0.f;
    }

    UPROPERTY()
    TObjectPtr<UConsideration> _considerationInstance;

    virtual float GetScore()
    {
        if (_considerationInstance != nullptr)
        {
            float rawScore = _considerationInstance->GetRawScore();
            rawScore = Inverted ? 1 - rawScore : rawScore;
            return rawScore;
        }
        return 0.f;
    }
```

Обчислення оцінки корисності задачі:

```
float UAction::EvaluateActionScore()
{
    if (ScoreType == EST_Constant)
        ActionScore = ConstantScore;

    if (ScoreType == EST_Evaluated && !Scorers.IsEmpty())
    {
        ActionScore = Scorers[0].GetScore();
        for (int i = 1; i < Scorers.Num(); i++)
        {
            float currentScore = Scorers[i].GetScore();

            if (Scorers[i].Operator == OR)
                ActionScore = (ActionScore + currentScore) - (ActionScore *
currentScore);
            else if (Scorers[i].Operator == AND)
                ActionScore *= currentScore;
        }
    }
    if (ModifierCurve != nullptr)
```

```
        ActionScore = ModifierCurve->Eval(ActionScore);
    return ActionScore * ActionPriority;
}
```

Запуск та завершення процесу виконання логіки задачі:

```
void UAction::Execute()
{
    IsFinished = false;
    if (_controller && _pawn)
        this->ExecuteAction(_controller, _pawn);
}

void UAction::FinishExecute(EExecutionResult execResult)
{
    IsFinished = true;
    OnActionFinished(_controller, _pawn, execResult);
    if (Timeout > .0f)
    {
        bCanBeEvaluated = false;
        GetWorld()->GetTimerManager().SetTimer(_timeoutTimer, this,
&UAction::ResetTimeout, Timeout);
    }
}
```

Процес вибору найбільш доцільної задачі:

```
TObjectPtr<UAction> FActionsPool::EvaluateActions()
{
    UAction *_bestAction = nullptr;
    float maxScore = 0;
    for (UAction *action : Actions)
    {
        if (!action || !action->CanBeEvaluated())
            continue;
        float currentScore = action->EvaluateActionScore();
        if ((currentScore > maxScore) && (currentScore > ScoreThresshold))
        {
            maxScore = currentScore;
            _bestAction = action;
        }
    }
    return _bestAction;
}

void UUtilityManager::EvaluateActions()
{
    _activeActions.RemoveAll([](UAction *_action)
                             { return _action->IsFinished; });

    _pools.Sort([](const FActionsPool &p1, const FActionsPool &p2)
                { return p1.Priority > p2.Priority; });

    for (FActionsPool &_pool : _pools)
    {
        UAction *_evaluatedAction = _pool.EvaluateActions();
        if (_evaluatedAction &&
            CanRunConcurent(_evaluatedAction))
        {
```

```
            _activeActions.Add(_evaluatedAction);
            _evaluatedAction->Execute();
            return;
        }
    }
}
```

Функції переривання виконання активних задач:

```
void UUtilityManager::AbortActiveActions()
{
    for (UAction *_action : _activeActions)
    {
        _action->FinishExecute(EExecutionResult::Aborted);
        UE_LOG(UtilityManagerLog, Log, TEXT("%s: action aborted"), *_action-
>GetName());
    }
}

void UUtilityManager::AbortAction(TSubclassOf<UAction> action)
{
    for (UAction *_action : _activeActions)
    {
        if (_action->GetClass() == action)
        {
            _action->FinishExecute(EExecutionResult::Aborted);
            return;
        }
    }
}

void UUtilityManager::AbortActionsFromPool(FName PoolName)
{
    FActionsPool *_pool = _pools.FindByPredicate([PoolName](const FActionsPool &p)
                                                 { return p.PoolName == PoolName; });
    if (_pool)
    {
        for (UAction *_action : _pool->GetActions())
            if (_action && _activeActions.Contains(_action))
                _action->FinishExecute(EExecutionResult::Aborted);
    }
}
```

Встановлення пріорітету пулу задач:

```
bool UUtilityManager::SetPoolPriority(FName PoolName, int32 Priority)
{
    FActionsPool *_pool = _pools.FindByPredicate([PoolName](const FActionsPool &elem)
                                                 { return elem.PoolName == PoolName;
});
    if (_pool)
    {
        _pool->SetPoolPriority(Priority);
        return true;
    }
    UE_LOG(UtilityManagerLog, Warning, TEXT("Pool with name [%s] not found"),
*PoolName.ToString());
    return false;
}
```

Оновлення таблиць кривих факторів оцінювання та модифікаторів оцінок корисності:

```
void UUtilityManager::PostEditChangeProperty(FPropertyChangedEvent
&PropertyChangedEvent)
{
    FName property = PropertyChangedEvent.GetPropertyName();
    if (((property == GET_MEMBER_NAME_CHECKED(FActionsPool, Actions) &&
ScorersCurveTable != nullptr) ||
         property == GET_MEMBER_NAME_CHECKED(UUtilityManager, ScorersCurveTable)) &&
        PropertyChangedEvent.ChangeType == EPropertyChangeType::ValueSet)
        UpdateScorersCurveTable();

    if (((property == GET_MEMBER_NAME_CHECKED(FActionsPool, Actions) &&
ActionModifiersCurveTable != nullptr) ||
         property == GET_MEMBER_NAME_CHECKED(UUtilityManager,
ActionModifiersCurveTable)) &&
        PropertyChangedEvent.ChangeType == EPropertyChangeType::ValueSet)
        UpdateModifiersCurveTable();
    Modify();
}

void UUtilityManager::UpdateScorersCurveTable()
{
    if (IsEmpty())
    {
        ResetConsiderations();
        return;
    }

    if (!IsValid(ScorersCurveTable))
    {
        UE_LOG(UtilityManagerLog, Warning, TEXT("Property [ScorersCurveTable] is
invalid!"));
        return;
    }

    for (FActionsPool &pool : ActionsPools)
    {
        /*Get scorers array from each referenced action*/
        for (const TObjectPtr<UAction> &action : pool.GetActions())
        {
            if (!action)
                continue;

            for (FScorer &scorer : action->GetScorers())
            {
                /*Get a scorer tag from each scorer in action*/
                FGameplayTag tag = scorer.ScorerTag;
                if (tag.IsValid() && !ScorersCurveTable-
>FindRichCurve(tag.GetTagName(), FString(), false))
                    CreateCurve(tag.GetTagName(), *ScorersCurveTable); // if curve is
not exist, create new curve in table
            }
        }
    }
}
```

```
void UUtilityManager::UpdateModifiersCurveTable()
{

    if (!IsValid(ActionModifiersCurveTable))
    {
        UE_LOG(UtilityManagerLog, Warning, TEXT("Property [ScorersCurveTable] is
invalid!"));
        return;
    }

    for (FActionsPool &pool : ActionsPools)
    {
        /*Get scorers array from each referenced action*/
        for (const TObjectPtr<UAction> &action : pool.GetActions())
        {
            if (!action)
                continue;
            FName _actionName = FName(action->GetName().LeftChop(4));
            if (!ActionModifiersCurveTable->FindRichCurve(_actionName, FString(),
false))
            {
                CreateCurve(_actionName, *ActionModifiersCurveTable);
            }
        }
    }
}
```

Код сервісного класу:

```
#pragma once
#include "../Core/UtilityManager.h"
#include "AIController.h"
#include "Service.generated.h"

class APawn;
class UUtilityManager;
class AAIController;

UCLASS(MinimalAPI, Abstract, EditInlineNew, Blueprintable, BlueprintType)
class UService : public UObject
{
    GENERATED_BODY()

private:
    TObjectPtr<UUtilityManager> _manager;
    TObjectPtr<AAIController> _controller;
    TObjectPtr<APawn> _pawn;

protected:

    UFUNCTION(BlueprintCallable, BlueprintPure, Category = "UtilitySystem|Service")
    APawn *GetPawn() { return _pawn;}
    UFUNCTION(BlueprintCallable, BlueprintPure, Category = "UtilitySystem|Service")
    UUtilityManager *GetUtilityManager() { return _manager; };
    UFUNCTION(BlueprintCallable, BlueprintPure, Category = "UtilitySystem|Service")
    AAIController *GetController() { return _controller; };

public: // Parent class overrides
    UService() : Super() {};
```

```
    FTimerHandle tickTimer;

    virtual UWorld *GetWorld() const override;
    virtual void PostInitProperties() override;
    virtual void Init(class UUtilityManager *inManager);
    virtual void Activate();
    virtual void Deactivate();
protected: // Properties
    UPROPERTY(EditDefaultsOnly, Category = "ServiceConfig")
    float UpdateInterval = .01f;
protected: // Events
    UFUNCTION(BlueprintImplementableEvent, Category = "UtilityAI|Service")
    void ServiceStart(APawn *Pawn, UUtilityManager *Manager);
    virtual void Start()
    {
        ServiceStart(_pawn, _manager);
    };
    UFUNCTION(BlueprintImplementableEvent, Category = "UtilityAI|Service")
    void ServiceUpdate(APawn *Pawn, UUtilityManager *Manager);
    virtual void Update()
    {
        ServiceUpdate(_pawn, _manager);
    };
};
```