Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Порівняльна реалізація алгоритмів симуляції штучного інтелекту в іграх

(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-21-5
Даніїл Красніков
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект
(повна назва освітньої програми)

Керівник ст. викл. Тетяна Мірошниченко
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ _____ Олег ЗОЛОТУХІН
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

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

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Краснікову Даніілу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Порівняльна реалізація алгоритмів симуляції штучного інтелекту в іграх

затверджена наказом університету від 19 травня 20 25 р. № 378Ст

2. Термін подання студентом роботи до екзаменаційної комісії 18 червня 20 25 р.

3. Вихідні дані до роботи дані публічних Інтернет-джерел та технічних книг щодо методів симуляції штучного інтелекту в комп'ютерних іграх

4. Перелік питань, що потрібно опрацювати в роботі

1) Аналіз предметної галузі та постановка задачі

2) Дослідження алгоритмів симуляції штучного інтелекту в комп'ютерних іграх

3) Реалізація алгоритмів симуляції штучного інтелекту в комп'ютерних іграх

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	19.05.2025	виконано
2	Аналіз предметної галузі та постановка задачі	25.05.2025	виконано
3	Дослідження алгоритмів	30.05.2025	виконано
4	Реалізація алгоритмів	06.06.2025	виконано
5	Написання пояснювальної записки	07.06.2025	виконано
6	Перевірка на академічний плагіат	12.06.2025	виконано
7	Нормоконтроль	12.06.2025	виконано
8	Підготовка презентації та доповіді	12.06.2025	виконано
9	Попередній захист	13.06.2025	виконано
10	Рецензування	15.06.2025	виконано
11	Захист перед ЕК	18.06.2025	

Дата видачі завдання 19 травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ст. викл. Тетяна Мірошниченко
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка: 75 с., 24 рис., 1 дод., 10 джерел.

АЛГОРИТМИ, КОМП’ЮТЕРНІ ІГРИ, ШІ, COMPUTE SHADER, C#, DOTS, UNITY 3D.

Об'єкт дослідження – штучний інтелект в комп’ютерних іграх.

Предмет дослідження – алгоритми симуляції штучного інтелекту в комп'ютерних іграх.

Мета роботи – вивчення наявних алгоритмів симуляції штучного інтелекту, пропозиція та розробка власних методів їхньої реалізації, що збільшить їхню швидкодію та/або дозволить спростити їхнє сприйняття.

Методи досліджень – аналіз публічних статей, репозиторіїв, фото та відео матеріалів, практична реалізація обраними технологіями.

У ході кваліфікаційної роботи було проаналізовано три алгоритми симуляції ШІ в іграх: State Machine, Behavior Tree, GOAP. За результатами аналізу визначено слабкі сторони кожного з алгоритмів та запропоновано способи виправлення цих мінусів. Було реалізовано ці алгоритми враховуючи запропоновані правки та проведено тести для оцінки ефективності кожного з методів.

ABSTRACT

Bachelor’s thesis contains: 75 pp., 24 fig., 1 ann., 10 references.

AI, ALGORITHMS, COMPUTE SHADER, C#, DOTS, UNITY 3D, VIDEO GAMES.

Object of study is artificial intelligence in video games.

Subject of study is artificial intelligence simulation algorithms in computer games.

Purpose of the work – to study existing algorithms for simulating artificial intelligence, propose and develop custom methods that increase performance and/or simplify their comprehension.

Research methods – analysis of public articles, repositories, photo and video materials, and practical implementation using selected technologies.

In the course of this qualification work, three algorithms for AI simulation in games were analyzed: State Machine, Behavior Tree, and GOAP. Based on the analysis, the weaknesses of each algorithm were identified, and methods for addressing these shortcomings were proposed. The algorithms were implemented with the suggested improvements, and tests were conducted to evaluate the effectiveness of each method.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект;

AI – Artificial Intelligence – штучний інтелект;

ECS – Entity-Component-System – сутність-компонент-система;

FPS – Frames per Seconds – кадри за секунду;

Unity – рушій розробки ігор Unity.

ВСТУП

Ігрова індустрія наразі знаходиться в своїй найбільш активній фазі. Щодня на віртуальних полицях інтернет магазинів з’являються сотні нових ігор. Це легко пояснити різноманітністю ігрового досвіду, який очікують гравці та можуть надати розробники. І справді, індустрія пройшла свій розвиток від Space Invaders та Tetris, до Sonic the Hedgehog і нарешті вийшла на рівень S.T.A.L.K.E.R. 2: Серце Чорнобиля та Balatro. Кожен проєкт розказує історії, які ніхто до цього не чув, та пропонує досвід, який до цього ніхто не відчував.

Важливою складовою історій є не ігрові персонажі – НІПи. Вони наповнюють світ, протистоять чи допомагають гравцю. За розвитком світу НІПів насправді цікаво спостерігати, якщо він зроблений добре. Існують ігри, які використовують такий підхід для покращення сприйняття світу гравцем. Серед яскравих прикладів неможливо не згадати Stardew Valley чи Animal Crossing. НІПи в цих іграх мають власний розклад дня, улюблені справи та предмети, схильні до певних дій. Через ці фактори вони здаються реальними людьми – добре пропрацьованими особистостями, а гравець відчуває себе на своєму місці – затишне та спокійне відчуття, що твої дії визначають лише твій стан. Взаємодія з НІПами приносить спокій та гра створює медитативну приємну атмосферу.

Окремо хочеться виділити шедевральну серію ігор The Last Of Us. Неможливо уявити цю історію без добре пропрацьованого ШІ ворогів, які враховують ігрові умовності та природньо реагують на дії гравця. Вбивство кожного ворога відчувається як важке рішення, обумовлене суворістю цього світу. Але і сама дія не є такою простою задачею – вороги активно використовують свої переваги: заражені нападають натовпом та переважають числом, коли люди продумують тактику, використовують навколишнє середовище у своїх цілях та враховують сили загону. Звична гравцям формула «вбив – отримав нагороду» також порушується: кожна

смерть супроводжується криками про поміч та оплакуванням полеглих товаришів.

І це одні з небагатьох прикладів, як добре створені не ігрові персонажі наповнюють світ сенсом, рухають сюжет та створюють певні емоції. Все це було б неможливим, якби у них не було власного штучного інтелекту.

Не зайвим буде уточнити, що існують ігри, які завойовують серця мільярдів навіть без персонажів. Наприклад, раніше згадувалася Balatro – картковий roguelite про покер. Однак навіть в ній є своєрідний ШІ – ігрова система, яка підкидає гравцю карти та визначає складність кожного наступного раунду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної галузі

Обрана предметна галузь, алгоритмічна симуляція штучного інтелекту в іграх, знаходиться на перетині двох: штучний інтелект та комп'ютерні ігри. Пропонується проводити їхній аналіз окремо.

1.1.1 Аналіз предметної галузі «Комп'ютерні ігри»

Комп'ютерні ігри – це розважальний чи розважально-навчальний (виховний) підвид виконуваних програм. Вони можуть включати в себе багато підсистем та складаються з:

- арт (англ. Art – художні елементи);
- тех (англ. Tech – технічна частина);
- тех арт (англ. Tech Art – технічний художник).

Пропонується розглянути ці відділи детальніше.

Кожна гра має візуальну частину, з якою користувач взаємодіє. Ця категорія об'єднує в собі двовимірні картинки (спрайти), анімації, тривимірні моделі, текстури (картинки, які надають колір моделі, визначають фізичні властивості моделі, які стосуються її відображення), ріг (англ. Rig – переведення статичної моделі в стан, в якому її можна анімувати, шляхом створення «кісток» та прив'язки кожної частини моделі до певної кістки), інтерфейс користувача, візуальні ефекти (англ. VFX чи Particles – часточки, які додають об'єм, наповненість дій та заповнюють їхній результат [3]).

Проте арт відділ включає в себе також геймдизайнерів. Гейм дизайн – це продумування механік гри, їхнє балансування, опрацьовування відгуків, створення схем рівнів. Цей відділ генерує основні ідеї, переводить їх на технічну мову, формуючи документацію проєкту, з якою потім працює тех

відділ. Геймдизайнери також прописують сюжети, діалоги, контролюють якість та умісність артової частини, допомагають відділу маркетингу з режисурою та зйомкою відео для реклами гри, заповнюють щоденники розробки.

Кожна гра має власний рушій, на якому ведеться розробка. Ігровий рушій – це програма, яка включає в себе спільний для всіх ігор функціонал, з додатковими розширеннями та полегшеннями. Серед популярних рушіїв є Unity [8], Godot [4] та Unreal Engine [9]. Кожен з них пропонує зручні та потужні інструменти відображення (рендеригну), фізики об'єктів (базові сили за законами механіки, прорахування колізії), та додатковий функціонал, який обов'язковий для розробки ігор.

Проте рушій – це лише зручна оболонка. Для її наповнення необхідний арт та програмний код. Код пишеться розробниками та кожен рушій має власну мову: для Unity це C#, Unreal використовує C++, спільнота Godot розвиває його інтеграцію з Python.

Для написання ефективного коду в ігровій індустрії необхідно знати не лише технічні особливості мови, а також самого рушія, так як він може накладати обмеження. Так, наприклад Unity, використовує віртуальне середовище Mono, замість звичного C# virtual machine. Через це у розробників немає доступу до надпотужного збірника сміття, який розроблявся компанією Microsoft спеціально для C#.

Відділ технічного арту займається величезною кількістю завдань. Основна їхня задача – інтеграція арту в фінальний продукт, його оптимізація, полірування інтерфейсу користувача, налаштування візуальної частини проєкту. Загалом, технічний артист починає свою роботу після того, як розробник та художник закінчили свою відповідну частину в конкретній задачі. Відділ тех арту об'єднує розробників і художників та допомагає проводити їм якісну комунікацію.

Над створенням ігрових проєктів також працюють тестувальники, менеджери проєктів та маркетинг.

1.1.2 Аналіз предметної галузі «Штучний інтелект»

Штучний інтелект – це загальна назва, яка об’єднує під собою три основні частини: комп’ютерний зір, нейронні мережі та машинне навчання. Розберемо ці частини окремо.

Комп’ютерний зір – це те, звідки штучний інтелект отримує інформацію і потенційно вивід в зовнішній світ. Ця система може в себе включати всеможливі набори датчиків (наприклад камери, датчики руху, газу, термометри тощо), консолі для введення користувачем (текстове поле для спілкування з LLM (Large Language Models) моделями) та способи виводу (мотори, сервоприводи, текстовий чи звуковий інтерфейс). Дані, отримані комп’ютерним зором, попередньо обробляються та виділяється інформація. Вона в свою чергу форматується до вигляду, в якому нейронній мережі буде це легше опрацьовувати, та передається на вхід.

Нейронні мережі по суті своїй є орієнтованими графами, які трансформують вхідну інформацію у відповідь. В основному тут використовується числовий підхід. Тобто інформація представляються у вигляді певної послідовності значень, які подаються на вхідні нейрони. Всередині графа відбуваються прості математичні операції над цими числами і в фінальний слой виводиться результат цих математичних обрахунків. В свою чергу, що саме означають вхідні та вихідні значення, їхня інтерпретація і перевод в подальші інструкції чи команди визначаються розробником. Наприклад в системі автопілотування агенту [2] використовувалися вхідні значення позиції (три на кожне значення світової координати) та один вихід, який був цілим числом. Кожне число виходу визначало дію: піти в сторону, зареєстровану під конкретним номером, чи стояти бездіяльно.

Перед тим, як увага перейде до розбору наступної складової, хотілося би більше поговорити про архітектуру нейронних мереж. Нейронні мережі наслідують структуру мозку біологічних істот, зображену на рисунку 1.1.

Так, кожна нейронна мережа складається так званих нейронів – найменші одиниці графу, які проводять розрахунки засновуючись на вхідних даних.

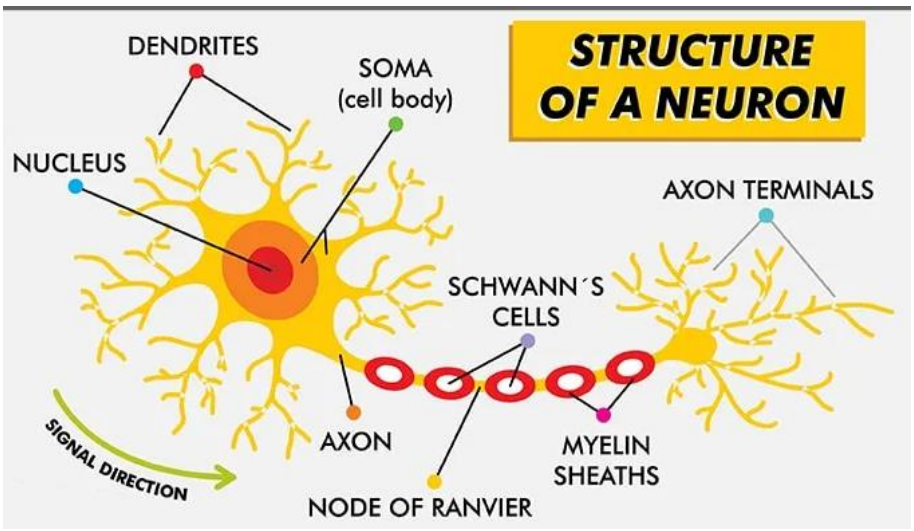

Рисунок 1.1 – Структура біологічного нейрону

Кожен нейрон має своє з’єднання (в біологічній аналогії – аксон), так званий перехід. Останній, в свою чергу має власну вагу. Як і в біологічній структурі, в процесі перетворень нейрони можуть блокуватися та не видавати значення. Ця поведінка визначається розрахунком суми всіх вхідних даних та біаса, а результат пропускається через нелінійну функцію активації (наприклад найпоширенішій тип – сігмоїдна функція). Це дозволяє використовувати більш складні патерни навчання та створювати зони нейронної мережі для вирішення різних задач. Все аналогічно тому, як в людському мозку йде умовне розділення на зони відповідальності (наприклад зорова, слухова та моторна кора).

Як писалося вище, нейрони організовуються в орієнтований граф, який в свою чергу розділяється на специфічну для штучних нейронних мереж річ – на шари. Їх приклад можна побачити на рисунку 1.2. Обов’язковими є вхідний та вихідний, мінімум один (відповідно їх може бути набагато більше) прихованих шарів, або ж шарів обробки.

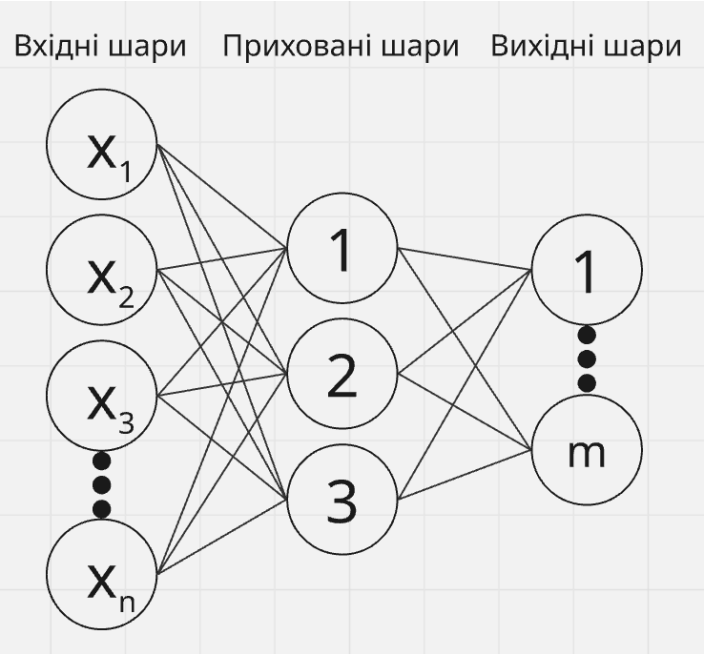

Рисунок 1.2 – Приклад слоїв в штучній нейронній мережі

Нейрони вхідного шару є пасивними – вони ніяк не зміють дані, а тільки комунікують (передають) їх на приховані слої. Загалом це і є їхньою задачею – отримати дані. Зазвичай кількість нейронів вхідного шару відповідає кількості зміних, проте для складних типів даних ситуація може бути іншою. Наприклад, розглянути всім відомий приклад з простою нейромережею, яка розпізнає числа. Для користувача зміна буде лише одна (фотографія чи малюнок) тоді як для нейронної мережі ці дані треба перевести в значення кольору кожного пікселя на екрані та передавати вже їх. Для квадратної фотографії з ребром 256 пікселів це буде 65536 нейронів вхідного шару.

Між вхідним та вихідним слоями лежать приховані, або ж слої обробки. Важливо відмітити, що кожен нейрон прихованого слою об’єднаний з кожним нейроном попереднього слою: вхідного чи іншого прихованого. Як вже вказувалося вище, основна обробка даних відбувається за рахунок зважених переходів між слоями. Вихідні з’єднання об’єднуються з кожним нейроном наступного слою і, вже зважені, формують його результат шляхом отримання суми всіх входів.

Вихідний шар збирає всі входи та є заключним етапом, з якого ми отримуємо результат. Для проблем класифікації достатньо використовувати лише один вихідний нейрон, результат якого вкаже на приналежність до класу/кластеру. Однак принцип залишається однаковим і для більшої кількості результатів: зібрати та отримати суму всіх зважених виходів від активних нейронів попереднього шару обробки.

Проте результат нейронних мереж з самого початку завжди буде повністю неправильним – точність системи надто низька на випадкових числах. В основному це через незбалансованість оцінок в середині самої нейронної мережі, якщо бути точнішими – коефіцієнтів переходів та функцій активацій кожного нейрону. Для невеликих систем це можливо виправити навіть ручним налаштуванням, але кількість часу, який піде на це, занадто невиправдана. Тому на допомогу приходить машинне навчання: автоматичний спосіб збільшити точність передбачень нейронної мережі шляхом балансування оцінок та біасів кожного нейрону.

Повертаючись ближче до теми кваліфікаційної роботи, штучний інтелект для ігор створюється в синергії технічного відділу та геймдизайну. Для забезпечення чесного досвіду гравця, геймдизайнери прописують очікувану поведінку кожної сутності та системи. Це довгий процес, який включає в себе визначення особистості та походження сутності (якщо це істота), продумування логічних та очікуваних дій в межах сюжету, їхню приоритетність і порядок. Технічний відділ використовує цю інформацію для визначення складності задачі та можливих спопобів її вирішення. Наступним кроком завжди стає імплементація обраного рішення розробниками, ітеративно покращуючи та підлаштовуючи реалізацію в комунікації з геймдизайнерами. Останнім кроком розробники, чи окремий відділ, відповідальний за штучний інтелект ігрових сутностей, проводять доналаштування та балансування системи обраного рішення, підводячи поведінку сутності до прописаної в технічній документації.

Проте реалії ринку розчаровують – ігри не можуть собі дозволити використання повноцінного ШІ. На це є декілька причин, серед яких можна виділити час, вартість розробки і підтримки, та ресурсну обмеженість пристроїв користувачів.

І справді, заробітня плата на позицію AI Engineer, в порівнянні зі звичайними розробниками, складає на приблизно 10% більше [5]. В ситуації сучасності, коли багатьом позиціям в геймдеві недоплачують значні суми, оплачувати роботу спеціаліста такого класу буде майже неможливою задачею.

Ресурсна обмеженість пристроїв звучить як жарт, враховуючи які досягнення наразі в сфері електронних компонентів для споживачів широкого кола збуту. Проте варто враховувати три фактори:

- більшість ігор виходять та розробляються для мобільних пристроїв;
- найбільш популярні складові для комп’ютерів не відповідають останнім вийшовшим відеокартам та процесорам [6];
- сучасним іграм і без того необхідні всі потужності, які має пристрій, тож забивати процесор чи відеокарту 1650-серії повноцінною нейромережею призведе до максимального зниження якості гри.

1.2 Постановка задачі

Зважаючи на всі вище перераховані факти, були виділені основні задачі кваліфікаційної роботи:

- пошук ринкових рішень, виділення популярних;
- аналіз підходів та реалізації ринкових рішень;
- запропонувати кращі способи імплементації алгоритмів;
- реалізувати запропоновані виправлення до алгоритмів;
- порівняння реалізації та пояснення їхнього результату.

2 ДОСЛІДЖЕННЯ АЛГОРИТМІВ СИМУЛЯЦІЇ ШТУЧНОГО ІНТЕЛЕКТУ В КОМП'ЮТЕРНИХ ІГРАХ

Отже, перед розробкою необхідно дослідити, які наразі є ринкові способи симуляції штучного інтелекту, їхні переваги та недоліки. На основі цього аналізу виділити складності та можливості контролю і перейти до імплементації обраних представників.

Важливо відмітити, що в кваліфікаційній роботі не будуть вважатися алгоритми пошуку шляху за спосіб симуляції штучного інтелекту. Основним критерієм буде можливість алгоритму самостійно приймати та виконувати рішення, тоді як пошук шляху краще вважати утилітою – підпрограмою, яка може допомогти основному алгоритму виконати чи оцінити свої дії.

2.1 Пошук та виділення категорій ринкових рішень

Пропонується проводити аналіз публічних рішень використовуючи платформу Unity Asset Store [7]. Unity Asset Store – це популярне рішення, якому довіряють геймдев-спеціалісти з усіх куточків світу та сфер, представлене компанією Unity.

За довгі роки розробники створили та виклали для продажі сотні рішень, які використовуються в іграх і досі. Деякі з них були викуплені та стали офіційною частиною Unity Engine та тепер знаходяться в безкоштовному доступі для всіх охочих, як для доробки, так і для простого використання в своїх проєктах.

Як можна побачити на рисунку 2.1, станом на 04.05.2025 Asset store пропонує 411 результатів імплементації поведінкових алгоритмів штучного інетелекту.

Якщо пролистати та проаналізувати декілька сторінок пошуку, то стає швидко зрозуміло, що в цей фільтр потрапляють алгоритми, які вирішені не

включати в поняття симуляції штучного інтелекту (пошук шляху, системи діалогів, ранні прототипи динамічної деформації 3д моделей).

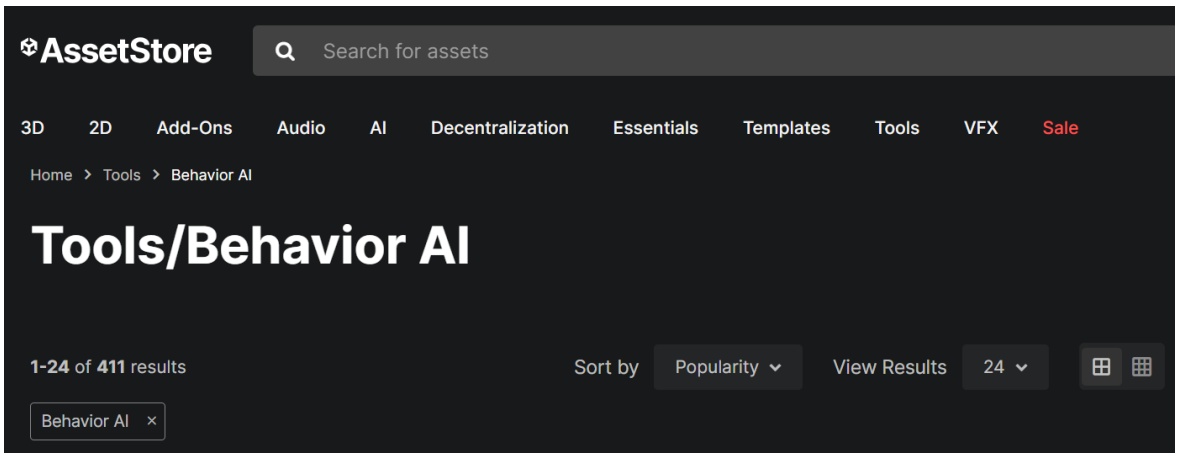

Рисунок 2.1 – Результати пошуку

Для кваліфікаційної роботи цікавить залишок запропонованих асетів. За швидким аналізом, з них також можна викинути деякі системи, наприклад повноцінні нейронні мережі, навантаження на систему від яких підходить лише для великих симуляцій систем, а не реальних ігрових проєктів. Отже, викинувши всі ці пропозиції, залишаються лише ринкові рішення, які і цікавили від початку – алгоритмічна симуляція штучного інтелекту.

Результат аналізу групи, що залишилася, щиро здивував. Основна маса всіх асетів пропонує використання патерну програмування State Machine. Менш популярні для продажу (і відповідно використання) стало сімейство Behavior-алгоритмів (Behavior Tree). І абсолютно непопулярними рішеннями стали алгоритми, які своєю потужністю дозволяють створювати повноцінні системи штучного інтелекту без обмежень та складнощів, які вони несуть (алгоритм GOAP – Goal Oriented Action Planning).

Пропонується почати аналіз з найбільш популярного алгоримту – State Machine, або Машина Станів.

Найкраще цей алгоритм зображує рисунок 2.2, на основі якого проводитиметься подальший аналіз. Одразу варто додати, що схема не є повною, та буде доповнена в розділах пізніше.

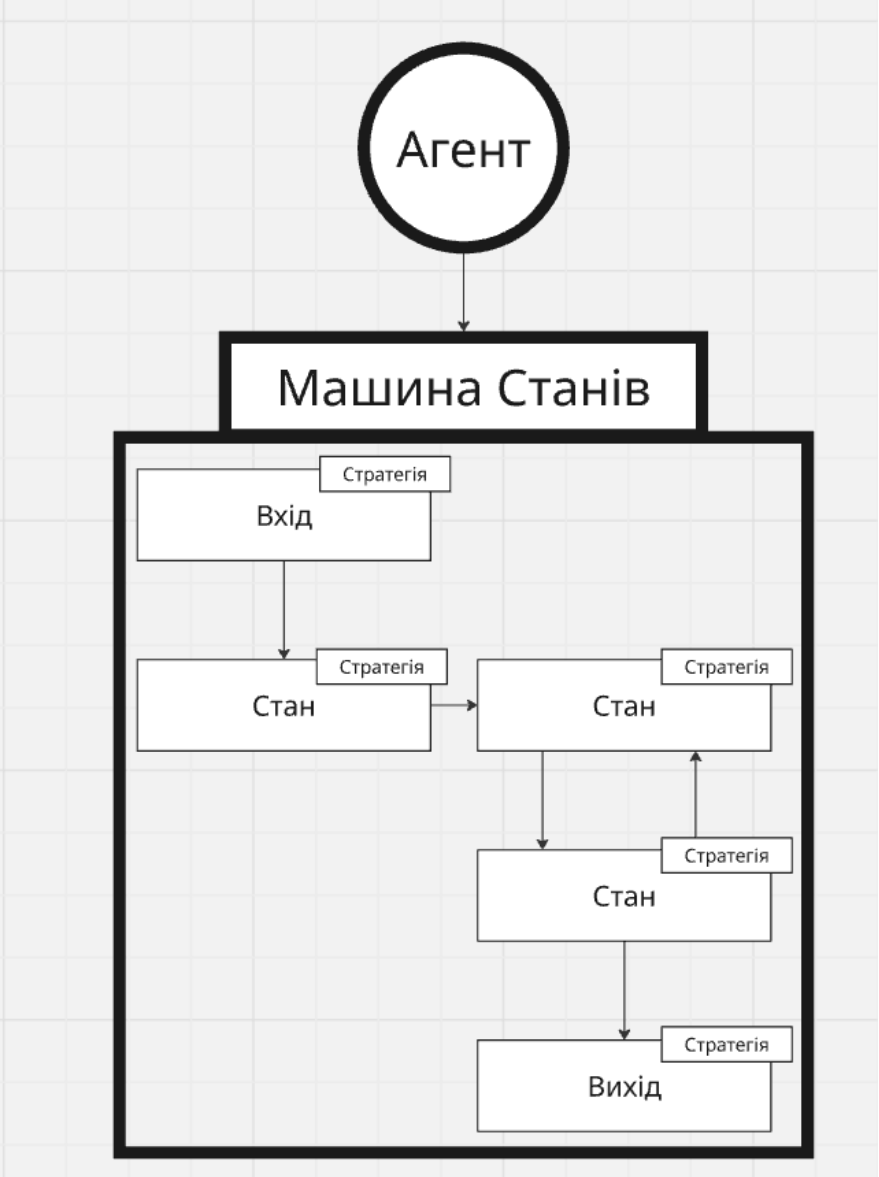

Рисунок 2.2 – Схема Машини Станів

По суті своїй це найпростіший спосіб створення багатьох систем, так як дозволяє використати всі плюси патерну програмування State, Стан. Найкращий опис цього патерну надає сервіс refactoring.guru: «Стан — це поведінковий патерн проектування, що дає змогу об’єктам змінювати

поведінку в залежності від їхнього стану. Ззовні створюється враження, ніби змінився клас об’єкта.» [1]. Тобто це дозволяє агенту повністю змінювати свої дії, імітуючи «поведінку», яка підлаштовується під ситуації. Машина станів в свою чергу оперує станами одного об’єкту, контролює його переходи та поточний стан, як зображено на рисунку 2.2, сповіщає всіх спостерігачів про зміну стану та запускає виконання дій, які відповідають цим станам.

Традиційно, архітектура самого алгоритму проста та складається з агента, машини станів, імплементації конкретних станів та їхніх відповідних стратегій, і списку умов переходів з одного стану в інший.

З очевидних плюсів такого підходу можна виділити його просту реалізацію. Також не варто забувати, що розробник завчасно задає стани, стратегії та умови переходів, і це дає повний контроль та відповідність задумкам геймдизайнерів.

Але останній плюс створює найбільший мінус машини станів – складність розширення. За бажанням додати новий стан та відповідну стратегію стоятиме довжелезний список конфігурації умов переходів, виставлення зв’язків та послідовностей. З особистого досвіду, коли розмір станів агенту переходить за 10, систему більше майже неможливо тримати в чистому та читабельному вигляді.

Зважуючи перераховані аргументи, машина станів відноситься до легких алгоритмів з повним контролем дій.

Наступними на черзі аналізу будуть Behavior-алгоритми. Основна їхня суть – визначати можливі дії та виконувати їх на основі умов, які були задані завчасно розробником.

Найлегше уявляти ці алгоритми у вигляді дерев чи графів, як зображено на рисунку 2.3. Таким чином інтуїтивно можна виділити складові та визначити їхню мінімальну необхідну кількість.

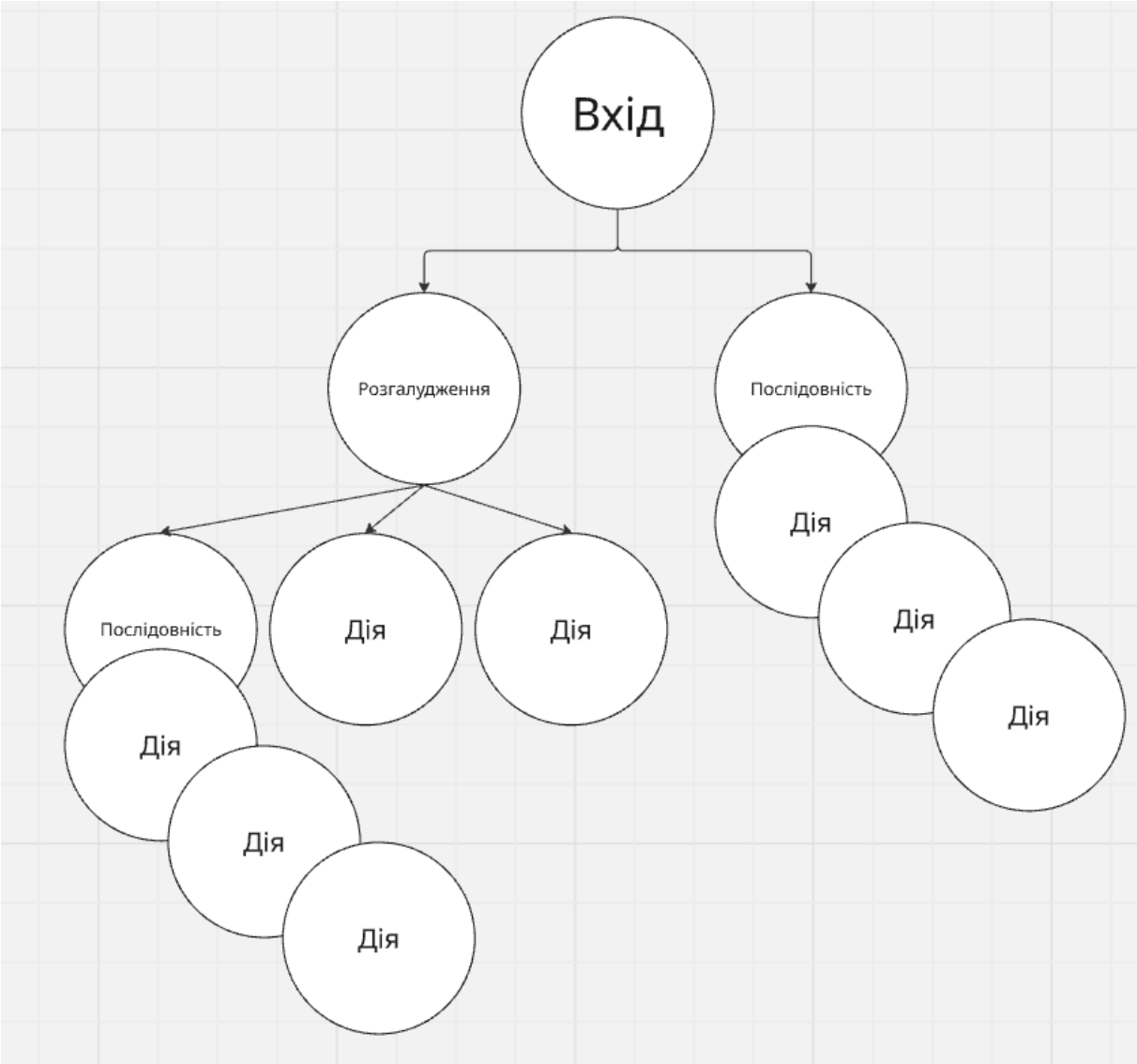

Рисунок 2.3 – Схема-приклад алгоритмів сімейства Behavior

Проте спосіб виконання таких дерев трохи відрізняється від графів – кожна нода (вона же ж вузол) повертає значення, відповідне статусу виконання поставленої перед нею задачі: успіх, провал та виконання. Для зрозумілості всі ноди виділені окремим списком:

- виконання дії;
- розгалуження;
- послідовність.

Для кращого розуміння є необхідність окремо визначити та розглянути особливості виділених типів.

Найпростішою для розуміння складовою графу є дія. Дія завжди є листом та в процесі свого виконання може повертати всі значення, залежно від стану. Після старту значення завжди знаходиться в стані «виконання», до моменту першої перешкоди (відповідно повертає «провал» та одразу припиняє подальше виконання дії) чи коли доходить до кінця (повертає «успіх» та припиняє подальше виконання, опційно скидаючи власний стан назад до початкового).

Розгалуження має в собі список листів нижчого рівня та поступово пробує запускати їхнє виконання. Після старту знаходиться в стані «виконання», проте як тільки змінюється стан поточного листа в черзі виконання, відбувається відповідна перевірка: якщо лист повернув «успіх», тоді виконання наступних листів переривається та саме розгалуження повертає «успіх»; якщо лист повертає «провал», тоді стан «виконання» залишається, але стан листу скидається до початкового і в обробку береться наступний. Якщо розгалуження проходить по всім листам та всі вони повертають «провал», тоді і саме розгалуження повертає «провал».

У розгалуження є ще декілька підвидів, наприклад зважене, випадкове та циклічне розгалуження. Зважене розгалуження перед запуском виконання листів сортує їх за пріоритетністю, що дозволяє перевіряти можливість та необхідність виконання важливих дій першими. Випадкове дозволяє досягти зворотнього результату – перед запуском виконання листи перемішуються у випадковому порядку і на основі цього вже стартують своє виконання. Циклічне в свою чергу вносить модифікацію в базовий потік виконання: після отримання «успіху» розгалуження скидає стан всіх листів та запускає їхнє виконання знову, залишаючись в стані «виконання» по суті вічно. Це дозволяє створювати нескінченні ланцюжки дій, які повинні повторюватися протягом всього часу. Така поведінка найбільше підходить нодам входу, проте з ними треба бути обережним, так як потік виконання по суті ніколи не можна зупинити.

Аналогічно до розгалуження, послідовність має в собі список всіх листів нижнього рівня. Умови стану мають свої особливості. Так як і всіх нодах до цього, після старту одразу переходить в стан «виконання». Проте обов’язкова умова послідовності – це успішне виконання всіх дій зі списку. Що приводить до наступних умов: якщо хоча б одна дія зі списку повертає «провал», тоді вся послідовність повертає «провал» та скидає стан листів; якщо дія повертає «успіх», тоді виконання переходить до наступної дії. Так повторюється, допоки список дій не закінчиться, і в кінці повертається «успіх».

До послідовності також можна виділити циклічний підвид. Аналогічно циклічній розгалуженості, воно повторюється нескінченно, проте умови циклічності всередині трохи відрізняються: якщо виконання доходить до кінця та повертає «успіх», тоді скидаємо стан та починаємо спочатку; якщо на будь-якому етапі отримуємо «провал», тоді знову скидаємо стани та починаємо спочатку.

Поєднуючи всі описані види нод можна досягати надзвичайних результатів. Алгоритми Behavior надають розробникам всі переваги графів та поєднують в собі потужність дерев і легкість візуального програмування. З цими алгоритмами можна дослівно перевести технічну мову очікуваної поведінки, прописаної геймдизайнерами, в мову візуальних з’єднань, швидко узгодити деталі, виправити на діаграмі і по досягненню порозуміння за короткий час перетворити діаграму на сам алгоритм.

Зважаючи на перераховані аргументи, Behavior алгоритми можна віднести до середніх за складністю (через особливості імплементації) та таких, що надають середній контроль над діями і результатом.

Останніми в черзі аналізу будуть найбільш просунуті алгоритми когнітивних моделей, найяскравішим представником яких є GOAP – Goal Oriented Action Planning.

Найчастіше для ігрових систем GOAP буде занадто складним рішенням, якого вони не потребують. Алгоритм дозволяє задавати цілі та

дії, і на основі передумов та наслідків збирає план для досягнення найбільш пріоритетних цілей. Проте саме це рішення є найближчим до нейронних мереж у звичайному розумінні. І, варто відзначити, видає одні з найкращих результатів, створюючи неочікувані плани дій, які будуть створювати цікавість гравцеві. Але краще спочатку розібратися зі складовими алгоритму, які зображені на рисунку 2.4.

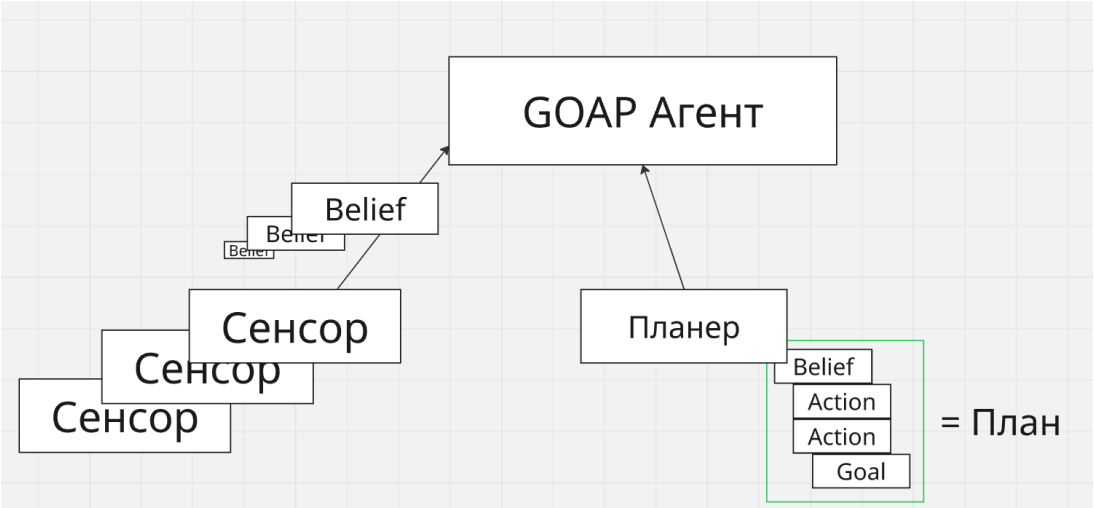

Рисунок 2.4 – Схема вигляду GOAP

В алгоритмі відокремлюється Агент. Агент – це сутність, яка задає запит на розрахунок цілі та виконує обрані дії.

Наступними будуть Beliefs – знання Агенту про свій стан та стан навколишнього світу. Наприклад кількість здоров’я, позиція ворога, точки відпочинку тощо.

Агент має список Actions, дій, які він може виконати в будь-який момент свого існування. В свою чергу Actions мають два списки Beliefs. Перший відповідає за передумови – яким повинен бути стан світу чи Агенту для того, щоб мати змогу виконати цю дію. Наприклад для дії «Відновити здоров’я» Агент повинен мати менше 15% здоров’я. Другий відповідає за наслідки цієї дії – як зміниться стан Агенту чи світу після (або в процесі) виконання дії. На тому ж самому прикладі з відновленням здоров’я, агент

впевнений, що після виконання дії його одиниці здоров’я піднімуться до 75%.

Останньою складовою алгоритму є Goals – цілі, яких повинен досягати Агент. По суті своїй, цілі – це обгортка над Beliefs, так як вони описують бажаний стан агенту чи навколишнього світу та задають пріоритетність досягнення цього стану. Далеко не відходячи від прикладу з відновленням здоров’я, у Агента є ціль «Бути здоровим», з Belief «очки здоров’я більше 50%» та найвищим пріоритетом.

На основі цих складових робиться сам алгоритм планеру. Його реалізація не є простою, проте порядок дій можна описати так:

- відсортувати цілі за пріоритетністю;
- взяти першу ціль зі списку;
- перевірити виконання її умов;
- якщо умови не досягнуті – обрати дію, наслідки якої змінюють стан до бажаного;
- перевірити умови виконання дії;
- якщо умови виконання дії також не досягнуті – обрати наступну дію, наслідки якої задовільнять передумови поточної;
- повторювати поки не закінчяться дії чи цілі.

Такі прості кроки дозволяють алгоритму обирати неочікувані дії для досягнення цілі та швидко, навіть в процесі виконання, змінювати свою поведінку. Це є одним з найбільших плюсів алгоритму. Проте складність імплементації приводить до висновку, що алгоритм належить до категорії складних, з відсутністю контролю над діями.

Варто підбити підсумок цього розділу. Серед ринкових рішень наразі можна виділити три основні способи вирішення проблеми симуляції штучного інтелекту: Машина Станів, Дерева Поведінки та Планування Дій Орієнтованих на Досягнення Цілі. Кожен з цих алгоритмів представляє свою власну категорію складності та контролю: легкість та повний контроль Машини Станів, середня складність та контроль Дерев Поведінки,

надзвичайна складність та відсутність контролю Планування Дій Орієнтованих на Досягнення Цілі.

Плюси Машини Станів:

- легка реалізація;
- повний контроль;
- чітке відображення послідовності та залежності дій.

Мінуси Машини Станів:

- неможливість розширення для великих систем;
- необхідність визначати умови переходів для всіх станів.

Плюси Дерев Поведінки:

- легка реалізація;
- легкість сприйняття;
- легко модифікувати поведінку чи додавати нову.

Мінуси Дерев Поведінки:

- реалізація алгоритму видається складною для новачків;
- на занадто великих гілках можуть початися проблеми з продуктивністю.

Плюси Планування Дій Орієнтованих на Досягнення Цілі:

- швидкість виконання;
- надзвичайна легкість модифікації поведінки;
- неочікувані стратегії.

Мінуси Планування Дій Орієнтованих на Досягнення Цілі:

- надзвичайно складна реалізація;
- складність розуміння алгоритму;
- відсутність контролю над діями.

2.2 Аналіз підходів реалізації алгоритмів

Отже, попереднім пунктом було визначено три алгоритми, які будуть далі аналізуватися: State Machine, Behavior Tree, GOAP. Тепер необхідно

проаналізувати класичні підходи їхньої реалізації та виділити їх слабкі сторони, щоб запропонувати кращі рішення.

2.2.1 Аналіз класичної реалізації State Machine

Класичним підходом до реалізації Машини Станів є прямолінійна імплементація відповідного патерну. Можна повернутись до рисунку 2.2 та більш детально проаналізувати його.

Агент – це клас, який має в собі деякі знання про світ та власні поля. Серед цих полів знаходиться посилання на об’єкт машини станів. В ігровому процесі, під час змін в навколишньому середовищі, Агент передає ці зміни в саму State Machine та чекає на її результат. Всередині Машини Станів відбуваються наступні дії:

- перевірка, в якому стані агент знаходиться зараз;
- пройтись по всім переходам;
- визначити, умови переходів, які виконуються;
- перервати виконання дії поточного стану;
- перевести агент в наступний стан;
- запустити виконання дії нового стану.

На словах та схемі, це все звучить просто. Проте, нажаль, класична імплементація змушує визначати умови переходу для всіх станів. Тобто, навіть якщо перехід неможливий та ніколи не відбудеться, нам необхідно визначити йому умову «завжди не виконується».

Такий підхід приводить до величезної кількості сміття та призводить до надскладних рівнів залежностей. Наприклад, за цим принципом модифікований рисунок 2.2 виглядатиме так, як зображено на 2.5.

Визначивши проблему, пропонується перейти до шляхів її вирішення.

Одним зі способів, який імплементований майже в кожне ринкове рішення, є введення неможливого переходу за замовчуванням. Це частково вирішує проблему: тепер у розробників немає необхідності задавати його

власноруч і рисунок 2.5 повертається в чистіший стан 2.2. Проте на цьому можна не зупинятися та додатково створити спеціальний віртуальний стан «будь-який стан». Цей віртуальний стан найкраще показаний на рисунку 2.6, скріншот з Unity.

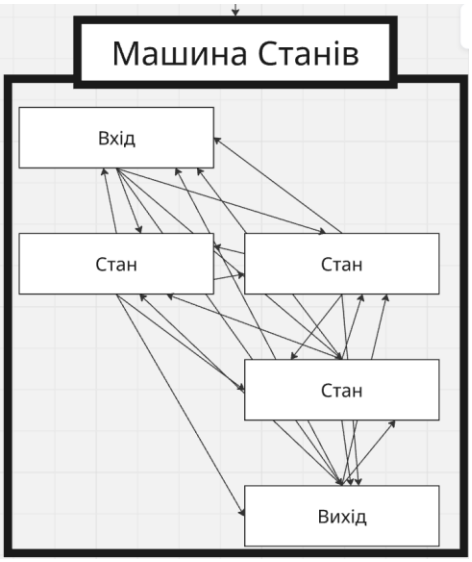

Рисунок 2.5 – Повна схема переходів між станами

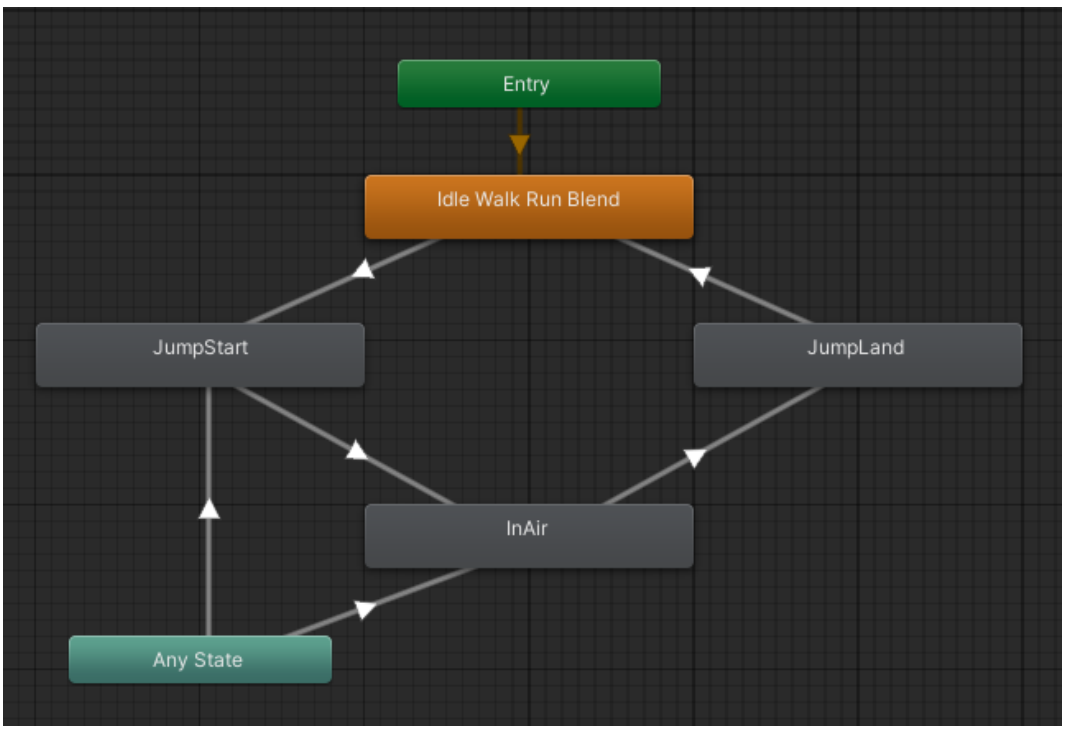

Рисунок 2.6 – Віртуальний стан Any State

Any State існуватиме лише для зручності. На знімку екрану з Unity можна побачити, що будь-який стан з наявних може перейти в стан JumpStart та InAir. Це дозволяє позбутися додаткової складності і необхідності створення переходів від станів Idle Walk Run Blend та JumpLand до вказаних станів. Таким чином, на моменті налаштування агенту до запуску гри, можна створити умови лише один раз, а потім система автоматично буде розгортати цей Any State назад в правила переходів стану «від кожного до кожного».

Проте за таким підходом розробник не прибирає основну незручність та проблему, обов’язкові переходи, а просто приховує її, обгортаючи для власного почуття краси та зручності. В цьому немає проблем, якщо мислити в межах парадигми об’єктно орієнтованого програмування. Саме тому пропонується перейти на абсолютно іншу парадигму Data-driven підходу. Якщо точніше, то хочеться поговорити про Entity Component System, скорочено ECS.

2.2.2 Аналіз ECS підходу до реалізації State Machine.

Для розуміння, як саме тут допоможе ECS пояснення почнеться з розповіді про парадигму. Сама Entity-Component-System [10] складається з трьох частин, які можна виділити з назви:

- Entity – сутність. Аналогічно ігровому об’єкту – це просто контейнер для компонентів. Проте GameObject дає набагато більше доступу та зберігає в собі як дані, так і функціонал. Це Створює додаткове навантаження. На противагу цьому Entity своїм існуванням задає виключно існування компонентів;
- Component – компонент. Компонент в ECS світі, на відміну від ООП, має в собі виключно дані. За рахунок цього відкривається прекрасна можливість оптимізації – розділення даних та функціоналу;

- System – системи. Дані з компонентів все одно повинні бути інтегрованими в якусь логіку. Як раз за це і відповідають системи – зібрати сутності з певним типом компоненту, провести ітерацію логіки, повернути сутність та компонент назад для обробки іншими системами.

Тепер можна повертатися назад до аналізу алгоритму. В ООП розробники представляють кожну сутність, та в деяких випадках навіть взаємодію між сутностями, у вигляді об’єктів, які тримають в собі і реалізацію, і дані. Як вже обговорювалося вище, ECS розділяє дані та їхню обробку. Тобто, якщо в ООП світі кожен стан мав свою стратегію (якусь дію, тобто обробку даних) та дані, ECS пропонує розділити це. І результат виглядає доволі просто – системи беруть на себе відповідальність за контроль над станом об’єкту та визначають в який стан він перейде далі. Як вже згадувалося вище, для роботи системі необхідні Entity з певними компонентами. Отже, розробник може використати компонент як флаг, в якому стані знаходиться зараз об’єкт. Це надає значну перевагу над ООП підходом – немає необхідності визначати перехід станів «кожен в кожен». Ба більше, це дозволяє обробляти один об’єкт декількома системами, що може бути як проблемою, так і додатковою перевагою над ООП реалізацією. Таким чином фінальну архітектуру ECS State Machine можна буде описати як: сутність; компоненти відповідного стану; системи, які визначають поведінку сутності з певним компонентом. Пройдемось трохи детальніше.

Як вже казалося в описі парадигми, сутність є просто контейнером для компонентів.

Самі компоненти – це об’єкти, які зберігають дані про сутність.

Системи, в цій архітектурі, беруть на себе роль обробки об’єктів певного стану та переведення їх в наступний за досягнення відповідних умов переходу, які також задаються всередині системи. Переважно умовами буде наявність чи відсутність якогось конкретного компоненту, а сам перехід ставатиметься шляхом прибирання компоненту поточного стану та додавання компоненту наступного стану.

Такий підхід вирішує найбільшу проблему Машини Станів – масштабованість та зручність. Отже, основна ціль була досягнута і можна переходити до наступного алгоритму.

2.2.3 Аналіз звичайної реалізації Behavior Tree

Behavior Tree містить в своїй назві «Дерево». Це наводить на цілком правильну думку – імплементація алгоритму буде схожою на реалізацію будь-якого іншого дерева в мовах програмування. Для цього необхідно представити предметну задачу у вигляді структури вузлів, нод. Кожен вузол матиме в собі посилання на материнський, тож таким чином можна будувати рекурсивні виклики від листя до кореня.

Проте така реалізація йде в розріз з ідеєю потоку виконання симуляції штучного інтелекту – зручніше, простіше та зрозуміліше йти згори вниз. Також на треба визначати розгалуження та йти в судженнях від кореня до листа. Саме тому для Поведінкових Дерев набагато краще використовувати абсолютно інший підхід, зображений на рисунку 2.7.

Аналогічно до Машини Станів, в системі є Агент – сутність, яка має в собі знання про свій стан та стан навколишнього середовища. Серед полів об’єкту Агента є посилання на Поведінкове Дерево.

Коли в світі, чи в стані Агенту, щось змінюється – він пропускає свої знання через поведінкове дерево та очікує на результат. В свою чергу дерево виконує код від кореня, поступово проходячи шлях від всіх розгалужень до відповідного листа дії. Проте як це відбувається? Кожна нода має в собі список всіх дітей. За принципом стану кожного вузла, який описувався в аналізі алгоритму, корінь запускає виконання. Тобто, дерево просто ітеративно проходиться по всім вузлам, які є в його списку, та пробує виконувати їх, до моменту, поки не повернеться «успіх» або «провал».

Рисунок 2.7 – Уточнена схема Поведінкового дерева

Теоретично це звучить легко, проте на моменті програмної імплементації виникне багато архітектурних деталей, які додатково потребуватимуть обговорення у відповідному розділі. На відміну від машини станів, рішення для якої поєднує в собі зміну парадигми та від цього повну зміну підходу до імплементації, особливість методу Поведінкових Дерев буде виключно в кодовій імплементації.

2.2.4 Аналіз звичайної реалізації GOAP

Канонічна реалізація GOAP якнайкраще зображує класичний ООП підхід до вирішення проблеми. Абсолютно кожна складова алгоритму зображується у вигляді об’єкту – Belief, як поточний стан світу чи Агенту;

Action, як дія, яку Агент потенційно може запустити за виконання умов списку Preconditions та яка можливо принесе наслідки зі списку Consequences; Goal, як ціль, яка по суті є обгорткою над Belief, що додатково задає пріоритет та описує стан світу чи Агенту, який хоче отримати в результаті. Таким чином різні абстракції постійно зображуються одним представленням, будь-то дані, як фактичні, так і очікувані, чи дії на основі та в сторону тих даних.

Засновуючись на цій концепції можна уявити GOAP як конструктор, який збирає логічний ланцюжок дій підлаштовуючись під вимоги та стан. Деталями цього конструктору, як зображено на рисунку 2.8, будуть дії та цілі.

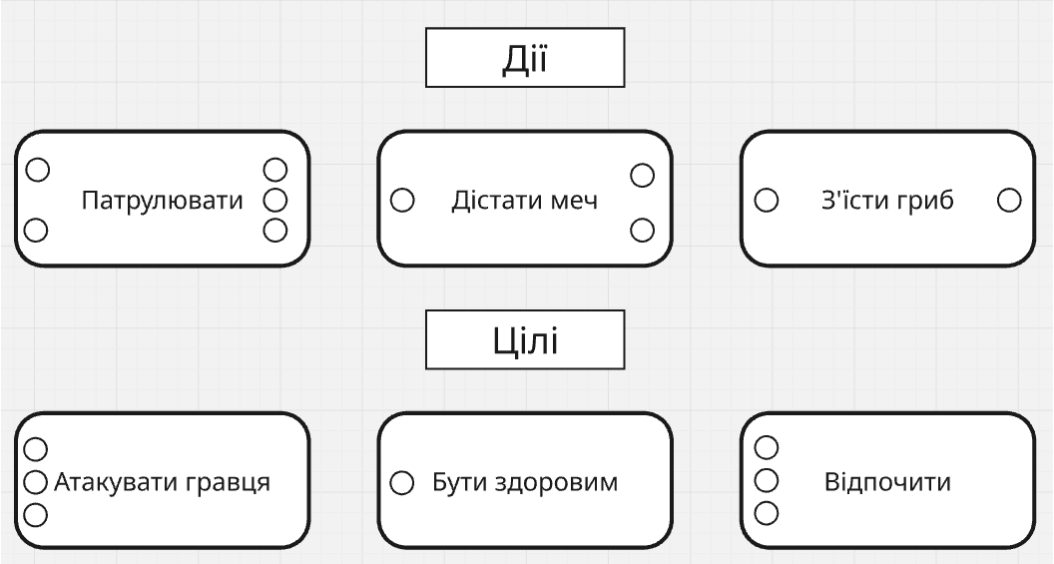

Рисунок 2.8 – Візуальна інтерпретація складових частин GOAP

Входи, які зображені кружечками на лівій стороні, – це списки очікуваних станів (Belief) Агенту та/або світу. В свою чергу виходи, відповідно кружечки на правій стороні, – списки очікуваних змін (Belief) в стані Агенту чи світу. Алгоритм обирає цілі та формує ланцюжок дій для кожної так, щоб списки (кількість входів та виходів) збігалися. Наприклад,

для рисунку 2.8 можна зробити декілька планів, залежно від стану Агенту та навколишнього світу, і це зображено на рисунку 2.9.

Коротко опишуться деталі до прикладу з рисунку. Агент має знання, що зараз його здоров’я менше 50%, отже є необхідність відновити його стратегією «з’їсти гриб». Після виконання цієї дії він досягає цілі «Бути здоровим.». Аналогічним чином працюють всі інші плани з рисунку: Агент має впевненість, що тепер його здоров’я більше 75% та витривалість більша 50%, тож він може перейти в режим патрулювання для досягнення цілі «Відпочинок». Проте за умови «Гравець поруч» Агнет може дістати меч та результат патрулювання буде «Атакувати гравця».

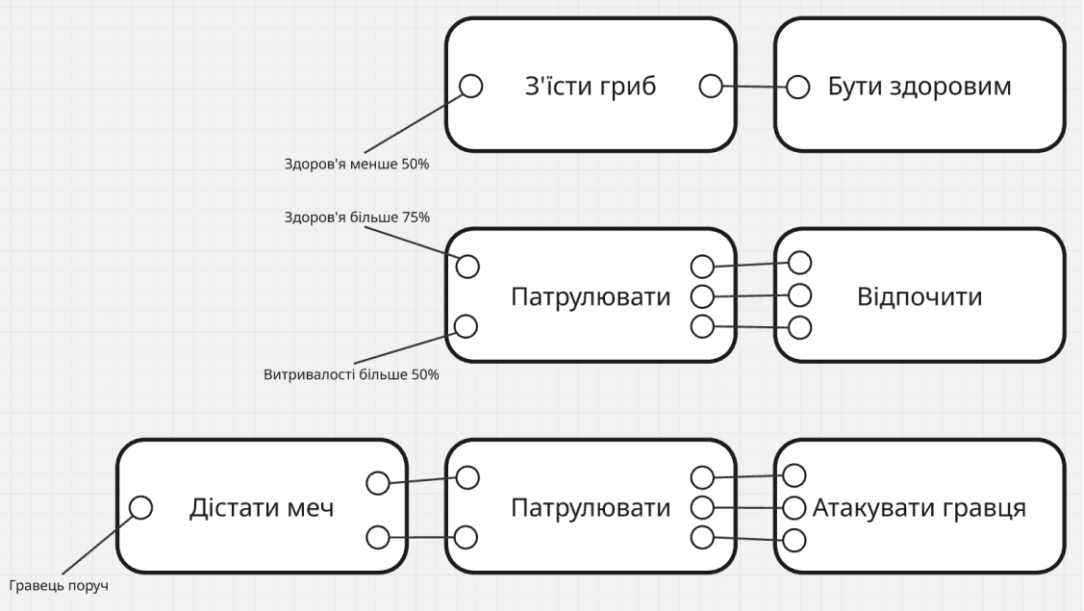

Рисунок 2.9 – Приклад планів, які може зібрати GOAP

Класичний метод реалізації звучить майже ідеально, до одного моменту – GOAP зобов’язаний пройтись по всім цілям. Так, цю проблему можна легко вирішити перериванням алгоритму в момент, коли алгоритм зміг зібрати план для цілі. Якщо їх перед цим відсортувати в порядку пріоритетності, тоді Агент виконуватиме ціль, яка матиме найбільший вплив в поточному моменті. Нажаль, все не так просто.

Тут виникає проблема складності дії. Кожен Action має свою вагу – умовне відносне значення складності виконання цієї дії. Наприклад, це може описувати складність у вигляді затратності ресурсів, час виконання дії тощо. В деяких ситуаціях алгоритм може зібрати план, який буде виконувати пріоритетну ціль, але перший доступний шлях її виконання буде надто важким. Тож для логічності та легкості треба буде обрати іншу, менш важливу ціль, план якої важитиме менше.

Проте прорахування всіх планів на головному процесорі може бути занадто ресурсно затратно – алгоритму необхідно пройтись по всім цілям та діям, перевірити всі умови, створити з цього плани, та перевірити чи оптимально буде виконувати цю ціль за отриманим планом. Саме тому треба знайти та запропонувати інший підхід до створення планів.

2.2.5 Аналіз реалізації GOAP за допомогою Compute Shader

Для реалізації GOAP було вирішено обрати підхід Compute Shader’ів. Проте перед тим, як поринути в особливості та плюси реалізації алгоритму цим підходом, пропонується розібратися, що це таке.

Отже, тепер можна повертатися до того, як саме пропонується перенести принципи роботи з обрахунковими шейдерами на алгоритм Планування Дій Орієнтованих на Цілі.

Основна концепція залишатиметься незмінною – все так же ж буде Агент, списки Дій, Знань та Цілей. Агент буде передавати їх на алгоритм планінгу GOAP, проте тепер сам алгоритм буде перенесений на підхід, схожий до ECS. Якщо точніше, знову йтиме використання Data-Driven архітектури.

В розділі з реалізацією буде пояснено, чому було обрано для всіх алгоритмів Unity та C#, але поки пропустимо це пояснення. Так як Compute Shader не підтримують посилальні типи даних, то для реалізації алгоритму треба виділяти дані з класів-об’єктів Belief, Action, Goal та Plan і пакувати

їх у структури. Наступним кроком дані об’єднуються у структури даних, контейнери, які передаються на відеокарту, де вже йде обробка та потім повертається назад результат. Сам алгоритм планеру майже не зміниться, лише рекурсія, яка присутня в класичному підході, переведеться на циклічно-ітеративну операцію.

Таким чином отримується алгоритм, який буде в паралель запускати обрахунок плану для всіх цілей, і як результат повертати одну ціль з найкращим планом. Це вирішує проблему довгого та дорого вартісного обрахунку плану на головному процесорі, та переводить це на максимально швидкі операції обробки умов та наслідків на відеокарту.

3 РЕАЛІЗАЦІЯ АЛГОРИТМІВ СИМУЛЯЦІЇ ШТУЧНОГО ІНТЕЛЕКТУ В КОМП'ЮТЕРНИХ ІГРАХ

Отже, закінчивши з аналітичною частиною та пояснивши всі алгоритми і парадигми, можна переходити до пояснення імплементації алгоритмів. Проте перед цим, треба обрати інструментарій, за допомогою якого це робитиметься.

3.1 Вибір інструменту реалізації

Для розробки ігор наразі рідко хто пише рушії власноруч. Набагато легше, дешевше та безпечніше обрати вже готовий рушій, який міститиме в собі необхідний функціонал для відображення об’єктів, базової взаємодії між ними та додаткових інструментів, які спрощуватимуть розробку надалі.

Треба обрати ігровий рушій, який зможе підтримувати одночасно декілька парадигм (ООП та обидві Data-Driven, як у вигляді ECS, так і у вигляді Compute Shader). На щастя, інструмент, з яким автор працює вже більше 5 років, вже має підтримку та власні пакети для цього. Мова йде про рушій Unity.

Окрім того, що Unity підтримує всі необхідні технології, можна виділити дуже багато її плюсів:

- використання мови програмування C#. Мова програмування C# – це об’єктно орієнтована мова програмування. Проте, не дивлячись на величезну кількість можливостей, у вигляді синтаксичного цукру та особливостей самої мови, C# залишається однією з найлегших мов для початківців. А за отримання досвіду у використанні цієї мови вона розкривається як надпотужний інструмент, з яким приємно працювати;
- офіційна підтримка. Служби підтримки самої Unity дуже доброзичливі та завжди стараються допомогти з вирішенням проблеми чи питання в найкоротші терміни;

- підтримка спільноти. Спільнота Unity – наразі одна з найбільших та налічує мільйони постійних користувачів. За роки існування рушія незалежні розробники створили неймовірну кількість рішень для Unity. Незліченна кількість проєктів стала найкращою перевіркою та загартувала сам рушій. Протягом років розробки спільнота створила величезну кількість обговорень на форумах, які досі є актуальними та допомагають визначати помилки і способи їх вирішення;
- наявність власного ECS рішення. Unity вже довгий час займається розробкою власної ECS системи під назвою DOTS – Data Oriented Technology Stack. DOTS наразі є однією з найкращих інтеграцій парадигми Data-Driven в світ самого рушія. І це є критичною точкою для цієї кваліфікаційної роботи;
- наявність повноцінної інтеграції з Compute Shaders. Цей пункт є об’єктивно надзвичайним дивом, враховуючи нішевість ситуацій, в яких з’являється необхідність використання комп’ют шейдерів. Проте Unity надає надзвичайно зручне API для роботи з цією технологією, кросплатформеність цього рішення та зрозумілу і добре задокументовану структуру роботи.

Зважаючи на все вище перераховане, стає очевидно, що для подальшої розробки використовуватиметься Unity та мову програмування C#.

3.2 Імплементація Машини станів

Отже, першим алгоритмом на черзі імплементації стає State Machine через ECS. Як вже згадувалося вище, Unity надає для цієї задачі пакет DOTS, на базі якого вестиметься подальша розробка цього алгоритму.

Пропонується почати з визначення, що саме робитиме Агент в цій симуляції. Тримаючи в голові, що Машина Станів є простим та обмеженим алгоритмом, було визначено для нього три цілі: відпочити, знайти шлях до

точки, піти до точки. Не дивлячись на простоту цих пунктів, це буде ідеальним способом показати потужність DOTS та запропонованої системи.

3.2.1 Класичний спосіб створення об’єктів

Першим пунктом буде створення та налаштування Агенту. Для цього необхідно розібратися в тому, як взагалі створюються об’єкти в Unity.

Після створення порожнього проєкту розробника зустрічає інтерфейс, зображений на рисунку 3.1.

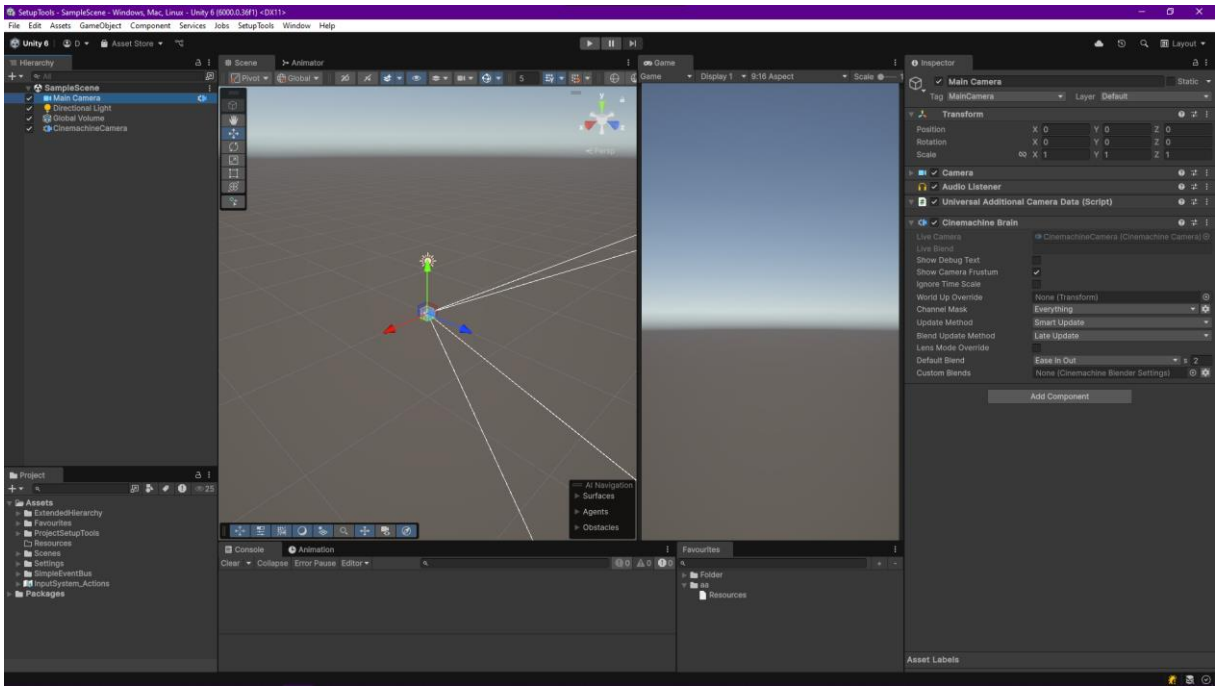

Рисунок 3.1 – інтерфейс Unity

Все це називається редактором. Редактор розбитий на вікна. Швиденько проясниться суть кожного:

– ієрархія (Hierarchy). Це вікно відображає, які сцени зараз відкриті, наявність об’єктів та їхню залежність одне від одного;

– вид сцени (Scene View). Це вікно дає редакторну камеру вільного польоту, та попередній перегляд об’єктів на сцені. За допомогою

невеликих налаштувань це вікно може показувати ґізми – віртуальні об’єкти для більш зручного орієнтування у ігровому світі. Це класична сітка, ручки для зміни координат об’єкту, а також границі 3D моделі та поля зору камери. Також вид сцени дозволяє передивлятися анімації та роботу Surface Shaders;

- вид гри (Game View). Це вікно можна перемикати між станом симулятора та більш класичним, який зображений на рисунку 3.1. Суть цього вікна – змоделювати ситуацію, як гравець бачитиме гру;
- проєкт (Project). Вікно проєкту – це відображення файлів та папок, які знаходяться в проєкті і доступні для використання;
- консоль (Console). Можна вважати обмеженим терміналом, так як в це вікно дозволяється лише виводити повідомлення, через API, яке надає саме Unity;
- інспектор (Inspector). Після того, як розробник обирає об’єкт зі сцени, це вікно стає активним та відображає усі компоненти, що знаходяться на об’єкті.

Для розробників класичним та більш звичним способом є створення ігрових сутностей в самому редакторі, у вікні ієрархії. Для цього можна зробити правий клік на порожньому місці, та перейти у відповідне меню створення об’єкту. Цей процес зображений на рисунку 3.2.

Unity пропонує насправді багато способів створити об’єкт. Звичайно це можна зробити через код. Деякі типи файлів підтримують перетягування з папки у вікно сцени, наприклад 3D моделі чи 2D спрайти.

Створювати можна також через два контекстних меню – кнопка «+» ліворуч в ієрархії та через верхнє меню Assets. Кожен з цих способів є абсолютно прийнятним та часто є лише вибором самого розробника, як буде зручніше.

Важливо згадати, що можна додавати власний функціонал в ці контекстні вікна. Це дозволить одним кліком створювати однотипні об’єкти, або сутності з однаковим набором компонентів.

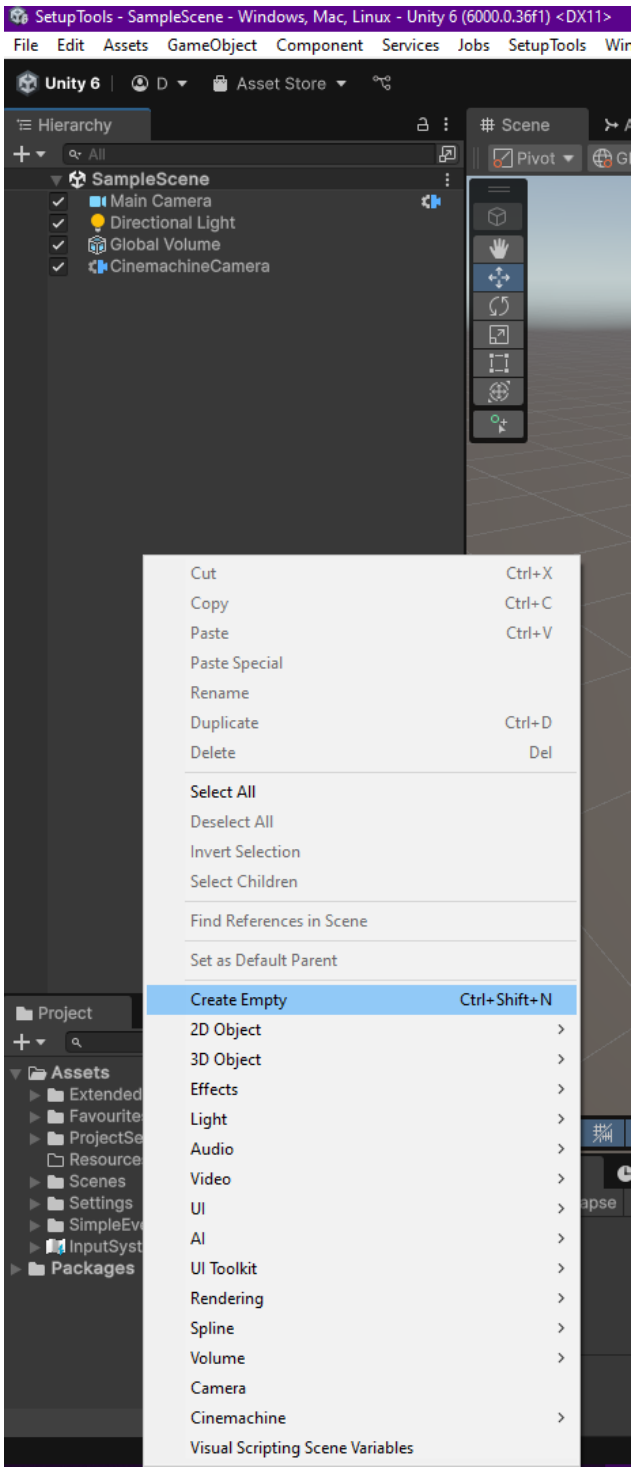

Рисунок 3.2 – Створення об’єкта в редакторі Unity

Після цього в нескінченній порожнечі редактора створиться об’єкт, який можна виділити лише у вікні ієрархії – це порожній об’єкт без будь-якого відображення. Він розглянеться детальніше, орієнтуючись на рисунок 3.3.

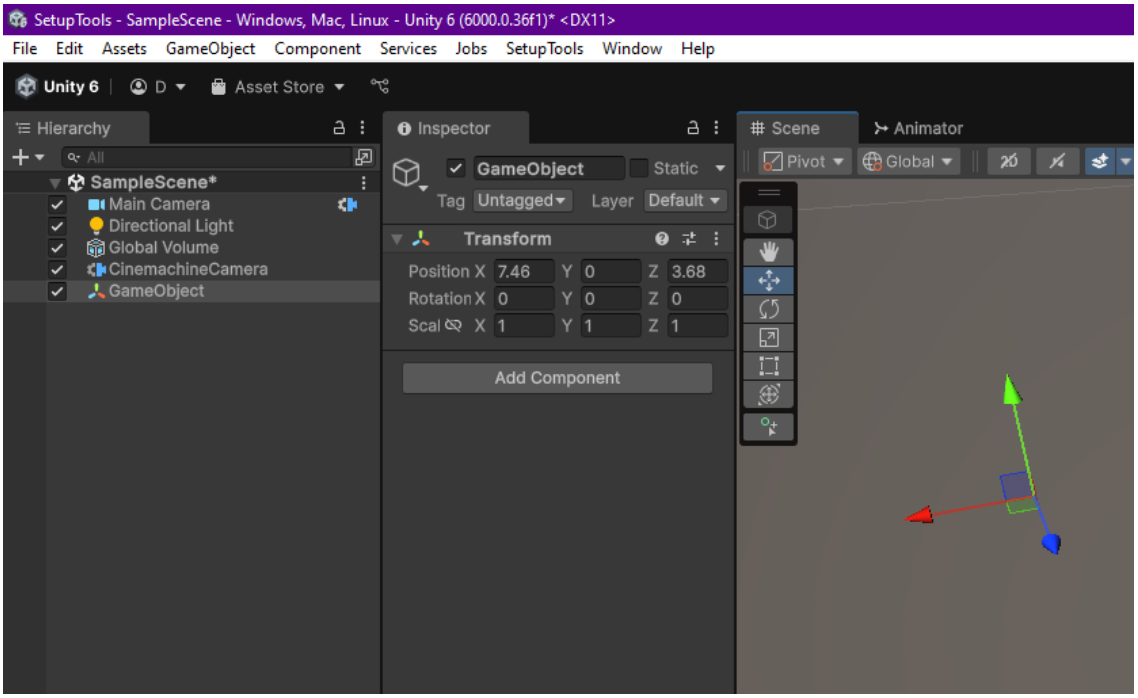

Рисунок 3.3 – Редактор після створення нового об’єкту

Новий об’єкт отримує назву GameObject. І це відображає, що саме створює Unity – ігровий об’єкт. Цей тип по суті є контейнером, для задання властивостей якого використовуються компоненти. Один з таких компонентів додається автоматично та є обов’язковим – Transform. Він описує позицію, розмір та значення повороту ігрового об’єкту в світі.

Unity надає 134 вбудованих компоненти, якими можна задати поведінку об’єкту. Це включає в себе всі можливі способи відображення, фізики, позиціонування в світі і так далі. Далі будуть використані деякі з них, для того щоб краще показати, як працює ця система. Наприклад, зробити так, щоб новостворений об’єкт можна було побачити, та провести фізичну взаємодію.

Для рендерингу об’єкта необхідно додати два компоненти: Mesh Renderer та Mesh Filter. Потім треба обрати 3D модель, яка буде відображатись та її матеріал. Для цього достатньо заповнити поле Mesh в компоненті Mesh Filter та Material в компоненті Mesh Renderer, приблизно як це зображено на рисунку 3.4.

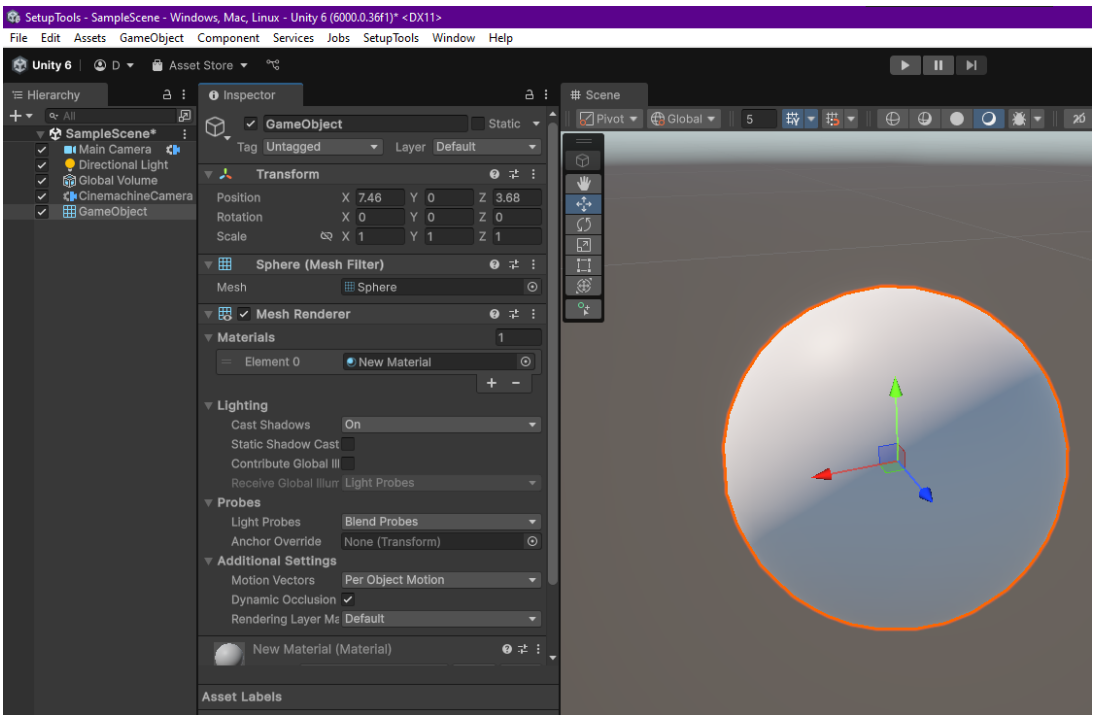

Рисунок 3.4 – Компоненти для відображення об’єкту

Перший крок виконаний, тепер можна додати фізичну взаємодію. По-перше для цього необхідно додати компонент Rigidbody, який відповідає за симуляцію фізики – як закони механіки діятимуть на цей об’єкт. Проте наразі цей об’єкт не має фізичної оболонки, тож він не буде взаємодіяти з іншими об’єктами в сцені. Для виправлення цього можна використовувати систему колайдерів, та додати відповідний колайдер до об’єкта. Тепер налаштування сфери з рисунку 3.4 виглядають так, як це зображено на рисунку 3.5.

Unity використовує фізичний рушій Physix, який маючи свої проблеми та особливості досі є хорошим наближенням та добре спрощує роботу з ігровою фізикою. Так, очевидно, що компонентна система дозволяє використовувати інші фізичні рушії, або навіть написати свій, але це не буде розглядатися в межах кваліфікаційної роботи. Класичної системи буде цілком достатньо для виконання поставлених задач перед всіма алгоритмами.

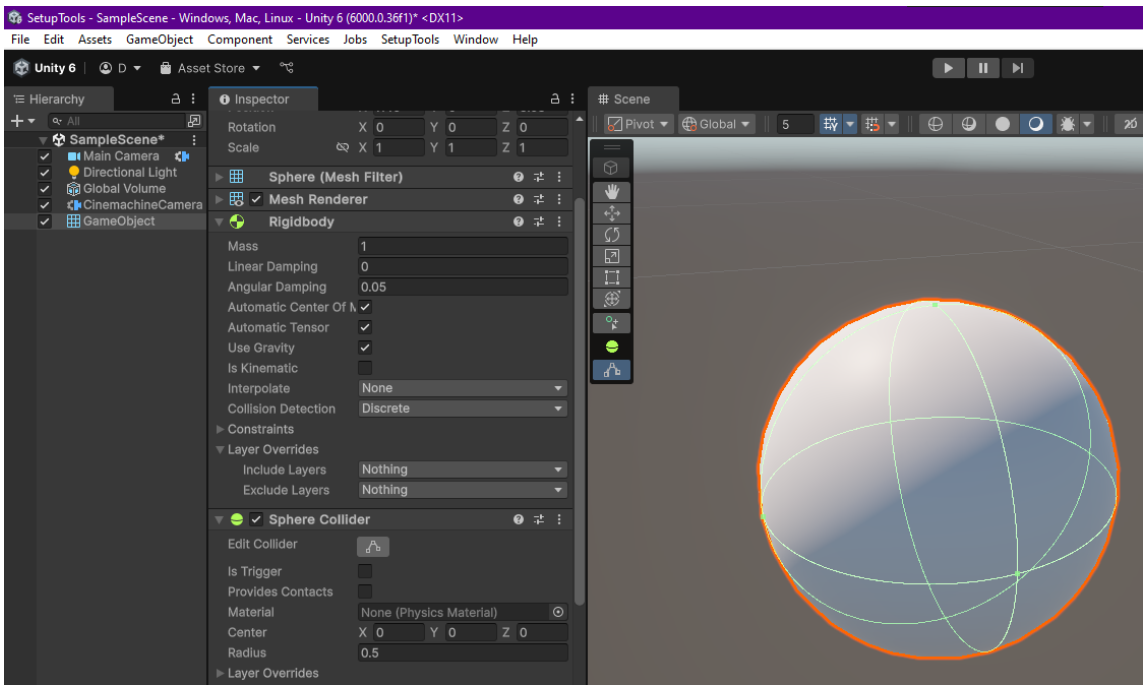

Рисунок 3.5 – Фізичні налаштування об’єкта

Отже, маючи базові знання про створення об’єктів в Unity, можна переходити до наступного кроку.

3.2.2 Створення об’єктів в DOTS

DOTS в свою чергу вносить зміни в цей простий процес створення об’єктів.

По-перше, розробник повинен створити під-сцену в межах обраної сцени. Під-сцени мають назву, яка збиває з пантелику. Це не спосіб організації об’єктів ієрархії. Під-сцена – це повноцінно окрема система, яка після запуску гри перетворює всі об’єкти, що знаходяться в ній, на ECS сутності. Цей процес називається запіканням.

По-друге, далі йде більш звичний потік – розробник може створити об’єкт, та надати йому властивостей відповідними компонентами. Але після запуску, за замовчуванням, в ECS сутність перенесеться лише відповідні компоненти для рендеру. Під капотом тут відбувається процес перенесення

звичайних Unity компонентів, які виконують всю роботу самостійно, об’єднуючи дані та функціонал, на ECS компоненти, які мають в собі виключно дані, та запуск відповідної системи. В свою чергу за процес конвертування компонентів відповідає спеціальний клас Baker. Для вбудованих компонентів Unity надає або неявне автоматичне перетворення (в основному компоненти рендерингу), або одразу з пакетом DOTS йдуть компоненти-запікачі, які необхідно додавати до кожного компоненту на ігровому об’єкті. Проте більшість часу розробник працює з самописними компонентами і повинен власноруч створювати Baker-клас, відповідну структуру, в яку будуть запікатися дані з компоненту, та виносити функціонал компоненту в систему.

3.2.3 Створення та налаштування ECS Агенту

Закінчивши з поясненням особливостей та відмінностей роботи зі створення об’єктів в ECS, нарешті можна переходити до створення Агенту.

Отже, як вже раніше зазначалося, агент матиме три стани-цілі: відпочинок (Idle), пошук шляху (Pathfinding) та рух до цілі (Movement). В розділі, де розбиралося вирішення проблеми Машини станів, було запропоновано спосіб визначення стану Агенту шляхом наявності-відсутності певного компоненту. Отже наразі стоїть ціль створити компоненти, які відповідатимуть цим станам.

Почнемо зі створення ігрового об’єкту, з яким йтиме взаємодія надалі. Спосіб створення описувався вище, тому пропонується одразу поглянути на результат, зображений на рисунку 3.6.

Тепер розглянеться сама ієрархія та компоненти. Агент складається з двох частин – батьківського контейнеру Mover_Root_0, який тримає на собі всі функціональні компоненти (інспектор 1 на рисунку 3.6), та ігрового об’єкту SM_Bean_Female_01(Clone), який по суті виконує лише роль

відображення (інспектор 2 на рисунку 3.6). Так, звичайно ніхто не заважає об’єднати ці дві сутності в одну, але це буде надто не зручно для роботи.

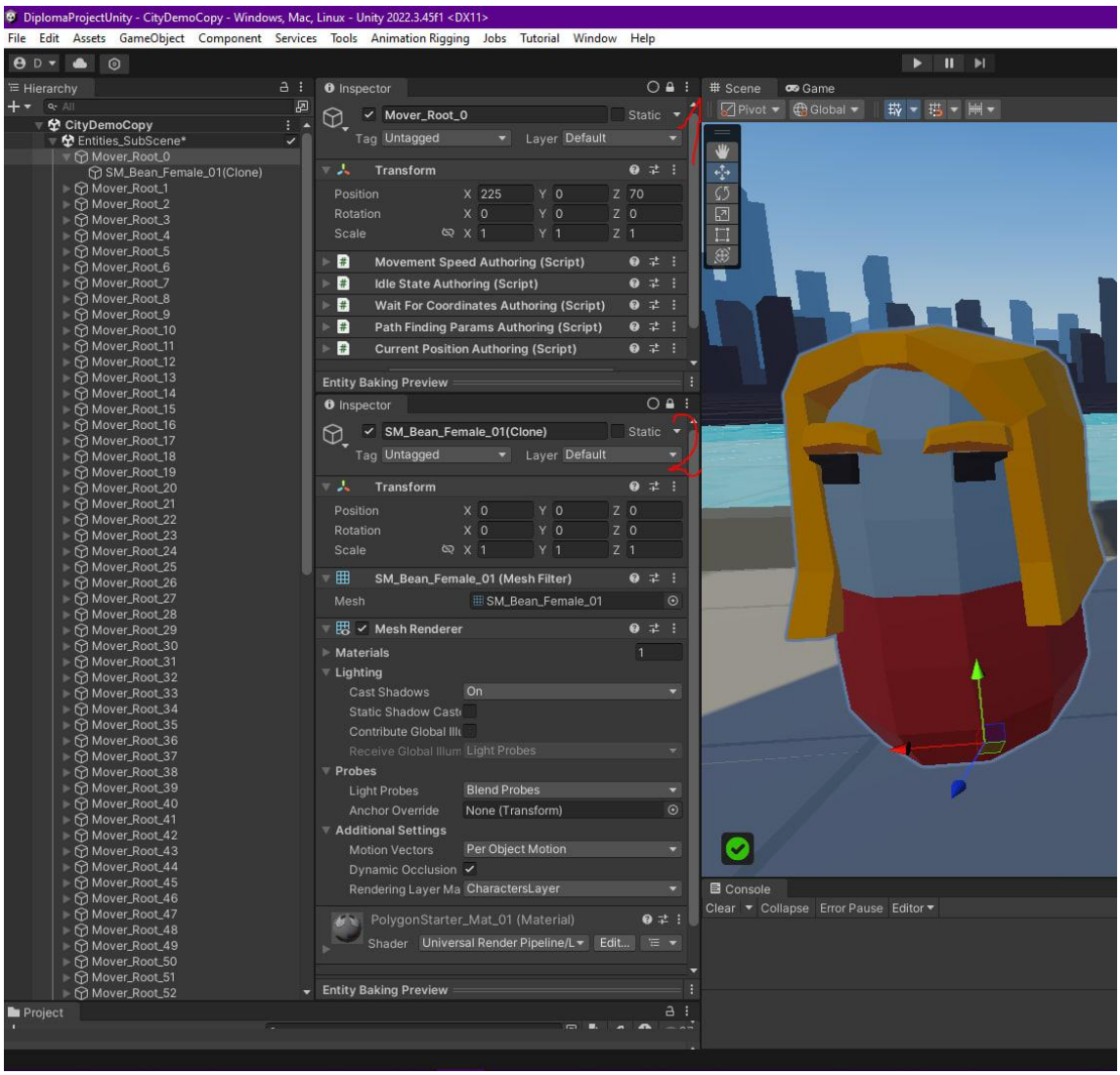

Рисунок 3.6 – ECS Агент

За замовчуванням Агент знаходиться в стані відпочинку, Idle. Нехай умовою виходу з цього стану буде досягнення внутрішнього таймеру 0. Отже, спочатку треба створити класичний компонент, який зберігатиме в собі час, та відповідний Baker, який перетворить це в ECS компонент. Цю частину можна побачити на лістингу 3.1.

Лістинг 3.1 – Створення та конвертування класичного компоненту в ECS

```
public class IdleStateAuthoring : MonoBehaviour
    {
        public class Baker : Baker<IdleStateAuthoring>
        {
            public override void Bake(IdleStateAuthoring
authoring)
            {
                var entity =
GetEntity(TransformUsageFlags.Renderable);
                AddComponent(entity, new IdleState()
                {
                    IdleTime = Random.Range(1, 2)
                });
            }
        }
    }

    public struct IdleState : IComponentData,
IEnableableComponent
    {
        public float IdleTime;
    }
```

При старті сцени (для простоти розуміння можна вважати, що це старт гри) внутрішня система Unity проходить та збирає всі компоненти, які мають цей Baker-клас, і конвертує MonoBehaviour компонент IdleStateAuthoring в IdleState ECS компонент. Останній, в свою чергу, має в собі значення часу, що визначає, скільки часу ця сутність повинна знаходитися в стані відпочинку.

Тепер стоїть задача створити систему, яка поступово зменшуватиме це значення таймеру. Реалізацію можна побачити на лістингу 3.2.

Лістинг 3.2 – ECS система для обробки Idle стану

```
public partial struct StateMachineSystem : ISystem
    {
        public void OnUpdate(ref SystemState state)
        {
```

Продовження лістингу 3.2

```
                var entityManager = state.EntityManager;

                // буде пояснено пізніше
                foreach (var (buffer, entity)
                         in
SystemAPI.Query<DynamicBuffer<PathFindingParams>>().WithEntity
Access())
                {
                    if (buffer.IsEmpty)
                    {

entityManager.SetComponentEnabled<IdleState>(entity, true);
                    }
                }

                foreach (var (idleState, entity)
                         in
SystemAPI.Query<RefRW<IdleState>>().WithEntityAccess())
                {
                    if (idleState.ValueRO.IdleTime >= 0)
                    {
                        idleState.ValueRW.IdleTime -=
SystemAPI.Time.DeltaTime;
                    }
                    else
                    {
                        idleState.ValueRW.IdleTime =
Random.Range(2, 4);

entityManager.SetComponentEnabled<IdleState>(entity, false);
     entityManager.SetComponentEnabled<WaitForCoordinates>(enti
ty, true);
                    }
                }
            }
```

В системі відбувається максимально проста логіка. Спершу треба дістати менеджер сутностей з API Unity. Перший цикл тимчасово буде опущений з пояснень, Другий цикл виконує роботу яка вже відноситься до цієї частини. Спершу система збирає всі сутності з компонентом IdleState. Далі простою перевіркою визначає наступні кроки: якщо у сутності час

очікування ще не менше нуля, тоді віднімається час, який зайняло очікування попереднього фрейму; інакше створюємо новий час очікування та менеджер сутностей вимикає цей компонент, одночасно вмикаючи інший. Цією рокіровкою досягається логіка зміни стану з одного на інший.

Після того, як Агент побув у стані відпочинку, можна переводити його в стан пошуку шляху, який відповідає компоненту WaitForCoordinates. Попередня система своєю логікою як раз перевела Агент в цей стан, увімкнувши компонент. Оголошення WaitForCoordinates та його Baker дуже схоже на лістинг 3.1, тому пропуститься. Натомість увага перейде одразу до системи, яка обробляє цей стан.

Пояснення почнуться з визначення цілі цієї системи: вона повинна обрати нові координати, в які рухатиметься Агент, взяти поточну позицію, знайти шлях від точки до точки та передати його агенту для стану руху.

За знаходження нової точки відповідає лістинг 3.3. Насправді, все класично просто: згенерувати випадкове число, яке буде відповідати індексу в масиві клітинок. Потім стоїть перевірка, чи можна стояти в цій клітинці: якщо так – цикл закінчується, фінальна позиція виставляється в отриману; якщо ні – продовжувати пошук.

Лістинг 3.3 – Знаходження нової точки

```
int2 endPosition;

                    while (true)
                    {
                        var index = random.NextInt(0,
tilemap.Length);
                        var tile = tilemap[index];
                        if (tile.Walkable)
                        {
                            endPosition = new int2(tile.X,
tile.Y);
                            break;
                        }
                    }
```

Отримання власної позиції Агента можна здійснити через інший компонент, який не включений у опис. Спосіб його оголошення такий же ж, як на лістингу 3.1, а спосіб використання доволі простий:

```
var position = currentPosition.ValueRO.Position;
```

Тепер відбувається виклик методу для розрахунку шляху, який можна побачити на лістингу 3.4.

Лістинг 3.4 – Метод пошуку шляху

```
public void FindPath(int2 startPosition, int2 endPosition, int2 gridSize, NativeList<int2> path)
            {
                var job = new FindPathJob()
                {
                    startPosition = startPosition,
                    endPosition = endPosition,
                    pathNodeArray = SetupGrid(endPosition, gridSize, tilemap),
                    gridSize = gridSize,
                    path = path
                };
                job.Run();
            }
```

Сам метод використовує мультипотокові розрахунки, збираючи необхідні данні в структуру FindPathJob. Далі запускає та очікує на кінець виконання. Для простоти тексту, опуститься частина з детальним описом алгоритму пошуку шляху. Достатньо буде знати, що система запускає самописний A*, який і виконує основну задачу.

По закінченню розрахунку шляху, система очищає буфер з координатами попереднього шляху (якщо там щось залишилося) та конвертує результат алгоритму пошуку шляху в більш конкретний для системи руху, заповнюючи буфер новими координатами.

Це можна побачити на лістингу 3.5.

Лістинг 3.5 – Заповнення буферу координатами

```
buffer.Clear();

                    for (int i = path.Length - 2; i >= 0; i--)
                    {
                        var pos = path[i];
                        buffer.Add(new PathFindingParams {
StartPosition = position, EndPosition = pos });
                        position = pos;
                    }
```

Тепер система може вимкнути компонент WaitForCoordinates і з заповненим буфером та вимкненими компонентами всіх інших станів, Агент переходить в стан руху. Отже можна переходити до розгляду системи руху. На диво це буде найскладніша та найдовша система з усіх наявних.

Почнімо з простого. Необхідно обробити ситуацію, коли буфер координат порожній, проте сутність має всі необхідні компоненти. В такому випадку достатньо просто пропустити її:

```
if (buffer.IsEmpty)
                    continue;
```

Тепер треба взяти першу координату з масиву, перевести її в світову позицію та розрахувати вектор, в якому необхідно рухати сутність. Цей процес описаний на лістингу 3.6.

Лістинг 3.6 – Отримання вектору руху

```
var targetPosition = buffer[0].EndPosition;
                    var worldTargetPosition =
TileMapUtils.TileToWorldPosition(targetPosition.x,
targetPosition.y).ToFloat3();

                    var moveVector =
math.normalize(worldTargetPosition -
transform.ValueRO.Position)
                                        *
SystemAPI.Time.DeltaTime * speed.ValueRO.Speed;
```

Далі необхідно перевірити, яка наразі відстань між обраною точкою, у світових координатах, та поточною позицією Агента. Якщо відстань менша певного граничного значення – тоді можна видаляти цю точку з буфера, додатково виставивши свою позицію у фінальну. Ця умова описана в лістингу 3.7.

Лістинг 3.7 – обробка умови знаходження біля точки призначення

```
    if (math.distance(worldTargetPosition,
transform.ValueRO.Position) <= THRESHOLD)
                    {
                        currentPosition.ValueRW.Position =
targetPosition;
                        buffer.RemoveAt(0);
                    }
```

Якщо ж відстань ще не досягла граничного значення, можна продовжувати рух. Для руху треба виконати дві речі: змінити позицію, змінити поворот, щоб дивитися в напрямку руху. Перша частина виконується легко: склали поточну позицію з вектором руху, записали в значення позиції. Це описано на лістингу 3.8.

Лістинг 3.8 – Зміна позиції

```
    var newPosition = transform.ValueRO.Position + new
float3(moveVector.x, 0, moveVector.z);
    transform.ValueRW =
transform.ValueRO.WithPosition(newPosition);
```

Тепер йде частина повороту. Спочатку нормалізуємо позиції призначення та поточної. Для цього достатньо буде ігнорувати у-координату їх обох. Лістинг 3.9 зображує цей процес.

Лістинг 3.9 – Нормалізація координат

```
var currentPosition3D = new
float3(currentPosition.ValueRO.Position.x, 0,
currentPosition.ValueRO.Position.y);
var targetPosition3D = new float3(targetPosition.x, 0,
targetPosition.y);
```

Тепер розраховуємо нормалізовану відстань до цілі та робимо перевірку: якщо всі компоненти цієї відстані дорівнюють нулю, тоді Агент прийшов в точку і немає необхідності змінювати поворот. Інакше обраховується кут повороту та виставляються значення, як зображено на лістингу 3.10.

Лістинг 3.10 – Обрахунок кут повороту

```
var directionToTarget = targetPosition3D -
currentPosition3D;

                        if (!math.all(directionToTarget ==
float3.zero))
                        {
                            var lookDirection =
math.normalize(directionToTarget);
                            var targetRotation =
quaternion.LookRotationSafe(lookDirection, math.up());

                            var currentRotation =
transform.ValueRO.Rotation;
                            var smoothedRotation =
math.slerp(currentRotation, targetRotation,
SystemAPI.Time.DeltaTime * 8);

                            transform.ValueRW =
transform.ValueRO.WithRotation(smoothedRotation);
                        }
```

З повною кодовою реалізацією можна познайомитись у додатках. А наразі – це вся реалізація алгоритму Машина Станів на ECS.

3.3 Імплементація Поведінкових Дерев

Отже тепер можна переходити до другого алгоритму – Behavior Tree. Нагадаю, що цей алгоритм віднесений до середніх за складністю та таких, що надають повний контроль. Зручність самого алгоритму дозволяє сильно розгулятися в цілях та реалізації. Тож пропонується спочатку визначити очікувану поведінку Агенту, намалювавши схему дерева вибору (зображена на рисунку 3.7).

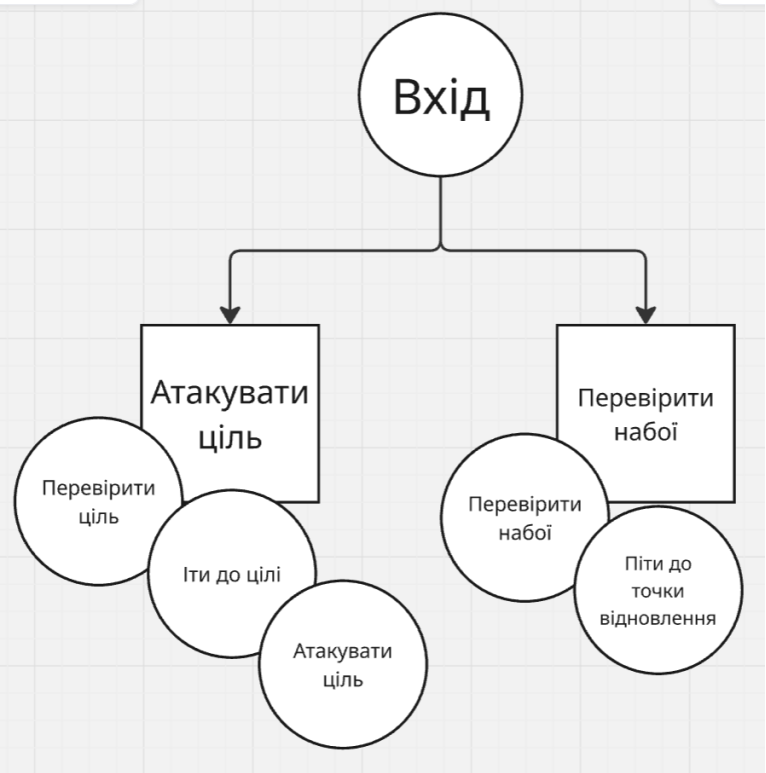

Рисунок 3.7 – Поведінка Behavior Tree Агента

Отже, аналізуючи схему, можна виділити три елементи:

- вхід, який за класифікацією буде розгалуженням та запускатиме одну з двох гілок;
- вузли «перевірити набої» та «атакувати ціль», які представлені послідовністю;
- листи, які є діями («іти до цілі», «перевірити ціль»).

Тепер можна переходити до самої реалізації. Для цього буде достатньо класичної системи Unity компонентів, тож керуватимусь описом створення звичайного ігрового об’єкта. Цей новий агент буде організовано аналогічним чином, як і в State Machine, дивитися на рисунок 3.8.

Тут варто трохи зупинитися та пояснити ієрархію. Батьківський компонент Enemy, як вже описувалося, зберігає на собі основні компоненти логіки: Nav Mesh Agent, Rigidbody та самописний компонент Enemy, до якого повернемося пізніше. Дочірній компонент Character_Male_Jacket_01, аналогічним чином як у ECS Агента, відповідає за відображення. Проте цей Агент має в собі анімації, тож потребує набагато складнішого налаштування, яке включає в себе модель та ріг, які як раз і зберігаються всередині цього ігрового об’єкту.

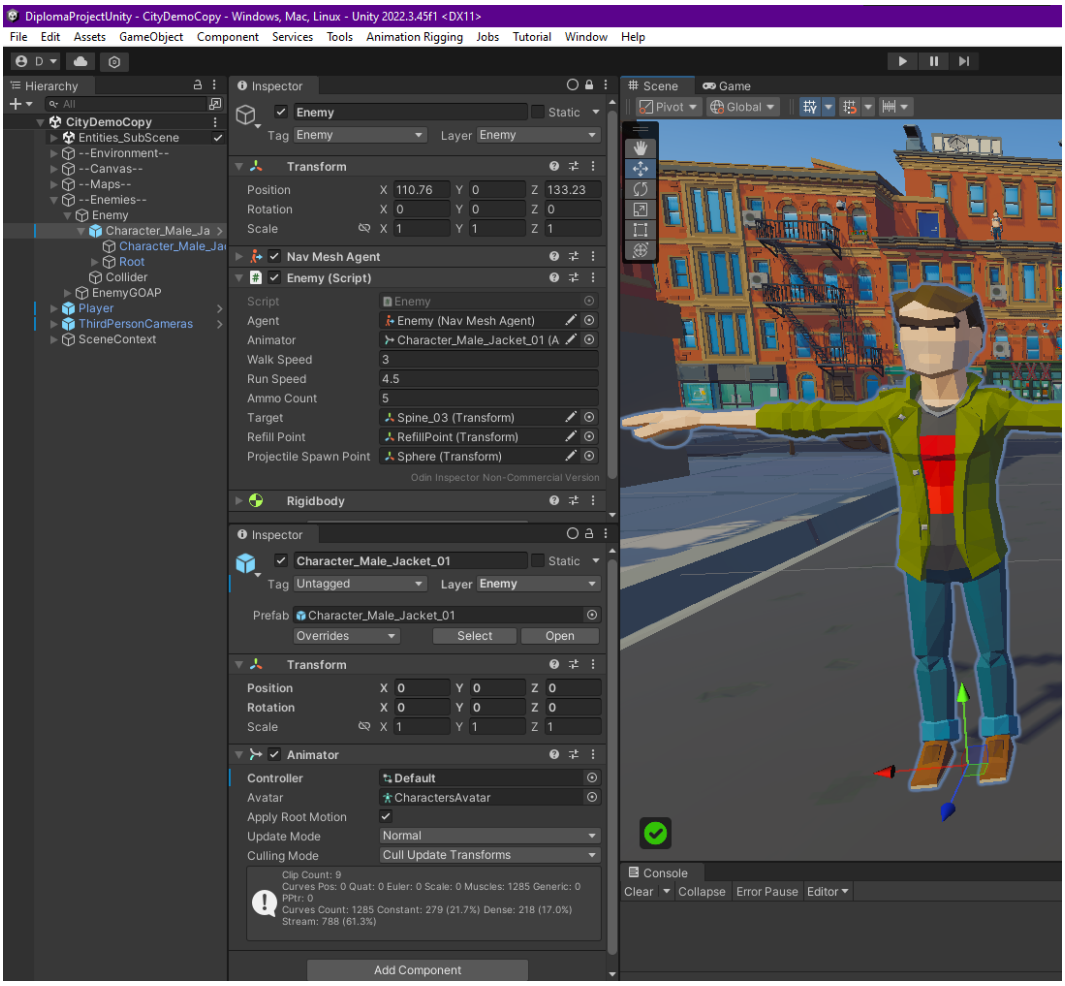

Рисунок 3.8 – Behavior Tree Агент

Тепер пропонується розібрати самого Агента. Nav Mesh Agent – це компонент з пакету навігації, який використовується для зручності роботи з пошуком шляху та рухом Агентів вздовж прокладеного маршруту. Про Rigidbody вже розповідалося вище, але конкретно в цій ситуації, окрім фізичної взаємодії з об'єктами, він допоможе в поведінковій гілці відновлення набоїв.

Сам скрипт Enemy є медіатором, який збирає в себе дані про світ та свій стан і передає їх в Behavior Tree. Розглядати його надто детально немає сенсу, поясняться лише частини пов'язані з самим алгоритмом.

На старті свого існування Агент ініціалізує системи, серед яких і «мозок», це показано на лістингу 3.11.

Лістинг 3.11 – Ініціалізація Агенту

```
private void Awake()
{
  _currentAmmo = _ammoCount;
  SetupBrain();
}
```

Метод SetupBrain необхідно розглянути більш детально, проте перед цим треба перемкнутися на загальну архітектуру імплементації алгоритму.

Отже, спочатку треба визначити базовий клас для кожного вузла. Він матиме в собі оголошення можливих станів («успіх», «провал» та «в процесі виконання», можна побачити на лістингу 3.12); ім'я вузла (зручно для дебагу та чистоти коду), список дочірніх нод, індекс поточної, яка обробляється; логіку створення ноди, додання дітей, обробки дітей та скидання стану. Детальніше можна ознайомитись на лістингу 3.13.

Лістинг 3.12 – Оголошення станів вузла

```
public enum Status
{
```

Продовження лістингу 3.12

```
            Success,
            Failure,
            Running
        }
```

Лістинг 3.13 – Оголошення базового класу Node

```
public class Node
    {
        public readonly string Name;
        public readonly List<Node> Children = new();
        protected int currentChild;
        public Node(string name)
        {
            Name = name;
        }
        public Node AddChild(Node child)
        {
            Children.Add(child);
            return this;
        }
        public virtual Status Process()
        {
            return Children[currentChild].Process();
        }
        public virtual void Reset()
        {
            currentChild = 0;
            Children.ForEach(c => c.Reset());
        }
    }
```

З цього базового класу дуже швидко можна зробити вузол листу, який матиме в собі стратегію, тобто дію, для виконання, та перевизначатиме тіло

методу Process і Reset на запуск відповідних методів стратегії. Ознайомитися з цим класом можна на лістингу 3.14.

Лістинг 3.14 – Оголошення вузла-листа

```
public class Leaf : Node
    {
        private IStrategy strategy;
        public Leaf(string name, IStrategy strategy) :
base(name)
        {
            this.strategy = strategy;
        }

        public override Status Process() =>
strategy.Process();
        public override void Reset() => strategy.Reset();
    }
```

Тепер треба створити вхідну точку, вузол входу. Цьому вузлу буде достатньо перевизначити метод Process так, як це описано на лістингу 3.15: поки індекс поточної ноди в обробці менший кількості всіх вузлів, запускаємо обробку цього вузла та повертаємо значення, яке повернув цей вузол.

Лістинг 3.15 – Перевизначення методу Process для класу BehaviourTree

```
public override Status Process()
        {
            while (currentChild < Children.Count)
            {
                var status =
Children[currentChild].Process();
                if (status != Status.Success)
                {
                    return status;
                }

                currentChild++;
            }
```

Продовження лістингу 3.15

```
            return Status.Success;
        }
```

Тепер достатньо зробити останню ноду, яка цікавитиме в цьому контексті – послідовність. Тут все просто – аналогічно треба перевизначити лише метод Process. Поки в послідовності не випаде «провал», алгоритм може обробляти листи по колу, що і зображено на лістингу 3.16.

Лістинг 3.16 – Перевизначення методу Process для класу LoopSequence

```
public override Status Process()
        {
            if (currentChild < Children.Count)
            {
                switch (Children[currentChild].Process())
                {
                    case Status.Running:
                        return Status.Running;
                    case Status.Failure:
                        Reset();
                        return Status.Failure;
                    default:
                        currentChild++;
                        return Status.Running;
                }
            }

            Reset();
            return Status.Running;
        }
```

Тепер можна повертатися до методу SetupBrain в класі Enemy. Першим ділом створюється вхідна нода BehaviourTree. Далі треба створити дві послідовності LoopSequence та заповнити їх листами-діями. В кінці достатньо додати їх як дітей в основну ноду – і процес стартового налаштування мозку готовий. З цим можна познайомитись на лістингу 3.17.

Лістинг 3.17 – Реалізація методу SetupBrain.

```
            private void SetupBrain()
            {
                _ai = new BehaviourTree("Main Tree");

                var attackSequence = new LoopSequence("Follow
target sequence")
                        .AddChild(new Leaf("Check target", new
ConditionStrategy(() => _target.gameObject.activeSelf)))
                        .AddChild(new Leaf("Follow target",
new FollowTransformStrategy(_agent, _target, _animator,
_walkSpeed, _runSpeed)))

                    .AddChild(new Leaf("Attack", new
AttackStrategy(_target, transform, _animator,
_projectileSpawnPoint, _ammoCount, _container)))
                    ;
                var checkAmmoSequence = new Sequence("Check
Ammo sequence")
                .AddChild(new Leaf("Check Ammo", new
ConditionStrategy(() => _currentAmmo <= 0)))
                    .AddChild(new Leaf("Go to refill", new
PatrolStrategy(_agent, new[] { _refillPoint }, _runSpeed)));

                var mainLoop = new LoopSelector("Main Loop
selector")
                    .AddChild(checkAmmoSequence)
                    .AddChild(attackSequence);
                _ai.AddChild(mainLoop)
                    .AddChild(new Leaf("End log", new
ActionStrategy(() => Debug.Log("End log"))));
            }
```

Тепер достатньо запустити мозок шляхом використання подійної функції з API Unity – Update. Цей метод викликається кожен ігровий кадр, тож це як раз те, що треба для обробки мозку:

```
private void Update() => _ai.Process();
```

Отже, таким простим чином вийшло створити потужний алгоритм, який є ідеальним способом симуляції ШІ для комп’ютерних ігор.

3.4 Імплементація Планування Дій Орієнтованих на Досягнення Цілі

Так як GOAP є найкращим доступним когнітивним алгоритмом, то для нього можна придумати насправді багато цілей, дій та знань. Але, щоб тримати пояснення в об’єктивних межах, робота обмежиться більш простим демонстративним прикладом. Для цілей пропонується обрати такі: «Нічого не робити», «Гуляти», «Здоровий», «Відпочивший», «Атакувати».

Тепер можна створити Агента. Процес аналогічний, а набір компонентів схожий на Behavior Tree, зображений на рисунку 3.9: батьківський об’єкт EnemyGOAP, який тримає на собі компоненти логіки (GOAP Agent, Nav Mesh Agent та Rigidbody); і дочірні об’єкти – колайдер, об’єкт відображення 3D моделі та анімацій, а також сенсори – річ, яка не проговорювалася до цього.

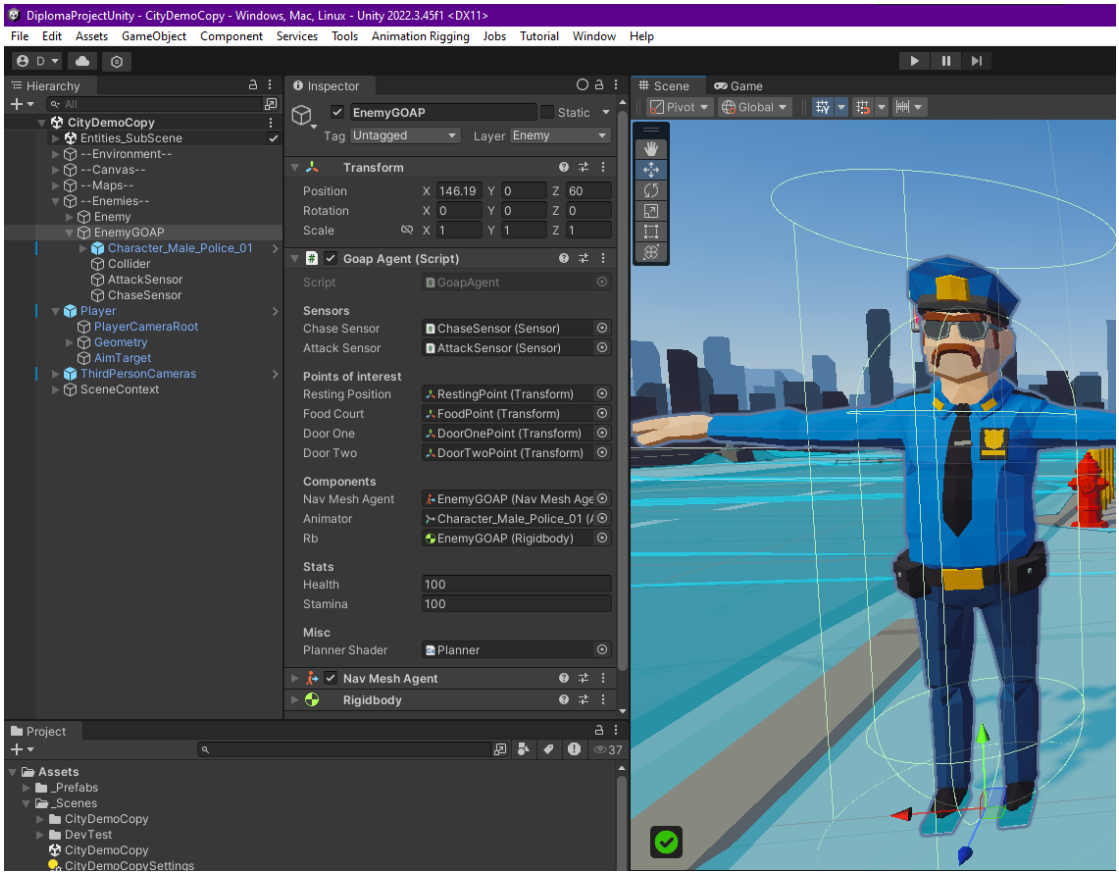

Рисунок 3.9 – GOAP Агент

Сенсори – одне з основних джерел інформації про зовнішній світ для Агента. Конкретно в цьому прикладі сенсори – це колайдери, які знаходяться в стані тригеру. Тобто вони реєструють колізію, але не зупиняють об’єкти від проходження через одне одного. Це дозволяє швидко і зручно реєструвати коли об’єкт знаходиться в певній зоні.

Сам клас сенсору максимально простий і включає в себе подію, на яку можуть підписати об’єкти з інших класів; метод з API Unity, який стартує таймер для перевірки на наявність мішені в тригер-зоні, та саме оновлення цілі. Для більш зручної взаємодії із зовнішнім світом, сенсор дає публічний доступ до координат цілі, та сам об’єкт.

Тепер розглянуться основні будівельні блоки GOAP, з якими надалі йтиме робота. Пояснення почнуться з класу AgentBelief. Основним, що має цей клас, є метод Evaluate, сигнатуру якого можна побачити нижче.

```
public bool Evaluate() => _condition();
```

Внутрішнє поле condition – це предикат, який передається в конструктор та визначає, чи правдиве це Belief в момент перевірки. Як вже згадувалося вище, для роботи з GOAP на комп’ют шейдерах необхідно буде перевести цей клас у структуру відповідник. З методом можна ознайомитись на лістингу 3.18.

Лістинг 3.18 – Переведення AgentBelief в структуру

```
public AgentBeliefStruct ToStruct()
        {
            return new AgentBeliefStruct
            {
                NameHash = Name.GetHashCode(),
                Location = Location,
                Condition = Evaluate() ? 1 : 0
            };
        }
```

Наступний будівельний блок алгоритму – клас AgentAction. Як вже проговорювалося в аналізі алгоритму, клас має в собі два списки –

Preconditions та Effects. Обидва списки є контейнерами, які зберігають AgentBelief. Аналогічним чином, як і в листі Behavior Tree, цей клас є обгорткою над стратегією, яку Агент може виконати за умови задоволення всіх Preconditions, та внести зміни-Effects у світ. І, аналогічно як для AgentBelief, мається необхідність перевести цей клас в структуру. З відповідним методом можна ознайомитись на лістингу 3.19.

Лістинг 3.19 – Переведення AgentAction в структуру

```
public AgentActionStruct ToStruct()
        {
            var precondition = Preconditions.Select(b =>
b.ToStruct()).FirstOrDefault();

            if (Preconditions == null ||
Preconditions.Count == 0)
                precondition.Condition = 1;
            return new AgentActionStruct
            {
                NameHash = Name.GetHashCode(),
                Cost = Cost,
                Precondition = precondition,
                Effect = Effects.Select(b =>
b.ToStruct()).FirstOrDefault()
            };
        }
```

Передостанній клас, з яким працюватиме планер – AgentGoal. Як вже казалося раніше, цей клас є обгорткою над AgentBelief із заданням їй пріоритету:

```
public int Priority { get; private set; }
public HashSet<AgentBelief> DesiredEffects { get; } =
new();
```

Цьому класу також необхідний метод переведення в структуру, описаний на лістингу 3.20.

Лістинг 3.20 – Переведення AgentGoal в структуру

```
public AgentGoalStruct ToStruct()
        {
            return new AgentGoalStruct
            {
                NameHash = Name.GetHashCode(),
                Priority = Priority,
                DesiredEffect = DesiredEffects.Select(e =>
e.ToStruct()).FirstOrDefault()
            };
        }
```

Закінчити опис будівельних блоків алгоритму хотілося б класом ActionPlan – клас, в який збирається послідовність дій для досягнення цілі, сама ціль, та загальну вартість плану. Сигнатура у нього проста та не потребує окремих пояснень, описана на лістингу 3.21.

Лістинг 3.21 – Оголошення класу ActionPlan

```
public class ActionPlan
    {
        public AgentGoal ActionGoal { get; }
        public Stack<AgentAction> Actions { get; }
        public float TotalCost { get; set; }

        public ActionPlan(AgentGoal actionGoal,
Stack<AgentAction> actions, float totalCost)
        {
            ActionGoal = actionGoal;
            Actions = actions;
            TotalCost = totalCost;
        }    }
```

З цима блоками можна почати створювати цілі для Агенту. Легко здогадатися: стартовій функції з Unity API створюються окремі виклики для цього.

Почнімо опис цієї частини зі знань Агенту про світ. Для зручності роботи в створенні знань було зроблено фабрику. Спосіб роботи її простий: передати ключ, передати умову, або точку та радіус. Всередині фабрика викликає конструктори та додає новостворене знання в словник:

```
factory.AddBelief(AGENT_RESTED_KEY, () => _stamina >= 80);
factory.AddBelief(AGENT_AT_DOOR_ONE_KEY, _doorOne, 3f);
```

Проте знань про світ необхідно набагато більше, тож розглянемо їх списком:

- NOTHING_KEY – в світі нічого не відбулося, чи дія Агенту нічого не змінить;
- AGENT_MOVING_KEY – Агент зараз в русі, чи дія призведе до руху;
- AGENT_IDLE_KEY – Агент стоїть на місці, не має активного шляху;
- AGENT_STAMINA_LOW_KEY – Агент має низьку витривалість (менше 30);
- AGENT_HEALTH_LOW_KEY – Агент має низьке здоров’я (менше 30);
- AGENT_HEALTHY_KEY – Агент у хорошому стані (здоров’я 60 або більше);
- AGENT_RESTED_KEY – Агент відпочив і має високу витривалість (80 або більше);
- AGENT_AT_DOOR_ONE_KEY – Агент перебуває поблизу першої двері (радіус ≤ 3f);
- AGENT_AT_DOOR_TWO_KEY – Агент перебуває поблизу другої двері (радіус ≤ 3f);

- AGENT_AT_RESTING_POINT_KEY – Агент біля точки для відпочинку (радіус ≤ 3f);
- AGENT_AT_FOOD_POINT_KEY – Агент біля точки з їжею (радіус ≤ 3f);
- ENEMY_IN_CHASE_RANGE_KEY – У зоні досяжності з’явився ворог для переслідування;
- ENEMY_IN_ATTACK_RANGE_KEY – Ворог потрапив у радіус атаки Агента;
- ATTACKING_ENEMY_KEY – Агент перебуває в стані атаки.

З цим набором знань Агент вже може формувати адекватні плани дій.

Переходимо до заповнення списку дій. Для зручності роботи з цією частиною системи було імплементовано патерн програмування Builder, тож створення дій виглядає так, як описано на лістингу 3.22.

Лістинг 3.22 – Створення дії для Агенту

```
     _actions.Add(new
AgentAction.Builder("FromDoorTwoMoveToRestingPosition")
     .WithStrategy(new MoveStrategy(_navMeshAgent, () =>
_restingPosition.position))
     .WithCost(2)
     .AddPrecondition(_beliefs[AGENT_AT_DOOR_TWO_KEY])
     .AddEffect(_beliefs[AGENT_AT_RESTING_POINT_KEY])
     .Build());
```

Дій, які може виконати агент, набагато більше, тож їх було винесено окремо: Relax, Wander, MoveToEatingPosition, Eat, MoveToRestingPosition, Rest, MoveToDoorOne, MoveToDoorTwo, FromDoorOneMoveToRestingPosition, FromDoorTwoMoveToRestingPosition, ChaseEnemy, AttackEnemy.

Останньою частиною підготовки до передачі в алгоритм є створення цілей. Аналогічно до дій, щоб з цим було зручніше працювати, було імплементовано патерн Builder, тож створення цілей виглядає як описано в лістингу 3.23.

Лістинг 3.23 – Приклад створення цілей

```
_goals.Add(new AgentGoal.Builder("Healthy")
.WithPriority(2)
.AddDesiredEffect(_beliefs[AGENT_HEALTHY_KEY])
.Build());
```

Цілі агенту вже були визначені на початку опису реалізації алгоритму, тому вони не будуть повторюватися тут.

Отже, маючи цілі та дії, можна передавати їх на алгоритм планеру. Ініціалізація планеру відбувається в стартовій функції, та не вимагає окремих пояснень – виклик конструктору. А ось виклик планування вже є найцікавішою частиною.

По-перше, необхідно перетворити списки дій та цілей на масиви структур. За це відповідає метод PrepareArrays, виклик та імплементацію якого можна побачити на лістингу 3.24.

Лістинг 3.24 – Імплементація методу PrepareArrays

```
     PrepareArrays(agent, goals, mostRecentGoal, out
AgentGoalStruct[] goalsStructs, out var actionsStructs);
     private void PrepareArrays(GoapAgent agent,
HashSet<AgentGoal> goals, AgentGoal mostRecentGoal, out
AgentGoalStruct[] goalsStructs, out AgentActionStruct[]
actionsStructs)
             {
                 var newGoals = new HashSet<AgentGoal>(goals);
                 newGoals.Remove(mostRecentGoal);
                 var orderedGoals = newGoals

                     .Where(g => g.DesiredEffects.Any(b =>
!b.Evaluate()))
                     .OrderByDescending(g => g ==
mostRecentGoal ? g.Priority - 0.01 : g.Priority)
                     .ToList();
                 var orderedActions = agent.Actions
                     .OrderBy(a => a.Cost);
                 goalsStructs = orderedGoals
                     .Select(g => g.ToStruct())
                     .ToArray();
                 actionsStructs = orderedActions
```

Продовження лістингу 3.24

```
                .Select(a => a.ToStruct())
                .ToArray();
        }
```

Спершу треба оновити список цілей та відсортувати його. Для цього достатньо скопіювати, видалити попередню ціль, та відсортувати за пріоритетом. Потім аналогічна операція проводиться над цілями. І в результаті по обом отриманим колекціям треба пройтися ще раз, щоб сконвертувати їх в масиви структур goalsStructs та actionsStructs.

Попереднім кроком інформація була переведена у вигляд, зрозумілий для Compute Shaders. Тепер необхідно обгорнути цю інформацію в буфер для передачі далі. За це відповідає метод PrepareComputeBuffers, з яким можна ознайомитися в лістингу 3.25.

Лістинг 3.25 – імплементація PrepareComputeBuffers

```
            private void
PrepareComputeBuffers(AgentGoalStruct[] goalsStructs,
AgentActionStruct[] actionsStructs)
            {
           _goalBuffer = new
ComputeBuffer(goalsStructs.Length,
Marshal.SizeOf<AgentGoalStruct>());
                _actionsBuffer = new
ComputeBuffer(actionsStructs.Length,
Marshal.SizeOf<AgentActionStruct>());
                _actionPlanBuffer = new
ComputeBuffer(PLAN_ACTIONS_LENGTH,
Marshal.SizeOf<AgentActionStruct>());
                _succeededGoalBuffer = new ComputeBuffer(1,
sizeof(int));

                _goalBuffer.SetData(goalsStructs);
                _actionsBuffer.SetData(actionsStructs);
                _actionPlanBuffer.SetData(new
AgentActionStruct[PLAN_ACTIONS_LENGTH]);
                _succeededGoalBuffer.SetData(new int[] {-1 });
```

Продовження лістингу 3.25

```
                _plannerShader.SetBuffer(_kernel, _goals,
_goalBuffer);
                _plannerShader.SetInt(_goalsCount,
goalsStructs.Length);

                _plannerShader.SetBuffer(_kernel, _actions,
_actionsBuffer);
                _plannerShader.SetInt(_actionsCount,
actionsStructs.Length);

                _plannerShader.SetBuffer(_kernel, _plan,
_actionPlanBuffer);
                _plannerShader.SetBuffer(_kernel,
_succeededGoalIndex, _succeededGoalBuffer);
            }
```

Перша частина відповідає за ініціалізацію буферів – необхідно передати кількість елементів в масиві та розмір одного елемента. Другою частиною виставляються дані (всі масиви у їхні буфери).

Кінець методу займається виставленням буферів у відповідні місця в самому планер-шейдері.

Виклик відбувається в одну строчку:

```
_plannerShader.Dispatch(_kernel, 1, 1, 1);
```

Dispatch – це основний метод виклику, який приймає в себе індекс кернелу та координати, виділені у відеокарті на обрахунки.

Наступна частина займається витягуванням плану та цілей з шейдеру, та очисткою буферів. Частина, представлена на лістингу 3.26 пояснень не потребує.

Лістинг 3.26 – Очистка буферів

```
        private void CleanupBuffers()
        {
            _actionsBuffer.Release();
            _goalBuffer.Release();
            _actionPlanBuffer.Release();
            _succeededGoalBuffer.Release();
        }
```

Залишок методу займається валідацією результату, збіркою плану дій та поверненням результату.

3.5 Порівняння результатів реалізації алгоритмів

Отже, отримавши реалізацію всіх методів, можна провести фінальний аналіз. Основна одиниця, яка цікавить розробників найбільше – це замір продуктивності, в одиницях FPS. З цим можна ознайомитись на рисунку 3.10.

Навіть без детального аналізу графіку можна побачити, що ECS система, як і очікувалося, видає найкращі результати – 1440 FPS. GOAP посідає друге місце зі значенням 1200, а Behavior Tree показує найслабші результати – лише 900. Проте зі збільшенням кількості агентів значення вирівнюються та досягають (на 10000 агентах) відповідно 90, 60 та 30, залишаючи той же ж порядок по місцям.

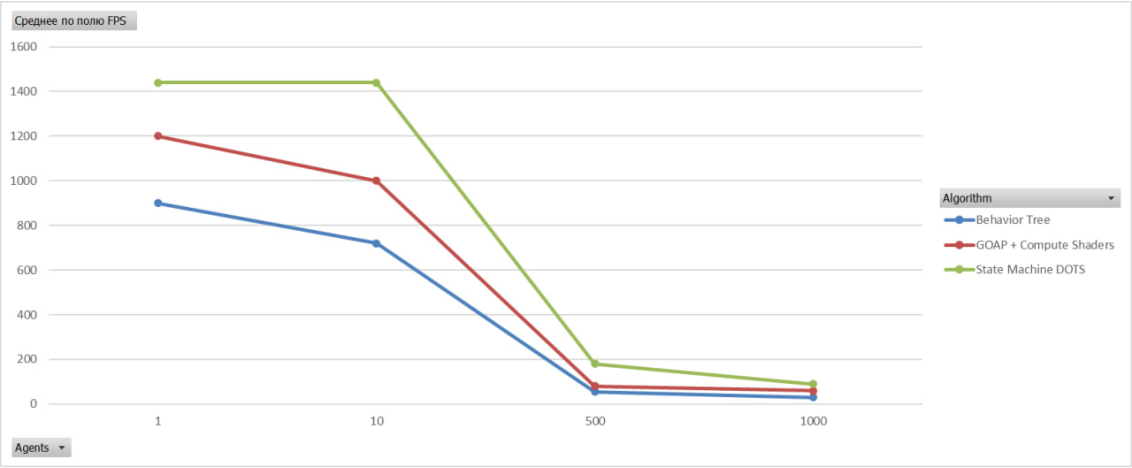

Рисунок 3.10 – Залежність FPS від кількості агентів

Рисунок 3.11 та 3.12 пояснює, в чому полягає проблема Behavior Tree – обробка займає надто багато часу головного процесору та прорахунки всередині вузлів алокують надзвичайно багато сміття.

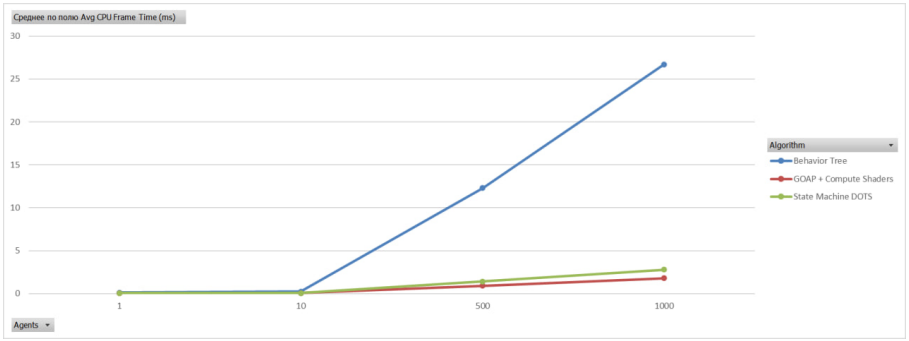

Рисунок 3.11 – Залежність часу обробки головним процесором від кількості агентів

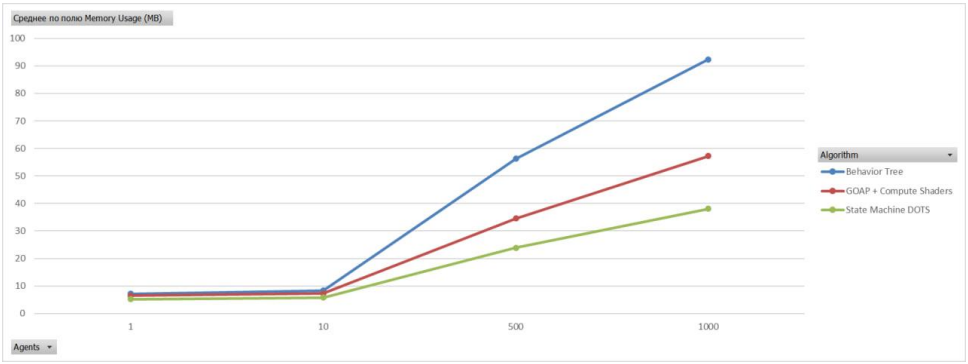

Рисунок 3.12 – Залежність використання оперативної пам’яті від кількості агентів

Чому це є такою проблемою? Як згадувалося вище, Unity використовує віртуальне середовище Mono. Це дозволяє досягти результату, щоб розроблені застосунки були кросплатформеними. Але це також надзвичайно обмежує розробників, через відсутність доступу до збірника сміття з C#, який виконує роботу алокацій та очистки пам’яті набагато швидше.

Аналогічну проблему зі сміттям, але в трохи менших масштабах, можна побачити і для двох інших алгоритмів. Проте це не зможе пояснити, чому GOAP сповільнюється більше, ніж State Machine. На цей раз допоможе графік рисунку 3.13. Там можна чітко побачити, що час навантаження на

відеокарту складає 5 мілісекунд, що створює проблеми для рендерингу, а це в свою чергу сповільнює загальний замір.

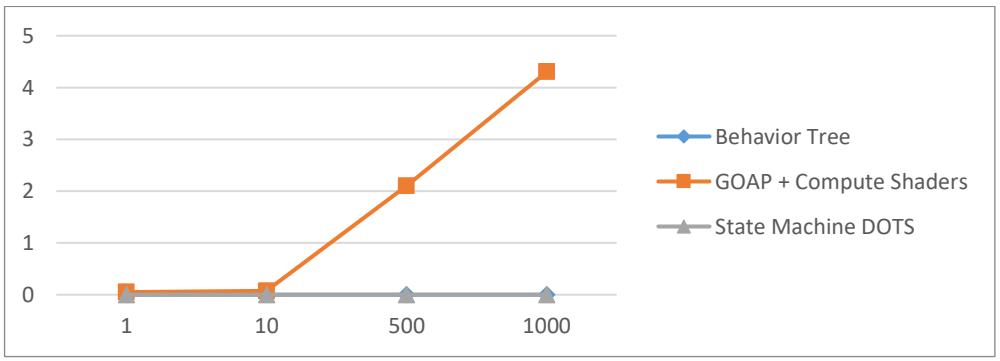

Рисунок 3.13 – Залежність навантаження на відеокарту від кількості агентів

Отже, в порівнянні можна побачити, що зміна підходів до реалізації спростила сприйняття алгоритмів та вирішила деякі з проблем класичних алгоритмів, але це досі не є панацеєю, так як на великих навантаженнях досі спостерігаються проблеми.

ВИСНОВКИ

Отже, протягом кваліфікаційної роботи було виконано комплексний аналіз популярних ринкових алгоритмів симуляції ШІ в комп’ютерних іграх: State Machine, Behavior Tree та GOAP. Алгоритми були виділені на основі аналізу популярних ринкових рішень. Кожен з виділених алгоритмів був розглянутий в контексті ринкових умов. Виявлено сильні та слабкі сторони класичної імплементації а також запропоновано шляхи їх вдосконалення шляхом зміни підходу до задачі чи загальної парадигми.

State Machine адаптовано під Data-Driven підхід в контексті ECS. Відмова від переходу «кожен до кожного» та винесення логіки у системи дозволило спростити розширення патернів поведінки агента та підвищило масштабованість.

Для Behavior Tree визначено ключове вузьке місце, яке погіршує продуктивність та ускладнює зручність використання – рекурсивність та фрагментація пам’яті та алокація об’єктів. Запропонував та реалізував оптимізовану послідовну обробку вузлів шляхом створення нерозривних списків всередині кожного вузла.

GOAP алгоритм також переведено на Data-Driven підхід, але на цей раз в контексті Compute Shader. Це дозволило паралельно будувати плани і як наслідок знизило навантаження на головний процесор.

В кінці проведено тестування та визначено, що ECS-реалізація State Machine забезпечує набагато кращі результати продуктивності у порівнянні з всіма іншими алгоритмами. Проте це все ще залишається простим алгоритмом, тож для просунутих систем, де важлива продуктивність, найкраще використовувати GOAP, який показав більш прийнятні результати, аніж Behavior Tree.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Стан. *Refactoring and Design Patterns*. URL: https://refactoring.guru/uk/design-patterns/state (дата звернення: 07.06.2025).

2. Code Monkey. How to use machine learning AI in unity! (ml-agents), 2020. *YouTube*. URL: https://www.youtube.com/watch?v=zPFU30tbyKs (date of access: 07.06.2025).

3. DaDi. 유니티 VFX 포트폴리오 / unity game effect portfolio, 2024. *YouTube*. URL: https://www.youtube.com/watch?v=Qt4KOrRSQ3I (дата звернення: 07.06.2025).

4. Godot Engine - Free and open source 2D and 3D game engine. *Godot Engine*. URL: https://godotengine.org/ (date of access: 07.06.2025).

5. Kolesnikova A. Here's how much AI engineers are getting paid. *Levels.fyi | Salaries & Tools to Level Up Your Career*. URL: https://www.levels.fyi/blog/ai-engineer-compensation.html (date of access: 07.06.2025).

6. Steam hardware & software survey. *Welcome to Steam*. URL: https://store.steampowered.com/hwsurvey/videocard/ (date of access: 07.06.2025).

7. Unity asset store. *The Best Assets for Game Making | Unity Asset Store*. URL: https://assetstore.unity.com/ (date of access: 07.06.2025).

8. Unity real-time development platform | 3D, 2D, VR & AR engine. *Unity*. URL: https://unity.com/ (date of access: 07.06.2025).

9. Unreal engine. *Unreal Engine*. URL: https://www.unrealengine.com/ (date of access: 07.06.2025).

10. An Investigation of Data Storage in Entity-Component Systems. 2022. URL: https://scholar.afit.edu/cgi/viewcontent.cgi?article=6355&context=etd (date of access: 07.06.2025).