Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

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

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ДОСЛІДЖЕННЯ МЕТОДІВ КЕРУВАННЯ ПОВЕДІНКОЮ NPC У ВІДЕОІГРАХ
(тема)

Виконав:
здобувач 2 року навчання,
групи ІНФм-24-1
Мірошниченко М. С.
(прізвище, ініціали)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Науковий керівник проф. Машталір С. В.
(посада, прізвище, ініціали)

Допускається до захисту

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

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти другий (магістерський)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Мірошниченку Максиму Станіславовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів керування поведінкою NPC у відеоіграх

затверджена наказом університету від 14 листопада 2025 року № 1045Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 14 листопада 2025 р.
3. Вихідні дані до роботи методи керування поведінкою NPC у відеоіграх (FSM, BT, Utility AI, HFSM), літературні джерела щодо сучасних підходів до побудови ігрового штучного інтелекту, математичні моделі станів та переходів, середовище розробки Unity та мова C#, інструменти навігації системи NPC.

4. Перелік питань, що потрібно опрацювати в роботі
1. Провести аналіз існуючих методів керування поведінкою NPC у відеоіграх.
2. Виконати огляд літературних джерел щодо побудови поведінкових моделей та ієрархічних машин станів.
3. Описати математичні моделі FSM та HFSM, сформувати ієрархічну структуру керування поведінкою NPC.
4. Сформувати покроковий алгоритм роботи HFSM та описати його особливості.
5. Розробка архітектури і програмної реалізації HFSM у середовищі Unity на мові C#.
6. Виконати порівняння якості та ефективності HFSM із іншими методами керування.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність проблеми методів керування поведінкою NPC у відеоіграх, об'єкт та мета дослідження, постановка задачі, блок-схема методів керування поведінкою NPC, UML-діаграми класів та переходів, висновки, перспективи та апробація роботи.

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

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	29.09.2025	
2	Аналіз завдання, підбір літератури	30.09.25-07.10.25	
3	Аналіз літератури з досліджуваної проблеми	08.10.25-14.10.25	
4	Дослідження особливостей методів керування NPC	15.10.25-20.10.25	
5	Розробка алгоритмів керування поведінкою NPC	21.10.25-27.10.25	
6	Програмна реалізація	28.10.25-05.11.25	
7	Обґрунтування отриманих результатів	06.11.25-11.11.25	
8	Оформлення пояснювальної записки	12.11.25-14.11.25	
9	Перевірка на нормоконтроль	11.12.25-13.12.25	
10	Перевірка на плагіат	13.12.25-14.12.25	
11	Рецензування	14.12.25-15.12.25	
12	Підготовка презентації та доповіді	15.12.25-16.12.25	
13	Занесення роботи в електронний архів	18.12.25	
14	Попередній захист кваліфікаційної роботи	18.12.25	

Дата видачі завдання 29 вересня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. Машталір С. В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 80 с., 2 табл., 18 рис., 39 джерел.

ВІДЕОІГРИ, ІГРОВИЙ ДВИГУН, НЕІГРОВІ ПЕРСОНАЖІ, UNITY, C#, BEHAVIOR TREE, HIERARCHICAL STATE MACHINE, UTILITY AI, NAVMESH, SCRIPTABLE OBJECTS, GAME AI.

Об’єктом дослідження є поведінка неігрових персонажів (NPC) у відеоіграх.

Предметом дослідження є методи моделювання та керування поведінкою NPC із використанням алгоритмів прийняття рішень та систем штучного інтелекту.

Метою дослідження є аналіз, порівняння та практична реалізація різних підходів до керування поведінкою NPC на основі розробки інтерактивного застосунку в середовищі Unity з використанням мови програмування C#.

Використано методи машини станів (Finite State Machine), дерева поведінки (Behavior Tree) та Utility AI.

Наукова новизна роботи полягає у створенні узагальненої системи керування NPC, що дозволяє комбінувати кілька підходів (FSM, Behavior Tree, Utility AI).

Взаємозв’язок з іншими роботами полягає у використанні сучасних підходів до побудови ігрового штучного інтелекту, які широко застосовуються в індустрії відеоігор.

Рекомендації щодо використання результатів роботи: розроблена система може бути використана як основа для створення складних NPC у відеоіграх.

У результаті дослідження створено прототип системи керування поведінкою NPC у середовищі Unity, що демонструє застосування різних підходів та дозволяє оцінити їхню ефективність.

ABSTRACT

Explanatory note to the qualification work: 80 pages, 2 table, 18 figures, 39 sources.

VIDEO GAMES, GAME ENGINE, NON-GAME CHARACTERS, UNITY, C#, BEHAVIOR TREE, HIERARCHICAL STATE MACHINE, UTILITY AI, NAVMESH, SCRIPTABLE OBJECTS, GAME AI.

The object of research is the behavior of non-player characters (NPCs) in video games.

Subject of research is the methods of modeling and controlling NPC behavior using decision-making algorithms and artificial intelligence systems.

The aim of the research is to analyze, compare, and practically implement various approaches to NPC behavior control based on the development of an interactive application in Unity using the C# programming language.

The methods applied include Finite State Machine (FSM), Behavior Tree, and Utility AI.

Scientific novelty of the work lies in the creation of a generalized NPC control system that allows combining several approaches (FSM, Behavior Tree, Utility AI).

Interconnection with other works is based on the use of modern approaches to game artificial intelligence widely applied in the video game industry.

Recommendations for using the results: the developed system can serve as a foundation for creating complex NPCs in video games.

As a result of the research, a prototype of an NPC behavior control system was developed in Unity, demonstrating the application of different approaches and allowing evaluation of their efficiency.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект

API – Application Programming Interface

AI – Artificial Intelligence (Штучний інтелект)

Agent – Окремий NPC або AI-одиниця

BT – Behavior Tree (Дерево поведінки)

Chase – Стан переслідування гравця

Context – Контекст стану, набір параметрів NPC

FSM – Finite State Machine (Машина станів)

HFSM – Hierarchical Finite State Machine (Ієрархічна машина станів)

Idle – Стан очікування NPC

NavMesh – Navigation Mesh (Навігаційна сітка)

NavMeshAgent – Компонент Unity для навігації NPC

NPC – Non-Player Character (Неігровий персонаж)

NPC Behavior – Поведінка NPC

Patrol – Стан патрулювання

State – Стан NPC у машині станів

Substate – Підстан, вкладений стан у HFSM

Superstate – Суперстан, стан верхнього рівня в HFSM

Transition – Перехід між станами

Trigger – Подія або умова, що викликає перехід

UI – User Interface (Користувацький інтерфейс)

Utility AI – Метод вибору поведінки на основі корисності

View Angle – Кут огляду NPC

View Distance – Дальність огляду NPC

ВСТУП

Комп’ютерна графіка розділяється за трьома основними напрямками: візуалізація, обробка зображень і розпізнавання образів.

Візуалізація – це створення зображення на основі якогось опису (моделі). Приміром, це може бути відображення графіку, схеми, імітація тривимірної віртуальної реальності в комп’ютерних іграх, у системах архітектурного проєктування.

Основне завдання розпізнавання образів – одержання семантичного опису зображених об’єктів. Мета розпізнавання може бути різна: як виділення окремих елементів на зображенні, так і класифікація зображення в цілому. У якомусь сенсі завдання розпізнавання є зворотним стосовно завдання візуалізації. Області застосування – системи розпізнавання текстів, створення тривимірних моделей людини по фотографіях.

Сучасна індустрія відеоігор є однією з найдинамічніших галузей цифрових технологій, у якій поєднуються досягнення програмування, комп’ютерної графіки, штучного інтелекту та психології сприйняття. Одним із ключових аспектів, що безпосередньо впливає на якість ігрового досвіду, є поведінка неігрових персонажів (NPC – Non-Player Characters). Від того, наскільки реалістично NPC реагують на події в ігровому світі, залежить занурення гравця, рівень взаємодії з ігровим середовищем та загальна динаміка гри.

Актуальність роботи полягає у зростанні потреби в розробці інтелектуальних систем керування поведінкою NPC, здатних діяти природно, адаптивно та варіативно. Якщо раніше неігрові персонажі мали обмежений набір дій і реагували на події за простими сценаріями, то сучасні гравці очікують глибокої взаємодії, живої реакції та складних патернів поведінки. Це зумовлює необхідність застосування більш гнучких підходів, таких як ієрархічні машини станів (Hierarchical State Machines), дерева поведінки

(Behavior Trees) та Utility AI, які забезпечують NPC можливістю самостійно обирати дії на основі внутрішніх станів, цілей і змін у навколишньому середовищі.

На сучасному етапі розвитку ігрового штучного інтелекту існує широкий спектр методів керування NPC. Найпростішими є Finite State Machines (FSM) – системи, у яких поведінка описується набором станів та переходів між ними. Вони відзначаються простотою реалізації, однак обмежені у масштабованості, особливо для складних систем. Behavior Trees дозволяють ієрархічно структурувати дії NPC, розділяючи складну поведінку на підзадачі, що робить їх ефективними у великих ігрових проєктах. У свою чергу, Utility AI ґрунтується на математичній оцінці «корисності» дій, завдяки чому NPC можуть динамічно приймати рішення на основі змінних факторів, наприклад, рівня здоров’я, відстані до ворога чи важливості завдання.

Огляд сучасного стану проблеми показує, що провідні розробники ігор поєднують кілька методів для досягнення максимальної гнучкості та правдоподібності NPC. Наприклад, у серіях Far Cry, The Last of Us, Horizon Zero Dawn використовується комбінація Behavior Trees і Utility AI для забезпечення складної поведінки супротивників, які можуть кооперуватися, ухвалювати тактичні рішення та реагувати на дії гравця. У незалежній розробці (indie) все більшого поширення набувають фреймворки на основі Scriptable Objects у Unity, які дозволяють створювати гнучкі, розширювані системи поведінки без надмірного ускладнення коду.

Попри значну кількість досліджень і практичних реалізацій, залишається актуальною наукова задача розробки універсальної, модульної системи керування поведінкою NPC, яка б поєднувала простоту розширення, ефективність виконання та природність дій персонажів. Складність полягає у пошуку оптимального балансу між формальними алгоритмами прийняття рішень і непередбачуваністю, що створює ефект «живої» поведінки.

Тому метою даного дослідження є аналіз, порівняння та практична реалізація різних методів керування поведінкою NPC у середовищі Unity із

використанням мови C#, а також створення прототипу, який демонструє роботу FSM, Behavior Tree та Utility AI у єдиній системі.

Завдання дослідження полягають у:

– проведенні огляду сучасних підходів до керування поведінкою NPC;

– реалізації та порівнянні трьох підходів: Finite State Machine, Behavior Tree та Utility AI;

– створенні модульної архітектури системи, що дозволяє легко додавати нові типи поведінки;

– оцінці ефективності реалізованих методів за критеріями гнучкості, стабільності та складності налаштування;

Практична цінність роботи полягає у створенні прототипу системи, яку можна використати як основу для розробки складних NPC у різножанрових іграх – від шутерів і рольових до симуляторів виживання. Отримані результати також можуть бути корисними при навчанні розробників штучного інтелекту та у створенні навчальних проєктів для студентів спеціальностей, пов’язаних із ігровими технологіями.

1 АНАЛІЗ ОСНОВНИХ МЕТОДІВ КЕРУВАННЯ ПОВЕДІНКОЮ NPC

1.1 Аналіз сучасних методів керування поведінкою NPC у відеоіграх

1.1.1 Метод керування NPC засобами дерева поведінки

Розвиток ігрових технологій супроводжується зростанням вимог до реалістичності та автономності неігрових персонажів (NPC). Від якості штучного інтелекту NPC залежить глибина занурення користувача, динамічність ігрового процесу та рівень взаємодії у віртуальному середовищі.

Дерева поведінки – це прості, ієрархічні структури даних для управління складністю шляхом надання графічного представлення та формалізації складних дій [1]. Їх можна описати як синтез HSM та інших методів штучного інтелекту, таких як методи планування. Вони більш інтуїтивні, ніж кінцеві автомати, і використовуються для зберігання та виконання планів, а не для їх генерації. Дерева поведінки можна розглядати як дерева І-АБО, які зберігають усі можливі плани, яким ігрова сутність може слідувати для досягнення своїх цілей. Саме сувора та статична ієрархічна природа дерев поведінки робить їх придатними для повторного використання та модуляризації.

Усі дерева поведінки – це кореневі деревоподібні структури, які перетинаються кожного разу, коли виконується логіка штучного інтелекту.

Дерева складаються з вузлів та гілок у відношеннях «батьків-дитина». Вони можуть мати будь-яку кількість рівнів, але дочірні вузли обмежені одним батьківським вузлом. Вузли поблизу кореня дерева представляють поведінку вищого рівня, ніж ті, що знаходяться поблизу листя.

Коли виконується цикл ШІ, батьківський вузол вибирає відповідний дочірній вузол для виконання, при цьому логічне значення (або результат, що все ще виконується) повертається від дочірнього вузла батьківському. Дерева поведінки можна розглядати як структури цілей, які відображають, як ціль високого рівня може бути розкладена на цілі нижчого рівня, доки не будуть досягнуті примітивні цілі, яких можна досягти за допомогою доступних дій.

Приклад дерева поведінки показано на рисунку 1.1.

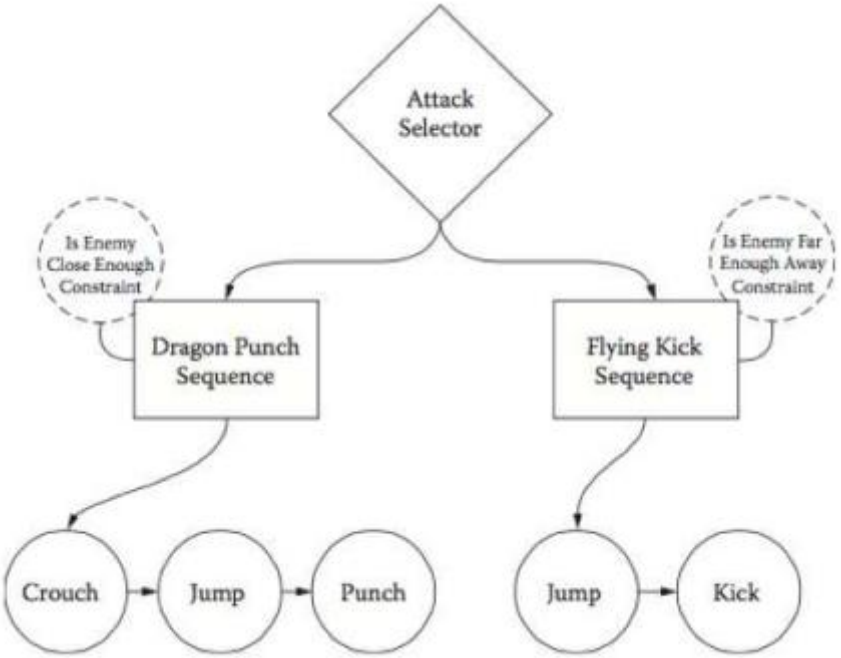

Рисунок 1.1 – Приклад дерева поведінки

Підвищення інтелекту ігрових персонажів постійно було ключовим пріоритетом як для фахівців галузі, так і для науковців-дослідників у галузі розробки ігор та суміжних дисциплін. З часом застосування різних методів штучного інтелекту в іграх значно змінилося. Дерева поведінки досі широко використовуються в численних галузевих застосуваннях і продовжують впливати на майбутні розробки ігор та академічні дослідження [2].

Незважаючи на значний прогрес у розробці ігор, такі проблеми, як неінтелектуальні неігрові персонажі (NPC), продовжують перешкоджати зануренню. Дослідження показують, що ці NPC, незалежно від того, чи є вони частиною оповіді, чи виконують функціональну роль, часто демонструють низький інтелект та негнучку тактику, що негативно впливає на ігровий досвід.

Наприклад, у таких іграх, як «Dark Souls» та «God of War: Ragnarōk», кооперативні NPC призначені для допомоги гравцям, але їхня низька продуктивність у складних сценаріях часто призводить до розчарування гравців.

Одним із ключових завдань сучасного геймдизайну є створення гнучких та адаптивних моделей поведінки NPC, здатних реагувати на змінні умови і дії гравця.

Дерево поведінки виконується циклічно, зазвичай на кожному кадрі або з певною частотою оновлення. NPC «опитує» дерево, щоб з'ясувати, яку дію потрібно виконати в поточний момент. Наприклад, дерево може мати гілку «Атакувати», що активується лише тоді, коли гравець знаходиться в полі зору персонажа, або гілку «Патрулювати», якщо ціль відсутня.

Таким чином, Behavior Tree дає змогу описати складну поведінку простою комбінацією умов і дій, не перевантажуючи систему переходами між станами, як це відбувається у FSM.

У структурі дерева поведінки кореневий вузол є відправною точкою для прийняття рішень. Умовні вузли використовуються для перевірки виконання певних умов, вузли дій використовуються для виконання певних дій, а складені вузли поєднують інші вузли для формування складніших моделей поведінки.

Дерево поведінки дозволяє розробникам модульно визначати та керувати поведінкою NPC, щоб вони могли гнучко реагувати на різні динамічні зміни та складні сцени в грі.

Основна ідея методу полягає у послідовному обході дерева: NPC аналізує стан середовища, перевіряє умови та виконує відповідну гілку дій.

Для прикладу, NPC може переходити між станами «патрулювання», «переслідування», «атака» і «втеча», залежно від того, чи бачить він гравця та який рівень його здоров'я [3].

Якщо NPC не бачить гравця – виконується поведінка патрулювання; якщо бачить – переходить до переслідування або атаки; якщо отримує пошкодження – переходить у стан втечі.

Математично модель дерева поведінки можна подати як множину:

$$B = (N, R, f), \tag{1.1}$$

де B – дерево поведінки;

N – множина вузлів;

R – відношення між вузлами;

f – функція виконання, що визначає результат кожного вузла(успіх, невдача, продовження).

Процес прийняття рішення описується рекурсивним обходом дерева згори донизу, де кожен селектор повертає перший успішний результат серед своїх підвузлів.

Дерева поведінки не лише покращують прийняття рішень штучним інтелектом, але й оптимізують процес розробки, роблячи поведінку NPC більш привабливою та реалістичною для гравців. Behavior Trees дозволяють розробникам розділяти складну поведінку на модулі та повторно використовувати їх у різних типах NPC.

Використання дерев поведінки підвищує стабільність логіки NPC, зменшує кількість помилок у прийнятті рішень та забезпечує більш природну поведінку персонажів у динамічному середовищі [4]. Подальші дослідження можуть бути спрямовані на інтеграцію елементів машинного навчання у структуру дерева поведінки, що дозволить NPC адаптувати свої рішення в реальному часі.

1.1.2 Метод керування NPC засобами скінченної машини станів(FSM)

Одним із найдавніших та водночас найпоширеніших методів моделювання поведінки неігрових персонажів (NPC) є машина скінченних станів (Finite State Machine, FSM). Цей підхід має глибоке теоретичне підґрунтя та використовується не лише у сфері розробки ігор, але й у системному програмуванні, робототехніці, автоматизації та штучному інтелекті.

FSM історично стала першим стандартом керування NPC у відеоіграх. Вона використовувалась ще у 90-х роках у класичних іграх DOOM, Quake, Half-Life, де вороги переходили між станами «патрулювання», «атака» та «втеча».

У сучасних проєктах FSM часто використовується для простих об’єктів – дронів, тварин, охоронців – або в комбінації з іншими підходами.

Машина станів – це модель системи**,** що може перебувати лише в одному стані в конкретний момент часу. Перехід між станами відбувається при виконанні певних умов або подій. Кожен стан описує певну поведінку NPC, а переходи між ними визначають логіку змін цієї поведінки [5, 6].

Принцип роботи FSM у контексті ігор:

Етап 1. Ініціалізація. NPC починає з базового стану (наприклад, «очікування» або «патрулювання»).

Етап 2. Виконання поточного стану. У межах стану виконується певна логіка – рух по маршруту, перевірка наявності ворогів, програвання анімації тощо.

Етап 3. Перевірка умов переходу. Кожен кадр або з певною частотою перевіряються умови, які можуть викликати перехід до іншого стану.

Етап 4. Перехід до нового стану. Якщо умова виконана, NPC «виходить» з поточного стану (викликається метод Exit), «входить» у новий (метод Enter) та починає виконувати нову поведінку.

Таким чином, FSM створює замкнену систему, у якій NPC реагує на зміни довкілля через чітко визначені переходи між поведінками.

Перевагою машини станів є її простота реалізації – цей підхід легко зрозуміти, реалізувати та налагодити, особливо на початкових етапах розробки. Логіка роботи системи є прозорою, оскільки кожен стан має чітко визначену поведінку, що значно спрощує аналіз і тестування. Крім того, машина станів відзначається високою продуктивністю, адже не потребує складних обчислень і може ефективно працювати навіть у системах із обмеженими ресурсами. Ще однією важливою рисою є детермінованість – поведінка NPC у такій системі є передбачуваною, що має особливе значення для ігор із точним балансом або змагальними елементами. Приклад роботи машини станів показано на рисунку 1.2.

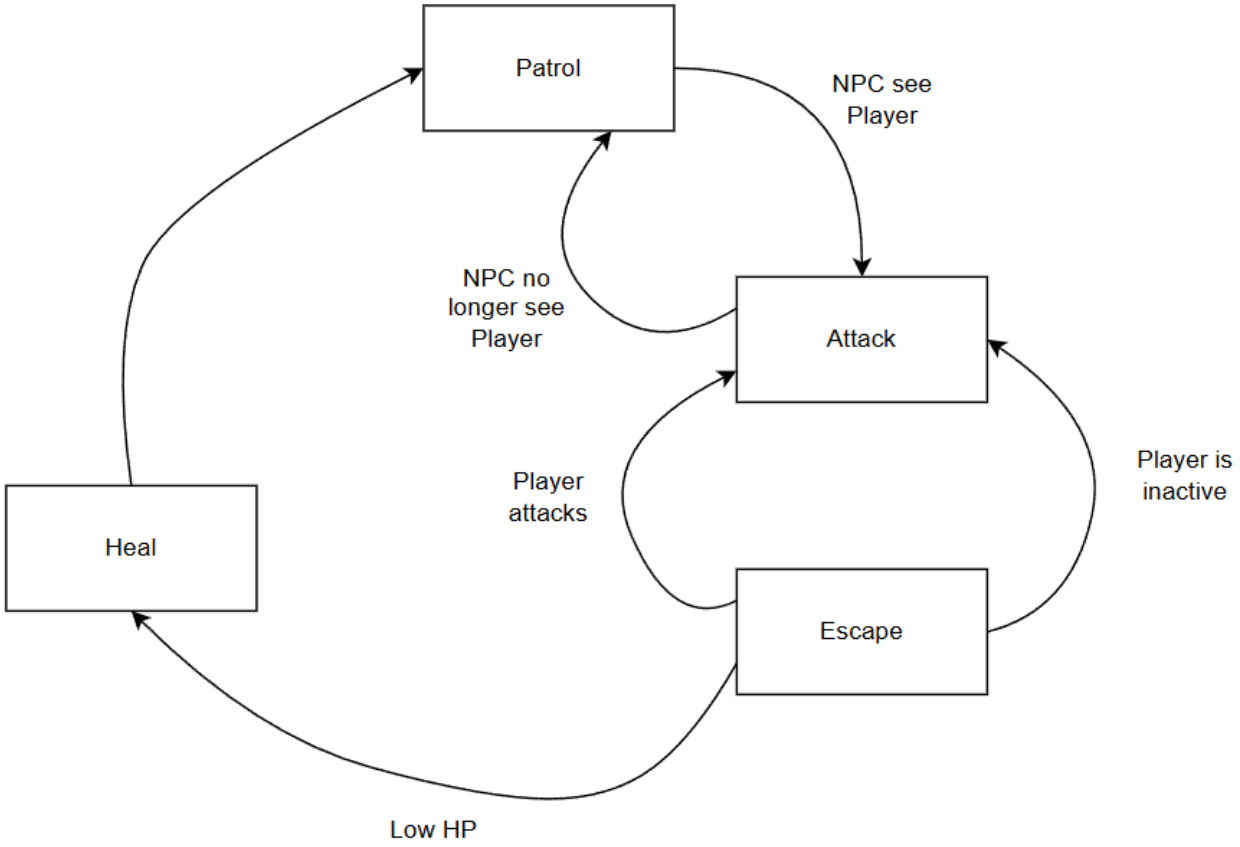

Рисунок 1.2 – Приклад переходу станів FSM

Водночас машина станів має і свої недоліки. Головним обмеженням є погана масштабованість: зі збільшенням кількості станів кількість зв’язків між ними зростає експоненційно, що ускладнює підтримку та розширення системи.

Додавання нових дій або умов часто вимагає значних змін у коді, роблячи систему менш гнучкою. Також FSM не здатна адаптуватися до контексту – вона лише реагує на наперед визначені події, не враховуючи складні ситуації чи змінні умови середовища. Через це поведінка NPC іноді виглядає занадто механічною або передбачуваною, якщо не додати елементів випадковості чи стохастичних переходів.

Метод машини станів є базовим, але фундаментальним інструментом у розробці ігрового штучного інтелекту. Його головна сила – у простоті, стабільності та зрозумілій логіці, що робить його ідеальним для початкової реалізації поведінки NPC [7].

Проте зі зростанням складності гри FSM стає менш ефективною, тому в сучасній практиці вона часто використовується як нижній рівень поведінкової системи, на якому реалізується базова логіка, у той час як більш високорівнева логіка контролюється Behavior Tree або Utility AI.

1.1.3 Метод керування NPC засобами Utility AI

Метод Utility AI (або «штучний інтелект на основі корисності») є сучасним підходом до моделювання поведінки неігрових персонажів, який забезпечує більш природну, гнучку та адаптивну взаємодію NPC із навколишнім середовищем. На відміну від машини станів або дерева поведінки, де рішення приймаються на основі фіксованих умов або структур переходів, Utility AI дозволяє NPC самостійно оцінювати можливі дії, обираючи ту, яка має найбільшу корисність у поточній ситуації.

Основна ідея методу полягає у тому, що кожна дія NPC (наприклад, «атакувати ворога», «відступити», «шукати укриття», «знайти їжу») оцінюється за допомогою функції корисності – математичної моделі, що визначає, наскільки доцільно виконати цю дію залежно від поточного стану світу, ресурсів або цілей персонажа. Приклад Utility AI показано на рисунку 1.3.

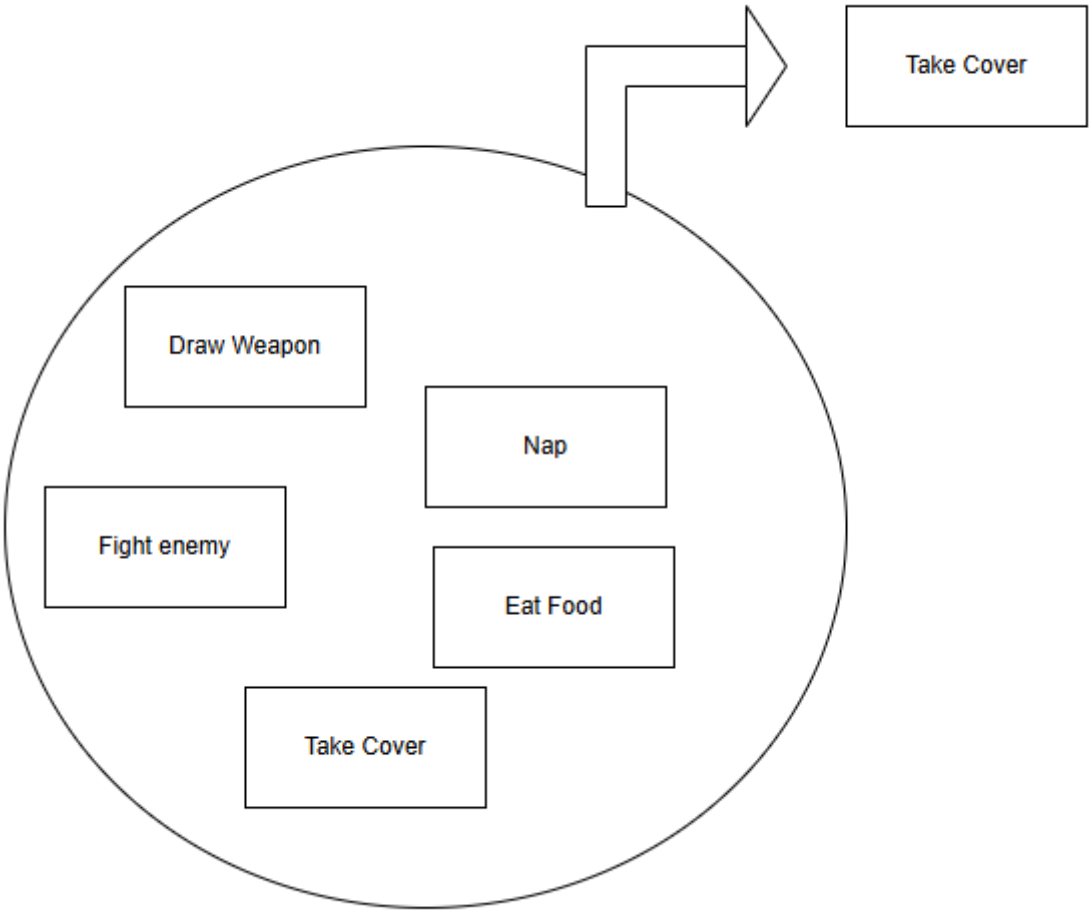

Рисунок 1.3 – Приклад оцінки дій Utility AI

Загальний принцип можна описати формулою:

$$U(a) = f(x_n), \tag{1.2}$$

де $U(a)$ – корисність дії a;

x_n – параметри, що характеризують ситуацію (здоров'я NPC, відстань до ворога, кількість боєприпасів, рівень небезпеки тощо);

f – функція виконання, може бути лінійною, експоненційною або логістичною, що дозволяє гнучко налаштовувати реакції NPC.

Процес прийняття рішення описується рекурсивним обходом дерева згори донизу, де кожен селектор повертає перший успішний результат серед своїх підвузлів.

Наприклад, якщо рівень здоров’я персонажа низький, а ворог близько, функція «відступу» може давати вищий показник корисності, ніж «атака» [8]. Якщо ж здоров’я повне, а ворог слабкий, кориснішою стає дія «атакувати». Такий підхід робить NPC не просто реактивним, а контекстно адаптивним, що створює враження більш «розумної» поведінки.

Однією з головних переваг Utility AI є гнучкість і масштабованість. Додавання нових дій або умов не потребує перебудови всієї логіки – достатньо визначити нову функцію корисності. Це робить систему легко розширюваною та придатною для складних ігор із великою кількістю взаємодіючих агентів. Крім того, завдяки безперервним оцінкам корисності поведінка NPC виглядає плавною, без різких змін між станами, що є типовим для FSM або Behavior Tree.

Utility AI часто використовується у сучасних відеоіграх, де важливо створити ілюзію розумного вибору. Наприклад, у стратегіях NPC можуть оцінювати економічні рішення, у шутерах – визначати оптимальні позиції для атаки або оборони, а в рольових іграх – вирішувати, коли допомагати союзнику чи ухилятися від бою.

Серед недоліків цього методу можна виділити складність налаштування функцій корисності. Для досягнення реалістичної поведінки потрібно ретельно підібрати ваги, формули та нормалізацію значень, інакше NPC може поводитися нелогічно або надто передбачувано. Також система може бути обчислювально затратною, якщо одночасно оцінюється велика кількість дій і параметрів.

У середовищі Unity Utility AI можна реалізувати за допомогою ScriptableObject для зберігання наборів дій та їхніх функцій корисності, а також спеціальних менеджерів прийняття рішень, що обчислюють найоптимальніший вибір у реальному часі [9]. Це дозволяє створювати адаптивні системи, де NPC

реагує на зміни ігрового світу не лише умовно, а на основі оцінки власних «інтересів» та «пріоритетів».

Таким чином, Utility AI поєднує елементи логіки, математики та психології прийняття рішень, забезпечуючи більш глибоке моделювання інтелектуальної поведінки NPC. Цей метод є сучасною альтернативою традиційним FSM і Behavior Tree, дозволяючи створювати гнучких, реалістичних і контекстно чутливих агентів у відеоіграх.

1.1.4 Метод керування NPC засобами ієрархічної машини станів(HFSM)

Ієрархічна машина станів (HFSM, Hierarchical Finite State Machine) є розширенням класичної машини скінченних станів, що дозволяє структурувати поведінку персонажа у вигляді багаторівневої ієрархії [33]. На відміну від базових FSM, де кожен стан є незалежною сутністю, HFSM вводить поняття суперстанів (або надстанів) і підстанів, що дозволяє групувати схожі стани та формувати узагальнені поведінкові моделі. Таким чином, HFSM дозволяє зменшити дублювання логіки, підвищити масштабованість системи та зробити поведінку NPC більш впорядкованою.

Основна ідея HFSM полягає у тому, що стан нижчого рівня може успадковувати загальні властивості та переходи від стану вищого рівня. Наприклад, NPC може мати суперстан «Бойова поведінка», який включає підстани «Переслідування», «Атака», «Відступ». Спільні перевірки, як-от визначення дистанції до цілі, логіка вибору зброї або оцінка рівня небезпеки, можуть бути зосереджені у суперстані, тоді як конкретна поведінка реалізується в підстанах. Такий підхід значно знижує складність розробки у порівнянні з традиційними FSM [10], де кожен стан вимушений дублювати частину загальної логіки. Приклад ієрархічної структури HFSM показано на рисунку 1.4.

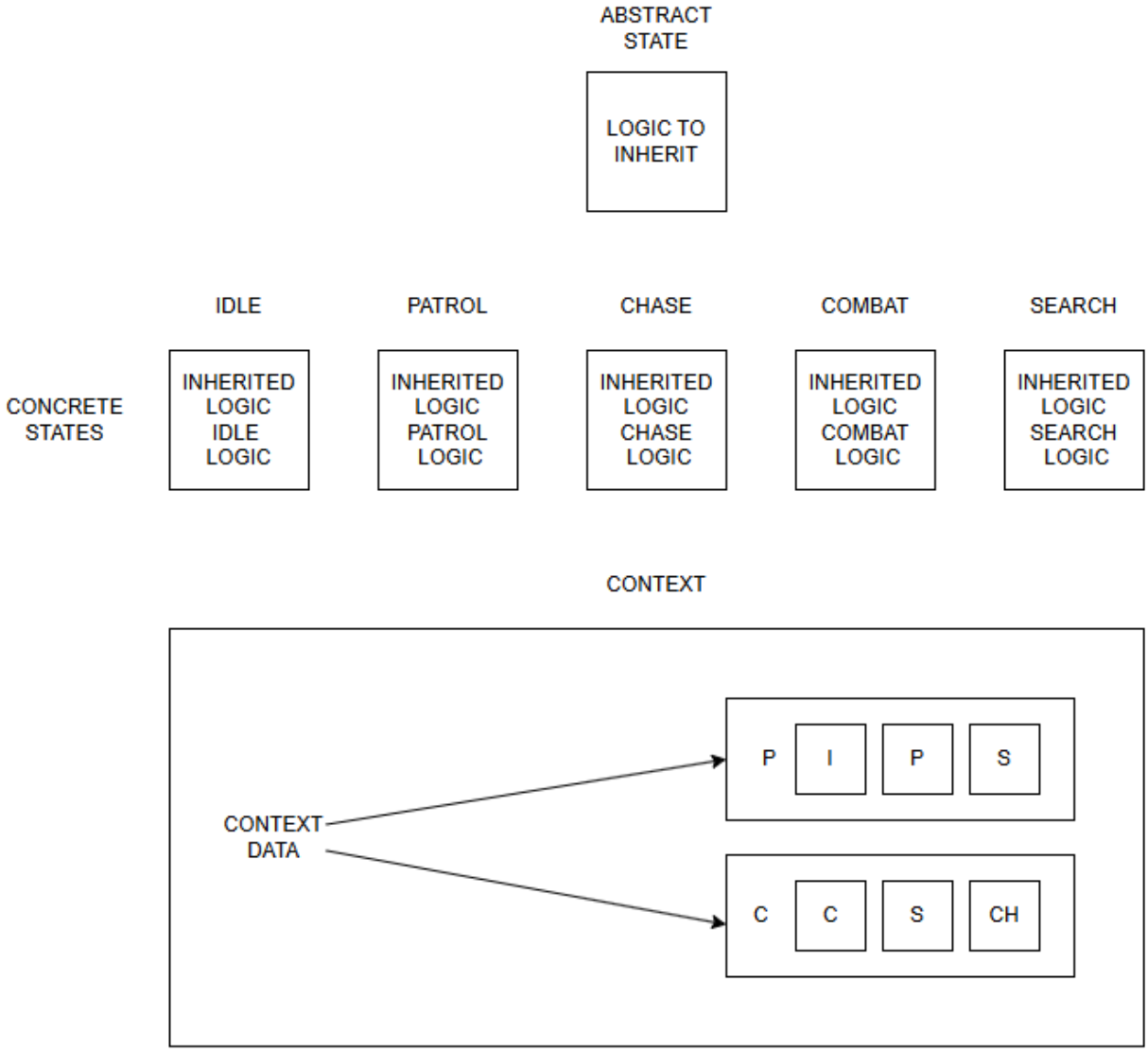

Рисунок 1.4 – Структура HFSM

Попри те, що HFSM є достатньо потужною технікою, вона рідше використовується у базових реалізаціях поведінки NPC у популярних ігрових рушіях. Причиною цього є те, що стандартні інструменти – зокрема Behavior Tree та класична FSM – широко підтримуються інженерними бібліотеками, документацією та інтегрованими редакторами. HFSM, натомість, у багатьох рушіях доводиться реалізовувати вручну, що вимагає певного технічного досвіду та продуманої архітектури.

У сучасних іграх метод HFSM застосовується переважно там, де необхідна висока передбачуваність поведінки [13], чітка структура, контроль на

низькому рівні та можливість легко відслідковувати стан NPC під час налагодження. HFSM добре підходить для моделей з великим набором взаємопов’язаних станів, наприклад: поведінка ворогів у слешерах, станова логіка супротивників у платформерах або персонажів, що виконують складні скриптові сценарії.

Однак класичні реалізації HFSM мають певні недоліки – зокрема громіздкість у структурі переходів [11], складність у відстеженні глибокої ієрархії та обмежену підтримку інструментами в ігрових рушіях. Це створює потребу у більш гнучких, оптимізованих та інструментально зручних підходах до побудови HFSM, які дозволяють розробникам ефективно масштабувати поведінку NPC без втрати керованості системи. Саме тому актуальним є створення вдосконаленої, оптимізованої моделі HFSM, орієнтованої на потреби сучасних відеоігор та інтеграцію в ігрові рушії на кшталт Unity.

1.2 Проблеми та обмеження існуючих рішень

Попри значний прогрес у розвитку систем керування поведінкою NPC, існуючі методи – такі як Behavior Trees, класичні машини станів (FSM), Utility AI та ієрархічні FSM (HFSM) – мають низку обмежень, що ускладнюють їх використання в сучасних відеоіграх. Кожен із підходів вирішує певні завдання, проте жоден із них не є універсальним, що зумовлює потребу у вдосконалених моделях та більш гнучких архітектурних рішеннях [12].

Основним обмеженнями стандартних машин станів є погана масштабованість, зі збільшенням кількості станів система швидко перетворюється на станову сітку з великою кількістю переходів. Дублювання логіки між станами, особливо коли NPC має схожі підтипи поведінки є ще однією проблемою. Також додається складність модифікацій, додавання нового стану або переходу часто потребує змін у багатьох частинах системи.

Відсутність ієрархії робить FSM менш ефективною під час роботи з багатоетапними або контекстно залежними поведінками.

ВТ є популярним інструментом, однак вони теж мають свої недоліки. Наприклад, занадто декларативний характер тож дерева поведінки добре структурують задачі, але гірше підходять для станів, що мають постійні внутрішні умови або довготривалі дії [15]. Важкість налагодження у великих проєктах робить її незручною для дебагу, дерево може містити сотні вузлів, що ускладнює пошук помилок. Існують також проблеми із пріоритезацією, якщо структура дерева недостатньо гнучка або реалізація не передбачає динамічних змін.

Utility AI пропонує гнучкий підхід на основі кількісної оцінки бажаності дій, але має свої обмеження. Складність у проєктуванні функцій корисності є важкою задачею, розробник змушений вручну створювати криві, ваги й формули. Високі вимоги до налаштування призводять до того, що навіть невеликі зміни у параметрах можуть суттєво вплинути на поведінку NPC [9]. Через відсутність явної структури станів NPC може непередбачувано перемикатися між діями, якщо система побудована без додаткових обмежень, а також складність відлагодження поведінки, що виникає з багатьох взаємопов’язаних факторів.

Попри очевидні переваги ієрархічної організації станів, класичні HFSM також мають свої недоліки. Складність реалізації через те, що базові ігрові рушії рідко пропонують готові інструменти, тож розробник вимушений створювати архітектуру з нуля. Громіздкість ієрархій у великих проєктах без чіткої структури HFSM може перетворитися на складну систему з важкопередбачуваною логікою. Жорстка структура переходів робить складнішим забезпечення гнучкості без ризику втратити контроль над поведінкою. Відсутність стандартів реалізації ускладнює підтримку й розширення системи в командних проєктах.

Усі ці проблеми свідчать про необхідність створення більш зручної, модульної та оптимізованої моделі керування поведінкою NPC, яка зберігає

структурованість HFSM, але усуває складність реалізації та забезпечує простоту масштабування.

1.3 Постановка задачі дослідження

Таким чином, дослідження методів керування поведінкою неігрових персонажів (NPC) є актуальним завданням у галузі розробки відеоігор. Від якості реалізації системи поведінки NPC безпосередньо залежить рівень занурення гравця, складність ігрового процесу та загальне враження від гри. Прийнято рішення щодо розроблення програмного застосунку, який дозволяє порівняти ефективність трьох підходів до моделювання поведінки NPC, а саме: машини станів (Finite State Machine), дерева поведінки (Behavior Tree) та системи на основі корисності (Utility AI).

Об’єктом дослідження є поведінка неігрових персонажів у відеоіграх.

Метою дослідження є аналіз, порівняння та практична реалізація різних методів керування поведінкою NPC у середовищі Unity з використанням мови програмування C#.

Для досягнення мети необхідно вирішити такі завдання:

– провести аналіз літературних джерел та існуючих підходів до моделювання поведінки NPC у відеоіграх;

– дослідити принципи побудови систем на основі FSM, Behavior Tree та Utility AI;

– сформувати алгоритми функціонування кожного з методів керування поведінкою NPC;

– розробити прототип програмного застосунку в Unity, що реалізує вибрані методи;

– здійснити тестування створених моделей і порівняти їх ефективність, гнучкість та зручність налаштування;

– зробити висновки щодо доцільності використання кожного з методів у різних типах відеоігор.

У результаті виконання роботи буде створено інтерактивний прототип, який демонструє особливості, переваги та недоліки кожного з розглянутих методів керування поведінкою NPC, що дозволить визначити найбільш оптимальний підхід для подальшого застосування в ігрових проєктах.

2 ТЕОРЕТИЧНІ ОСНОВИ КЕРУВАННЯ ПОВЕДІНКОЮ NPC ЗАСОБАМИ HFSM У ВІДЕОІГРАХ

2.1 Концепція та поняття HFSM

2.1.1 Основна ідея методу HFSM

Ієрархічна машина станів (HFSM) є розвитком класичної скінченної машини станів і пропонує більш гнучкий, масштабований і структурований підхід до моделювання поведінки агентів [33]. Основна ідея HFSM полягає у введенні багаторівневої структури, де поведінка організується не у вигляді плоского набору незалежних станів, а у формі ієрархії, що складається із суперстанів і підстанів. Завдяки цьому різні рівні логіки можуть співіснувати та виконуватися паралельно, утворюючи стратегію й тактику поведінки одночасно.

На відміну від традиційної FSM, у якій кожен стан є автономною одиницею [14], HFSM дозволяє групувати пов’язані стани в більші поведінкові блоки. Суперстани визначають загальний контекст, у якому перебуває агент, наприклад «патрулювання», «переслідування» чи «пошук», а підстани описують конкретні дії в межах цього контексту. Завдяки такій структурі NPC може одночасно виконувати локальну поведінку і реагувати на глобальні умови, що робить модель природнішою та гнучкішою.

HFSM дозволяє уникнути однієї з головних проблем класичної FSM – надмірного збільшення кількості станів. Коли поведінка ускладнюється, плоска FSM дуже швидко перетворюється на заплутану мережу переходів [15], у якій важко розібратися і яку складно розширювати. Ієрархічний підхід вирішує цю проблему шляхом інкапсуляції логіки у суперстанах: загальні переходи й умови перевіряються на верхньому рівні, а детальна поведінка – на нижньому. Таким чином, код стає чистішим, поведінка NPC – модульнішою, а система – значно легшою для подальшої підтримки. Приклад можна побачити на рисунку 2.1.

Рисунок 2.1 – Приклад надмірної кількості станів

У реальних іграх NPC повинні враховувати набагато більше станів, ніж у нашому наведеному вище прикладі [16]. Він реагує на багато реакцій гравців, що робить його кращим агентом. Отже, зі зростанням складності кількість станів починає швидко зростати, що ускладнює керування. Якщо ми хочемо внести навіть невелику зміну, нам потрібно врахувати всі залежні переходи. Щоб вирішити цю проблему, використоємо ієрархічний автомат станів. HFSM передбачає ієрархію, де FSM вкладені в інший FSM [17]. Групуючи стани разом з однаковими вихідними переходами, можна зменшити кількість переходів. Це початкова інтерпретація ієрархії на рівні станів. Вкладені стани прийматимуть лише підмножину алфавіту введення/виведення. Якщо потрібно внести зміни,

нам потрібно лише редагувати певний рівень ієрархії та нижче нього. Це спрощує внесення змін та розуміння коду, а також зменшує складність. На рисунку 2.2 наведено приклад того, як може виглядати структура проєкту HFSM.

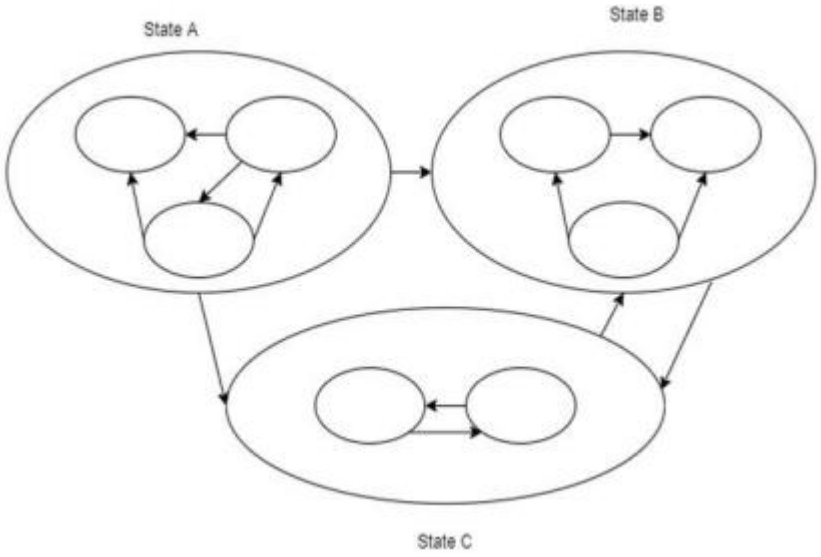

Рисунок 2.2 – Приклад структури HFSM

Ключовою перевагою HFSM є те, що підстани автоматично успадковують контекст свого суперстану [18]. Наприклад, суперстан «Переслідування» може визначати загальні правила взаємодії із ціллю, такі як максимальна дистанція або обмеження шляху. Усі підстани на зразок «біг навпростець», «обхід перешкоди» чи «перерахунок маршруту» отримують доступ до цих даних без дублювання коду. Це значно зменшує кількість взаємозалежностей між станами та дозволяє легко додавати нові поведінки [19].

Ієрархічна машина станів також забезпечує природні переходи між рівнями. Якщо поточний підстан не знаходить підходящої умови для переходу, запит автоматично передається на вищий рівень. Це дозволяє NPC не «застрягати» у дрібних діях та гарантовано повертатися до ширшого контексту поведінки, що моделює більш логічну, послідовну та адаптивну реакцію на зміну ситуації.

У підсумку HFSM є не просто розширенням FSM, а потужним архітектурним підходом, який дозволяє поєднати чіткість скінченних станів з гнучкістю ієрархічних структур [20]. Завдяки цьому ієрархічні машини станів ідеально підходять для створення поведінки NPC, оскільки дозволяють побудувати модульну, універсальну і легко масштабовану систему, що відображає складні багаторівневі реакції персонажів.

2.1.2 Алгоритм методу HFSM

Алгоритм функціонування ієрархічної машини станів ґрунтується на послідовній обробці логіки на кількох рівнях ієрархії, що дає змогу формувати як загальний поведінковий контекст, так і конкретні дії NPC [23]. Робота HFSM починається з ініціалізації, під час якої створюється кореневий стан, до якого одразу додається перший активний суперстан та відповідний підстан. На цьому етапі кожен стан отримує можливість виконати власну функцію початкової підготовки через метод Enter, що дозволяє завантажити необхідні дані або встановити параметри для подальшої роботи.

Після ініціалізації HFSM переходить у режим циклічного оновлення, який відбувається на кожному кадрі. У цьому циклі спочатку виконується логіка активного суперстану, що відповідає за глобальні перевірки умов і зміну загального контексту поведінки. Суперстан може визначати, чи варто продовжувати поточну поведінку, чи, можливо, середовище змінилося настільки, що NPC повинен перейти до іншого типу дій [25]. Після виконання логіки суперстану оновлюється підстан, який відповідає за конкретну поведінку: рух, обробку анімації, зміну напрямку, пошук цілі або іншу локальну дію.

Однією з ключових частин алгоритму є перевірка умов переходу. Кожний стан, незалежно від рівня, може визначати власні умови, за яких поведінка NPC має змінитися. Наприклад, підстан руху може перевіряти, чи досягнуто точки

патруля, а суперстан – чи з’явився гравець у полі зору. Якщо умова виконана, HFSM виконує послідовність виходу з поточного стану через метод Exit, після чого активується інший стан з викликом Enter. Цей процес дозволяє забезпечити стабільність і передбачуваність поведінки NPC, оскільки кожний стан відповідає за власний цикл життєдіяльності. Приклад блок схеми наведено на рисунку 2.3.

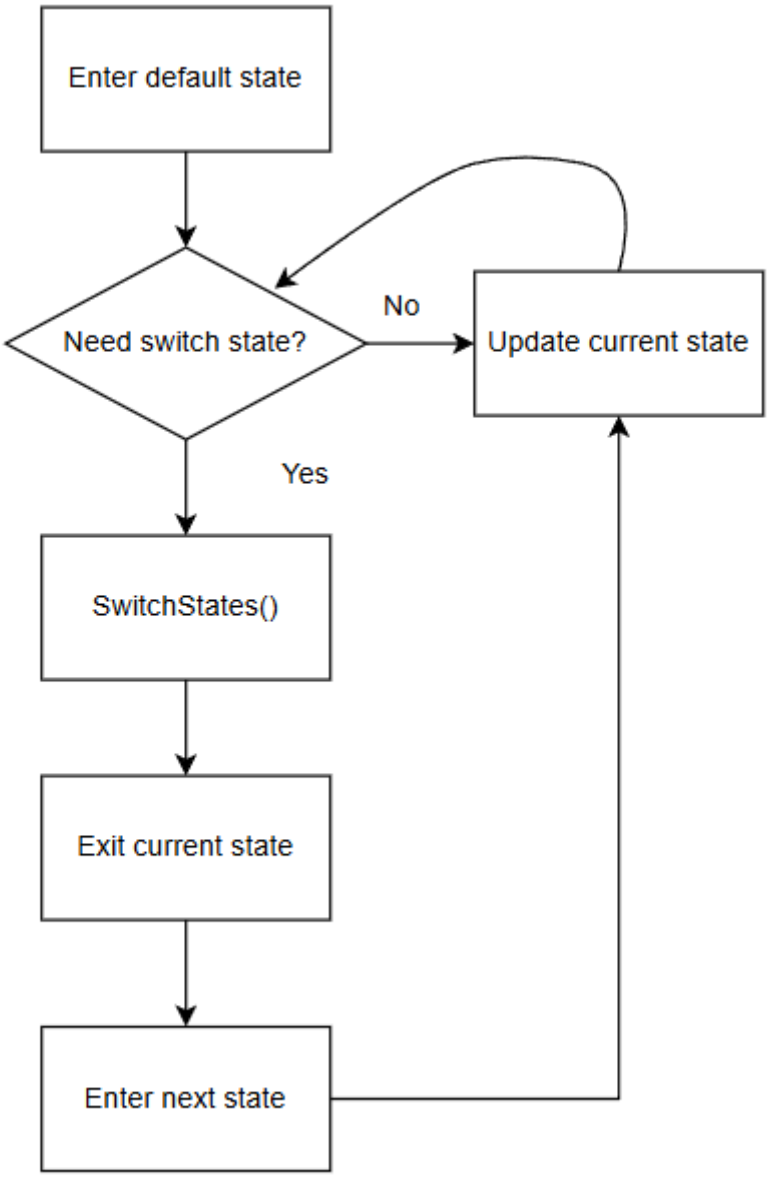

Рисунок 2.3 – Блок схема HFSM

Особливістю алгоритму HFSM є механізм підйому до вищого рівня. Якщо поточний підстан не знайшов умови для переходу, запит автоматично передається суперстану. Таким чином HFSM запобігає ситуаціям, коли NPC міг би застрягти у локальній логіці, і гарантує, що поведінка завжди повертається до більш загальної моделі. Це також робить можливим плавний перехід між різними суперстанами, наприклад між переслідуванням і патрулюванням або між пошуком і поверненням до базової позиції [27].

Метод Ієрархічних Скінченних Автоматів (HFSM) є вдосконаленим алгоритмічним підходом до керування поведінкою NPC, який розширює концепцію класичного Скінченного Автомата (FSM) шляхом запровадження ієрархічної структури станів [28]. Ця структура дозволяє значно спростити моделювання складної логіки, оскільки вона організована у вигляді дерева, де загальна поведінка інкапсулюється у Суперстанах (батьківських станах), а детальні дії – у Підстанах (дочірніх станах). Суперстан визначає спільну логіку та умови для всіх своїх підстанів, забезпечуючи механізм успадкування поведінки та даних, що сприяє повторному використанню коду і підвищенню модульності системи.

Алгоритм HFSM виконується ітеративно в ігровому циклі. На кожній ітерації відбувається сувора пріоритезація перевірки умов переходу. Спочатку перевіряються Глобальні Переходи, які мають найвищий пріоритет і можуть відбутися незалежно від поточного активного стану NPC, що є критично важливим для негайної реакції на зовнішні, небезпечні події (наприклад, перехід до стану «Смерть») [31].

Якщо глобальний перехід не відбувся, алгоритм переходить до перевірки логіки на рівні активного Суперстану. Тут ключовим є механізм вихідних переходів Суперстану: якщо умова, визначена на цьому рівні, виконується (наприклад, «ворог втік»), NPC негайно виходить з будь-якого Підстану, в якому він перебував, забезпечуючи високий пріоритет логіки вищого рівня над поточною детальною дією. Це запобігає застряганню NPC у дрібних станах, коли потрібна кардинальна зміна поведінки.

Тільки якщо не спрацювали ні глобальні переходи, ні вихідні переходи Суперстану, алгоритм перевіряє внутрішні переходи активного Підстану. Ці переходи керують деталізацією поведінки (наприклад, перехід від «Йти до цілі» до «Атакувати, досягнувши дистанції»). Якщо відбувається успішний перехід, виконуються відповідні методи OnExit() для старого стану та OnEnter() для нового. Якщо ж жодна умова переходу не виконується на жодному рівні, виконується основний код активного Підстану (OnUpdate()), який здійснює необхідні дії NPC протягом поточного кадру. Таким чином, HFSM вирішує проблему експоненційного зростання кількості зв’язків у традиційних FSM, замінюючи складну «плоску» мережу станів на організовану ієрархію з чітко визначеною пріоритетністю логіки [34].

Під час переходу між суперстанами HFSM не лише змінює локальну дію NPC, а й перебудовує весь контекст поведінки. Вхід у новий суперстан оновлює логіку прийняття рішень, ініціалізує нові параметри, а також автоматично активує відповідний підстан. Така поведінка робить NPC більш адаптивним, оскільки перехід між типами активності супроводжується перезавантаженням внутрішніх механізмів і водночас зберігає структуру ієрархії [33].

Завершення роботи HFSM відбувається шляхом коректного виходу зі станів, що гарантує звільнення ресурсів і завершення всіх активних процесів. Це є важливим аспектом, оскільки дозволяє уникнути залишкових ефектів поведінки, які могли б вплинути на подальшу логіку NPC.

У підсумку алгоритм HFSM забезпечує гнучку, багаторівневу й структуровано організовану модель поведінки, яка одночасно дозволяє NPC реагувати на глобальні зміни й виконувати конкретні локальні дії. Він забезпечує плавність переходів, природність рішень та можливість легкого масштабування, що робить його одним з найбільш ефективних інструментів для побудови сучасного ігрового штучного інтелекту.

2.2 Принципи ієрархічної організації станів у HFSM

Ієрархічна організація станів у HFSM забезпечує структуроване, модульне та добре кероване представлення поведінки NPC, що особливо важливо у складних інтерактивних системах. На відміну від класичних FSM, де кількість переходів швидко зростає зі збільшенням кількості станів, HFSM дозволяє оптимізувати архітектуру шляхом групування логічно пов’язаних станів у багато-рівневі структури. Такий підхід не лише спрощує розробку, але й забезпечує максимальну гнучкість та легке розширення системи в майбутньому [33].

Функціональні принципи ієрархічної організації станів у HFSM:

− використання суперстанів як контейнерів для логічно об’єднаних груп поведінок NPC, які містять узагальнену логіку та правила;

− можливість агрегувати повторювані перевірки у верхніх рівнях ієрархії (наприклад, наявність цілі, стан здоров’я, глобальні обмеження), що дозволяє уникати дублювання коду в підстанах;

− реалізація глобальних дій у суперстанах, які виконуються незалежно від активного підстану, що забезпечує узгодженість поведінки в умовах динамічної зміни ситуацій;

− створення підстанів, які відповідають за конкретні реакції NPC – переслідування, атака, оборона, патрулювання, відступ тощо – при цьому наслідуючи поведінкові обмеження свого суперстану;

− формування активної конфігурації станів, де NPC одночасно перебуває на кількох рівнях ієрархії, що дозволяє розділяти високорівневу логіку (наприклад, «бойова поведінка») та низькорівневу («атака ближнього бою», «постійний моніторинг ворога»);

− забезпечення передбачуваності роботи системи за рахунок чітко сформованих внутрішніх та зовнішніх переходів між станами, що дозволяє NPC адекватно реагувати на зміну ігрового середовища;

− можливість створення складних поведінкових моделей без неконтрольованого зростання кількості переходів, характерного для звичайних FSM;

− логічне та структурне розділення поведінок NPC, що робить загальну архітектуру більш зрозумілою для розробників та значно спрощує подальше масштабування.

Нефункціональні властивості та переваги HFSM у контексті керування NPC:

− висока масштабованість: можливість легко додавати нові стани, суперстани або рівні ієрархії без необхідності переписувати наявну логіку;

− універсальність: структура HFSM підходить для NPC будь-якого рівня складності – від простих ворогів до просунутих агентів з багатоступеневою поведінкою;

− розширюваність: архітектура підтримує додавання нових поведінкових модулів, а також інтеграцію зовнішніх алгоритмів (навігація, планування, прийняття рішень);

− висока читабельність та підтримуваність: ієрархічна логіка зменшує ризик плутанини у великій кількості станів, полегшуючи розуміння роботи системи навіть новим членам команди;

− модульність: кожен суперстан може розглядатися як окремий модуль логіки, який легко перенести в інший проєкт або адаптувати під нові сценарії;

− прозорість поведінки: розробник чітко розуміє, які стани NPC активні в даний момент і які правила керують переходами між ними, що полегшує відлагодження та аналіз поведінки;

− стійкість до розширення і підтримки: у HFSM складні поведінки не призводять до експоненційного зростання кількості переходів, тому система залишається керованою навіть у дуже великих проєктах;

− логічна узгодженість поведінки NPC: незалежно від кількості підстанів, поведінка на рівні суперстану залишається стабільною, що гарантує передбачувані реакції в будь-яких сценаріях.

Ієрархічна організація станів у HFSM формує потужний інструментарій для моделювання складних поведінок NPC у відеоіграх. Вона поєднує структурованість, гнучкість, масштабованість та можливість повторного використання логіки. Завдяки цьому HFSM є одним із найбільш ефективних та універсальних способів керування поведінкою ігрових агентів у сучасних інтерактивних системах.

2.3 Моделювання поведінки NPC на основі HFSM

Моделювання поведінки NPC за допомогою HFSM передбачає формалізацію логіки дій персонажа у вигляді багаторівневої структури, де кожен рівень відповідає певному ступеню абстракції. Суперстани описують загальні поведінкові режими NPC, тоді як підстани деталізують конкретні дії, реакції та сценарії. Такий підхід дозволяє створити керовану, передбачувану та масштабовану систему поведінки, яка легко адаптується до складних динамічних умов гри.

Unity підтримує мову програмування C#, використовуючи методи об’єктно-орієнтованого програмування, засновані на принципах SOLID, що дозволяє легко створювати, модифікувати та розширювати систему [16]. Принцип SOLID – це точний принцип програмування, започаткований Робертом К. Мартіном, який базується на п’яти принципах, зокрема:

– (S)RP або принцип єдиної відповідальності: кожен клас повинен мати одну відповідальність, одне завдання або одну мету;

– (O)CP або принцип відкритості/закриття: розширити поведінку класу без його зміни;

– (L)SP або принцип заміщення Ліскова: цей принцип гарантує, що будь-який клас, який є дочірнім до батьківського класу, може використовуватися замість свого батьківського класу без неочікуваної поведінки;

– (I)SP або принцип розділення інтерфейсів: не змушуйте жодного клієнта реалізовувати інтерфейс, який йому не відповідає. Цей принцип допомагає зменшити побічні ефекти та частоту необхідних змін;

– (D)IP або принцип інверсії залежностей: високорівневі модулі не повинні залежати від низькорівневих модулів, але вони повинні залежати від абстракцій [36].

Першим етапом моделювання є визначення загальних поведінкових режимів NPC, які в HFSM реалізуються через суперстани. Це можуть бути такі глобальні стани, як бойова поведінка, поведінка поза боєм, патрулювання або дослідження території. Кожен суперстан формує логічну основу для всієї групи дій, що відбуватимуться усередині нього. На нижчому рівні розташовані підстани, які відповідають за конкретні тактичні рішення та дії, наприклад переслідування цілі, ближній бій, відступ, взаємодію з об’єктами або відновлення ресурсів. Така ієрархія дозволяє NPC одночасно залишатися у високорівневому логічному стані й виконувати конкретну дію, що забезпечує природність та узгодженість поведінки.

Переходи між станами в HFSM формуються на основі внутрішніх і зовнішніх умов. Зовнішні умови можуть включати виявлення гравця, рівень загрози чи зміну дистанції до цілі, у той час як внутрішні враховують параметри самого NPC: здоров’я, енергію, внутрішні таймери або запаси ресурсів. Завдяки такому підходу система може здійснювати як локальні переходи між підстанами одного суперстану, так і глобальні зміни поведінкового режиму при активації іншого суперстану. Це забезпечує плавність та логічність змін поведінки, навіть у складних ситуаціях.

Важливою складовою є механізм обробки подій, який дозволяє NPC динамічно реагувати на зміни в середовищі. Події можуть перехоплюватися як на рівні окремих підстанів, так і на рівні суперстанів. Наприклад, сигнал про появу гравця в полі зору може активувати підстан переслідування в рамках бойового суперстану, тоді як критичне зниження здоров’я може спровокувати перехід до стану відступу або навіть до суперстану «поведінка поза боєм». Така

модель робить NPC здатним поводитися контекстно, реагувати на непередбачувані ситуації та демонструвати більш «живу» поведінку.

Однією з найбільших переваг HFSM є повторне використання логіки та модульність. Оскільки підстани можуть бути вкладені у різні суперстани, а загальні перевірки винесені у верхні рівні ієрархії, система уникає дублювання коду та спрощує підтримку [33]. Наприклад, підстан переслідування може застосовуватися як у бойовому режимі, так і у режимі розвідки, без необхідності створення окремих реалізацій. Це робить HFSM універсальною основою для моделювання NPC різної складності – від простих ворогів до складних агентів зі складними поведінковими патернами.

Алгоритмічна структура HFSM базується на циклічному визначенні активного суперстану, виборі відповідного підстану та виконанні його логіки, паралельно моніторячи умови для можливих переходів. У разі активації нових умов система змінює підстан або суперстан, а потім знову повторює цикл. Такий процес забезпечує NPC стабільну, узгоджену та водночас гнучку поведінку, здатну адаптуватися до нових обставин у режимі реального часу.

Розглянемо алгоритм роботи HFSM:

Крок 1. Визначити активний кореневий стан (Root State), який відповідає за глобальний режим поведінки NPC, наприклад «Патрулювання» або «Бойова поведінка».

Крок 2. Ініціалізувати активний підстан кореневого стану відповідно до умов сценарію. Наприклад, у суперстані «Патрулювання» активується підстан «Рух за маршрутом».

Крок 3. Виконати логіку поточного підстану: обчислення навігаційних маршрутів, перевірка умов переходу, аналіз сенсорних даних тощо.

Крок 4. Перевірити локальні умови переходу підстану (наприклад, NPC досяг patrol-point або бачить гравця).

Якщо умова досягнута, виконати локальний перехід до іншого підстану в межах того самого суперстану.

Крок 5. Якщо локальні умови не спрацьовують, перевірити глобальні умови переходу суперстану (наприклад, рівень загрози, дистанція до гравця, стан здоров’я NPC).

Крок 6. Якщо виконано глобальну умову, завершити роботу поточного суперстану, деактивувати всі його підстани та перейти до нового суперстану (наприклад, з «Патрулювання» → у «Переслідування»).

Крок 7. Ініціалізувати структуру підстанів для нового суперстану (наприклад, у «Переслідуванні» активується підстан «Рух до цілі»).

Крок 8. Виконати логіку нового активного підстану та продовжувати обробку поведінки NPC у циклі оновлення.

Крок 9. Якщо стан або підстан завершує роботу (за таймером, досягнутим маршрутом або умовою переходу), виконати процедуру виходу (ExitState), звільнити ресурси або підготувати NPC до наступного стану.

Крок 10. Перевірити, чи необхідно повернутися до попереднього кореневого стану згідно з умовами гри.

Якщо так – активувати попередній суперстан і ініціалізувати його відповідний підстан.

Крок 11. Якщо умови для зміни станів завершені, зафіксувати поточну конфігурацію HFSM і продовжувати виконання NPC у цій поведінковій моделі.

Завдяки своїй багаторівневій структурі та чіткій організації логіки HFSM дозволяє формалізувати навіть найскладніші сценарії поведінки NPC, забезпечуючи модульність, легкість масштабування та повторне використання логічних блоків. Це робить HFSM одним із найефективніших методів для створення реалістичних і добре контрольованих моделей поведінки персонажів у сучасних ігрових проєктах.

2.4 Математична модель ієрархічної машини станів

2.4.1 Модель FSM

$$FSM = (S, S_0, T, C, A), \tag{2.1}$$

де S – кінцева множина станів;

S_0 – початковий стан системи;

T – множина переходів між станами;

C – множина умов активації переходів;

A – множина дій, що виконуються у станах.

Оновлення стану FSM відбувається за правилом:

$$s(t+1) = \begin{cases} s_j, \text{якщо активний перехід } t_{ij} \\ s_i, \text{якщо переходу немає.} \end{cases}, \tag{2.2}$$

У FSM кожен стан є ізольованою одиницею та не має вкладених станів. Це означає, що вся логіка поведінки NPC у конкретний момент часу визначається виключно активним станом, перехід між станами здійснюється лише на одному рівні, а поведінка не може бути розділена на підрівні – для кожної дрібної зміни необхідно створювати окремий стан. Оскільки FSM не підтримує ієрархії, множина переходів збільшується пропорційно до кількості станів [15].

Діаграма станів чи граф переходів – графічне уявлення множини станів і переходів. Є розміченим орієнтованим графом, вершини якого є станами, а дуги є переходами з одного стану в інший. Приклад графу переходів показано на рисунку 2.4.

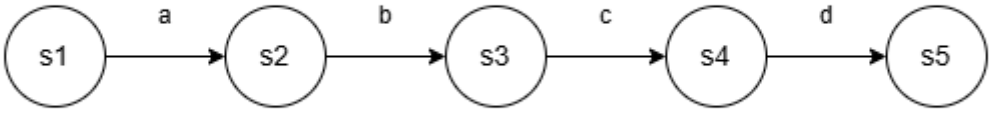

Рисунок 2.4 – Граф переходів

Таблиця переходів – табличне уявлення станів і переходів. Як правило, у такій таблиці для кожного рядка відповідає один стан, а для кожного шпальти відповідає один допустимий вхідний сигнал. У комірці на перетині рядка та стовпця записується стан результату. Приклад показано у таблиці 2.1.

Таблиця 2.1 – Таблиця переходів

Початковий стан	**Вхідний сигнал a**	**Вхідний сигнал b**	**Вхідний сигнал c**	**Вхідний сигнал d**
S1	S2	S1	S1	S1
S2	S2	S3	S2	S2
S3	S3	S3	S4	S3
S4	S4	S4	S4	S5

Модель FSM описує поведінку NPC як послідовність плоских станів без ієрархічних зв’язків. Це робить FSM ефективною у простих сценаріях, однак у разі складних моделей NPC така структура швидко втрачає гнучкість, потребує великої кількості переходів і ускладнює масштабування [13]. Саме ці недоліки мотивують перехід до HFSM як більш адаптивної та структурованої моделі.

2.4.2 Модель HFSM

Ієрархічна машина станів (HFSM) є узагальненням класичної скінченної машини станів (FSM) та використовується для формального опису складної поведінки NPC у вигляді багаторівневої структури взаємопов’язаних станів. На

відміну від звичайної FSM, яка оперує лише плоским набором станів, HFSM дозволяє організувати стан простору у вигляді дерева, де кожен стан може містити підстани, утворюючи ієрархію довільної глибини. Така модель забезпечує кращу масштабованість, повторне використання логіки та адаптивність до змін середовища [33].

Математична модель HFSM забезпечує формальну основу для опису поведінки NPC у вигляді структурованої системи станів, дозволяючи моделювати складні сценарії, уникати надмірностей та покращувати керованість логіки поведінки в ігровому середовищі [25]. Ієрархічний підхід суттєво підвищує масштабованість моделі та забезпечує можливість повторного використання логіки на різних рівнях деталізації.

$$HFSM = (S, S_{root}, T, C, H), \tag{2.3}$$

де S – множина станів;

S_{root} – кореневий стан;

T – множина можливих переходів;

C – умови активації переходів;

H – відношення ієрархії між станами.

Активний стан s_i оновлюється за правилом:

$$s(t+1) = \begin{cases} s_j, \text{якщо активний перехід } t_{ij} \\ s_i, \text{інакше.} \end{cases}, \tag{2.4}$$

Таким чином, математична модель HFSM забезпечує формальну основу для опису поведінки NPC у вигляді структурованої системи станів, дозволяючи моделювати складні сценарії, уникати надмірностей та покращувати

керованість логіки поведінки в ігровому середовищі. Ієрархічний підхід суттєво підвищує масштабованість моделі та забезпечує можливість повторного використання логіки на різних рівнях деталізації.

2.5 Аналіз існуючих реалізацій HFSM у сучасних ігрових рушіях

Ієрархічні машини станів (HFSM) використовуються у сучасних ігрових рушіях для керування поведінкою NPC, проте рівень їхньої інтеграції та підтримки значно варіюється залежно від платформи та доступних інструментів. У Unity, одному з найпоширеніших рушіїв, відсутній повноцінний вбудований механізм для ієрархічних машин станів. Розробники переважно використовують Animator Controller, який дозволяє створювати базові FSM з елементами ієрархії у вигляді підстанів, проте ці інструменти обмежені здебільшого роботою з анімаціями. Для складніших моделей поведінки широко застосовуються сторонні рішення – такі як PlayMaker, NodeCanvas, Behaviour Machine – які забезпечують створення HFSM у вигляді візуальних графів. Основним недоліком є те, що такі інструменти часто недостатньо гнучкі та важко масштабуються при великій кількості поведінкових сценаріїв, а також можуть ускладнювати інтеграцію з кастомним ігровим кодом.

У Unreal Engine реалізація HFSM виглядає більш просунутою завдяки вбудованим засобам State Machine в Animation Blueprint, а також системі Behavior Trees, яку часто поєднують із FSM для створення багаторівневої логіки. Проте навіть тут HFSM здебільшого пов’язана з анімаційною системою, що ускладнює її використання для логіки NPC поза межами анімаційних переходів. Хоча Unreal Engine пропонує широкі можливості створення складних поведінкових моделей, він також потребує значних зусиль із налаштування та підтримки.

В інших рушіях, таких як CryEngine або Godot, HFSM зазвичай реалізується вручну або через сторонні модулі. Такий підхід забезпечує максимальну гнучкість і контроль, але водночас збільшує складність розробки, тестування та подальшої підтримки. Більшість ААА-студій створюють власні внутрішні системи HFSM, адаптовані під конкретні вимоги проєкту, оскільки універсальні інструменти не завжди здатні охопити всі необхідні сценарії.

Узагальнюючи, можна виділити основні властивості сучасної підтримки HFSM у різних рушіях:

− HFSM залишається одним з найефективніших підходів для організації складної та багаторівневої поведінки NPC;

− сучасні рушії не забезпечують повноцінної універсальної нативної підтримки HFSM, що змушує розробників створювати власні або частково кастомізовані реалізації;

− основними викликами залишаються масштабованість, інтеграція з різними системами гри, підтримка багаторівневої логіки та забезпечення гнучкості;

− у розробці складних ігор виникає потреба у створенні оптимізованої HFSM-моделі, що здатна адаптуватися під різні типи NPC та складні сценарії, – саме така модель буде представлена у третьому розділі дипломної роботи.

3 РЕАЛІЗАЦІЯ МОДЕЛІ КЕРУВАННЯ ПОВЕДІНКОЮ NPC ЗАСОБАМИ HFSM

3.1 Технічне середовище для реалізації HFSM

3.1.1 Аналіз існуючих інструментів для реалізації HSFM

Для реалізації ієрархічних машин станів (HFSM) у сучасних ігрових проєктах розробники використовують різні підходи та інструменти, залежно від можливостей рушія та конкретних вимог до поведінки NPC. Аналіз існуючих рішень дозволяє зрозуміти переваги та обмеження готових систем і обґрунтувати необхідність створення власної оптимізованої моделі.

У Unity одним із найпоширеніших інструментів для побудови логіки станів є Animator Controller. Він забезпечує базову реалізацію FSM і часткову підтримку ієрархічних структур завдяки механізму суперстанів. Перевагами цього підходу є наявність зручного візуального редактора, тісна інтеграція з анімаційною системою та простота створення нескладних моделей поведінки. Проте для складних NPC Animator Controller виявляється недостатньо гнучким: його важко масштабувати, він не призначений для багаторівневої логіки та сильно прив’язує поведінку до анімаційних переходів, що обмежує можливості розробника.

Для подолання цих обмежень у Unity широко застосовуються сторонні плагіни, такі як PlayMaker, NodeCanvas або Behaviour Machine. Вони надають розробнику візуальні графи, підтримку ієрархічних станів і можливість швидкого створення й тестування моделей поведінки NPC. Однак подібні рішення також мають суттєві недоліки: залежність від сторонніх інструментів, обмежена можливість глибокої кастомізації та труднощі зі створенням масштабованої системи для великої кількості NPC із різною логікою.

У рушії Unreal Engine підтримка складної поведінки NPC реалізується через поєднання State Machine у Animation Blueprint та Behavior Trees. Така комбінація дозволяє будувати ієрархічні структури, поєднувати високорівневі

рішення з деталізованими діями та інтегрувати логіку з анімацією персонажа. Проте й тут є обмеження: поведінка NPC часто залишається жорстко прив’язаною до анімаційної системи, що утруднює створення чистої логіки без врахування анімаційних переходів, а також знижує гнучкість у керуванні активними станами.

У великих AAA-проєктах розробники переважно створюють власні кастомні реалізації HFSM. Це дає повний контроль над структурою системи, дозволяє впроваджувати гнучкі патерни, оптимізувати продуктивність і масштабувати логіку відповідно до потреб гри. При цьому ці системи є складними в розробці, потребують ретельного проєктування архітектури, значного часу на тестування та підтримку, проте забезпечують найвищий рівень адаптивності та керованості.

Підсумовуючи аналіз, можна зазначити, що готові інструменти добре підходять для швидкого створення прототипів, але мають суттєві обмеження у гнучкості та масштабованості. Саме тому складні проєкти та реалістичні NPC зазвичай потребують власної реалізації HFSM. У контексті дипломного дослідження найбільш доцільним є створення власної ієрархічної машини станів у Unity, що дає змогу повністю контролювати побудову структури, формування ієрархії станів та логіку переходів, поєднуючи програмну гнучкість C# з можливостями рушія.

3.1.2 Вибір середовища та технологій для проведення дослідження

У межах кваліфікаційної роботи було розроблено метод керування поведінкою NPC на основі ієрархічної машини станів (HFSM). Вибір програмного забезпечення та технологій відіграв вирішальну роль, оскільки ефективність реалізації залежить від гнучкості архітектури, можливості масштабування, інтеграції з іншими компонентами гри та зручності тестування. Після аналізу існуючих підходів та інструментів було обрано середовище Unity

2024 та мову програмування C#, що забезпечують необхідні можливості для побудови HFSM будь-якої складності.

Unity виступає основною платформою реалізації, оскільки поєднує сучасний ігровий рушій з широким набором інструментів для роботи з 2D- та 3D-графікою, компонентною архітектурою, анімаційною системою та фізичним модулем. Рушій дозволяє безперешкодно інтегрувати користувацький код на C#, що відкриває можливість створення власних алгоритмів обробки станів, подій та переходів. Крім того, Unity забезпечує зручні засоби налагодження, профілювання та тестування NPC у різних сценаріях, що стало важливою перевагою під час розробки HFSM. Гнучкість рушія також дозволяє розширювати функціональність за допомогою сторонніх бібліотек або власних модулів, інтегруючи HFSM без будь-яких обмежень з боку інструментарію.

Основною мовою програмування було обрано C#, оскільки вона глибоко інтегрована з API Unity і забезпечує високу продуктивність при роботі з ігровими об’єктами. Мова підтримує об’єктно-орієнтовану парадигму, що дає змогу природно моделювати стани, суперстини, підстани та переходи HFSM. Завдяки можливості створення модульної архітектури з повторним використанням логіки C# значно полегшує розробку складних поведінкових моделей NPC, а також спрощує відлагодження та подальшу підтримку системи.

Замість використання стандартних рішень Unity на кшталт Animator Controller чи сторонніх візуальних плагінів було створено власну програмну реалізацію HFSM. Це дозволило забезпечити повний контроль над структурою ієрархії, логікою роботи станів, управлінням активними станами та обробкою подій. На відміну від готових інструментів, власна HFSM не має зовнішніх обмежень, легко масштабується під різні типи NPC, дозволяє додавати нові стани та поведінкові гілки без перебудови всієї архітектури. Такий підхід забезпечує гнучкість, необхідну для створення реалізованої в роботі моделі поведінки, та гарантує можливість подальшого розширення системи.

Реалізацію завдання дослідження можна розбити на наступні етапи:

Етап 1. Аналіз поведінки NPC та визначення необхідної ієрархії станів (суперстани, підстани, глобальні та локальні умови переходів).

Етап 2. Створення структури базового стану (Base State), яка включає методи EnterState, ExitState, UpdateState, CheckSwitchStates та можливість вкладеності станів.

Етап 3. Реалізація кореневих суперстанів NPC (наприклад, «Патрулювання», «Переслідування», «Бій») та визначення логіки їх переходів.

Етап 4. Реалізація підстанів кожного суперстану (наприклад: Idle, PatrolMove, ChaseMove, AttackDecision) та налаштування ієрархії.

Етап 5. Розробка і реалізація обробки подій та сенсорних перевірок (виявлення гравця, колізії, дистанції, станів таймерів).

Етап 6. Побудова класу StateMachine, який координує активні стани, їх взаємодію та переходи між рівнями ієрархії.

Етап 7. Інтеграція HFSM із системами Unity (NavMeshAgent, анімації, фізика, UI для дебагу).

Етап 8. Тестування NPC у різних сценаріях, порівняння продуктивності FSM і HFSM, аналіз стабільності та передбачуваності поведінки.

Етап 9. Оптимізація архітектури HFSM, виправлення дублювання логіки, удосконалення умов переходів та повторне тестування.

Етап 10. Підготовка результатів дослідження та документування архітектури HFSM у межах кваліфікаційної роботи.

Обрана комбінація Unity, C# та власної HFSM-архітектури надала оптимальний баланс між продуктивністю, масштабованістю та простотою реалізації. Вона дозволила створити ефективну, керовану та зрозумілу структуру поведінки NPC, яка безпосередньо взаємодіє з іншими системами рушія – анімацією, фізикою, навігацією та логікою ігрового світу. Це створює надійну технічну основу для побудови HFSM та подальшого дослідження її ефективності в умовах ігрового середовища.

3.2 Побудова архітектури HFSM

У даному дослідженні була реалізована ієрархічна машина станів (HFSM), яка покращує класичну FSM за рахунок впорядкування поведінки NPC у багаторівневу структуру. Структура HFSM проєктується таким чином, щоб кожен стан мав власну відповідальність, а переходи між станами були передбачуваними, локалізованими та легко розширюваними.

Спроєктована архітектура складається з таких основних елементів:

– фабрики станів;

– базового стану, що реалізує логіку, спільну для всіх станів;

– суперстанів і підстанів, які формують ієрархію поведінки;

– машини станів (контексту), яка керує NPC, зберігає дані, визначає переходи та викликає оновлення станів.

Усі компоненти взаємодіють за принципами ООП, забезпечуючи високу гнучкість і можливість розширення.

3.2.1 Контекст та фабрика станів

Модуль NPCStateMachine виконує роль контексту HFSM, а також центру управління NPC. Машина станів не містить логіки поведінки – вона лише надає доступ до даних. Уся поведінка винесена у стани, що відповідає принципам Single Responsibility.

Цей модуль містить усі дані та посилання. Так як уся поведінка знаходиться у станах, то за допомогою такої структури можна реалізувати будь який контекст. Завдяки наслідуванню від MonoBehaviour через контекст стани можуть отримувати доступ до будь яких даних.

В даному прикладі реалізовано NPC із можливостю патрулювання та переслідування. UML-діаграму класу показано на рисунку 3.1.

Лістинг 3.1 Контекст NPCStateMachine:

```
using System.Collections;
using TMPro;
using UnityEngine;
using UnityEngine.AI;

public class NPCStateMachine : MonoBehaviour
{
    NavMeshAgent _agent;
    public Transform chaseTarget;
    [SerializeField] float _viewAngle;
    [SerializeField] float _viewDistance;
    [SerializeField] float _stopChaseDistance;
    [SerializeField] float _idleTime;
    public Transform[] patrolPoints;
    [SerializeField] int _currentPointIndex = 0;

    [Header("UI")]
    public TextMeshProUGUI currentSuperStateText;
    public TextMeshProUGUI currentSubStateText;

    NPCBaseState _currentState;
    NPCStateFactory _states;

    [SerializeField] bool _canSeeTarget;
    [SerializeField] bool _isIdle;

    public float IdleTime
    {
        get { return _idleTime; }
        set { _idleTime = value; }
    }

    public NavMeshAgent Agent
    {
        get { return _agent; }
        set { _agent = value; }
    }

    public bool IsIdle
    {
        get { return _isIdle; }
        set { _isIdle = value; }
```

```
    }

    public int CurrentPointIndex
    {
        get { return _currentPointIndex; }
        set { _currentPointIndex = value; }
    }

    public bool CanSeeTarget
    {
        get { return _canSeeTarget; }
        set { _canSeeTarget = value; }
    }

    public float StopChaseDistance
    {
        get { return _stopChaseDistance; }
        set { _stopChaseDistance = value; }
    }

    public NPCBaseState CurrentState
    {
        get { return _currentState; }
        set { _currentState = value; }
    }

    private void Awake()
    {
        _states = new NPCStateFactory(this);
        _currentState = _states.Patrol();
        _currentState.EnterState();
    }

    private void Start()
    {
        Agent = GetComponent<NavMeshAgent>();
        patrolPoints =
GameObject.Find("PatrolPoints").GetComponentsInChildren<Transform>();
    }

    private void Update()
    {
        CanSeeTarget = CanSeeTargetCheck();
        _currentState.UpdateStates();
    }
```

```
        public bool CanSeeTargetCheck()
        {
            Vector3 directionToPlayer = chaseTarget.position - transform.position;
            if (directionToPlayer.magnitude > _viewDistance)
                return false;
            float angle = Vector3.Angle(transform.forward, directionToPlayer);
            if (angle > _viewAngle / 2f)
                return false;
            if (Physics.Raycast(transform.position + Vector3.up,
directionToPlayer.normalized, out RaycastHit hit, _viewDistance))
            {
                if (hit.transform == chaseTarget)
                    return true;
            }

            return false;
        }
        public void UpdateSuperStateText(string superStateText)
        {
            currentSuperStateText.text = superStateText;
        }
        public void UpdateSubStateText(string subStateText)
        {
            currentSubStateText.text = subStateText;
        }
    }
```

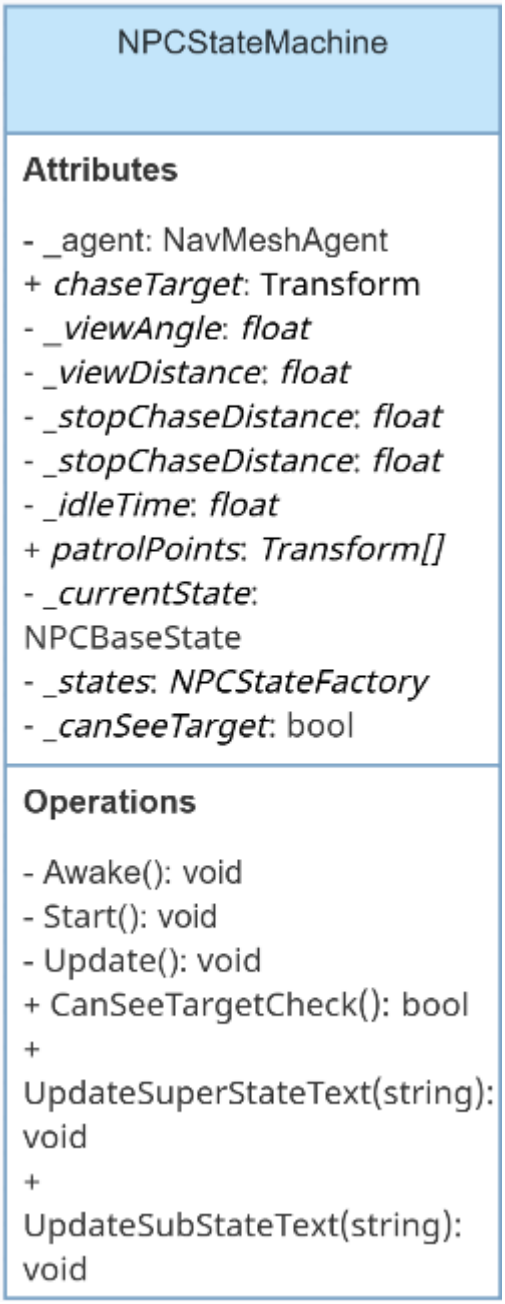

Рисунок 3.1 – UML діаграма NPCStateMachine

Модуль NPCStateFactory виконує роль фабрики станів у HFSM. Він відповідальний за створення станів та ініціалізацію. Завдяки такому підходу кожен стан має доступ до контексту та іншим станам, що робить перемикання між станами дуже легким та гнучким. UML-діаграму класу показано на рисунку 3.2.

Лістинг 3.2 Фабрика станів NPCStateFactory:

```
public class NPCStateFactory
{
    NPCStateMachine _context;

    public NPCStateFactory(NPCStateMachine currentContext)
    {
        _context = currentContext;
    }

    public NPCBaseState Chase()
    {
        return new NPCChaseState(_context, this);
    }
    public NPCBaseState Patrol()
    {
        return new NPCPatrolState(_context, this);
    }
    public NPCBaseState Idle()
    {
        return new NPCIdleState(_context, this);
    }
    public NPCBaseState PatrolMove()
    {
        return new NPCPatrolMoveState(_context, this);
    }
}
```

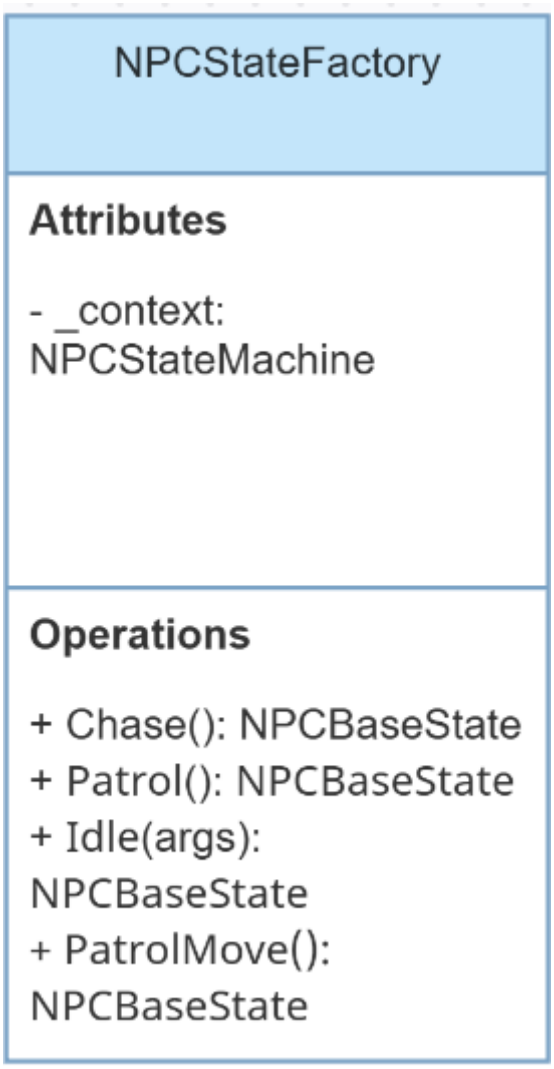

Рисунок 3.2 – UML діаграма NPCStateFactory

3.2.2 Абстрактний базовий стан

Абстрактний клас NPCBaseState визначає набір методів, які повинен реалізувати кожен конкретний стан, незалежно від того, є він кореневим чи підстаном. Це забезпечує уніфікованість роботи всієї HFSM.

− EnterState() – входження в новий стан;

− UpdateState() – містить логіку оновлення всередині стану;

− ExitState() – потрібна для очищення даних перед виходом зі стану;

− CheckSwitchStates() – перевіряє чи є необхідність переходу в новий стан;

− InitializeSubStates () – ініціалізація підстанів в залежності від умов;

− UpdateStates() – оновлення логіки стану та підстану;

− SwitchState() – перехід в новий стан;

− SetSuperState() – встановлення суперстану для цього стану;

− SetSubState() – встановлення підстану для цього стану.

Повний код розглянутих функцій наведено у лістингу 3.3. UML-діаграму класу показано на рисунку 3.3.

Лістинг 3.3 Реалізація NPCBaseState:

```
public abstract class NPCBaseState
{
    protected bool _isRootState = false;
    protected NPCStateMachine _ctx;
    protected NPCStateFactory _factory;
    protected NPCBaseState _currentSuperState;
    protected NPCBaseState _currentSubState;

    protected NPCBaseState(NPCStateMachine currentContext,
NPCStateFactory npcStateFactory)
    {
        _ctx = currentContext;
        _factory = npcStateFactory;

    }

    public abstract void EnterState();
    public abstract void UpdateState();
    public abstract void ExitState();
    public abstract void CheckSwitchStates();
    public abstract void InitializeSubState();
    public void UpdateStates()
    {
        UpdateState();
        if (_currentSubState != null)
        {
            _currentSubState.UpdateStates();
        }
    }
    protected void SwitchState(NPCBaseState newState)
    {
```

```
        ExitState();
        newState.EnterState();

        if (_isRootState)
        {
            _ctx.CurrentState = newState;
        }
        else if (_currentSuperState != null)
        {
            _currentSuperState.SetSubState(newState);
        }
    }
    protected void SetSuperState(NPCBaseState newSuperState)
    {
        _currentSuperState = newSuperState;
    }
    protected void SetSubState(NPCBaseState newSubState)
    {
        _currentSubState = newSubState;
        newSubState.SetSuperState(this);
    }
}
```

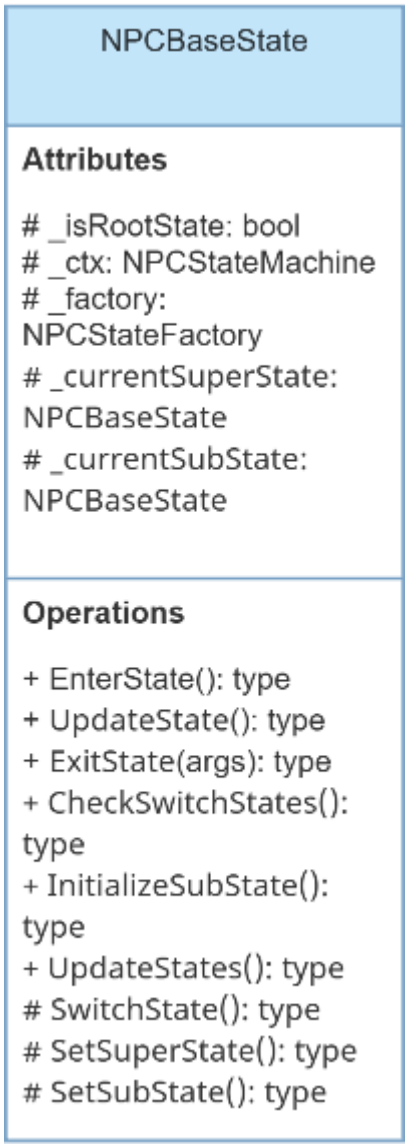

Рисунок 3.3 – UML діаграма NPCBaseState

3.2.3 Структура реалізації суперстанів

Суперстан – це стан, який містить у собі інші стани, відомі як підстани. Він діє як контейнер або складений стан (Composite State). Суперстан об’єднує групу пов’язаних станів та переходів. Це дозволяє абстрагувати загальну поведінку. Всі підстани всередині суперстану успадковують його поведінку та переходи. Якщо підстан не обробляє певну подію, вона автоматично «піднімається» до його суперстану для обробки. Це часто називають поведінковим успадкуванням. Перехід, визначений на рівні суперстану, означає, що будь-який підстан, що знаходиться всередині цього суперстану,

може перейти до іншого стану (навіть за межами суперстану) за цією умовою. Це значно зменшує кількість переходів, які потрібно визначати для кожного підстану окремо.

В реалізованому прикладі суперстан визначається за допомогою зміної _isRootState значення котрої ініціалізується під час створення стану через фабрику. Метод InitializeSubState відповідає за встановлення підстанів для цього суперстану.

− EnterState() – входження в новий стан;

− UpdateState() – містить логіку оновлення всередині стану;

− ExitState() – потрібна для очищення даних перед виходом зі стану;

− CheckSwitchStates() – перевіряє чи є необхідність переходу в новий стан.

UML-діаграми класів показано на рисунках 3.4-3.5.

Лістинг 3.4 – Реалізація суперстану NPCPatrolState:

```
public class NPCPatrolState : NPCBaseState
{
    public NPCPatrolState(NPCStateMachine currentContext,
NPCStateFactory npcStateFactory) : base(currentContext, npcStateFactory)
    {
        _isRootState = true;
        InitializeSubState();
    }

    public override void CheckSwitchStates()
    {
        if (_ctx.CanSeeTarget)
        {
            SwitchState(_factory.Chase());
        }
    }

    public override void EnterState()
    {
        if (_isRootState)
        {
            _ctx.UpdateSuperStateText("Patrol");
```

```
        }
        else
        {
            _ctx.UpdateSubStateText("Patrol");
        }
    }

    public override void ExitState()
    {

    }

    public override void InitializeSubState()
    {
        if (_ctx.IsIdle)
        {
            SetSubState(_factory.Idle());
        }
        else
        {
            SetSubState(_factory.PatrolMove());
        }
    }

    public override void UpdateState()
    {
        CheckSwitchStates();
    }
}
```

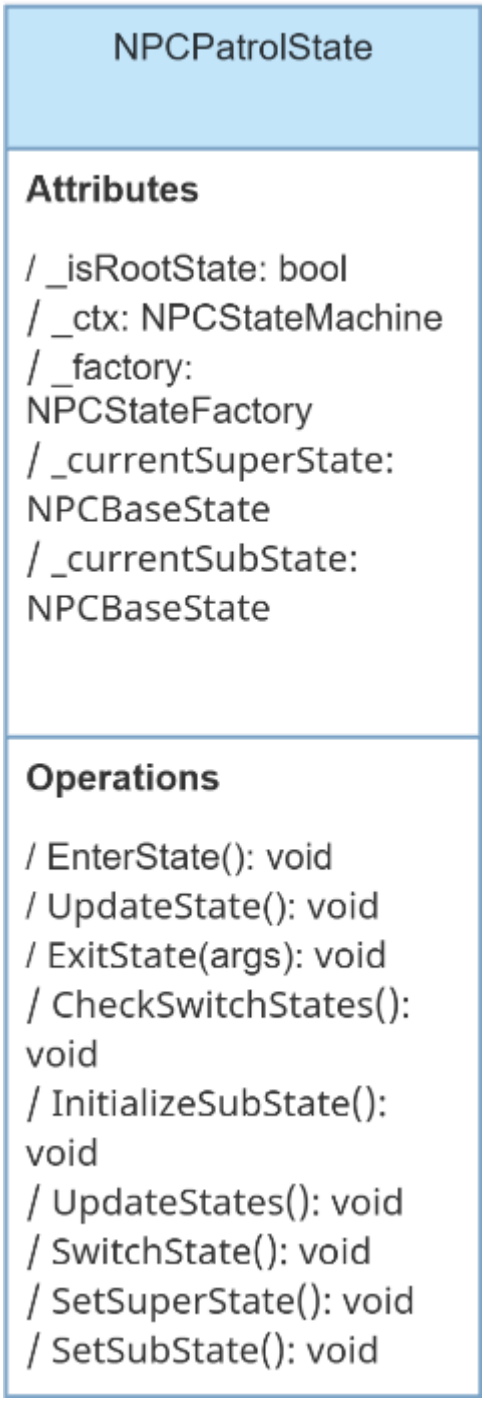

Рисунок 3.4 – UML-діаграма NPCPatrolState

Лістинг 3.5 – Реалізація суперстану NPCChaseState:

```
using UnityEngine;

public class NPCChaseState : NPCBaseState
{
    public NPCChaseState(NPCStateMachine currentContext, NPCStateFactory
npcStateFactory) : base(currentContext, npcStateFactory)
    {
        _isRootState = true;
        InitializeSubState();
    }

    public override void CheckSwitchStates()
    {
        float distance = Vector3.Distance(_ctx.transform.position,
_ctx.chaseTarget.transform.position);
        if (distance > _ctx.StopChaseDistance * 1.1f)
        {
            SwitchState(_factory.Patrol());
        }
    }

    public override void EnterState()
    {
        if (_isRootState)
        {
            _ctx.UpdateSuperStateText("Chase");
        }
        else
        {
            _ctx.UpdateSubStateText("Chase");
        }
    }

    public override void ExitState()
    {

    }

    public override void InitializeSubState()
    {

    }
```

```
    public override void UpdateState()
    {
        CheckSwitchStates();
        HandleChase();
    }

    private void HandleChase()
    {
        _ctx.Agent.SetDestination(_ctx.chaseTarget.position);
    }
}
```

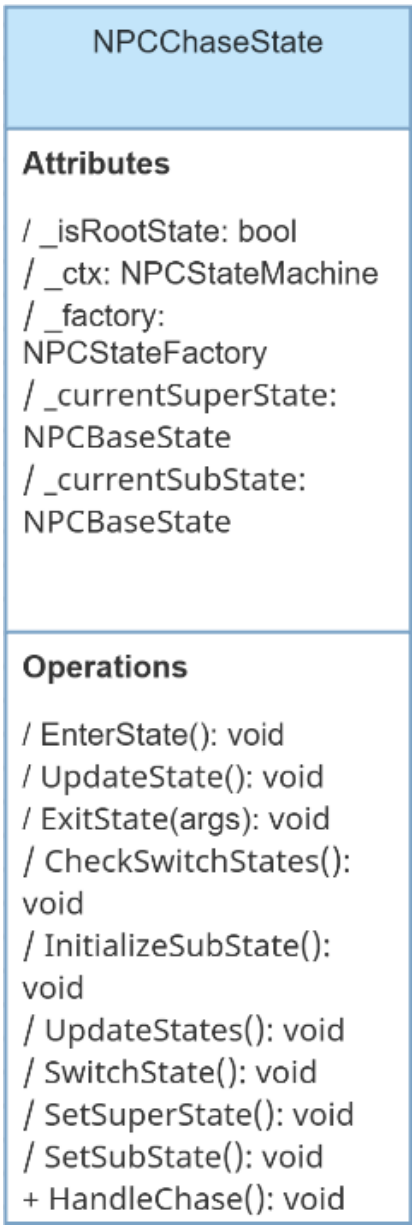

Рисунок 3.5 – UML-діаграма NPCChaseState

3.2.4 Структура реалізації підстанів

Підстан – це звичайний стан, який знаходиться всередині іншого стану (суперстану). Підстан являє собою деталізований, специфічний стан поведінки, що знаходиться в контексті свого суперстану. Підстан також може бути суперстаном, створюючи багатошарову ієрархію. Коли HFSM знаходиться у певному підстані (листовому стані), він також (неявно) вважається таким, що знаходиться в усіх його батьківських суперстанах. Тобто, в будь-який момент часу активною є вся «гілка» ієрархії від кореня до поточного листового стану. Підстани визначають свою власну поведінку (дії, переходи) та/або перевизначають поведінку, успадковану від суперстану.

В реалізованому прикладі кожний підстан має тільки той код котрий відповідає за обробку саме цього стану, що дозволяє легко структурувати код та уникнути проблем із зв’язаним кодом.

UML-діаграми класів показано на рисунках 3.6-3.7. На рисунку 3.8 показано загальну діаграму реалізованих класів.

Лістинг 3.6 – Реалізація підстану NPCIdleState:

```
using System.Collections;
using UnityEngine;

public class NPCIdleState : NPCBaseState
{
    public NPCIdleState(NPCStateMachine currentContext, NPCStateFactory
npcStateFactory) : base(currentContext, npcStateFactory)
    {

    }

    public override void CheckSwitchStates()
    {
        if (!_ctx.IsIdle)
        {
            SwitchState(_factory.PatrolMove());
        }
```

```
    }

    public override void EnterState()
    {
        if (_isRootState)
        {
            _ctx.UpdateSuperStateText("Idle");
        }
        else
        {
            _ctx.UpdateSubStateText("Idle");
        }
        _ctx.StartCoroutine(IdleTimer());
    }

    public override void ExitState()
    {

    }

    public override void InitializeSubState()
    {
        throw new System.NotImplementedException();
    }

    public override void UpdateState()
    {
        CheckSwitchStates();
    }

    public IEnumerator IdleTimer()
    {
        yield return new WaitForSeconds(_ctx.IdleTime);
        _ctx.IsIdle = false;
    }
}
```

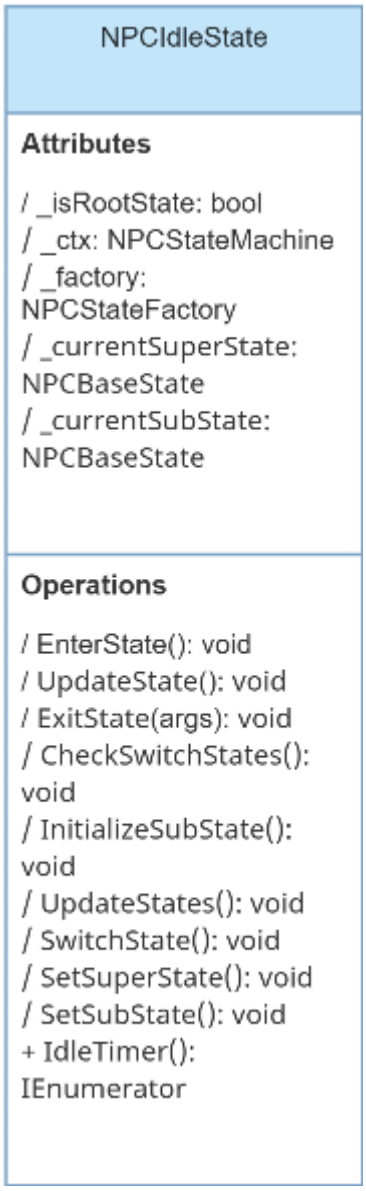

Рисунок 3.6 – UML-діаграма NPCIdleState

Лістинг 3.7 – Реалізація підстану NPCPatrolMoveState:

```
public class NPCPatrolMoveState : NPCBaseState
{
    public NPCPatrolMoveState(NPCStateMachine currentContext, NPCStateFactory npcStateFactory) : base(currentContext, npcStateFactory)
    {

    }

    public override void CheckSwitchStates()
    {
```

```
            if (_ctx.IsIdle)
            {
                SwitchState(_factory.Idle());
            }
        }

        public override void EnterState()
        {
            if (_isRootState)
            {
                _ctx.UpdateSuperStateText("PatrolMove");
            }
            else
            {
                _ctx.UpdateSubStateText("PatrolMove");
            }

_ctx.Agent.SetDestination(_ctx.patrolPoints[_ctx.CurrentPointIndex].position);
        }

        public override void ExitState()
        {

        }

        public override void InitializeSubState()
        {

        }

        public override void UpdateState()
        {
            CheckSwitchStates();
            HandlePatrol();
        }


        private void HandlePatrol()
        {
            if (_ctx.Agent.remainingDistance <= _ctx.Agent.stoppingDistance + 0.1f
&& !_ctx.IsIdle)
            {
                _ctx.CurrentPointIndex++;
                if (_ctx.CurrentPointIndex < _ctx.patrolPoints.Length)
                {
```

```
_ctx.Agent.SetDestination(_ctx.patrolPoints[_ctx.CurrentPointIndex].position);
                }
                else
                {
                    _ctx.CurrentPointIndex = 0;
                    _ctx.IsIdle = true;
                }
            }
        }
    }
```

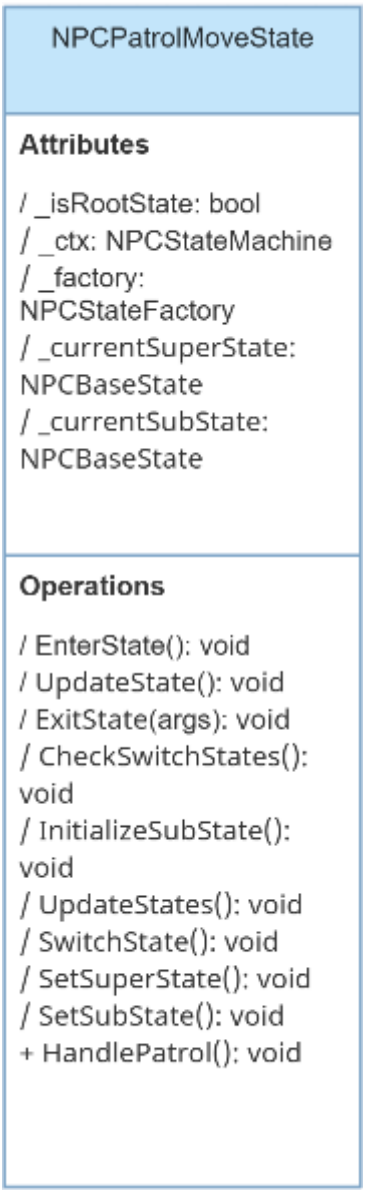

Рисунок 3.7 – UML-діаграма NPCPatrolMoveState

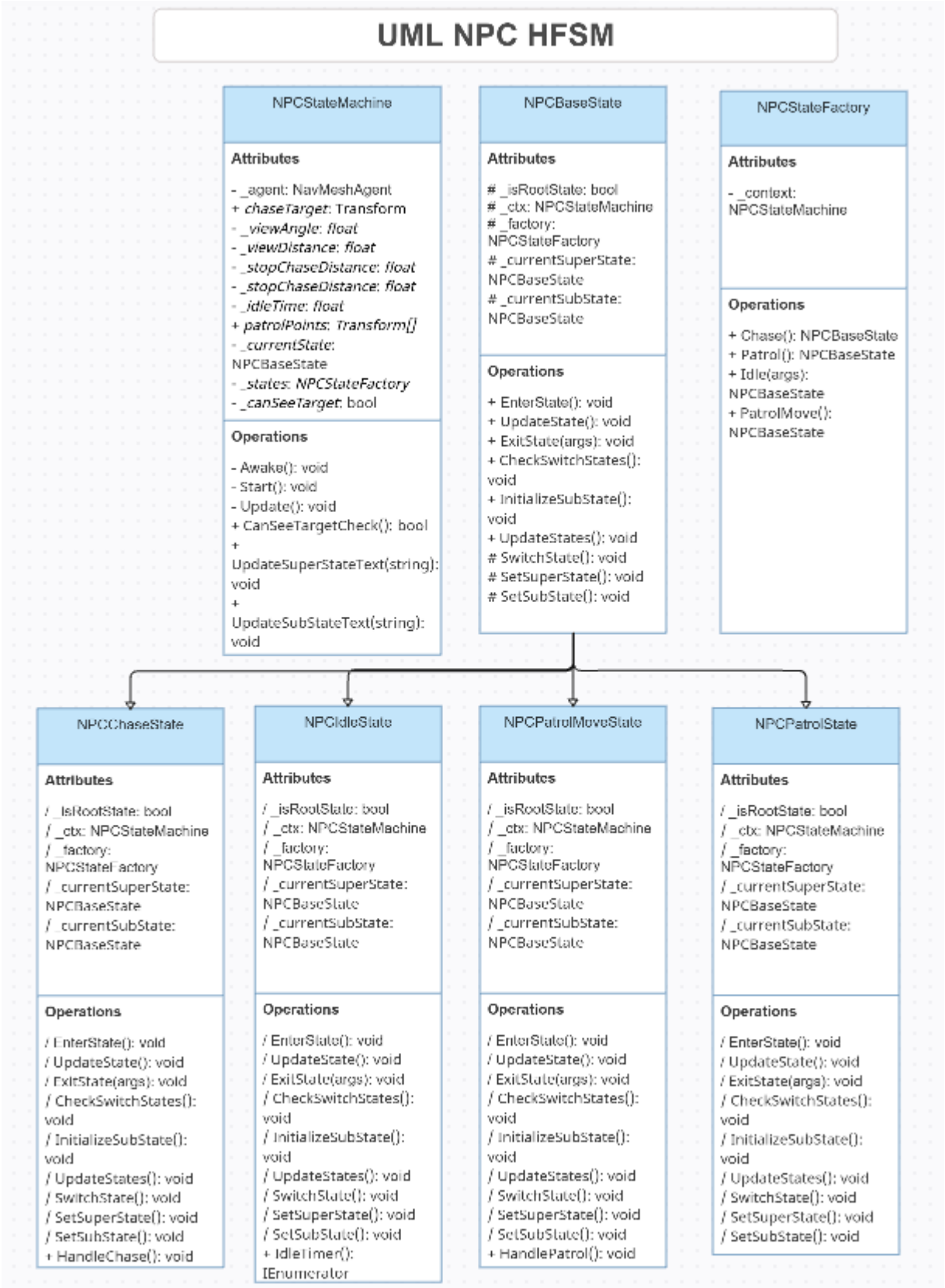

Рисунок 3.8 – UML-діаграма класів

3.3 Підготовка NPC та задач для тестування

У даному підрозділі описана підготовка середовища для тестування та налаштування NPC.

Для демонстрації результатів було створено сцену котра містить:

− площину для руху NPC та гравця (рис. 3.10);

− NPC із можливостю руху за допомогою AI Pathfinding;

− гравця із базовим контролером руху та камери;

− деякі перешкоди для руху NPC.

На рисунку 3.9 показано структуру файлів та папок у проєкті. Було створено префаби NPC та гравця, матеріали до них, та скрипти для реалізації HFSM.

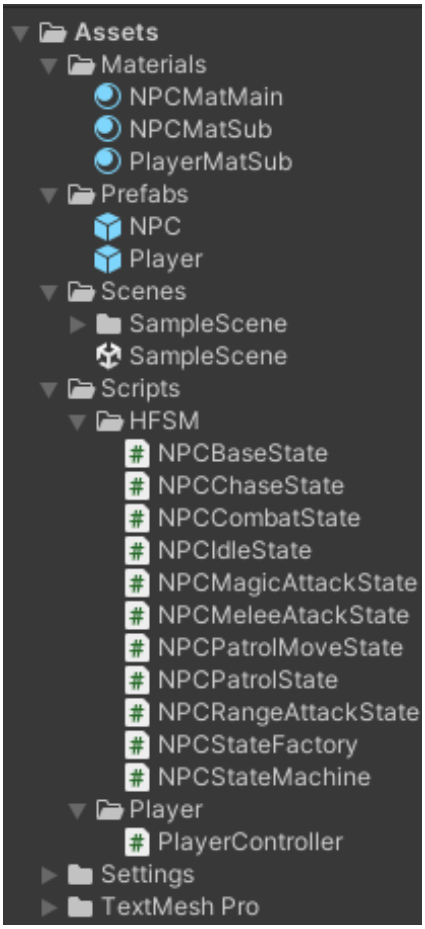

Рисунок 3.9 – Структура файлів програми

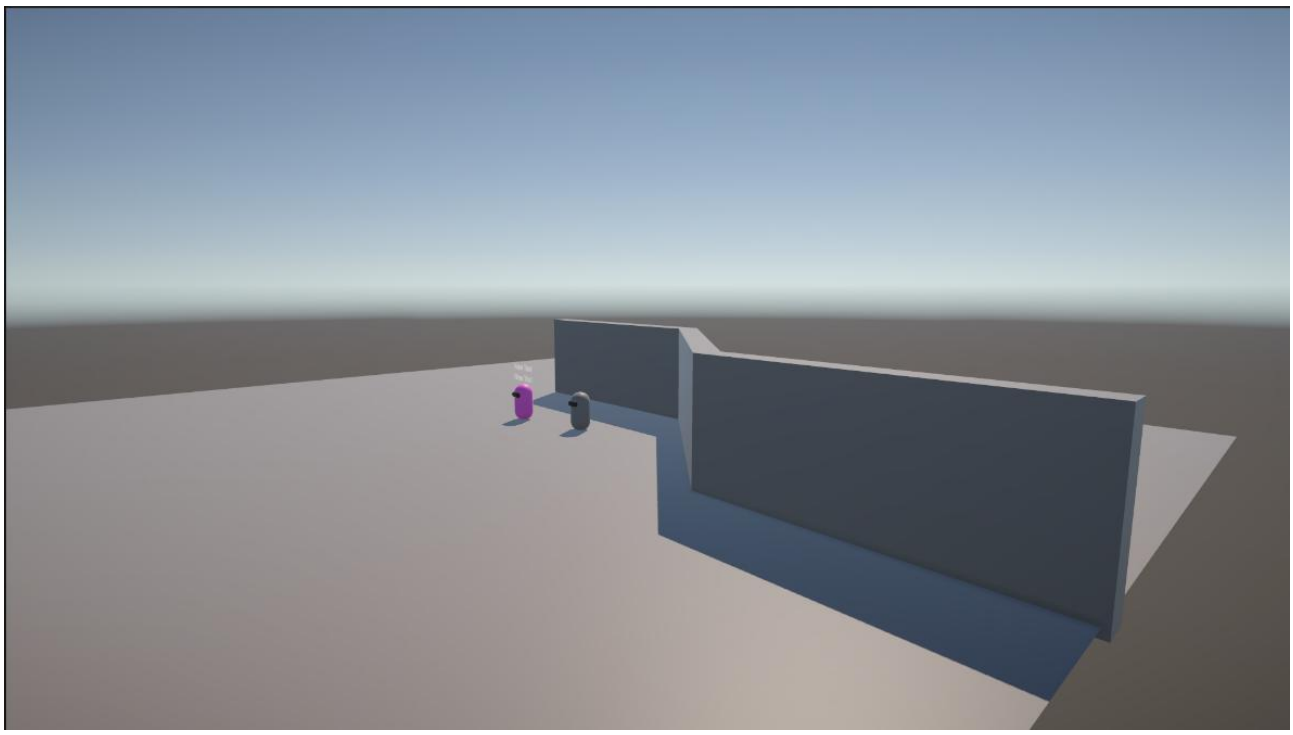

Рисунок 3.10 – Загальний вигляд сцени для тестування

Завданням дослідження є порівняти швидкість реакції HFSM, порівняно до інших методів. Визначити плавність переходів та оцінити читаність і підтримуваність архітектури, протестувати вкладені стани, оцінити, наскільки легко додавати нові підстани та root-стани.

3.4 Порівняння та оцінка якості моделей

Після завершення тестування було проведено оцінку якості для кожного з методів та виконано порівняння.

Оцінка проводилась за таким критеріями:

− читабельність;

− архітектурна чистота;

− масштабованість;

− модульність;

− швидкість реакції;

− плавність переходів;

− складність реалізації;

− придатність для складних сценаріїв;

− гнучкість.

Характеристики розраховувались шляхом порівняння різних методів по заданих критеріях. Результати порівняння показано у таблиці 3.1.

Таблиця 3.1 – Характеристики та параметри детекторів

Критерій оцінювання	**Utility AI**	**FSM**	**BT**	**HFSM**
Масштабованість	Висока	Середня	Висока	Дуже висока
Читабельність	Низька	Середня	Середня	Висока
Архітектурна чистота	Висока	Середня	Середня	Висока
Придатність для складних сценаріїв	Висока	Низька	Середня	Висока
Складність реалізації	Дуже висока	Низька	Висока	Середня
Гнучкість	Дуже висока	Низька	Висока	Дуже висока
Модульність	Висока	Низька	Низька	Дуже висока
Плавність переходів	Висока	Низька	Середня	Висока
Швидкість реакції	Дуже висока	Середня	Середня	Висока

Порівняння моделей керування поведінкою NPC показало, що класична FSM та BT ефективні лише для простих систем із невеликою кількістю станів. У міру зростання складності поведінки виникають типові проблем, такі як збільшення кількості переходів та дублювання логіки та труднощі з підтримкою. FSM виявилася менш масштабованою і менш передбачуваною у складних сценаріях.

У свою чергу, HFSM продемонструвала суттєві переваги завдяки ієрархічній структурі. Логіка поведінки стала більш модульною, організованою та гнучкою, що дозволило зменшити дублювання коду та підвищити

стабільність реакції NPC на події. HFSM легше розширюється, краще піддається тестуванню та забезпечує більш чітку структурованість переходів між станами.

В свою чергу Utility AI також показала високі результати, але на відміну від інший моделей, ця модель має дуже високий рівень складності реалізації, підтримки та розширення та використовується у моделювання адаптивної поведінки NPC котра базується на ML.

Результати порівняння свідчать, що HFSM є більш ефективною моделлю для створення середніх і складних поведінкових систем NPC, забезпечуючи баланс між продуктивністю, передбачуваністю та гнучкістю. Для простих NPC різниця мінімальна, проте зі збільшенням складності HFSM демонструє явну перевагу над FSM.

3.5 Перспективи подальших досліджень

Розроблений програмний застосунок має значний потенціал для подальшого розвитку методу керування NPC, який відкриває можливість вдосконалення та перспективи подальшого дослідження даного питання.

Перспективи подальших досліджень у напрямі розроблення ієрархічних машин станів для керування поведінкою NPC є достатньо широкими та відкривають можливості для вдосконалення як самої HFSM-моделі, так і її інтеграції з іншими компонентами ігрового середовища. Подальший розвиток може бути спрямований на поглиблену оптимізацію обробки подій, удосконалення системи переходів між станами та розширення механізмів адаптивної поведінки NPC. Особливий інтерес становить поєднання HFSM із методами машинного навчання або нейронними мережами, що могло б забезпечити NPC можливістю самонавчання та зміни поведінкових патернів залежно від дій гравця чи стану ігрового світу.

Важливим напрямом розвитку є створення універсальних інструментів для автоматизації побудови HFSM, включаючи візуальні редактори, системи

валідації переходів та генерацію коду на основі поведінкових діаграм. Це дозволить значно прискорити розробку та знизити ризик логічних помилок у складних ієрархіях станів. Окремої уваги потребує дослідження методів інтеграції HFSM із сучасними системами навігації, анімації та мережевими складовими, що дасть змогу підвищити реалістичність NPC у багатокористувацьких середовищах.

Додатковим напрямом є розроблення масштабованих архітектур для керування великою кількістю NPC із різними ролями та моделями поведінки, що дозволило б застосовувати HFSM у проєктах із високою щільністю населення ігрового світу. Перспективним також є дослідження підходів до автоматичного профілювання та оптимізації продуктивності HFSM задля ефективної роботи на платформах із різними апаратними обмеженнями.

Також є перспектива інтеграції елементів ML у цю систему, щоб зробити перехід між станами більш адаптивним під навколишнє середовище та різноманітні події.

Подальші дослідження у сфері HFSM можуть охоплювати як поглиблення теоретичних аспектів, так і практичне вдосконалення інструментів, що використовуються при створенні систем поведінки NPC. Це відкриває можливості для формування більш інтелектуальних, адаптивних та реалістичних персонажів, здатних реагувати на складні ігрові ситуації та забезпечувати високий рівень взаємодії з гравцем.

ВИСНОВКИ

У рамках кваліфікаційної роботи проведено дослідження методів реалізації ієрархічних машин станів для керування поведінкою NPC та розроблено програмну модель HFSM у середовищі Unity. У ході виконання роботи вирішено низку взаємопов’язаних завдань, що забезпечили досягнення поставленої мети.

Проведено аналіз літературних джерел та сучасних підходів до побудови машин станів і систем керування поведінкою NPC, що дало можливість визначити основні моделі, принципи ієрархізації станів та особливості їхнього застосування в ігровій індустрії. Здійснено порівняння традиційних FSM, Behavior Trees та HFSM, що дало можливість обґрунтувати вибір HFSM як найбільш гнучкого та масштабованого рішення для складної багаторівневої поведінки.

Досліджено існуючі інструменти реалізації HFSM у популярних ігрових рушіях, включаючи Unity, Unreal Engine та сторонні бібліотеки. На основі аналізу виявлено їхні сильні та слабкі сторони, обмеження у гнучкості, складність масштабування та прив’язку до анімаційних систем, це дозволило доцільність створення власної реалізації HFSM у рамках даної роботи.

Розроблено архітектуру ієрархічної машини станів, яка включає механізми суперстанів, підстанів, внутрішніх переходів, обробки подій та активації активного шляху станів. Побудовано покроковий алгоритм роботи HFSM та здійснено його візуалізацію у вигляді блок-схеми, що дозволило чітке розуміння взаємодії між станами та можливість подальшої оптимізації.

Створено програмну реалізацію HFSM у середовищі Unity з використанням мови C#. Реалізована система забезпечує гнучке додавання нових станів, повторне використання поведінкових компонентів, підтримку складної ієрархії та коректне оброблення переходів різних рівнів. Проведено тестування системи на прототипі NPC, що дозволило довести працездатність архітектури та її здатність масштабуватися під різні типи ігрових агентів.

Наукова новизна роботи полягає у розробленні модульної архітектури HFSM, адаптованої до інтеграції з компонентами Unity, зокрема анімацією, фізикою та навігацією, а також у практичному підході до побудови ієрархічної системи керування для NPC із можливістю розширення та повторного використання. Запропонована модель дозволяє створювати більш реалістичну, прогнозовану та керовану поведінку ігрових персонажів.

Практична значущість виконаного дослідження полягає у можливості застосування розробленої HFSM-архітектури для широкого спектра ігрових проєктів різної складності. Отримані результати можуть бути використані для створення адаптивної поведінки NPC, прототипування ігрової логіки, навчання молодих розробників та подальшого вдосконалення систем штучного інтелекту в інтерактивних застосунках.

Результати дослідження апробовано у вигляді 2 тез доповідей під час XI Міжнародної науково-практичної конференції «World science: problems, issues and prospects for development» [38] та IX Міжнародної студентської наукової конференції «Глобалізація наукових знань: міжнародна співпраця та інтеграція галузей наук» [39].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Plass, J. L., Homer, B. D., MacNamara, A., Ober, T., Rose, M. C., Pawar, S., Hovey, C. M., & Olsen, A. (2020). Emotional design for digital games for learning: The effect of expression, color, shape, and dimensionality on the affective quality of game characters. Learning and Instruction, 70, 101194.

2. Hazra, T., & Anjaria, K. (2022). Applications of game theory in deep learning : a survey (Issue 60). Multimedia Tools and Applications.

3. Riyan, T. S., Pardede, A. M. H., & Manik, F. Y. (2023). Implementation of Finite State Machine Models on the Artificial Intelligence System of Characters in The Game " MMORPG " using RPG Maker. 3(1), 2–6.

4. Mustafa E. Hierarchical state macine (HFSM) realization with polymorphism and inheritance. Embedded World Conference, Germany.

5. Zhang, B., Member, S., Du, X., Zhao, J., & Zhou, J. (2020). Impedance Modeling and Stability Analysis of a Three-phase Three-level NPC Inverter Connected to the Grid. 6(2), 270–278.

6. Yoones A Sekhavat. «Behavior trees for computer games». In: International Journal on Artificial Intelligence Tools 26.02 (2017), p. 1730001.

7. Dmitrii Iarovoi, Richard Hebblewhite, and Phoey Lee Teh. «AI's Influence on Non-Player Character Dialogue and Gameplay Experience». In: Science and Information Conference. Springer. 2024, pp. 76– 92.

8. Da Silva, G. A. (2021). Development of Non-Player Character with Believable Behavior. SBGames.

9. Marek Kopel and Tomasz Hajas. «Implementing AI for non-player characters in 3D video games». In: Intelligent Information and Database Systems: 10th Asian Conference, ACIIDS 2018, Dong Hoi City, Vietnam, March 19-21, 2018, Proceedings, Part I 10. Springer. 2018, pp. 610–619.

10. Game Developer. Behavior trees for AI: How they work. Accessed: September 24, 2024. 2024. URL: https: / /www.gamedeveloper.com/programming/behavior-treesfor-ai-how-they-work.

11. Miguel Nicolau et al. «Evolutionary behavior tree approaches for navigating platform games». In: IEEE Transactions on Computational Intelligence and AI in Games 9.3 (2016), pp. 227–238.

12. Gertjan M. Chaslot et al. «Monte-Carlo Tree Search: A New Framework for Game AI». In: Proceedings of the National Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE). AAAI Press, 2008, pp. 216–221. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/18700.

13. H. L. Cheng, R. K. G. V. Q. Teixeira, and E. M. F. P. S. M. V. A. Oliveira. «Monte Carlo Tree Search in Imperfect Information Games». In: arXiv preprint arXiv:2103.04931 (2021). URL: https://ar5iv.org/html/2103.04931.

14. Mela Karim Barznjiمنمهلاكهريمبهرزەنجـــــــە, Hemn. (2019). Artificial Intelligence and Game Development.

15. Syahputra, Mohammad &Arippa, A &Rahmat, Romi&Andayani, Ulfi. (2019).

16. Syahputra, Mohammad &Arippa, Historical Theme Game Using Finite State Machine for Actor Behaviour, Journal of Physics: Conference Series.

17. Adegun, Adekanmi&Ogundokun, Roseline &Ogbonyomi, Samuel & Sadiku, Peter. (2020). Design and Implementation of an Intelligent Gaming Agent Using A* Algorithm and Finite State Machines. International Journal of Engineering Research and Technology. 13. 191.

18. Aiolli F., Palazzi C.E. (2008) Enhancing Artificial Intelligence in Games by Learning the Opponent’s Playing Style. In: Ciancarini P., Nakatsu R., Rauterberg M., Roccetti M. (eds) New Frontiers for Entertainment Computing. ECS 2008. IFIP International Federation for Information Processing, vol 279. Springer, Boston,

19. W. Hu, Q. Zhang and Y. Mao, "Component-based hierarchical state machine – A reusable and flexible game AI technology," 2011 6th IEEE Joint International Information Technology and Artificial Intelligence

20. Solihin, Ade, Eka Wahyu Hidayat and Aldy Putra Aldya. «Application of the Finite State Machine Algorithm on 2D Platformer Rabbit Games vs Zombies.»

21. A F Pukeng et al 2019 J. Phys.: Conf. Ser. 1341 042006

22. K Fathoni et al 2020 J. Phys.: Conf. Ser. 1577 012018

23. Reza Andrea, SeftyWijayanti, Nursobah, " Finite State Machine Model in Jungle Adventure Game an Introduction to Survival Skills", International Journal of Information Engineering and Electronic Business(IJIEEB), Vol.13, No.4, pp. 55-61, 2021.

24. Rahadian, M. F., Suyatno, A., & Maharani, S. (2016). Penerapan Metode Finite State Machine Pada Game «The Relationship.» Informatika Mulawarman : Jurnal Ilmiah Ilmu Komputer, 11(1), 14.

25. Millington I. Artificial Intelligence for Games. San Francisco: Morgan Kaufman Publisher. 2006.

26. Czerwinski R, Kania D. Synthesis Method of High Speed Finite State Machines. Bulletin of the Polish Academy of Sciences, Technical Sciences, 58(5), 635-644, 2010.

27. Kielar PM, Handel O, Biedermann DH, Borrmann A. Concurent Hierarchical Finite State Machines for Modeling Pedestrian Behavioral Tendencies. Transportation Research Procedia, 2, 576-584, 2014

28. Chow S-H, Ho Y-C, Hwang T. Low Power Realization of Finite State Machines-A Decomposition Approach. ACM Transaction on Design Automation of Electronic Systems, 1 (3), 315-340, July 1996

29. Arif YM, Hariadi M, Nugroho SMS. Integrasi Hierarchy Finite State Machine dan Logika Fuzzy untuk Desain Strategi NPC Game. Matics 4(3), Maret 2011.

30. Tremblay J, Dragert C, Verbrugge C. Target Selection for AI Companions in FPS Games. Int. Con. on FDG, 2014.

31. Alur R, Kannan S, Yannakakis M Communicating Hierarchical State Machines. Int. Colloquim of Automata, anguages and Programming, Springer pp. 169-178, 1999.

32. Risler M. Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines. Doctoral Dissertation, Technischen Universitat Darmstadt, 2009.

33. Kurt A, Ozguner U. Hierarchical Finite State Machines for Autonomous Mobile Systems. Control Engineering Practice, 21 (2), pp. 184-194. February 2013

34. Chow TS. Testing Software Design Modelled by Finite-State Machines. IEEE Transactions on Software Engineering, 4(3), 178-187. 1978.

35. Salauyou V and Grzes T. FSM state assignment methods for low-power design. Int. Con. Computer Information Systems and Industrial Management Appliations, pp. 345-350, 2007.

36. Lee B, Lee EA. Interaction of Finite State Machines and Concurrency Models. Circuits, Systems and Computers 2, 1998.

37. Girault A, Lee B, Lee EA. Hierarchical Finite State Machines with Multiple Concurrency Models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 18(6), 742-760, 1999.

38. Мірошниченко, М. С. (2025). Моделювання поведінки неігрових персонажів засобами дерева поведінки. Глобалізація наукових знань: міжнародна співпраця та інтеграція галузей наук: матеріали IX Міжнародної студентської наукової конференції (с. 300–302). Вінниця: ТОВ «УКРЛОГОС Груп».

39. Мірошниченко, М. С. (2025). Моделювання поведінки неігрових персонажів засобами H-FSM. World Science: Problems, Issues and Prospects for Development: Proceedings of the XI International Scientific and Practical Conference (pp. 63–65). Sofia, Bulgaria: International Science Group. DOI: 10.46299/ISG.2025.2.11.