ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Система генерації поведінки ігрового ворога для First-person shooter на основі скінчених автоматів»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Микола ЛЕВЧУК
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-62
Микола ЛЕВЧУК

Керівник: Тимур ДОВЖЕНКО
к.т.н., доцент

Рецензент: _____
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Левчуку Миколі Петровичу

1. Тема кваліфікаційної роботи: «Система генерації поведінки ігрового ворога для First-person shooter на основі скінчених автоматів»

керівник кваліфікаційної роботи Тимур ДОВЖЕНКО, к.т.н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, теорія штучного інтелекту в ігровій розробці, методологія скінчених автоматів, принципи розробки ігрового ші, вимоги до інтелектуальної поведінки ворогів у відеоіграх.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження методів штучного інтелекту в ігровій індустрії.
2. Аналіз архітектур поведінки ігрових ворогів.
3. Розробка алгоритму керування станами ворога.

5. Перелік ілюстративного матеріалу: *презентація*

1. Огляд сучасних підходів до створення ігрового ШІ.
2. Математична модель скінченого автомата.
3. Схема архітектури системи генерації поведінки ворога.
4. Діаграма переходів між станами ворога.
5. Демонстрація роботи системи на прикладах.
6. Результати тестування та порівняння з іншими підходами.

6. Дата видачі завдання «16» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
s1	Аналіз наявної науково-технічної літератури	16.10-23.10.2024	
2	Вивчення матеріалів для аналізу розвитку моделей поведінки ворогів у іграх	23.10-28.10.2024	
3	Дослідження методів проєктування скінченних автоматів	28.10 – 02.11.2024	
4	Аналіз особливостей впливу архітектури скінченних автоматів на поведінку ігрових ворогів	02.11 – 11.11.2024	
5	Розробка та тестування системи генерації поведінки ворога на основі скінченних автоматів	11.11 – 18.11.2024	
6	Тестування системи у FPS-прототипі	19.11.2024	
7	Оформлення роботи: вступ, висновки, реферат	20.11 – 24.11.2024	
8	Розробка демонстраційних матеріалів	24.11 – 26.11.2024	
9	Попередній захист роботи	27.11.2024	

Здобувач вищої освіти _____ Микола ЛЕВЧУК
(підпис)

Керівник
кваліфікаційної роботи _____ Тимур ДОВЖЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 75 стор., 2 табл., 7 рис., 31 джерело.

Мета роботи – підвищити реалістичність та ефективність взаємодії ігрових ворогів з гравцем у First-person shooter за рахунок використання системи генерації поведінки, заснованої на скінчених автоматах.

Об'єкт дослідження – процес генерації поведінки ігрових ворогів у жанрі First-person shooter.

Предмет дослідження – методи та технології генерації поведінки ігрових ворогів у жанрі First-person shooter на основі скінчених автоматів.

У роботі використано різноманітні методи, такі як скінченні автомати, алгоритми моделювання поведінки, апарат об'єктно-орієнтованого програмування та технології ігрового розроблення в середовищі Unity. Головний акцент зроблено на використанні скінченних автоматів для створення ефективної та адаптивної системи поведінки ігрового ворога в First-person shooter.

Проведено аналіз сучасних методів моделювання поведінки ворогів у відеоіграх. Вивчено ключові підходи до генерації поведінки NPC, включаючи штучні нейронні мережі, дерева прийняття рішень та інші алгоритми. Детально проаналізовано особливості використання скінченних автоматів для задач генерації поведінки в First-person shooter-іграх.

Розроблено та оптимізовано алгоритм генерації поведінки ворога, що базується на скінченних автоматах. Проєктування включає визначення основних станів, умов переходу між ними (наприклад, патрулювання, переслідування, атака) та інтеграцію цієї системи в ігрове середовище Unity.

Реалізовано алгоритм, який дозволяє створювати ігрових ворогів з адаптивною поведінкою, що відповідає сценаріям в іграх. Застосовано об'єктно-орієнтоване програмування на мові C# для реалізації логіки поведінки та взаємодії ігрового ворога із гравцем і навколишнім середовищем.

Проведено експерименти для валідації розробленої системи та оцінки її ефективності. Тестування виконано на реальних сценаріях First-person shooter гри, що включають різні рівні складності та поведінкові шаблони ворогів. Оцінено продуктивність системи, точність реакції ігрових ворогів та відповідність їх поведінки очікуванням гравців.

Результати дослідження підтверджують ефективність використання скінченних автоматів для створення системи генерації поведінки ворогів. Розроблена система відповідає сучасним вимогам до реалізму та адаптивності ігрових ворогів у відеоіграх.

КЛЮЧОВІ СЛОВА: ГЕНЕРАЦІЯ ПОВЕДІНКИ, ІГРОВІ ВОРОГИ, СКІНЧЕННІ АВТОМАТИ, ПРОЄКТУВАННЯ ШТУЧНОГО ІНТЕЛЕКТУ, UNITY, C#, ІГРОВА ЛОГІКА, АНАЛІЗ МЕТОДІВ ГЕНЕРАЦІЇ ПОВЕДІНКИ.

ABSTRACT

Text part of the master's qualification work: 75 pages, 7 pictures, 2 table, 33 sources.

The purpose of the work is to increase the realism and efficiency of interaction between game enemies and the player in a First-person shooter by using a behavior generation system based on finite state machines.

Object of research is the process of generating the behavior of game enemies in the First-person shooter genre.

Subject of research are methods and technologies for generating the behavior of game enemies in the First-person shooter genre based on finite automata

The work uses a variety of methods, such as finite state machines, behavior modeling algorithms, object-oriented programming, and game development technologies in the Unity environment. The main emphasis is placed on the use of finite automata to create an effective and adaptive system of enemy behavior in a first-person shooter.

The analysis of modern methods of modeling enemy behavior in video games is carried out. The key approaches to generating NPC behavior, including artificial neural networks, decision trees, and other algorithms, are studied. The features of using finite automata for behavior generation tasks in first-person shooter games are analyzed in detail.

Developed and optimized an algorithm for generating enemy behavior based on finite state machines. The design includes the definition of basic states, conditions of transition between them (e.g., patrolling, pursuit, attack) and integration of this system into the Unity game environment.

An algorithm has been implemented that allows creating game enemies with adaptive behavior that corresponds to scenarios in games. Object-oriented programming in C# was used to implement the logic of behavior and interaction of the game enemy with the player and the environment.

Experiments were conducted to validate the developed system and evaluate its effectiveness. The testing was performed on real scenarios of a first-person shooter game, including different levels of complexity and behavioral patterns of enemies. The performance of the system, the accuracy of the reaction of game enemies, and the compliance of their behavior with the expectations of players were evaluated.

The results of the study confirm the effectiveness of using finite state machines to create a system for generating enemy behavior. The developed system meets the modern requirements for realism and adaptability of game enemies in video games.

KEYWORDS: BEHAVIOR GENERATION, GAME ENEMIES, FINITE AUTOMATA, ARTIFICIAL INTELLIGENCE DESIGN, UNITY, C#, GAME LOGIC, ANALYSIS OF BEHAVIOR GENERATION METHODS.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI – штучний інтелект (Artificial Intelligence).

FPS – шутер від першої особи (First-Person Shooter).

FSM – скінчений автомат (Finite-state machine).

NPC – неігровий персонаж (Non-Playable Character).

UI - користувацький інтерфейс (User Interface).

ВСТУП

Як відомо, створення ігор є складним та трудомістким процесом, що охоплює різноманітні сфери: від графічного дизайну та написання сценаріїв до програмування. Важливу роль у цій індустрії відіграє розробка штучного інтелекту. ШІ охоплює такі аспекти відеоігор, як керування анімацією, управління транспортними засобами, навігацію на місцевості, планування, а також тактичне й стратегічне мислення та навчання. Розробка нових методів і технологій ШІ є ключовим викликом у створенні будь яких ігор, в тому числі і шутерів від першої особи, оскільки ШІ представляє собою складну систему команд, яка базується на прийнятті рішень у певних ситуаціях.

Сфера штучного інтелекту значно ширша, ніж просто ігрові ШІ. Ще в 1950 році Алан Тьюрінг, піонер у цій галузі, описував інтелектуального агента як істоту, чия поведінка не відрізняється від людської [3]. Ігрові ШІ доповнюють це визначення, роблячи акцент не лише на інтелекті, а й на розвагах. Їхня мета – не просто здолати гравця, а й створити захопливий та динамічний ігровий процес, перетворившись на гідних й цікавих супротивників.

Для створення штучного інтелекту в іграх часто використовуються прості, але ефективні методи, такі як кінцеві автомати (FSM) та дерева прийняття рішень. Ці методи дозволяють неігровим персонажам (NPC) та іншим супротивникам демонструвати базову поведінку, реагуючи на дії гравця та навколишнє середовище. Відеогра складається з двох основних компонентів: контексту та геймплею. Геймплей включає правила, цілі та інші елементи, які визначають виклики та завдання, з якими стикаються гравці. Контекст же охоплює все те, що формує оточення, в якому розгортається гра, включаючи персонажів, сюжет та інші подробиці. Ця магістерська робота присвячена вивченню ігрового ШІ, який зосереджений на вирішенні ігрових задач, таких як перемога над суперником у бою або навігація в лабіринті. На противагу цьому, контекстний ШІ орієнтований на виконання завдань, пов'язаних з контекстом гри, наприклад, змусити персонажа

виконати певний набір дій під час проходження платформи, реагуючи на вибір гравця. Відеоігри одного жанру, як правило, мають схожі виклики, які ґрунтуються на спільних концепціях. Це створює можливість визначити загальні проблеми та розробити базові поведінкові моделі, які можна застосовувати до різних ігор в рамках жанру. Наприклад, у шутерах від першої особи "один на один" гравці стикаються з такими задачами, як вибір зброї, прогнозування дій суперника та навігація на карті. У кожну мить гравцеві потрібно оцінювати ситуацію, обирати найефективнішу зброю, передбачати пересування противника та знаходити оптимальний маршрут. Ці задачі можна розбити на більш дрібні, такі як оцінка швидкості стрільби зброї, стан здоров'я противника та розташування аптечок. Ці концепції є спільними для багатьох шутерів від першої особи і дозволяють розробити ефективні поведінкові моделі, які можна адаптувати до різних ігор. Такі рішення вже існують для певних завдань, наприклад, навігації, і використовуються в багатьох відеоіграх. Більше того, люди-гравці часто без проблем переносять досвід з однієї відеоігри на іншу в межах того ж жанру. Гравець, який грав у шутери від першої особи, зазвичай буде краще грати в нову гру цього жанру, ніж той, хто не має досвіду, а іноді навіть краще за того, хто має досвід, але в іншій грі. Це свідчить про те, що вивчена поведінка може бути ефективно застосована в різних іграх, які мають подібні концепції, навіть без знання деталей конкретної гри. Звичайно, деталі ігрового світу можуть потребувати коригування базової концептуальної поведінки або навіть її повного перевизначення. Тому створення міжігрового ШІ може бути ґрунтоватися на виявленні та вирішенні концептуальних проблем, а не на деталях конкретних ігрових екземплярів. Від'єднання ШІ або його частини від розробки конкретної відеоігри дозволить уникнути обмежень, які зазвичай накладаються на розробників, і забезпечить безперервний та ретельний процес його проектування.

Об’єкт дослідження – процес генерації поведінки ігрових ворогів у жанрі First-person shooter.

Предмет дослідження – методи та технології генерації поведінки ігрових ворогів у жанрі First-person shooter на основі скінчених автоматів.

Мета роботи – підвищити реалістичність та ефективність взаємодії ігрових ворогів з гравцем у First-person shooter за рахунок використання системи генерації поведінки, заснованої на скінчених автоматах.

Для досягнення мети вирішувалися наступні завдання:

1. Проведено дослідження та аналіз існуючих наукових і технічних джерел, щоб отримати уявлення про сучасні підходи до моделювання поведінки ігрових ворогів, особливості використання скінченних автоматів та їх застосування у іграх;
2. Застосовано теоретичні знання з літературного огляду для створення прототипів системи генерації поведінки NPC. Виконано тестування різних моделей поведінки в ігрових сценаріях;
3. Розроблено архітектуру системи на основі скінченних автоматів, включаючи стани (наприклад, патрулювання, переслідування, атака) та умови їх переходів.
4. Використано кількісні та якісні метрики (наприклад, час реакції NPC, кількість коректних дій у заданих ситуаціях, загальний вплив на геймплей) для оцінки ефективності та реалістичності системи.
5. Проведено порівняльний аналіз продуктивності системи на основі скінченних автоматів із іншими підходами до генерації поведінки ігрових ворогів, такими як дерева прийняття рішень, алгоритми на основі штучного інтелекту.

Вирішення цих задач сприяє створенню ефективної системи генерації поведінки ігрових ворогів у жанрі First-person shooter, що забезпечує реалістичну та динамічну взаємодію з гравцем і підвищує рівень занурення у гру.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Огляд та аналіз категорій комп’ютерних ігор, в яких застосовується штучний інтелект

Розвиток комп'ютерних технологій та штучного інтелекту суттєво трансформує ігрову індустрію, надаючи розробникам унікальні можливості для створення більш реалістичних, динамічних та розумних ігрових світів. Категорії комп'ютерних ігор відрізняються не лише жанровими особливостями, але й ступенем складності впровадження штучного інтелекту в ігровий процес.

Абстракція є ключовим елементом концептуалізації відеоігор, що дозволяє розробникам створювати універсальні рішення для широкого кола ігрових проблем, не зациклюючись на деталях конкретної гри. Візьмемо для прикладу проблему сортування масиву. Її можна реалізувати різними способами, залежно від типу елементів у масиві. Однак, завдяки абстракції типу даних та функції порівняння, програмісту не потрібно писати окреме рішення для кожного типу масиву. Достатньо написати універсальне рішення, яке буде працювати з будь-яким типом даних. Абстракція стає потужним інструментом для оптимізації розробки програмного забезпечення, зводячи до мінімуму дублювання коду та роблячи його більш універсальним. Апаратне абстрагування є ще одним яскравим прикладом застосування абстракції в програмуванні. Цей підхід полягає у представленні фізичних компонентів комп'ютера як абстрактних пристроїв, що значно спрощує процес розробки програмного забезпечення. Різні фізичні компоненти, які виконують схожі функції, наприклад, пристрої зберігання даних, можуть бути об'єднані в один абстрактний тип - "пристрій зберігання даних". Це дає можливість розробникам програмного забезпечення створювати універсальні програми для роботи з даними, які не залежать від конкретного типу фізичного носія. Такий механізм широко використовується в багатьох операційних системах, таких як NetBSD 5 [8] та сімейство Windows NT [9]. Завдяки абстрактному рівню

апаратного забезпечення, розробники можуть вільно взаємодіяти з різними типами компонентів сховища, не заглиблюючись у їхні специфічні особливості. Це робить процес розробки програмного забезпечення більш гнучким, ефективним та універсальним.

Перевагами апаратного абстрактування є спрощення розробки, бо розробникам не потрібно буде писати окремий код для кожного типу фізичного компонента. Також це наділяє програми універсальністю, бо програми, написані з використанням апаратного абстрактування, можуть працювати з різними типами обладнання. Також треба зазначити, що зростає і продуктивність. Завдяки абстракції, розробники можуть зосередитися на функціональності програмного забезпечення, а не на специфіці апаратного забезпечення.

Приклади використання апаратного абстракції:

- Інтерфейси програмування додатків (API) для роботи з файлами та пристроями зберігання даних.
- Драйвери для різних типів апаратного забезпечення.
- Віртуальні машини, які емулюють апаратне забезпечення на програмному рівні.

Важливо зазначити, що апаратне абстрагування не завжди є ідеальним рішенням. У деяких випадках може знадобитися прямий доступ до фізичних компонентів комп'ютера для досягнення максимальної продуктивності або реалізації специфічних функцій.

Концепція створення універсального проміжного програмного забезпечення штучного інтелекту (ШІ) [11] для відеоігор вже є доволі старою. Вже у 2002 році Міжнародна асоціація розробників ігор (IGDA) заснувала Комітет зі стандартизації інтерфейсу штучного інтелекту (AIISC). Його основною метою було розробити стандартизований інтерфейс ШІ (AIinterface), який можна було б легко реалізувати та навіть передавати код ШІ на аутсорсинг [10].

Робота комітету AIISC була спрямована на створення єдиних підходів до використання штучного інтелекту в іграх. Зокрема, комітет розробив стандарти для забезпечення взаємодії ШІ з ігровим світом, управління персонажами та об’єктами,

навігації в віртуальному просторі, використання таких інструментів як кінцеві автомати та системи, засновані на правилах, а також для планування складних послідовностей дій.

Важливо зазначити, що група, яка працювала над системами, що базуються на правилах, згодом була розпущена.

Комітет AIISC прагнув не лише створити стандартний інтерфейс ШІ для відеоігор, але й розробити стандартизовані алгоритми ШІ [12]. Це було ґрунтовано на припущенні, що встановлення стандартів ШІ може стимулювати розробку спеціалізованого обладнання для ШІ, що може значно покращити його продуктивність.

В наслідку робота комітету AIISC мала значний вплив на розвиток ШІ у відеоіграх. Стандарти, розроблені цим комітетом, стали основою для багатьох сучасних технологій ШІ, які використовуються у відеоіграх.

Ідея проміжного програмного забезпечення ШІ для відеоігор вже давно обговорюється не лише в рамках роботи Комітету AIISC, але й у науковій літературі. Дослідники активно вивчають технічні проблеми та підходи до створення такого програмного забезпечення, що може значно спростити розробку ШІ для відеоігор.

Один з ключових моментів, який підкреслюється в наукових дослідженнях, полягає у необхідності знайти баланс між простотою та складністю ШІ-систем для відеоігор. З одного боку, розробники часто потребують простих рішень, таких як кінцеві автомати, для реалізації базової поведінки ШІ. З іншого боку, для більш складних завдань, як-от стратегічне планування або навчання на основі досвіду, можуть знадобитися більш досконалі когнітивні моделі.

Замість комплексних агентських рішень, які можуть бути занадто жорсткими та обмеженими, дослідники пропонують використовувати бібліотеки функціональних можливостей. Цей підхід забезпечує більшу гнучкість, дозволяючи розробникам збирати рішення ШІ з окремих модулів, що відповідають їхнім потребам.

Деякі дослідники також вважають, що для досягнення максимальної продуктивності ШІ у відеоіграх може знадобитися створення спеціалізованого обладнання. Це схоже на те, як поява основних графічних процесорів (GPU) революціонізувала комп'ютерну графіку, надавши їй нові можливості.

Розробка ефективного проміжного програмного забезпечення ШІ може мати значний вплив на індустрію відеоігор. Це може знизити витрати та час розробки ШІ, підвищити його якість у відеоіграх та стимулювати інновації. Завдяки своїй гнучкості, модульності та можливості стимулювати інновації, ШІПЗ може стати ключовим фактором у створенні більш захоплюючих та інтелектуальних відеоігор у майбутньому.

Відкритий стандарт інтерфейсу ШІ (OASIS) пропонується як спосіб спростити інтеграцію ШІ у відеоігри. Структура OASIS розроблена для підтримки представлення знань, міркувань та навчання. Вона складається з п'яти шарів, кожен з яких виконує певні функції і деякі шари відповідають за різні рівні абстракції, наприклад, рівень об'єктів або рівень домену , а інші шари надають різні послуги, такі як доступ до даних, переклад або арбітраж цілей.

OASIS пропонує модульну архітектуру для розробки ШІ в іграх. Нижній шар забезпечує прямий інтерфейс для взаємодії з ігровим двигуном, надаючи інструменти для отримання інформації про стан гри, керування персонажами та об'єктами, а також для отримання відгуків на дії ШІ. Верхній шар фокусується на більш абстрактних завданнях, таких як представлення знань про світ, планування послідовностей дій та прийняття рішень на основі аналізу отриманої інформації. Такий поділ відповідальності спрощує процес розробки ШІ, підвищує гнучкість системи та стимулює розробку нових і інноваційних рішень.OASIS може мати значний вплив на індустрію відеоігор, роблячи розробку ШІ доступнішою та стимулюючи створення більш складних та цікавих ігрових механік.

В останні роки спостерігається зростання популярності ШІ-проміжного програмного забезпечення (ШІПЗ) у відеоігрових двигунах. Такі платформи, як Unity [13], Unreal Engine [14], та Godot [15] , окрім набору інструментів для

створення реалістичних віртуальних світів, дедалі більше орієнтуються на розробку реалістичних агентів за допомогою ШІ.

Окрім ШІ, ще один підхід, який поділяє мету полегшення розробки відеоігор, - це використання ігрових патернів. Ці шаблони дизайну дозволяють розробникам документувати повторювані проблеми та рішення дизайну, роблячи їх доступними для використання у різних іграх. Це допомагає розробникам краще розуміти нюанси дизайну, які стосуються конкретного жанру, та приймати більш обґрунтовані рішення.

Робота, представлена в тексті, пропонує формалізований підхід до ігрових патернів, який допомагає розширити знання про дизайн ігор. Цей формалізм описує ігрові патерни за допомогою чотирьох ключових елементів, таких як назва **(ц**ей елемент чітко та лаконічно описує патерн), проблема (Цей елемент описує мету патерну, а також перешкоди, з якими можна зіткнутися при його застосуванні, та контекст, в якому він використовується), рішення **(**цей елемент описує абстрактні механізми та особливості, які використовуються для вирішення описаної проблеми), Наслідки (цей елемент описує вплив вибору дизайну на інші аспекти розробки, а також його переваги та недоліки).

Формалізований підхід до ігрових патернів може допомогти розробникам поліпшити комунікацію завдяки чіткому та лаконічному описуванні патернів можна посприяти кращому розумінню між розробниками, що може призвести до більш ефективної співпраці, повторно використовувати дизайнерські рішення можна використовувати повторно в різних проектах, що економить час та ресурси.

Бьорк [16] пропонує розрізняти два аспекти дизайну ігор: структурний каркас та ігрові патерни. Структурний каркас описує основні компоненти гри, тоді як ігрові патерни описують взаємодію гравців з цими компонентами під час гри.

Структурний каркас Бьорка поділяється на три категорії компонентів:
Обмежуючі компоненти: Ці компоненти описують дозволені дії в грі, такі як правила, режими гри, а також обмеження, які застосовуються до гравців.

Тимчасові компоненти: Ці компоненти пов'язані з часовим перебігом гри, включаючи дії, які виконують гравці, події, які відбуваються, та загальний темп гри.

Цільовими компонентами описують конкретні ігрові елементи, такі як гравці, персонажі, предмети та цілі, до яких вони прагнуть.

Ігрові патерни:

Ігрові патерни, за визначенням Бьорка, представляють собою повторювані схеми взаємодії гравців з елементами гри. Ці патерни дозволяють аналізувати та класифікувати різноманітні ігри, виявляючи спільні риси в їхньому геймплеї. Вони охоплюють широкий спектр аспектів, від способу дослідження віртуального світу (наприклад, "відкритий світ", де гравець має повну свободу дій) до механіки вирішення завдань (як у патерні "головоломка", що передбачає використання логіки та кмітливості). Розуміння ігрових патернів допомагає розробникам створювати більш передбачувані та цікаві ігрові досвіди, а гравцям – краще розуміти, чому певні ігри їм подобаються. Крім того, аналіз патернів дозволяє виявляти нові тенденції в розробці ігор та прогнозувати майбутнє індустрії.

Як гравці борються з ворогами? Наприклад, патерн "бос-файт" описує битву з потужним ворогом, яка вимагає від гравців спеціальних навичок та стратегії.

Переваги використання структурного каркасу та ігрових патернів:

- Покращення комунікації: Чіткий опис компонентів гри та ігрових патернів може допомогти розробникам краще розуміти один одного та чітко формулювати свої ідеї.
- Повторне використання дизайнерських рішень: Задокументовані патерни можна використовувати повторно в різних проектах, що економить час та ресурси.
- Створення більш послідовних вражень від гри: Використання спільних патернів може допомогти створити більш послідовний та передбачуваний досвід для гравців.

- Підвищити якість дизайну: Формальний аналіз патернів може допомогти розробникам виявити потенційні проблеми та покращити загальну якість дизайну ігор.

Щодо зразків ігрового дизайну, вони не включають проблемні та вирішувальні елементи, як це зроблено у роботі B. Kreimeier, "The case for game design patterns". Ці зразки описуються за допомогою п'яти елементів: імені, опису, наслідків, використання шаблону та відносин. Елемент "наслідки" зосереджений більше на характеристиках шаблону, ніж на його впливі на розробку та інших варіантах дизайну, які слід враховувати, що є роллю елемента "використання шаблону". Елемент "відносини" описує взаємозв'язки між зразками, такі як підшаблони в межах зразків та суперечливі зразки.

В роботі C.M. Olsson, S. Bjork та S. Dahlskog "Концептуальна модель взаємозв'язків [17]: розуміння шаблонів та механізмів у дизайні ігор" (Proceedings of the DiGRA International Conference (DiGRA 14), 2014) пропонується інтегрувати шаблони дизайну до концептуальної моделі взаємозв'язків. Це дозволяє чіткіше розмежувати поняття ігрових моделей та ігрової механіки. Модель описує механіку гри як результат застосування шаблонів до ігрових моделей через шар контекстуалізації. Цей шар відповідає за конкретизацію та адаптацію загальних шаблонів до конкретної гри. З іншого боку, з конкретної реалізації ігрової механіки (представленої в моделі як код) можна виокремити нові шаблони. Таким чином, концептуальна модель взаємозв'язків встановлює динамічний зв'язок між ігровими моделями, шаблонами та механікою, що дає змогу краще зрозуміти принципи дизайну ігор.

Поряд із загальними підходами до ШІ у відеоіграх, існують й ті, що орієнтовані на вирішення конкретних проблем. Це логічно, адже такі підходи пропонують універсальні рішення поширених проблем ШІ у відеоіграх, сприяючи загальному розвитку штучного інтелекту в цій галузі.

Одним із прикладів є створення моделей для інтелектуальних персонажів відеоігор, що стало предметом численних досліджень та пропонує різні методи. Мови поведінки, наприклад, дають можливість дизайнерам інтуїтивно визначати

поведінку агента, враховуючи загальні процеси. Лоялл та Бейтс [18] запропонували цільову архітектуру реактивного агента, яка дозволяє йому розпізнавати та реагувати на події, що динамічно змінюють відповідність поточної поведінки. Матеас та Штерн [19], а також Матеас та Штерн описали ABL – реактивну мову планування, призначену для створення реалістичних агентів, що здатні координувати свої дії з іншими персонажами.

У роботі Фанге [20] запропоновано концепцію "калькуляції ситуації", яка дозволяє персонажам відеоігор вести міркування та приймати рішення на високому рівні. Завдяки цій концепції персонаж сприймає світ як послідовність ситуацій, розуміючи, як вони можуть змінюватися під впливом його дій. Це дає йому можливість планувати свої дії та досягати поставлених цілей. Для формального опису поведінки автономних персонажів у Фанге використовується мова когнітивного моделювання (CML). Ця мова, що ґрунтується на принципах обчислення ситуації та інтервальних методів, дає персонажам можливість створювати плани дій навіть у дуже складних і динамічних ігрових світах.

Оркін [21] висловив думку, що планування в режимі реального часу є більш ефективним підходом до визначення поведінки агента, ніж використання сценаріїв або кінцевих автоматів. Це пов'язано з тим, що планування в режимі реального часу дає змогу агенту динамічно реагувати на непередбачувані ситуації, що робить його поведінку більш природною та реалістичною. Оркін запропонував модульну цільово орієнтовану архітектуру планування дій для ігрових агентів, подібну до тієї, що використовується у роботі Матеаса та Стерна. Основна відмінність цієї архітектури від мови ABL полягає в чіткому розмежуванні реалізації та даних. У ABL дизайнери безпосередньо реалізують поведінку агента, тоді як в архітектурі Оркіна реалізація здійснюється програмістами, а поведінка визначається дизайнерами за допомогою даних. Таким чином, архітектура Оркіна дає змогу дизайнерам зосередитися на визначенні бажаної поведінки агента, не заглиблюючись у технічні деталі реалізації. Це робить процес розробки ігрових агентів більш гнучким та ефективним.

Андерсон [22] розробив альтернативну мову для дизайну розумних персонажів – мову визначення аватари (AvDL). Ця мова дає змогу описувати як детерміновану, так і цільову поведінку віртуальних сутностей, тобто персонажів і об'єктів у віртуальному світі. Для розширення можливостей AvDL Андерсон створив TheSimpleEntityAnnotationLanguage (SEAL). Ця мова дозволяє інтегрувати визначення поведінки безпосередньо в об'єкти віртуального світу. Це робиться шляхом анотації об'єктів, що дає змогу персонажам обмінюватися з ними інформацією та реагувати на їхню поведінку[30]. Таким чином, AvDL та SEAL разом дають дизайнерам потужний інструментарій для створення складних та реалістичних персонажів, які можуть динамічно взаємодіяти з віртуальним світом.

Навчання агентів також є дієвим підходом до створення інтелектуальних персонажів відеоігор. Завдяки здатності навчатися та адаптуватися до свого оточення, агенти можуть вирішувати проблеми більш гнучко та універсально. Відеоігри протягом останнього десятиліття викликали значний інтерес у спільноті машинного навчання. Багато дослідників намагалися інтегрувати навчання у відеоігри з різним ступенем успіху. Деякі з використовуваних методів схожі на ті, що описані раніше, адже вони засновані на використанні абстракцій та концепцій для узагальнення знань та адаптації до широкого спектру ігрових ситуацій. Важливо зазначити, що навчання є складним завданням, яке потребує значних ресурсів та обчислювальних потужностей. Тому воно не завжди є практичним для створення персонажів у реальному часі. Однак, з розвитком технологій машинного навчання та штучного інтелекту, навчання агентів може стати все більш досконалим та ефективним методом створення інтелектуальних персонажів відеоігор.

Методи міркувань, що базуються на конкретних випадках, використовуються для узагальнення інформації про стан гри. Це дозволяє ШІ приймати більш послідовні рішення в подібних, але не ідентичних ситуаціях. Чанг [23] розглянув можливість використання розпізнавання планів на основі конкретних випадків для зменшення передбачуваності комп'ютерних гравців у стратегіях в реальному часі. Ага запропонував підхід до вивчення та вибору плану,

який використовується агентом, що навчається перемагати проти різних опонентів ШІ в грі WarCraft. Цей підхід ґрунтується на аналізі конкретних випадків, що дозволяє агенту динамічно адаптувати свою стратегію до поточної ситуації. В роботі Ontanon et al. (2007) [24] розроблена та успішно протестована в грі Wargus структура планування для стратегічних ігор в реальному часі. Ця структура дозволяє агентам автоматично отримувати знання про поведінку з анотованих повторень експертів. Матеас [25] (2009) провів додаткові дослідження, використовуючи тестову платформу Wargus. Він продемонстрував, як концептуальні сусідські райони можуть використовуватися для пошуку підходів, що базуються на конкретних випадках. Важливо зазначити, що методи міркувань, що базуються на конкретних випадках, є лише одним із багатьох методів, розроблених для покращення поведінки ШІ у відеоіграх. Їх детальний опис виходить за рамки цієї відповіді, проте розуміння їх принципів роботи дає змогу краще уявити широту та глибину досліджень у цій галузі.

Підходи трансферного навчання ґрунтуються на використанні досвіду, набутого в одному завданні, для покращення поведінки в інших. Шарма (2009) поєднав міркування на основі конкретних випадків з навчанням з підкріпленням, щоб досягти трансферного навчання в MadRTS. Це дозволило покращити результати агента в наступних іграх проти ШІ.

Lee-Urban et al. (2011) використали трансферне навчання в MadRTS за допомогою модульної архітектури, що інтегрує планування ієрархічної мережі завдань (HTN) та навчання концепціям. Шапіро (2012) досяг передачі структурних навичок та понять між різними завданнями за допомогою когнітивної архітектури.

Незважаючи на те, що технологія машинного навчання може призвести до створення універсального ШІ, який можна використовувати в різних іграх, наразі вона не досягла достатнього рівня якості. Деякі методи успішно застосовуються в деяких комерційних іграх, проте їм ще далеко до того, щоб стати стандартними.

З іншого боку, ігрові двигуни є більш практичним підходом до факторингу процесів розробки ігор для покращення їх якості. Їх широко використовують, але вони є всеосяжними інструментами, які розробники повинні застосовувати до

всього ігрового дизайну, а не лише до ШІ. До того ж, вони не дають свободи у фундаментальній архітектурі агентів, якими керують.

Рисунок 1.1 знайомить нас з найпопулярнішими жанрами відеоігор сьогодення по платформах.

Рис. 1.1. Найпопулярніші жанри відеоігор на основних ігрових платформах [7]

1.2 Аналіз моделей поведінки ігрових ворогів

Моделювання поведінки ігрових ворогів є однією з ключових задач при розробці комп'ютерних ігор, зокрема у жанрі First-person shooter. У таких іграх вороги відіграють критичну роль у створенні викликів для гравців, підтриманні напруженості та забезпеченні глибокого занурення в ігровий процес [2]. Для того щоб ігровий світ сприймався живим і динамічним, розробники приділяють особливу увагу розробці моделей поведінки ворогів, які можуть адаптуватися до дій гравця, взаємодіяти із середовищем та демонструвати різноманітність стратегій.

Загальна характеристика поведінки ігрових ворогів залежить від кількох ключових факторів. У іграх вороги виконують різноманітні ролі, залежно від їхньої важливості для сюжету та загальної динаміки геймплея. Деякі вороги створюють загальний рівень напруження, виступаючи як "фонова небезпека" — вони постійно змушують гравця бути насторожі, але не є ключовими для розвитку історії. Інші ж

мають центральну роль у сюжеті, вирізняються складною поведінкою та унікальними здібностями, додаючи глибини ігровому процесу.

Складність взаємодії з ворогами також варіюється. Прості вороги зазвичай мають лінійну поведінку, наприклад, вони атакують гравця, як тільки той потрапляє у їхнє поле зору. Натомість складні вороги проявляють адаптивність: вони можуть використовувати укриття, координувати свої дії з іншими ворогами або змінювати тактики залежно від обставин, що робить битви з ними більш непередбачуваними та цікавими.

Тип середовища, в якому діють вороги, значною мірою впливає на їхню поведінку. У відкритих просторах вони схильні переслідувати гравця, намагаючись оточити його або змусити рухатися у невигідному напрямку. У закритих просторах вороги можуть ефективно використовувати вузькі проходи, пастки чи інші елементи середовища, що додає складності і робить битви більш напруженими.

Говорячи про основні підходи до створення моделей поведінки то потрібно зазначити, що моделі поведінки ворогів еволюціонували разом із розвитком комп’ютерних ігор, забезпечуючи дедалі складнішу та реалістичнішу взаємодію з гравцями. Найпростішим підходом є скриптовані моделі, де поведінка ворога визначається наперед заданими сценаріями. Наприклад, ворог починає стріляти після того, як гравець перетне певну зону. Скрипти легко реалізувати та тестувати, але вони роблять ворогів механічними й неспроможними адаптуватися до змінних умов.

Дерева прийняття рішень [30] дозволяють створювати складніші сценарії поведінки завдяки умовним операціям. Вороги, наприклад, можуть ухвалювати рішення атакувати чи ховатися залежно від рівня здоров’я. Цей підхід забезпечує гнучкість, але його складність швидко зростає зі збільшенням кількості умов. Розширенням дерев прийняття рішень є поведінкові дерева, які додають модульності та повторного використання. Вороги можуть виконувати комбінації дій, як-от знаходити укриття, перезаряджатися та атакувати. Цей підхід є масштабованим і використовується в багатьох сучасних іграх, хоча й потребує більше ресурсів.

Скінченні автомати є ще одним популярним методом [26], який базується на переходах між фіксованими станами, такими як "Патрулювання", "Атака" чи "Втеча". Ці переходи залежать від змін у середовищі чи дій гравця. Цей підхід є надійним і простим у реалізації, але має обмежену адаптивність і часто потребує додаткових алгоритмів для складних ситуацій.

Сучасні ігри активно впроваджують алгоритми штучного інтелекту для забезпечення більш реалістичної поведінки ворогів. Такі вороги здатні аналізувати дії гравця, передбачати його наступні кроки та використовувати дані для оптимізації своїх стратегій. Наприклад, у серії ігор Halo вороги координують атаки та взаємодіють із середовищем.

Ще одним проривом у моделюванні поведінки є алгоритми машинного навчання, які дозволяють ворогам адаптувати свої стратегії до поведінки гравця у реальному часі. Це забезпечує високу адаптивність і непередбачуваність, хоча й вимагає значних обчислювальних ресурсів і складності у реалізації. Таким чином, розвиток моделей поведінки ворогів є важливим елементом еволюції індустрії комп’ютерних ігор, забезпечуючи глибокий і захопливий досвід для гравців.

Важливі аспекти, які потрібно враховувати при створенні моделей поведінки:

1. Адаптивність:

Вороги повинні змінювати свою стратегію залежно від дій гравця. Це вимагає створення складних алгоритмів аналізу та прийняття рішень.

2. Балансування складності:

Занадто сильний ворог може демотивувати гравця, тоді як занадто слабкий не створить виклику.

3. Продуктивність:

Складні алгоритми повинні працювати без затримок навіть у високонавантажених сценах.

4. Реалістичність:

Вороги мають діяти так, щоб їхня поведінка здавалася природною, уникаючи "штучного" враження.

5. Взаємодія з іншими NPC та ворогами:

У багатокористувацьких іграх або іграх із великою кількістю NPC вороги повинні демонструвати злагоджену взаємодію один з одним. Це може включати такі елементи, як командні атаки, підтримка союзників, розподіл ролей у групі.

Наприклад, у серії ігор Left 4 Dead, вороги координують свої дії, щоб організувати засідки, примушуючи гравця змінювати свою тактику.

6. Унікальність поведінки:

Щоб забезпечити різноманітність, різні типи ворогів повинні демонструвати унікальні риси. Наприклад, один ворог може бути агресивним і завжди атакувати, тоді як інший уникає прямого бою і використовує хитрощі.

Такі підходи додають глибини ігровому процесу, оскільки гравець змушений адаптуватися до різних ворогів, які діють за унікальними правилами.

З розвитком обчислювальних можливостей з'являються нові підходи до створення моделей поведінки ігрових ворогів. Деякі з ключових тенденцій включають:

1. Інтеграція ШІ з процедурною генерацією:

У деяких іграх вороги створюються динамічно разом із середовищем, що дозволяє гравцеві стикатися з унікальними ситуаціями у кожному новому проходженні. Наприклад, гра No Man’s **Sky** використовує процедурну генерацію для створення середовища, тобто кожна планета, істота, корабель, мультитул та інші предмети створюються процедурно з використанням алгоритмів у самій грі, що забезпечує різноманітність у вигляді кожного створеного предмета, але аналогічні підходи можуть бути застосовані й для генерації поведінки ворогів.

2. Імітація реального світу:

Використання фізичних моделей і симуляцій для реалізації реалістичної поведінки ворогів. Наприклад, у грі Half-Life 2 вороги враховують фізику навколишнього середовища, використовуючи об'єкти як укриття чи зброю.

3. Навчання з підкріпленням (Reinforcement Learning):

Це один із найбільш перспективних підходів, який дозволяє ворогам адаптуватися до гравця у реальному часі. Вони вчаться вибирати оптимальні дії шляхом спроб і помилок, отримуючи "нагороди" за правильні дії.

Недоліком такого підходу є високі обчислювальні витрати та складність налаштування, проте унікальні результати виправдовують ці витрати.

4. Соціальна поведінка:

Деякі ігри починають інтегрувати механіки, які дозволяють ворогам демонструвати соціальну поведінку, наприклад, збиратися у групи, спілкуватися або ділитися ресурсами. Це додає глибини їхній взаємодії з гравцем.

5. Розширена симуляція середовища:

У деяких іграх вороги реагують не лише на гравця, але й на змінні умови середовища, такі як погодні явища, час доби чи навіть інші NPC. Це робить їх поведінку більш непередбачуваною і цікавою.

Застосування скінчених автоматів у моделях поведінки

Одним із найбільш популярних підходів для моделювання поведінки ворогів є використання скінчених автоматів. Цей підхід дозволяє структурувати поведінку у вигляді набору станів, між якими відбуваються переходи залежно від умов.

1. Переваги використання скінчених автоматів:
    - Простота реалізації: цей підхід легко інтегрувати у більшість ігрових рушіїв, таких як Unity чи Unreal Engine.
    - Прозорість: поведінка ворога легко налаштовується і тестується.
    - Висока продуктивність: порівняно з більш складними моделями, скінченні автомати мають мінімальні вимоги до ресурсів.
2. Приклад реалізації:
    - У грі DOOM (2016) вороги демонструють складну поведінку, яка базується на наборі станів. Вороги можуть атакувати, переслідувати гравця, ухилятися від ударів і ховатися за об'єктами. Усі ці дії впорядковуються за допомогою скінчених автоматів.
3. Недоліки:
    - Обмеженість: ускладнення алгоритмів може призвести до надмірної кількості станів, що ускладнює підтримку і розвиток.

- Відсутність адаптивності: скінченні автомати зазвичай працюють за фіксованими правилами, що робить їх менш ефективними в умовах динамічного середовища.

Подальші перспективи та дослідження

Удосконалення моделей поведінки ігрових ворогів є динамічною сферою досліджень. Новітні технології та підходи дозволяють створювати ворогів, які виглядають ще більш реалістичними і викликають справжній інтерес у гравців. Деякі перспективи, що можуть вплинути на майбутнє розвитку цієї галузі:

Підходи також можна комбінувати. Сучасні системи дедалі частіше поєднують кілька моделей для досягнення найкращих результатів. Наприклад, можна використовувати скінченні автомати для базових станів ворога, але в критичних ситуаціях додавати адаптивність через машинне навчання.

Використання генеративних моделей для створення абсолютно нових патернів поведінки, які враховують не лише поточний стан ворога, але й аналізують історію дій гравця. Це може дозволити ворогам "вчитися" протягом гри і ставати більш небезпечними для досвідчених гравців.

Майбутні моделі можуть враховувати не тільки положення об'єктів, але й їхні фізичні характеристики. Наприклад, ворог може використовувати певні поверхні для створення засідок або навіть брати до уваги динамічні зміни в рівні (руйнівні об'єкти, зміну ландшафту тощо).

У майбутньому вороги можуть функціонувати як частина єдиної інтелектуальної системи, що ділиться інформацією в реальному часі. Наприклад, один ворог виявляє гравця, а інші координуються, щоб оточити його.

Моделі поведінки ігрових ворогів, розроблені для жанру First-person shooter, можуть бути адаптовані для використання в інших жанрах:

Стратегії в реальному часі (RTS): Тут вороги повинні демонструвати командну роботу та здатність до управління ресурсами.

Рольові ігри (RPG): Моделі можуть адаптуватися до різних ситуацій, враховуючи складність світу і взаємодію персонажів.

Survival Horror: У таких іграх акцент робиться на створенні ворогів, які викликають страх, використовуючи їхню непередбачуваність і постійний тиск на гравця.

Вплив моделювання ворогів на ігрову індустрію

Економічний аспект. Якісно опрацьовані вороги підвищують загальний рівень захоплення грою, що безпосередньо впливає на її комерційний успіх. Гравці часто цінують ігри за "розумність" їхніх ворогів.

Реіграбельність. Адаптивні та непередбачувані вороги мотивують гравців проходити гру декілька разів, оскільки кожен досвід буде унікальним.

Розвиток технологій. Інновації в моделюванні ворогів часто стають каталізатором для вдосконалення ігрових рушіїв, алгоритмів і навіть апаратного забезпечення.

Отже, аналіз моделей поведінки ігрових ворогів показує, наскільки складними і багатогранними є завдання, пов’язані з їхньою розробкою. Від базових скінчених автоматів до складних мультиагентних систем, сучасні технології дозволяють створювати ворогів, які можуть здивувати навіть найвимогливішого гравця.

Подальший розвиток цієї галузі лежить у застосуванні інтегрованих рішень, які поєднують у собі адаптивність, реалістичність і економічну ефективність. З кожною новою ітерацією ігор ми наближаємося до створення ворогів, які не тільки виглядають реалістично, але й діють як розумні противники, здатні створити незабутній ігровий досвід.

1.3 Огляд технологій та інструментів для реалізації штучного інтелекту в ігровій індустрії

Штучний інтелект (ШІ) в ігровій індустрії є одним з найважливіших елементів сучасних ігор, оскільки він безпосередньо впливає на поведінку і взаємодію персонажів, ворогів, NPC (неігрових персонажів) і навіть на навколишнє середовище [2]. ШІ дозволяє створювати не лише прості сценарії взаємодії, а й

складні, адаптивні моделі поведінки, що значно покращує досвід гравця, надаючи йому нові виклики, непередбачуваність і глибину в ігрових процесах [33].

Завдяки застосуванню ШІ в іграх стає можливим створення персонажів, які здатні до самонавчання, прогнозування дій гравця і реагування на змінні умови. Це дозволяє інтегрувати складніші сценарії гри, де гравець не лише взаємодіє з фіксованим набором поведінкових моделей, а й стикається з ворогами і ситуаціями, що постійно змінюються, залежно від його власних дій. Наприклад, вороги можуть адаптувати свою тактику, враховуючи попередні дії гравця, чи NPC можуть розвивати більш складну поведінку в залежності від обставин, що в свою чергу змушує гравця адаптуватися та розвивати нові стратегії.

Застосування ШІ також дозволяє автоматизувати значну кількість ігрових процесів, зменшуючи необхідність вручну задавати складні сценарії для кожного елемента гри. Ігрові системи, що базуються на ШІ, можуть автоматично генерувати нові ситуації, змінювати рівень складності, адаптувати ігрові механіки та підтримувати динамічний баланс у грі. Це не лише значно підвищує реалістичність світу гри, але й забезпечує більшу інтересність і повторюваність ігрового процесу, оскільки кожен новий прохід через гру може бути унікальним.

У галузі розробки ШІ для ігор використовується безліч різних підходів і технологій. Зокрема, можна виділити традиційні скриптовані моделі, які працюють за певними заздалегідь визначеними сценаріями, а також більш складніші підходи, такі як дерева прийняття рішень, поведінкові дерева та скінченні автомати. Ці методи дозволяють значно розширити можливості ігрових персонажів, забезпечити більшу гнучкість у їхній поведінці і значно підвищити рівень адаптивності до змінних умов гри.

Крім того, існують сучасні методи, такі як машинне навчання та глибоке навчання, які дозволяють створювати більш складні і адаптивні моделі, що не потребують постійного втручання розробника. Завдяки використанню цих технологій, персонажі в іграх можуть вчитися з досвіду, прогнозувати можливі стратегії гравця, а також адаптувати свої дії до нових ситуацій, створюючи таким чином більш реалістичне і насичене ігрове середовище.

Одним із найбільш потужних інструментів для створення ШІ в іграх є **Unity** — популярний ігровий движок, який надає розробникам потужні засоби для реалізації складних моделей поведінки персонажів і ворогів [13]. Unity дозволяє інтегрувати в ігри різні підходи до розробки ШІ, починаючи від простих скриптованих рішень і закінчуючи складними моделями на основі машинного навчання та глибоких нейронних мереж.

Unity пропонує розробникам величезну кількість готових інструментів, серед яких можна виділити вбудовані бібліотеки для роботи з фізикою, анімацією, а також гнучкі засоби для налаштування поведінки персонажів, управління штучним інтелектом і тестування взаємодії з навколишнім середовищем. Крім того, Unity активно підтримує інтеграцію з різними зовнішніми бібліотеками для машинного навчання, такими як TensorFlow та PyTorch, що дозволяє розробникам використовувати більш складні методи для навчання агентів в реальному часі.

Ще однією важливою складовою є Unity ML-Agents, набір інструментів і бібліотек, що дозволяє розробникам інтегрувати машинне навчання безпосередньо в ігри. ML-Agents підтримує такі алгоритми, як Proximal Policy Optimization (PPO) та Deep Q-Learning (DQN), що дозволяє створювати навчальних агентів, які можуть адаптувати свою поведінку на основі даних про навколишнє середовище, тим самим створюючи більш реалістичні і інтелектуальні моделі взаємодії в ігровому процесі. За допомогою ML-Agents розробники можуть створювати агентів, які не тільки виконують певні завдання, але й вчаться на власних помилках, приймаючи нові рішення на основі попереднього досвіду.

Використання технологій машинного навчання в Unity дозволяє не тільки створювати більш складні моделі поведінки, а й автоматизувати процеси тестування та оптимізації ігор. Вчити агентів за допомогою алгоритмів машинного навчання дозволяє знизити кількість необхідного коду для ручного створення складних сценаріїв і значно спростити процес розробки ігор, дозволяючи зосередитися на більш креативних аспектах гри.

Використання Unity для розробки ШІ в іграх

Unity є одним з найпоширеніших і найпотужніших інструментів для створення ігор завдяки своїй доступності, гнучкості та багатофункціональності. Розробники використовують Unity для створення як простих, так і складних ігрових механік, зокрема втілення штучного інтелекту. Завдяки широкому набору функцій, Unity дозволяє створювати адаптивні моделі поведінки ворогів, NPC, а також інших елементів ігрового світу, роблячи ігри більш інтелектуальними та реалістичними. Вбудовані інструменти і функції дозволяють реалізувати різні підходи до ШІ, від простих скриптованих рішень до складних алгоритмів машинного навчання, що додає багатогранності в розробку сучасних ігор.

Одним з основних переваг Unity є її багатофункціональність. Платформа підтримує не тільки стандартні методи ШІ, як-от дерева прийняття рішень або скінченні автомати, але й дозволяє розробляти складніші рішення на основі поведінкових дерев та алгоритмів глибокого навчання. Це дає змогу розробникам адаптувати поведінку персонажів відповідно до змінних умов гри, що значно підвищує динамічність ігрового процесу. Наприклад, в іграх з великими відкритими світами або складними тактичними боївками використання складних моделей ШІ дозволяє ворогам адаптувати свою поведінку не лише залежно від дій гравця, але й на основі попередніх дій або результатів боїв.

Потужні візуальні інструменти Unity, як Animator або NavMesh, дозволяють створювати та симулювати поведінку агентів у реальному часі, без необхідності написання складного коду для кожної ситуації. Наприклад, за допомогою NavMesh можна забезпечити рухливість персонажів у тривимірному просторі, враховуючи фізику навколишнього середовища. Це дає змогу агентам, таким як вороги чи союзники, орієнтуватися в грі та уникати перешкод, завдяки чому їхня поведінка виглядає природно і реалістично. Додатково, система NavMesh Agent дозволяє агентам ефективно слідувати за гравцем, переміщатися через складні ландшафти або навіть взаємодіяти з іншими об'єктами в ігровому середовищі.

Ще одним важливим аспектом є проста інтеграція Unity з зовнішніми бібліотеками, такими як TensorFlow, PyTorch та ML-Agents, що дозволяє розробникам впроваджувати більш складні алгоритми машинного навчання в

процес розробки ігор. За допомогою таких інструментів можна створювати адаптивних агентів, які здатні вчитися на основі досвіду. Це дозволяє не лише автоматизувати поведінку ворогів або NPC, але й виводити ШІ на новий рівень, де агент може самостійно оцінювати ситуацію, враховувати змінні фактори і оптимізувати свої дії для досягнення заданої мети.

ML-Agents є однією з найбільш корисних інтеграцій для розробників, які бажають впровадити машинне навчання в ігрові процеси. Цей інструмент дає можливість проводити тренування агентів за допомогою алгоритмів, таких як Proximal Policy Optimization (PPO) або Deep Q-Learning (DQN), прямо в Unity. За допомогою ML-Agents можна створювати агентів, які навчаються виконувати завдання, враховуючи всі доступні параметри навколишнього середовища, і адаптуються до нових умов в режимі реального часу. Наприклад, в екшн-іграх це може бути ворог, який вивчає стратегії гравця та змінює тактику атаки залежно від дій гравця, тим самим створюючи більш складний та непередбачуваний досвід гри.

Особливістю ML-Agents є можливість інтегрувати навчання агентів безпосередньо в ігровий процес, що дозволяє створювати більш адаптивні та інтелектуальні системи штучного інтелекту. Розробники можуть використовувати ML-Agents для тренування агентів в умовах реального ігрового світу, не обмежуючи їх лише теоретичними моделями. Цей підхід дозволяє розвивати агента, спостерігаючи, як він реагує на взаємодію з гравцем, змінюючи свою поведінку в залежності від досвіду, що створює реалістичне та непередбачуване середовище для гравця.

Завдяки такій інтеграції, Unity пропонує розробникам величезні можливості для створення динамічних, адаптивних і цікавих ігор. Крім того, Unity дозволяє знижувати складність розробки таких ігор, оскільки процеси навчання і тестування ШІ можна значно спростити, використовуючи вже готові рішення, а також гнучко налаштовувати їх під потреби конкретного проєкту. Технології машинного навчання, вбудовані в Unity, дозволяють значно покращити не лише ігрову механіку, а й створити більш складні і багатогранні взаємодії між гравцем і світом гри.

Unity ML-Agents є потужним інструментом, який відкриває перед розробниками можливість впровадження складних алгоритмів машинного навчання в ігрові проєкти. Цей набір бібліотек і середовищ, розроблений для Unity, дозволяє створювати інтелектуальних агентів, здатних адаптуватися до змінних умов ігрового середовища. Завдяки ML-Agents розробники можуть значно розширити межі можливого в дизайні поведінки персонажів і сценаріїв гри.

Основна перевага Unity ML-Agents полягає в тому, що він використовує методи навчання з підкріпленням (Reinforcement Learning, RL). Ця методика дозволяє агентам навчатися, отримуючи "нагороди" або "штрафи" за свої дії, залежно від того, наскільки вони наближаються до досягнення поставленої мети. Наприклад, ворог у грі може отримувати нагороду за успішне переслідування гравця або за влучну атаку, водночас зазнаючи штрафів за невдалі дії. Такий підхід дозволяє створювати агентів, які можуть вдосконалювати свої навички під час гри, стаючи все більш ефективними.

Unity ML-Agents підтримує низку сучасних алгоритмів навчання з підкріпленням, серед яких:

- Proximal Policy Optimization (PPO): один із найпопулярніших алгоритмів RL, відомий своєю стабільністю і здатністю швидко навчати агентів. PPO ефективно використовується для створення NPC з високим рівнем адаптивності.
- Deep Q-Learning (DQN): алгоритм, який поєднує методи Q-навчання з глибинними нейронними мережами. Він особливо корисний для агентів, які діють у складних середовищах із великою кількістю можливих станів.
- Asynchronous Advantage Actor-Critic (A3C): цей алгоритм дозволяє одночасно тренувати кілька агентів у різних середовищах, прискорюючи процес навчання та забезпечуючи кращу узагальненість моделі.

Навчання агентів у різноманітних середовищах Unity ML-Agents надає можливість створювати спеціалізовані середовища для тренування агентів. Розробники можуть налаштовувати параметри середовища, такі як розміри карт, кількість об’єктів, що взаємодіють, та різноманітність

сценаріїв. Наприклад, в симуляції боїв можна змінювати кількість ворогів, їхні характеристики або навіть умови освітлення, щоб агент навчався адаптуватися до найрізноманітніших ситуацій.

Використання ML-Agents у практичних сценаріях ML-Agents дозволяє не лише створювати ворогів, але й розробляти NPC із соціальними та кооперативними поведінками. Наприклад, NPC можуть координувати свої дії в групі, розподіляти завдання або навіть реагувати на емоційний стан гравця. У спортивних симуляторах це може бути тренування командних стратегій, де персонажі розвивають тактики, зважаючи на сильні й слабкі сторони супротивника.

У жанрі FPS вороги, створені за допомогою ML-Agents, можуть не тільки переслідувати гравця, але й передбачати його дії, займати стратегічні позиції, використовувати укриття та комбінувати атаки. Такі агенти, що навчилися на великій кількості сценаріїв, забезпечують динамічність ігрового процесу, викликаючи у гравця почуття виклику та захоплення.

Використовуючи інструмент, розроблений компанією Unity, можна тренування агентів у реальному часі Одна з найбільших переваг Unity ML-Agents – це можливість візуалізувати процес навчання агента. Розробники можуть спостерігати за тим, як агент взаємодіє з середовищем, аналізувати його дії та вносити корективи в налаштування середовища або алгоритму. Це значно прискорює процес розробки й забезпечує більш передбачувані результати.

Інтеграція з іншими технологіями Unity ML-Agents легко інтегрується з іншими платформами та бібліотеками для машинного навчання, такими як **TensorFlow** і **PyTorch**. Це дає змогу експортувати моделі, навчені в Unity, для подальшого використання в інших додатках або використовувати вже готові моделі в ігрових проєктах.

Таким чином, Unity ML-Agents є надзвичайно потужним інструментом для створення інтелектуальних агентів, які можуть забезпечити реалістичність, динамічність і непередбачуваність ігрового процесу. Це робить його незамінним

для сучасних розробників ігор, які прагнуть розширити межі стандартних ігрових механік.

Ігрові движки є основними платформами для створення інтерактивних ігор, і використання штучного інтелекту (ШІ) стало одним із ключових аспектів їхнього розвитку. Движки, такі як Unreal Engine, CryEngine, Godot та інші, активно впроваджують технології ШІ для створення інтелектуальних персонажів, ворогів, NPC (неігрових персонажів), а також для автоматизації складних процесів ігрового світу. Вони забезпечують розробникам потужні інструменти для впровадження ШІ, які варіюються від базових методів до складних систем, що використовують машинне навчання.

Одним із найпотужніших і популярних ігрових движків, який значно виділяється своїми можливостями у сфері ШІ, є **Unreal Engine**. Цей движок надає широкий спектр інструментів для розробки моделей поведінки агентів, що робить його привабливим вибором для великих і складних проєктів.

Одним із найбільш поширених підходів у реалізації штучного інтелекту на основі Unreal Engine є Behavior Trees (поведінкові дерева). Це інструмент, який дозволяє створювати складні й багаторівневі моделі поведінки агентів. Поведінкові дерева працюють за принципом розгалуження логіки: кожна "гілка" дерева відповідає певній дії, яку агент повинен виконати за певних умов. Наприклад, ворог може перейти від стану патрулювання до стану атаки, якщо виявить гравця в зоні видимості.

Перевагою Behavior Trees є їхня гнучкість і модульність. Розробники можуть легко створювати повторно використовувані компоненти дерева та адаптувати поведінку агентів до змінних умов гри. Це дає змогу масштабувати логіку поведінки, додаючи нові умови або розгалуження без необхідності повної переробки системи. Behavior Trees активно використовуються в іграх з динамічними ігровими сценаріями, наприклад, у шутерах, RPG і стратегіях.

Unreal Engine також надає AI Perception System, яка відповідає за симуляцію відчуттів персонажів, таких як зір, слух і навіть нюх. Ця система дозволяє агентам сприймати ігровий світ через датчики, і на основі отриманих даних змінювати свою

поведінку. Наприклад, ворог може реагувати на шум, створений гравцем, або помічати його рухи в полі видимості.

AI Perception System забезпечує реалістичну поведінку ворогів і NPC, створюючи у гравця відчуття, що персонажі дійсно "живуть" у грі та реагують на його дії, а не просто виконують заздалегідь прописані скрипти.

Для розробки адаптивних і складних моделей поведінки Unreal Engine підтримує інтеграцію з системами машинного навчання, такими як **TensorFlow**, **PyTorch**, а також сторонніми бібліотеками для навчання з підкріпленням. Ця інтеграція дозволяє розробникам створювати агентів, які можуть навчатися на основі взаємодії з ігровим середовищем і вдосконалювати свої навички в процесі гри.

Машинне навчання дає змогу розробляти ворогів, які адаптуються до дій гравця, змінюють свої стратегії та навіть вивчають нові способи досягнення мети. Наприклад, у шутерах вороги можуть аналізувати тактики гравця та змінювати свої схеми атак залежно від ефективності обраних гравцем методів.

Unreal Engine дозволяє створювати симуляційні середовища для тренування агентів. Ці середовища можуть бути використані для навчання агентів різним навичкам, таким як уникнення перешкод, прийняття рішень у бойових умовах або взаємодія з іншими NPC. Навчання можна проводити в режимі реального часу, спостерігаючи за прогресом агентів та вносячи корективи в їхні алгоритми.

Завдяки потужним інструментам для реалізації ШІ Unreal Engine широко використовується у великих AAA-проєктах. Наприклад, в іграх серії Gears of War вороги демонструють складну координацію дій, а в The Last of Us Part II NPC реагують на дії гравця з вражаючим рівнем реалістичності, завдяки поєднанню Behavior Trees та AI Perception System.

Unreal Engine надає розробникам потужні інструменти, що дозволяють створювати реалістичні моделі поведінки, які можна легко адаптувати до потреб конкретної гри. Інтеграція з технологіями машинного навчання, гнучкість Behavior Trees і можливість створення реалістичних сенсорних систем роблять цей движок

універсальним для розробки як невеликих інді-проєктів, так і масштабних ААА-ігор.

Таким чином, використання ШІ в ігрових движках, зокрема в Unreal Engine, дозволяє створювати персонажів і сценарії, що значно збагачують ігровий процес і забезпечують його реалістичність, складність і динамічність.

Штучний інтелект, заснований на методах глибокого навчання, займає провідні позиції в сучасній ігровій індустрії завдяки здатності аналізувати та вивчати складні патерни поведінки. Це дозволяє створювати персонажів і ігрові сценарії, які є надзвичайно адаптивними, реалістичними та багатогранними. Використання бібліотек глибокого навчання, таких як TensorFlow, PyTorch і інші, відкриває розробникам доступ до нових можливостей і технологічних досягнень у цій сфері.

TensorFlow є однією з найвідоміших і найпоширеніших бібліотек для розробки моделей глибокого навчання. Вона була створена компанією Google і підтримує широкий спектр алгоритмів, включаючи навчання з підкріпленням, згорткові нейронні мережі (CNN), рекурентні нейронні мережі (RNN) та трансформери. Ця бібліотека має потужний API, який дозволяє легко інтегруватися з різними платформами, включаючи ігрові движки, такі як Unity або Unreal Engine.

Одним із ключових застосувань TensorFlow у ігровій індустрії є створення ворогів, які можуть адаптуватися до дій гравця в реальному часі. Наприклад, за допомогою методів навчання з підкріпленням, агент може навчитися визначати найефективніші стратегії для досягнення цілей. У симуляційних іграх TensorFlow дозволяє моделювати складні системи, такі як економіка чи екосистеми, які реагують на зміни, внесені гравцем.

TensorFlow також підтримує TensorFlow Lite, що дає змогу розробляти легковагові моделі для мобільних платформ, забезпечуючи високу продуктивність навіть на пристроях із обмеженими ресурсами. Завдяки цьому ігри на мобільних телефонах можуть використовувати складні алгоритми ШІ без значних втрат у швидкодії.

PyTorch, розроблений компанією Facebook, є іншою провідною бібліотекою для глибокого навчання, яка використовується як дослідниками, так і розробниками ігор. Головною перевагою PyTorch є її інтуїтивний інтерфейс і динамічне обчислення графів, що дозволяє легко експериментувати з новими ідеями та підходами. PyTorch є ідеальним вибором для прототипування моделей, оскільки розробники можуть швидко тестувати різні архітектури нейронних мереж і оптимізувати їх.

У контексті ігрової індустрії PyTorch забезпечує високу гнучкість при створенні інтелектуальних агентів. Наприклад, у стратегічних іграх PyTorch може бути використаний для навчання моделей, які аналізують хід гри гравця, прогнозують його подальші дії та адаптують свою тактику. У спортивних симуляторах PyTorch допомагає створювати суперників, здатних грати на різних рівнях складності, змінюючи свій стиль гри залежно від рівня майстерності гравця.

Як TensorFlow, так і PyTorch легко інтегруються з популярними ігровими движками через API та сторонні модулі. Unity, наприклад, підтримує Unity ML-Agents, який надає засоби для використання TensorFlow та PyTorch у тренуванні агентів. Unreal Engine, у свою чергу, підтримує інтеграцію з цими бібліотеками через плагіни або сторонні інструменти.

Наприклад, за допомогою PyTorch розробники можуть створювати складні сценарії навчання з підкріпленням, у яких агенти навчаються проходити рівні або вирішувати головоломки, адаптуючись до умов гри. TensorFlow можна використовувати для створення генеративних моделей, які автоматично генерують нові рівні або створюють процедурно згенерованих персонажів із унікальними характеристиками.

Обидві бібліотеки підтримують розширені можливості, такі як мультиагентне навчання, коли кілька агентів взаємодіють у спільному середовищі, або навчання зі зворотним зв’язком, що дозволяє агентам коригувати свої дії на основі оцінок ефективності.

TensorFlow і PyTorch також підтримують розробку моделей генеративного змагання (GAN), які можуть бути використані для створення реалістичних текстур,

анімацій або навіть цілих персонажів, здатних реагувати на гравця в реальному часі.

І TensorFlow, і PyTorch забезпечують розробникам багатий набір інструментів для створення інтелектуальних і адаптивних систем, які значно підвищують якість ігрового процесу. Завдяки можливостям глибокого навчання, ігри стають більш реалістичними, а персонажі – більш "живими", що сприяє створенню унікального ігрового досвіду для гравців.

Штучний інтелект (ШІ) відіграє ключову роль у формуванні сучасного ігрового процесу, забезпечуючи багатогранність ігрових сценаріїв і створюючи унікальний досвід для кожного гравця. Одним із найпотужніших напрямків його застосування є адаптивність ворогів і неігрових персонажів (NPC), що дає змогу значно розширити межі можливостей традиційного геймплею.

Завдяки алгоритмам ШІ вороги в іграх можуть змінювати свою тактику і стратегії в реальному часі залежно від дій гравця. Це усуває передбачуваність, яка часто зустрічається в іграх із фіксованими сценаріями. Наприклад, якщо гравець обирає агресивний стиль гри, вороги можуть почати діяти більш обережно, організовуючи засідки чи використовуючи групові атаки. Навпаки, у випадку, коли гравець зосереджений на захисті, вороги можуть посилювати натиск або застосовувати нові методи атаки.

Такі адаптивні системи дозволяють створювати ворогів, які виглядають "розумними", і забезпечують постійний виклик для гравця, зберігаючи його інтерес до гри протягом усього проходження. Це особливо важливо для жанрів FPS, RPG та стратегій, де рівень складності та поведінка ворогів безпосередньо впливають на задоволення від гри.

ШІ також сприяє розвитку нових ігрових механік. Наприклад, генерація процедурних рівнів на основі алгоритмів дозволяє створювати унікальні карти, що підлаштовуються під стиль гри конкретного користувача. Уявіть собі платформер, у якому перешкоди на кожному рівні змінюються залежно від того, наскільки успішно гравець проходив попередні етапи. Це створює динамічний геймплей, який не дозволяє гравцеві звикнути до шаблонних ситуацій.

Ще однією цікавою механікою є створення завдань на основі поведінки NPC. Наприклад, NPC можуть аналізувати дії гравця і змінювати свої завдання, щоб адаптуватися до нових обставин. У симуляціях життя, як-от "The Sims", ШІ може використовувати такі алгоритми, щоб змінювати пріоритети персонажів залежно від їхніх потреб або дій гравця, створюючи більш природну взаємодію.

Алгоритми ШІ активно застосовуються для створення динамічних завдань і сценаріїв, які змінюються під час гри. Наприклад, у стелс-іграх вороги можуть організовувати пошуки гравця на основі останнього місця його появи. У цьому випадку ШІ враховує не лише поточні дії гравця, але й історію його взаємодій із середовищем.

Також ШІ може використовуватися для створення багатокрокових завдань, які потребують від гравця взаємодії з іншими NPC. Наприклад, у RPG-іграх алгоритми можуть автоматично генерувати квести, які враховують як рівень персонажа гравця, так і його попередні дії в грі. Це дозволяє створювати більш глибокі та унікальні сюжети, які залежать від виборів гравця.

Особливо важливу роль ШІ відіграє у багатокористувацьких іграх, де він може виконувати роль балансу між гравцями. Наприклад, якщо команда гравця відстає за очками, ШІ може зменшити складність ворогів або надати команді додаткові ресурси. Водночас вороги під управлінням ШІ можуть імітувати поведінку реальних гравців, створюючи враження, що в грі бере участь більша кількість людей.

Використання алгоритмів ШІ для розвитку геймплею дозволяє не лише підвищити рівень реалізму та складності гри, а й забезпечити більш глибоке занурення в ігровий процес. Гравці відчувають, що світ гри "живий", реагує на їхні дії, а NPC є не просто фоновими персонажами, а інтерактивними елементами, які впливають на загальний сюжет.

Завдяки прогресу в області ШІ, ігрова індустрія отримала унікальні інструменти для створення динамічного, непередбачуваного та інтерактивного геймплею, який тримає гравців у напрузі і забезпечує незабутні враження.

2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ГЕНЕРАЦІЇ ПОВЕДІНКИ ІГРОВИХ ВОРОГІВ

2.1 Використання дерев прийняття рішень

Дерева прийняття рішень (Decision Trees) є одним із найпоширеніших і базових підходів до реалізації штучного інтелекту (ШІ) у різних сферах, зокрема в ігрових системах. Цей метод забезпечує чіткий і структурований спосіб прийняття рішень, що ґрунтується на наборі умов та пов’язаних із ними дій. Дерева рішень дозволяють моделювати поведінку персонажів у складних ситуаціях, зберігаючи водночас простоту розробки та зрозумілість для програмістів і дизайнерів.

Дерево прийняття рішень можна уявити як дерево з гілками, де кожна гілка представляє можливий шлях. Починаючи з кореня, система послідовно перевіряє умови, представлені внутрішніми вузлами. Залежно від результату перевірки, вибирається наступний шлях. Коли алгоритм досягає листового вузла, приймається конкретне рішення.

Цей підхід ідеально підходить для ситуацій, коли потрібно приймати рішення на основі послідовності перевірок. Наприклад, NPC у грі може використовувати дерево рішень, щоб визначити, чи повинен він патрулювати, переслідувати гравця або ховатися, залежно від рівня загрози.

Однією з головних переваг дерев прийняття рішень є їхня простота. Їх легко зрозуміти навіть тим розробникам, які не мають глибоких знань у галузі ШІ. Кожна гілка дерева відображає логічну послідовність дій, що робить цей підхід інтуїтивно зрозумілим. Завдяки своїй наочності дерева рішень часто використовуються на етапі планування та проектування ігрових систем, допомагаючи команді краще зрозуміти, як NPC має взаємодіяти з гравцем або іншими об’єктами гри.

Дерева прийняття рішень можуть бути використані в широкому спектрі ігрових жанрів, від стратегій і RPG до шутерів і симуляторів. Наприклад у стелс-

іграх NPC можуть визначати, чи варто розпочати переслідування, викликати підкріплення або повертатися до звичайного патрулювання. У спортивних симуляторах, таких як футбольні ігри, дерева рішень допомагають гравцям приймати тактичні рішення, наприклад, обирати між пасом, ударом або захистом.

Хоча дерева прийняття рішень є порівняно простим інструментом, вони здатні моделювати досить складну поведінку. Наприклад, вони можуть включати багато умов і дій, створюючи багаторівневу структуру, яка дозволяє NPC реагувати на різні ситуації в грі. Водночас обчислювальна вартість таких дерев залишається невеликою, що є важливим фактором для підтримання продуктивності гри.

Уявімо NPC у грі, який охороняє базу. Дерево прийняття рішень може виглядати так:

Чи бачить NPC гравця?

- Якщо так, переходить до наступної умови.
- Якщо ні, продовжує патрулювати.

Чи гравець знаходиться в зоні атаки?

- Якщо так, NPC розпочинає атаку.
- Якщо ні, переслідує гравця, намагаючись наблизитися.

Незважаючи на простоту, дерева прийняття рішень мають свої обмеження. Зі збільшенням кількості умов і можливих дій дерево може стати надзвичайно великим і важким для підтримки. Це явище називається експоненційним зростанням складності. Крім того, NPC, які використовують лише дерева рішень, можуть виглядати передбачуваними та неадаптивними, якщо дерево не враховує широкий спектр сценаріїв.

Отже, дерева прийняття рішень є основою для створення базових моделей поведінки в ігрових системах. Вони поєднують простоту реалізації з достатньою функціональністю для вирішення багатьох задач. Попри певні обмеження, цей метод залишається популярним інструментом для розробників, особливо на етапах проектування та тестування. Їхня інтеграція з іншими підходами, такими як

поведінкові дерева чи алгоритми машинного навчання, відкриває ще більше можливостей для створення інтелектуальних ігрових систем.

2.2 Застосування поведінкових дерев

Поведінкові дерева (Behavior Trees) є одним із найефективніших інструментів для створення складних моделей поведінки в ігрових системах. Вони часто використовуються для управління штучним інтелектом (ШІ) NPC (неігрових персонажів) у таких жанрах, як стратегії, шутери, рольові ігри та симулятори. Цей підхід поєднує модульність, гнучкість і масштабованість, дозволяючи розробникам створювати інтелектуальних агентів, здатних виконувати різноманітні завдання.

Поведінкове дерево — це ієрархічна структура, що організовує логіку прийняття рішень і виконання дій для NPC. Вона складається з вузлів різних типів, які визначають, як і коли виконуються певні дії. Основними елементами дерева є:

- Кореневий вузол, що запускає процес прийняття рішень.
- Контрольні вузли (Selectors, Sequences), які визначають порядок виконання дочірніх вузлів.
- Умовні вузли, що перевіряють стан гри або NPC.
- Дії, які виконуються NPC (наприклад, "Рухатися до точки" або "Атакувати гравця").

Основний принцип роботи поведінкового дерева полягає в проходженні через його вузли в заданій послідовності. Процес починається з кореневого вузла й далі переходить до дочірніх, перевіряючи умови та виконуючи дії залежно від заданих правил. Якщо умова у вузлі не виконується, дерево переходить до наступного вузла, поки не знайде дію, яку можна виконати.

Приклад поведінкового дерева для охоронця може виглядати так:

1. Перевірити, чи бачить охоронець гравця.
    - Якщо так, перейти до вузла "Переслідувати гравця".
    - Якщо ні, перейти до вузла "Патрулювати".

2. Чи знаходиться гравець у зоні атаки?
    - Якщо так, виконати вузол "Атакувати".
    - Якщо ні, повернутися до переслідування.

Поведінкові дерева пропонують низку переваг для розробників ігор. Їхня модульна структура дозволяє легко змінювати та доповнювати логіку поведінки NPC. Завдяки контролюючим вузлам, таким як Selectors і Sequences, NPC можуть динамічно адаптуватися до змін у грі. Крім того, візуальне представлення поведінкових дерев полегшує їх розуміння та модифікацію.

Поведінкові дерева широко застосовуються для створення різних типів NPC у відеоіграх, забезпечуючи їм здатність приймати рішення на основі складних багаторівневих логік. Завдяки своїй структурі вони дозволяють моделювати поведінку як простих ворогів, так і складних персонажів із багатофункціональною взаємодією. У шутерах, наприклад, поведінкові дерева використовуються для управління рішеннями NPC, які можуть вибирати між такими діями, як пошук укриття, стрільба або переслідування гравця. Ці дії залежать від рівня загрози, який NPC оцінює на основі поточної ситуації, що робить їхню поведінку динамічною та адаптивною.

В іграх жанру RPG поведінкові дерева виконують роль інструменту, який визначає, як персонажі взаємодіють із гравцем або навколишнім середовищем. Наприклад, NPC може прийняти рішення ініціювати діалог із гравцем, розпочати торгову взаємодію або, за певних умов, атакувати. Важливо, що поведінкові дерева дозволяють створювати ієрархію рішень, де кожен вузол відповідає за певну умову чи дію, що забезпечує логічність і узгодженість у поведінці персонажів.

У стратегічних іграх, де NPC керують великими арміями або складними економічними системами, поведінкові дерева дозволяють реалізувати складну тактичну логіку. Наприклад, NPC може ухвалювати рішення щодо того, чи варто атакувати базу супротивника, збирати ресурси для зміцнення економіки чи будувати оборонні споруди для захисту. У таких випадках поведінкові дерева

допомагають NPC ефективно реагувати на зміни в умовах гри, дотримуючись загальної стратегії.

Таким чином, використання поведінкових дерев у сучасних ігрових системах стало невід'ємною частиною процесу створення інтелектуальних NPC. Вони забезпечують не лише логічність і передбачуваність дій персонажів, а й дозволяють створювати складні, динамічні моделі, які роблять ігровий процес більш цікавим та насиченим.

Приклад роботи поведінкового дерева в грі

Уявімо гру, де гравець намагається втекти від ворога:

1. Selector: Вибирає одну з кількох можливих дій.

   Умовний вузол: "Чи бачить NPC гравця?"

   - Якщо так, виконується Sequence для переслідування гравця.
   - Якщо ні, NPC повертається до патрулювання.

2. Sequence: Послідовність дій для переслідування.

   - Рухатися до гравця.
   - Атакувати, якщо знаходиться на близькій відстані.

Поведінкові дерева, хоч і є потужним інструментом, мають свої обмеження. Зі зростанням складності дерева його підтримка стає все більш складною. Крім того, поведінка NPC, заснована виключно на деревах прийняття рішень, може бути передбачуваною та менш цікавою для гравця.

Отже, поведінкові дерева є потужним і гнучким інструментом для створення інтелектуальних агентів у відеоіграх. Вони дозволяють моделювати широкий спектр поведінкових сценаріїв, зберігаючи при цьому простоту й модульність. Завдяки своїм перевагам цей підхід активно використовується в сучасній ігровій індустрії та продовжує залишатися основою для розробки складних ігрових механік.

2.3 Метод нейронних мереж ML-Agents

Метод нейронних мереж у контексті Unity ML-Agents є одним із найсучасніших підходів до генерації поведінки ігрових ворогів [4]. ML-Agents — це набір інструментів і бібліотек, що дозволяє розробникам застосовувати методи машинного навчання для створення інтелектуальних агентів у ігровому середовищі. Завдяки цьому підходу NPC можуть демонструвати адаптивну поведінку, вивчаючи найефективніші стратегії через взаємодію з ігровим світом.

Unity ML-Agents використовує нейронні мережі та алгоритми навчання з підкріпленням (Reinforcement Learning, RL), де агент отримує нагороди або штрафи залежно від своїх дій. Основними етапами роботи з ML-Agents є:

Створюється симуляція, де штучний агент діє в оточенні інших об'єктів. Агент збирає інформацію про середовище за допомогою сенсорів, які фіксують такі дані, як розташування противників, наявність перешкод та кількість доступних ресурсів. На підставі зібраних даних агент приймає рішення про свої наступні дії. Ці рішення можуть бути простими (наприклад, рух вліво) або складними (наприклад, виконання стрибка з певною силою). Кожне рішення оцінюється системою, яка визначає, наскільки воно було корисним для досягнення поставленої мети. Ця оцінка використовується для навчання агента. Для навчання застосовується нейронна мережа, яка постійно вдосконалюється за допомогою спеціальних алгоритмів підкріплення, таких як Proximal Policy Optimization або Deep Q-Learning

Щодо переваг ML-Agents то це потужний інструмент, який дозволяє створювати неігрових персонажів (NPC) з надзвичайно реалістичною та адаптивною поведінкою. Завдяки машинному навчанню, NPC можуть навчатися в динамічних ігрових середовищах, реагуючи на зміни в реальному часі, наприклад, на дії гравця або зміну ігрового сценарію. Це означає, що NPC можуть ховатися за укриттями, стратегічно переслідувати гравця або використовувати складні тактики в бою, ніби вони були живими людьми. Крім того, ML-Agents можуть самостійно освоювати нові ситуації та вдосконалювати свої стратегії, аналізуючи попередній досвід. Такий підхід не тільки робить гру більш цікавою та непередбачуваною, але

й суттєво скорочує час розробки, оскільки немає необхідності вручну програмувати кожну дію NPC.

Хоча ML-Agents пропонують потужний інструментарій для створення інтелектуальних NPC, цей метод має свої обмеження. Високі вимоги до обчислювальних ресурсів та необхідність глибоких знань у машинному навчанні роблять його недоступним для всіх розробників. Крім того, непередбачувана поведінка навчених агентів може створювати додаткові труднощі в процесі розробки.

ML-Agents ефективно застосовуються в іграх різного жанру:

- Шутери: Вороги можуть адаптуватися до стилю гри користувача, використовуючи стратегії для уникнення атак або фокусування на слабких місцях гравця.
- RPG: NPC можуть змінювати свою поведінку залежно від репутації гравця, його рівня чи спорядження.
- Стратегії: Вороги можуть вивчати, як краще атакувати базу гравця, розробляти стратегії захоплення ресурсів або уникати пасток.
- Симулятори: ML-Agents дозволяють створювати складних агентів, які поводяться реалістично у процедурно згенерованих світах.

ML-Agents підтримують кілька алгоритмів навчання, що дозволяє налаштовувати систему відповідно до потреб проекту:

- Proximal Policy Optimization (PPO): Оптимізований алгоритм навчання з підкріпленням, що забезпечує стабільність тренувань.
- Deep Q-Learning (DQN): Алгоритм, який поєднує нейронні мережі та Q-навчання для вибору найкращої дії в кожній ситуації.
- Self-Play: Техніка, де агенти навчаються через гру один з одним, дозволяючи створювати NPC з високим рівнем складності.

У порівнянні з традиційними методами, такими як скінченні автомати чи поведінкові дерева, ML-Agents забезпечують набагато більшу гнучкість та адаптивність, але вимагають більше ресурсів для розробки та тренування. На відміну від алгоритмів із жорстко заданою логікою, цей підхід дозволяє NPC

"вчитися" з часом, демонструючи поведінку, яка є важко передбачуваною та більш реалістичною.

Отже, метод нейронних мереж ML-Agents є одним із найпотужніших інструментів для генерації поведінки ігрових ворогів. Незважаючи на високі вимоги до ресурсів і складність впровадження, цей підхід відкриває нові можливості для створення складних, динамічних та реалістичних NPC, що значно підвищує якість ігрового досвіду.

2.4 Системи на основі правил (Rule-based Systems)

Системи на основі правил (Rule-based Systems) [32] є класичним підходом до генерації поведінки ігрових ворогів і NPC. Цей метод ґрунтується на визначенні чітких наборів умов і дій, які виконуються залежно від стану гри чи поведінки гравця. Такий підхід широко використовується завдяки простоті реалізації та передбачуваності результатів.

Основою роботи систем на основі правил є набір логічних тверджень, які визначають, які дії слід виконати за певних умов. Кожне правило складається з умови та відповідної дії. Коли умова виконується, система виконує вказану дію. Правила можуть бути організовані в ієрархію для створення більш складних поведінкових моделей.

Традиційні системи на основі правил дозволяють розробникам легко визначити, як NPC повинні поводитися в різних ситуаціях. Це забезпечує високий рівень контролю над поведінкою персонажів. Однак, така детермінованість може призвести до передбачуваної та менш реалістичної поведінки, особливо в динамічних ігрових світах. Крім того, створення складних сценаріїв за допомогою правил може вимагати великої кількості логічних конструкцій, що ускладнює підтримку та модифікацію системи.

Хоча системи на основі правил забезпечують певний рівень контролю над поведінкою персонажів, їх можливості обмежені. Жорстка логіка "якщо-то" не

дозволяє створювати складні та непередбачувані поведінки, які є характерними для живих істот.

Системи на основі правил мають ряд недоліків. Їхня робота обмежується рамками заздалегідь визначених правил, що може призводити до нелогічної поведінки NPC в непередбачених ситуаціях. Крім того, зі збільшенням кількості правил зростає ризик появи конфліктів між ними, що ускладнює управління системою. Також, такі системи не здатні вчитися на помилках і адаптуватися до змін в ігровому світі, що обмежує їхню реалістичність.

Приклади використання в іграх

1. Шутери: Rule-based Systems використовуються для створення ворогів, які атакують гравця в межах певної зони видимості або реагують на звук, створений гравцем.
2. RPG: NPC виконують певні сценарії, такі як патрулювання, торгівля чи видача завдань, залежно від стану гри.
3. Стратегії: Правила визначають поведінку ворогів, наприклад, атаку на базу гравця при досягненні певного рівня сили.

Системи, засновані на правилах, зазвичай реалізуються за допомогою умовних операторів, таблиць рішень або спеціалізованих продукційних систем. Умовні оператори, такі як if-else конструкції, є найпростішим способом реалізації правил. Таблиці рішень дозволяють наочно представити логіку поведінки у вигляді таблиці, де кожен рядок відповідає певному правилу. Продукційні системи, такі як Drools чи CLIPS, є більш складними інструментами, які забезпечують гнучке керування великою кількістю правил та складними логічними виразами.

У порівнянні з методами, такими як поведінкові дерева чи скінченні автомати, Rule-based Systems мають простішу структуру, але поступаються в гнучкості. Вони ефективні для створення базової поведінки NPC, але не підходять для складних сценаріїв, де потрібна адаптивність або самонавчання.

Отже, системи на основі правил залишаються ефективним та простим інструментом для створення поведінки ігрових ворогів. Вони підходять для невеликих або середніх проектів, де необхідна детермінована поведінка NPC, яка

легко піддається контролю та налаштуванню. Однак для створення більш складних, адаптивних систем їх варто поєднувати з іншими методами, такими як поведінкові дерева чи нейронні мережі.

2.5 Методи на основі скінчених автоматів

Скінченні автомати (Finite State Machines, FSM) [1, 5] є одним із найпоширеніших підходів до моделювання поведінки персонажів у відеоіграх. Цей метод базується на ідеї поділу поведінки персонажа на стани, між якими здійснюються переходи залежно від певних умов. Завдяки своїй простоті, передбачуваності й ефективності скінченні автомати широко використовуються для створення як базових, так і складних моделей поведінки.

Скінченний автомат складається з трьох основних елементів: станів, умов переходів і подій, що запускають ці переходи. Кожен стан визначає певну поведінку або дію персонажа, наприклад, патрулювання, атаку чи втечу. Перехід між станами залежить від подій, що відбуваються у грі, таких як наближення гравця, отримання пошкоджень або досягнення певного пункту маршруту.

У шутерах скінченні автомати часто використовуються для управління ворогами. Наприклад, ворог може перебувати у стані патрулювання, поки не помітить гравця. Після цього відбувається перехід до стану переслідування, а якщо дистанція зменшується до критичної, ворог починає атакувати. Якщо гравець завдасть значних пошкоджень, ворог може перейти у стан втечі або пошуку укриття.

В іграх жанру стелс скінченні автомати забезпечують поведінку охоронців. Охоронець може перемикатися між станами патрулювання, перевірки підозрілих звуків, погоні за гравцем та повернення до звичайного режиму патрулювання після завершення тривоги. Такий підхід дозволяє легко організувати логіку ворогів, забезпечуючи передбачувану реакцію на дії гравця.

У платформерах NPC, які є ворогами, часто використовують FSM для базових дій, таких як пересування по заздалегідь визначеній траєкторії, стрибки на перешкоди або атака гравця, якщо той наближається до певної зони.

Перевагою використання скінченних автоматів є те, що вони забезпечують простоту реалізації й тестування поведінки NPC. Їхня модульна структура дозволяє розробникам легко додавати нові стани та переходи, що робить цей метод гнучким. FSM також мають низькі вимоги до ресурсів, що є важливим для оптимізації продуктивності гри, особливо в проектах із великою кількістю NPC.

Ще однією перевагою є передбачуваність і логічність. Оскільки всі можливі стани та переходи визначаються заздалегідь, розробники можуть бути впевнені, що NPC діятиме відповідно до очікуваної логіки.

Основним недоліком скінченних автоматів є їх обмежена гнучкість. Для складних ігор із великою кількістю станів та умов FSM можуть стати громіздкими й важкими для управління. Наприклад, якщо NPC має виконувати десятки різних дій, то кількість станів і переходів між ними може зрости до сотень, що ускладнює тестування та внесення змін.

Крім того, FSM зазвичай не враховують випадкові фактори або складні сценарії, які можуть виникнути у динамічному ігровому середовищі. У таких випадках розробникам може знадобитися використання додаткових алгоритмів або комбінування FSM із іншими підходами, наприклад, з поведінковими деревами чи машинним навчанням.

Для подолання недоліків класичних скінченних автоматів часто використовуються ієрархічні скінченні автомати (Hierarchical FSM, HFSM). Цей підхід дозволяє об'єднувати групи станів у більші класи або категорії, що спрощує управління складними поведінковими логіками. Наприклад, усі дії, пов’язані з атакою, можна об’єднати в одну групу, а переходи між підстанами обробляти окремо.

Іншим підходом є комбінування FSM із іншими методами, такими як дерева прийняття рішень або поведінкові дерева. Це дозволяє створювати більш адаптивні моделі, зберігаючи при цьому переваги кожного з методів. Наприклад, FSM може

відповідати за базові стани NPC, тоді як дерево прийняття рішень використовується для деталізації конкретних дій у межах кожного стану.

Отже, методи на основі скінченних автоматів залишаються ефективним інструментом для створення поведінки NPC у відеоіграх. Вони прості у реалізації, дозволяють легко керувати поведінкою ворогів і NPC, забезпечують передбачуваність і стабільність. Незважаючи на певні обмеження, скінченні автомати є фундаментальною технологією в ігровій індустрії й залишаються актуальними завдяки своїй надійності та гнучкості в комбінації з іншими підходами.

2.6 Порівняння методів генерації поведінки ігрових ворогів

Розробка поведінки ігрових ворогів є важливим аспектом створення захопливого та реалістичного геймплею. Сучасна ігрова індустрія використовує різні методи для реалізації цієї поведінки, кожен із яких має свої переваги, обмеження і сфери застосування. У цьому розділі проведемо детальний аналіз і порівняння основних методів: систем на основі правил, дерев прийняття рішень, поведінкових дерев, скінчених автоматів і методів на основі машинного навчання.

Системи на основі правил – це простий і ефективний спосіб створити передбачувану поведінку ігрових персонажів. Вони легко налаштовуються і працюють швидко, але мають два основних обмеження. По-перше, вони не дозволяють створювати розумних противників, які здатні адаптуватися до дій гравця. По-друге, зі збільшенням кількості правил, що керують їхньою поведінкою, система може стати надмірно складною для розуміння та модифікації.

Дерева прийняття рішень — це потужний інструмент для структурування процесів прийняття рішень. Представляючи рішення у вигляді дерева, де кожен вузол відповідає певній умові, цей метод дозволяє легко відстежити логіку прийняття рішень. Швидкість обробки інформації є ще однією перевагою дерев прийняття рішень, оскільки навіть великі структури обробляються досить швидко. Проте, при моделюванні складних сценаріїв, дерева можуть ставати надмірно

громіздкими, що ускладнює їхнє використання. Крім того, дерева прийняття рішень не є достатньо адаптивними, оскільки зміна умов вимагає внесення змін у структуру дерева.

Поведінкові дерева — це гнучкий і масштабований підхід до створення складних поведінкових моделей, який дозволяє легко організувати та модифікувати логіку. Однак, початкове налаштування і зростання обчислювальних витрат при розширенні дерева можуть бути обмеженнями для великих проектів.

Скінченні автомати — це математична модель, яка використовується для опису дискретних систем, що змінюють свій стан у часі. Кожен стан системи відповідає певному набору умов, а переходи між станами визначаються зовнішніми подіями. Цей метод відрізняється простотою реалізації і дозволяє легко зрозуміти логіку поведінки системи. Однак, скінченні автомати мають певні обмеження. Зі збільшенням кількості можливих станів і переходів, модель стає все більш складною для розуміння і модифікації. Крім того, скінченні автомати не дуже добре підходять для моделювання складних і динамічних систем, оскільки не дозволяють ефективно описувати адаптивну поведінку.

Методи машинного навчання відкривають нові можливості для створення реалістичних ігрових персонажів. Завдяки нейронним мережам, вороги можуть навчатися на основі взаємодії з оточенням і демонструвати динамічну та адаптивну поведінку. Це дозволяє створювати більш захоплюючі ігровий досвід. Проте, розробка таких систем є складним завданням, яке вимагає глибоких знань в області машинного навчання. Крім того, навчання нейронних мереж потребує значних обчислювальних ресурсів і часу. Також варто зазначити, що поведінка персонажів, керованих нейронними мережами, може бути менш передбачуваною порівняно з персонажами, керованими традиційними методами

Кожен із методів має свої сфери застосування. Наприклад, системи на основі правил підходять для простих ігор із передбачуваною поведінкою NPC. Дерева прийняття рішень і поведінкові дерева забезпечують хороший баланс між простотою та гнучкістю, що робить їх ефективними для середніх проектів. Скінченні автомати краще використовувати для створення базових NPC із

детермінованою логікою. Методи машинного навчання підходять для великих проектів із потребою в адаптивності та високій складності, але вони вимагають значних ресурсів.

На таблиці 2.1 описано кожний метод та його ключові недоліки

Таблиця 2.1

Порівняння методів з їх описом та недоліками

Метод/модель	Опис	Ключові недоліки
Нейронні мережі	Система, що навчається на основі вхідних даних	- Висока обчислювальна складність під час навчання та прогнозування в реальному часі - Потреба у великих наборах даних для навчання -Непередбачуваність поведінки персонажа в умовах, відмінних від навчальних даних
Системи на основі правил (Rule-based Systems)	Набір предвизначених правил "якщо-то" для визначення поведінки	- Обмежена гнучкість через суворі прив’язки до умов - Складність розширювання зі збільшенням гри, що ускладнює керування системою, робить її важчою для читання, налагодження та оновлення
Дерева поведінки (Behavior Trees)	Ієрархічна структура вузлів, що визначає послідовність дій та прийняття рішень ігрових персонажів	- Важко відлагоджувати великі дерева - Високі витрати пам'яті при створенні великих дерев з багатьма вузлами

Продовження таблиці 2.1

Порівняння методів з їх описом та недоліками

Метод/модель	Опис	Ключові недоліки
Скінченні автомати (FSM)	Система станів та переходів між ними на основі умов. Кожен стан представляє конкретну поведінку (атака, патрулювання, відступ тощо)	- Стає важко керованою, якщо кількість станів перевищує десятки або сотні.І

2.7 Математична модель генерації поведінки ворогів на основі скінчених автоматів

Скінченний автомат є фундаментальною математичною моделлю, яка ефективно використовується для генерації поведінки ігрових ворогів у комп'ютерних іграх. Ця модель дозволяє описувати поведінку ворога як набір станів із визначеними правилами переходів між ними, залежно від поточного контексту гри.

Основні елементи математичної моделі

1. Модель передбачає дискретний набір станів, які відповідають різним аспектам поведінки ворога, наприклад:
   - Idle: ворог неактивний.
   - Patrolling: ворог патрулює територію.
   - Chasing: ворог переслідує гравця.
   - Shooting: ворог атакує гравця.
   - Dialogue: ворог взаємодіє з гравцем через діалог.

Правила переходів між станами залежать від набору вхідних параметрів (x), таких як:

- Відстань до гравця (d)
- Рівень доброзичливості (α)
- Шанс на взаємодію (pinteraction)

Функція переходів визначає, який стан обере ворог, виходячи з поточного стану та зовнішніх умов.

Ймовірність переходу зі стану s_i до s_j обчислюється на основі ваг переходів w_{ij} і функції умовних факторів $f(x)$

$$P(S_j|S_i, x) = \frac{w_{ij} \cdot f(x)}{\sum_k w_{ik} \cdot f(x)} \quad (2.1)$$

Функція умовних факторів f(x)

Ця функція враховує параметри навколишнього середовища, зокрема:

Вплив відстані

$$f_\alpha(\alpha) = \begin{cases} 1, & \text{якщо } d \leq d_{shot}, \\ \frac{d_{chase} - d}{d_{chase} - d_{attack}}, & \text{якщо } d_{attack} < d \leq d_{chase}, \\ 0, & \text{якщо } d > d_{chase} \end{cases} \quad (2.2)$$

Вплив доброзичливості

$$f_\alpha(\alpha) = \begin{cases} 1 - \frac{\alpha}{100}, & \text{якщо } \alpha \geq 0, \\ 1 + \frac{\alpha}{100}, & \text{якщо } \alpha < 0. \end{cases} \quad (2.2)$$

Шанс на взаємодію

$$f_p(Pinteraction) = Pinteraction \quad (2.3)$$

Ця формула означає, що вплив шансу взаємодії безпосередньо пропорційний самому значенню. Коли $Pinteraction$ збільшується (ймовірність взаємодії стає більшою), вплив також прямо пропорційно зростає і коли $Pinteraction$ = 0 — з ворогом неможливо буде взаємдіяти.

Розрахунок переходів

Стан ворога на наступному кроці визначається за формулою:

$$s(t+1) \arg max \{P(s_j|s_i, x)\} \quad (2.4)$$

Ця формула визначає наступний стан $s(t+1)$ і обирає той стан s_j який має максимальну ймовірність переходу $P(s_j|s_i,x)$ виходячи з поточного стану s_i зовнішнього впливу x.

$P(s_j|s_i,x)$ - умовна ймовірність переходу зі стану s_i до стану s_j, враховуючи зовнішній вплив x

Приклад обчислення

Вхідні дані:

- Відстань до гравця: d = 6, порогові значення d_{shot} = 3, d_{chase} = 10
- Рівень доброзичливості: α = 50
- Шанс на взаємодію: $P_{interaction}$ = 0.8

Розрахунок умовних факторів:

- $f_d(6) = \frac{10-6}{10-3} = 0.57$
- $f_\alpha(50) = 1 - \frac{50}{100} = 0.5$
- $f_p(0.8) = 0.8$

Загальний вплив:

$$f(x) = f_d(6) \cdot f_\alpha(50) \cdot f_p(0.8) = 0.57 \cdot 0.5 \cdot 0.8$$

Ймовірність переходів:

На основі заданих ваг переходів w_{ji} і обчисленого $f(x)$, визначаються ймовірності переходу між станами.

Скінченні автомати пропонують структурований і формалізований підхід до моделювання поведінки ігрових персонажів. Їх легко інтегрувати в ігрові системи, а передбачувана поведінка персонажів спрощує процес розробки та тестування. Крім того, скінченні автомати ефективно обробляють вхідні дані в реальному часі.

Однак, для складних ігор з великою кількістю факторів, що впливають на поведінку персонажів, модель скінченного автомата може стати надмірно складною для управління.

Крім того, скінченні автомати не здатні самостійно навчатися і адаптуватися до змін в ігровому середовищі, що обмежує їх застосування в іграх з високим рівнем динаміки.

Отже, математична модель на основі скінченних автоматів є потужним інструментом для створення систем поведінки ігрових ворогів, особливо в проектах середньої складності. Вона забезпечує баланс між простотою реалізації та ефективністю, але може бути вдосконалена шляхом інтеграції із сучасними технологіями, такими як навчання з підкріпленням чи нейронні мережі.

3 РОЗРОБКА СИСТЕМИ ГЕНЕРАЦІЇ ПОВЕДІНКИ ІГРОВИХ ВОРОГІВ

3.1 Проєктування системи генерації поведінки ворогів

Система генерації поведінки ворогів була спроєктована на основі скінченних автоматів, що дозволяє розділити поведінку на окремі стани та визначити правила переходу між ними. Такий підхід забезпечує структурованість коду, зручність його масштабування та модифікації.

Основною метою системи є моделювання реалістичної реакції ворогів на дії гравця, включаючи патрулювання території, переслідування гравця, атаку та інші сценарії.

Архітектура системи базується на трьох основних компонентах:

- Контекст ворога (EnemyContext): Цей компонент забезпечує єдину точку доступу до всіх необхідних даних та функціональних можливостей ворога. Сюди входять дані від сенсорів, системи навігації, анімації та конфігураційні параметри. Контекст ворога дозволяє різним частинам системи отримувати необхідну інформацію про ворога.
- Машина станів (StateMachine): Машина станів керує поточним станом ворога та визначає, коли і як ворог переходить з одного стану в інший. Кожен стан має свої умови переходу, які визначають, коли ворог повинен змінити свою поведінку.
- Класи станів: Кожен конкретний стан ворога (наприклад, патрулювання, переслідування, стрільба) представлений окремим класом. Ці класи успадковують спільні методи та структуру від базового класу BaseState. Класи станів визначають, які дії повинен виконувати ворог в кожному конкретному стані.

Принцип роботи системи заключається в тому, що на початку гри кожен ворог ініціалізує машину станів, яка завантажує список доступних станів. Алгоритм роботи виглядає наступним чином:

Ініціалізація:

- При створенні ворога визначається його початковий стан, наприклад, патрулювання.
- Усі необхідні параметри передаються через контекст ворога.

Виконання стану:

- Кожен кадр гри викликаються методи Update і HandleConditions поточного стану.
- Update відповідає за виконання основної логіки стану (наприклад, рух до точки патрулювання).
- HandleConditions перевіряє умови для переходу до іншого стану (наприклад, виявлення гравця).

Перехід між станами:

- Якщо умови переходу виконуються, машина станів змінює поточний стан на новий через метод SwitchState.
- Поточний стан завершується викликом методу Exit, після чого новий стан ініціалізується методом Enter.

Система містить кілька станів, кожен з яких відповідає за конкретний аспект поведінки ворога

PatrollingState (Стан патрулювання):

- Вороги пересуваються між випадковими точками в межах заданого радіусу.
- Якщо гравець потрапляє в поле зору, ворог переходить у стан переслідування.

ChasingState (Стан переслідування):

- Вороги слідують за гравцем, якщо він перебуває в зоні видимості, але поза зоною атаки.

- Якщо гравець потрапляє у зону стрільби, ворог переходить у стан атаки.

ShootingState (Стан стрільби):

- Вороги стріляють у гравця, використовуючи таймер для визначення інтервалу між пострілами.
- Якщо гравець виходить із зони стрільби, ворог повертається до переслідування.

Для гнучкого налаштування поведінки використовуються конфігураційні класи, які зберігають параметри для кожного стану:

- PatrollingConfig: швидкість руху та радіус патрулювання.
- ChasingConfig: швидкість переслідування.
- ShootingConfig: швидкість руху, інтервал між пострілами, ефекти стрільби.
- DispositionConfig: рівень агресивності ворога та шанс на взаємодію.

Ці параметри задаються через Unity Editor, що спрощує процес тестування і налаштування.

Завдяки модульності системи додавання нових станів чи зміна логіки існуючих не потребує значних змін у коді. Наприклад, можна додати:

- DialogueState (Стан взаємодії): для моделювання діалогів з гравцем.
- FleeingState (Стан втечі): якщо ворог отримує надто багато ушкоджень.

На рисунку 3.1 зображено діаграму станів, яка ілюструє роботу скінченного автомата, який використовується для управління поведінкою ігрового ворога. Ця діаграма відображає всі можливі стани ворога та переходи між ними, залежно від умов, що виникають у ігровому середовищі.

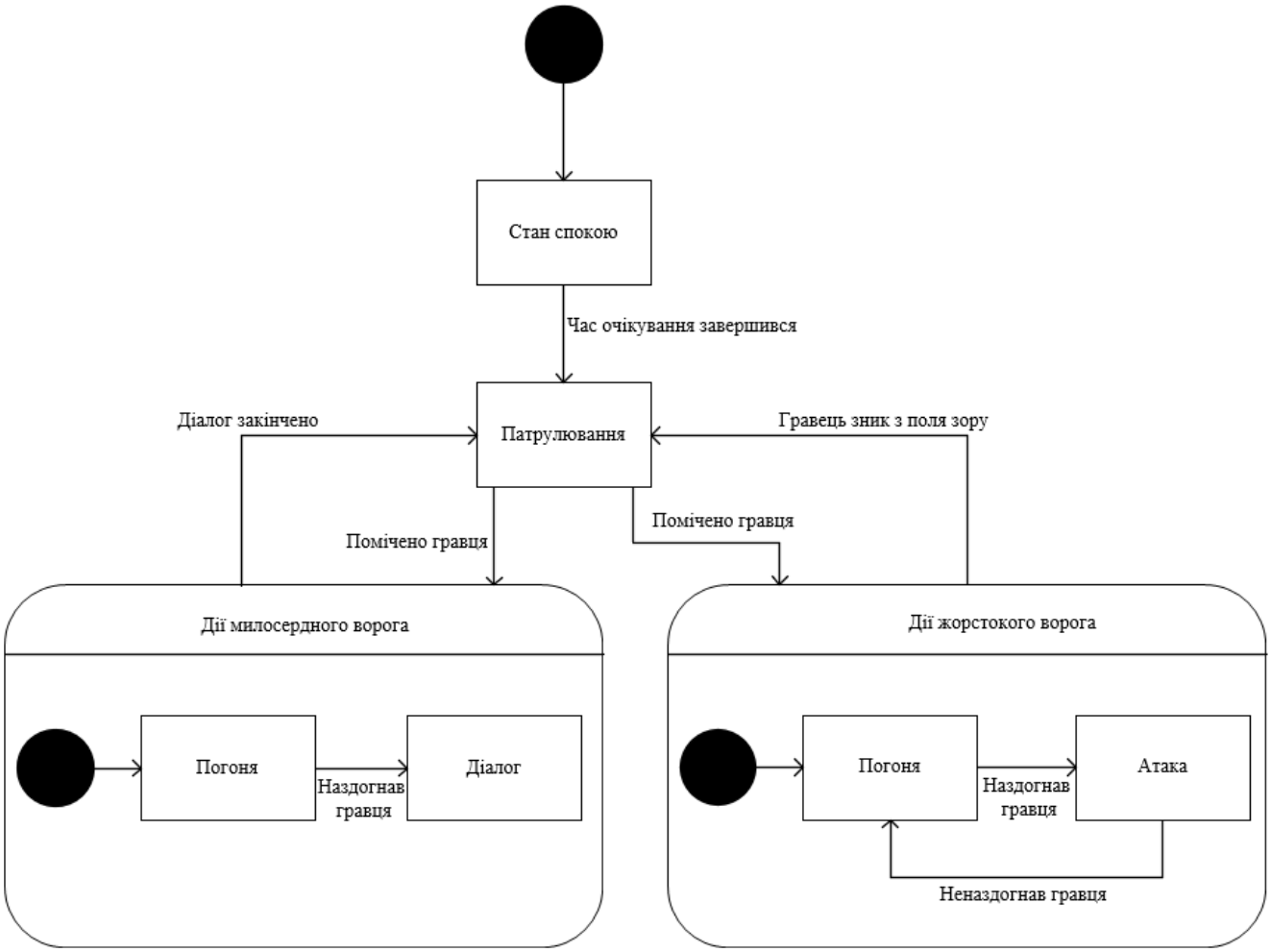


Рис 3.1 Діаграма станів

На рисунку 3.2 представлено діаграму класів, яка ілюструє архітектуру програмної реалізації системи генерації поведінки ворога в грі. Вона демонструє основні класи, їхні властивості та методи, а також зв'язки між ними

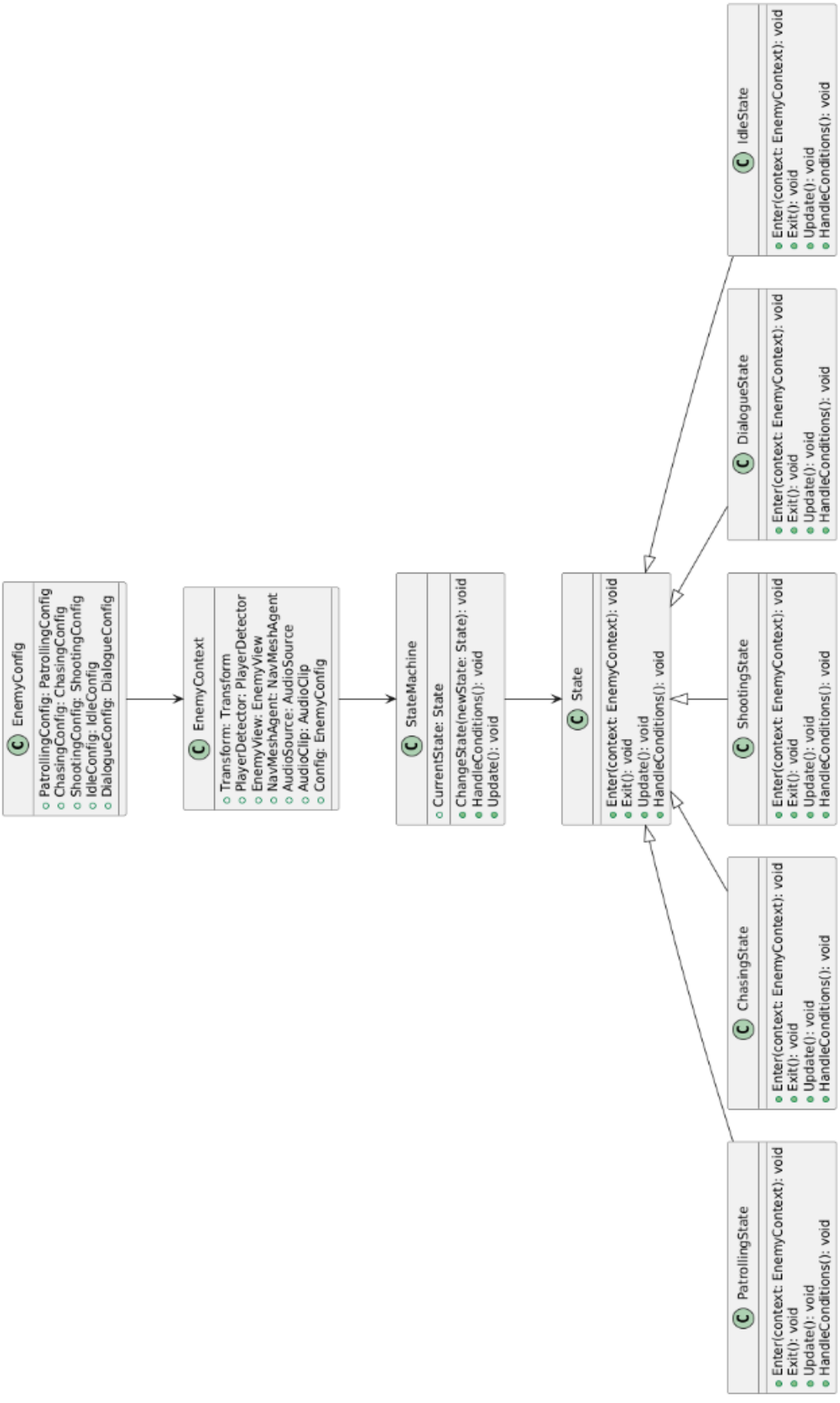


Рис 3.2 Діаграма класів

Отже, проектування системи генерації поведінки ворогів на основі скінченних автоматів забезпечує структурованість і гнучкість, дозволяє легко адаптувати систему до нових вимог і створює основу для складних сценаріїв поведінки ворогів в-іграх.

3.2 Вибір програмних засобів для реалізації системи

Для розробки системи було обрано ігровий рушій Unity. Цей вибір обумовлений кількома факторами. По-перше, Unity є надзвичайно гнучким інструментом, що дозволяє розробляти ігри різних жанрів, включаючи шутери від першої особи. По-друге, вбудована система навігації NavMesh забезпечує динамічне планування шляхів для ігрових персонажів. Крім того, підтримка мови програмування C# робить розробку поведінкових моделей більш зручною. Також варто відзначити масштабованість Unity, яка дозволяє створювати як невеликі прототипи, так і складні ігрові системи. Не останню роль відіграє велика спільнота розробників Unity, яка забезпечує доступ до великої кількості ресурсів, документації та сторонніх плагінів.

C# був обраний як основна мова програмування для розробки скриптів завдяки своїм перевагам. Об’єктно-орієнтований підхід, підтримка в середовищі Unity та простий, читабельний синтаксис роблять C# зручним інструментом для створення модульних і добре структурованих скриптів. Крім того, C# підтримує багатопотоковість, що дозволяє реалізовувати складні поведінкові моделі, такі як затримки між атаками. Висока продуктивність мови забезпечує ефективну роботу скриптів в ігровому середовищі.

Щоб забезпечити різноманітну та передбачувану поведінку ворогів, було застосовано комбінацію інструментів. Скінченні автомати дозволяють точно визначити і моделювати різні стани ворога (наприклад, патрулювання, переслідування, атака), а система навігації NavMesh забезпечує плавне пересування ворогів по ігровому світу з урахуванням перешкод.

Для виявлення гравця ми використали систему детекторів, яка реалізована на основі тригерів та перевірки відстаней.

Для додання динаміки та реалістичності поведінки ворогів було використано Unity Animator для управління анімаціями, Particle System для створення візуальних ефектів та AudioSource для додавання звукових компонентів.

Щоб оптимізувати роботу системи та виявити потенційні проблеми, було застосовано інструменти профілювання та відлагодження. Unity Profiler дозволяє аналізувати продуктивність різних компонентів системи та виявляти проблені місця, які можуть знижувати швидкість виконання. Для відлагодження логіки та перевірки коректності переходів між станами ми використовуємо функцію Debug.Log.

3.3 Опис структури та основних компонентів системи

Для створення системи генерації поведінки ворогів на основі скінчених автоматів у FPS-грі необхідно реалізувати чітку та модульну архітектуру. Такий підхід забезпечує зручність у розробці, тестуванні та майбутньому розширенні системи. Кожен модуль має виконувати конкретну функцію і взаємодіяти з іншими компонентами через стандартизовані інтерфейси.

У таблиці 3.1 описані основні модулі системи, їх призначення, функціонал і способи взаємодії.

Таблиця 3.1

Таблиця компонентів системи

Назва модуля	**Призначення**	**Функціонал**	**Взаємодія**
Контекст ворога	Зберігання даних про поточний стан ворога	Зберігає активний стан, посилання на скінченний автомат, обробляє зміну станів	Використовується скінченним автоматом для визначення і зміни стану
Скінченний автомат	Управління логікою поведінки ворога	Виконує переходи між станами на основі умов, визначених у конфігурації	Взаємодіє з модулями станів і контексту ворога
Модулі станів	Реалізація окремих дій ворога	Реалізує логіку станів: патрулювання, переслідування, атаки, діалогу	Взаємодіє з контекстом ворога для доступу до даних і змін станів
Модуль виявлення гравця	Визначення положення гравця	Використовує коллайдери, тригери та перевірки відстані	Передає дані про позицію гравця скінченному автомату
Модуль пересування (NavMesh)	Здійснення переміщень ворога	Використовує Unity NavMesh Agent для пошуку маршрутів і обходження перешкод	Отримує дані про цілі від стану PatrollingState чи ChasingState
Модуль конфігурації станів	Задає параметри та правила поведінки ворога	Зберігає конфігураційні дані для станів (PatrollingConfig, ChasingConfig, ShootingConfig), забезпечує доступ до них через властивості	Взаємодіє зі станами через StateMachine, використовується EnemyContext для отримання параметрів поведінки
Модуль налагодження	Тестування та моніторинг станів системи	Записує дані про переходи між станами, тестує поведінку ворога	Взаємодіє зі всіма модулями для збору даних

3.4 Опис розроблених класів

1. BaseState

Абстрактний базовий клас для всіх станів ворога. Забезпечує структуру для реалізації поведінки ворога в кожному стані.

Методи:

Enter() — викликається при вході в стан, виконує підготовчі дії.

Exit() — викликається при виході зі стану, очищує стан.

HandleConditions() — перевіряє умови для переходу в інший стан.

Update() — оновлює логіку стану кожен кадр.

2. PatrollingState

Реалізує поведінку патрулювання ворога, встановлюючи цільові точки для переміщення в межах заданої області.

Методи:

Enter() — встановлює швидкість і анімацію для патрулювання.

Update() — викликає метод Patrolling() для руху до цільової точки.

HandleConditions() — перевіряє, чи побачив ворог гравця, і перемикається на стан переслідування.

Patrolling() — визначає нову цільову точку, якщо поточна досягнута.

SearchForWalkingPoint() — знаходить випадкову точку для патрулювання.

DefineDistanceToWalkPoint() — перевіряє, чи ворог досяг цільової точки.

3. ChasingState

Реалізує поведінку переслідування гравця, фокусуючи рух ворога на місцезнаходження гравця.

Методи:

Enter() — встановлює швидкість і анімацію для переслідування.

Update() — викликає метод ChasePlayer() для оновлення руху.

HandleConditions() — перевіряє, чи гравець досягнутий або втратився з поля зору.

ChasePlayer() — спрямовує ворога до поточного місцезнаходження гравця.

4. ShootingState

Реалізує поведінку стрільби ворога по гравцю.

Методи:

Enter() — налаштовує анімацію та ефекти стрільби.

Update() — викликає AttackPlayer() для виконання пострілів.

HandleConditions() — перевіряє, чи гравець вийшов за межі радіуса стрільби.

AttackPlayer() — створює снаряд, відтворює звук і візуальні ефекти пострілу.

ResetAttack() — скидає затримку, щоб дозволити нові постріли.

5. DialogueState

Стан для взаємодії ворога з гравцем через діалог.

Методи:

Enter() — ініціює діалог або інші взаємодії.

Update() — обробляє поточну логіку діалогу (наприклад, чекання відповіді).

HandleConditions() — перевіряє, чи завершився діалог або гравець вийшов із зони взаємодії.

6. EnemyContext

Контейнер для всіх компонентів і залежностей ворога, використовується для спрощення доступу до даних у станах.

Методи:

Awake() — ініціалізує всі залежності та створює машину станів.

Update() — передає оновлення до поточного стану через машину станів.

7. StateMachine

Контролер станів, що керує переходами між станами та викликає відповідні методи для поточного стану.

Методи:

SwitchState<T>() — змінює поточний стан на новий тип.

HandleConditions() — перевіряє умови для переходу між станами.

Update() — викликає метод Update() активного стану.

8. Конфігураційні класи

Набір класів, що налаштовують параметри для різних станів ворога.

Методи:

PatrollingConfig — встановлює швидкість і радіус патрулювання.

ChasingConfig — визначає швидкість переслідування.

ShootingConfig — налаштовує затримку між пострілами, силу снарядів і звукові ефекти.

DispositionConfig — визначає характер ворога (агресивний, доброзичливий тощо).

3.5 Оцінка продуктивності

Було проаналізовано продуктивність системи генерації поведінки ігрового ворога на основі скінченних автоматів за двома основними метриками з використанням 100 ігрових ворогів на сцені.

Час реакції переходу між станами.Час, необхідний для прийняття рішення про зміну стану, вимірювався за допомогою класу, що використав метод Time.realtimeSinceStartup. Для кожного стану було визначено час виконання методом LogDecisionTime. На рисунку 3.3 зображенно виклики в консоль і скільки часу на це знадобилось. В результаті у ворожого персонажаце зайало 0.16 мілісекунд.

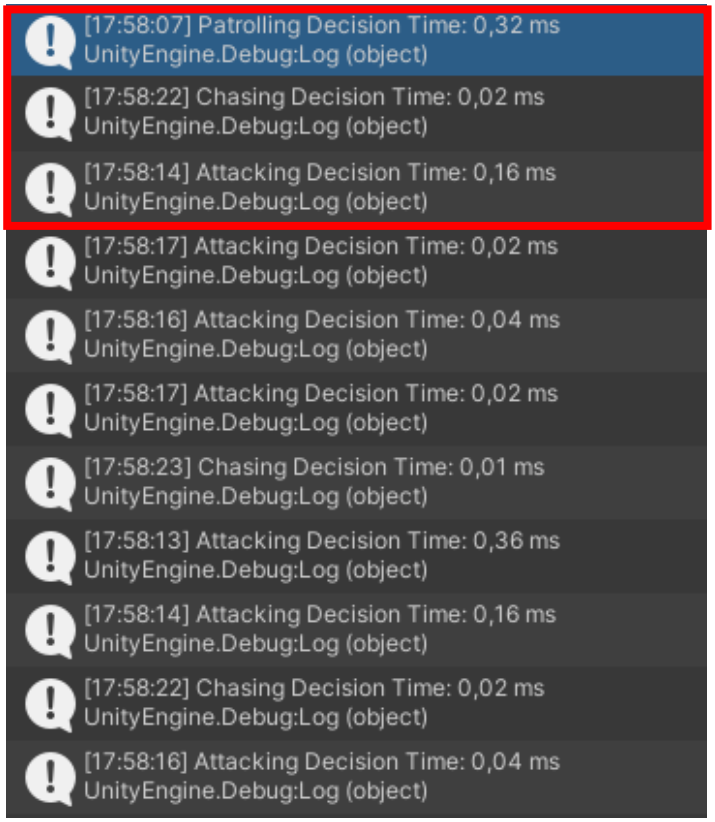


Рис.3.3 Логи часу переходу між станами

Використання пам’яті системи фіксувалось за допомогою вбудованого в Unity інструмента Profiler. Максимальне значення, яке вдалось зафіксувати – 258.1 кілобайт.

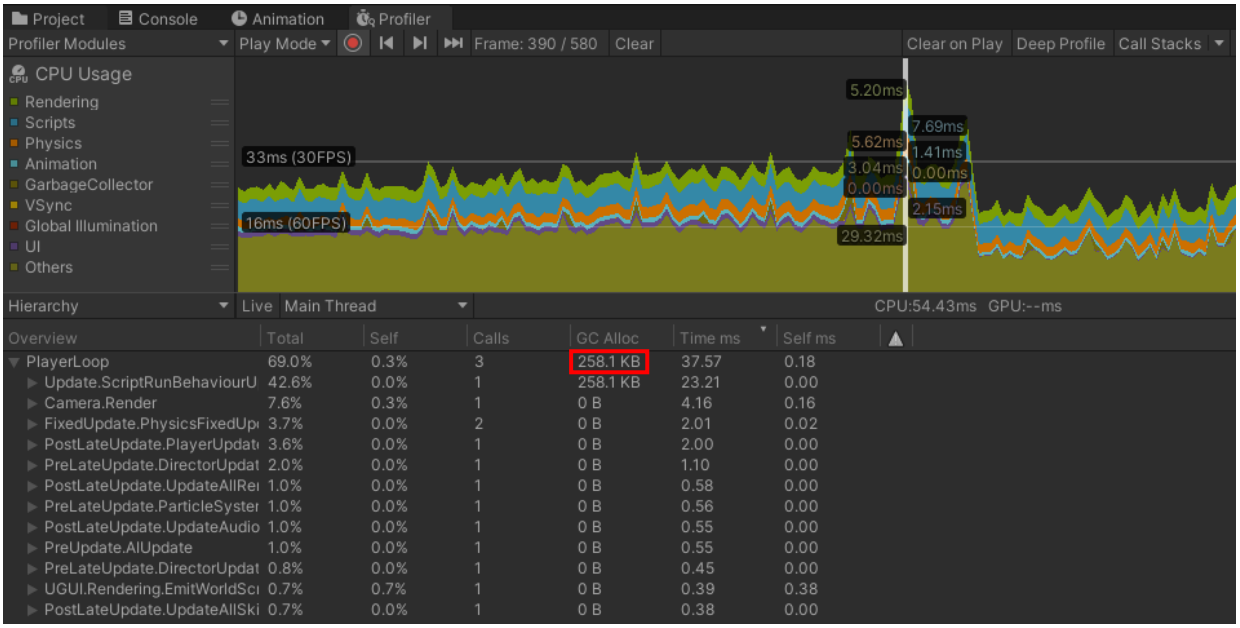


Рис.3.3 Графік використання пам’яті Unity Profiler

На рисунку 3.4 та 3.5 наглядно показано порівняння методів за метриками часу реакції та споживання пам’яті.

Рис. 3.4 Час реакції

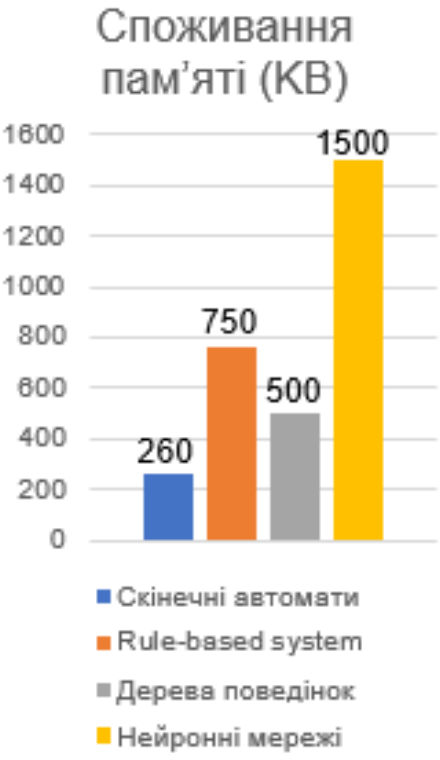


Рис. 3.5 Споживання пам’яті

Оцінка показників часу реакції:

1. Скінченні автомати забезпечують найшвидшу зміну станів — 0.16 мс, завдяки своїй простій структурі.
2. Rule-Based системи демонструють час реакції 1.5 мс через обробку складних умов.
3. Поведінкові дерева мають середній час реакції — 0.6 мс, що залежить від кількості вузлів.
4. ML-Agents є найповільнішими — 3 мс, оскільки залежать від обчислень нейронної мережі.

Оцінка показників споживання пам’яті:

1. Скінченні автомати використовують найменше ресурсів — 258 KB.
2. Rule-Based системи потребують більше пам’яті — 750 KB через велику кількість правил.
3. Поведінкові дерева споживають 500 KB, залежно від складності дерева.
4. ML-Agents мають найбільше споживання — 1536 KB через зберігання моделі нейронної мережі.

Результати демонструють, що розроблена система на основі скінченних автоматів є високопродуктивною як за часом реакції, так і за ефективністю використання пам’яті. Такі показники підтверджують доцільність використання FSM для реалізації поведінки ігрових ворогів у FPS-проектах.

ВИСНОВКИ

У ході виконання дипломної роботи було проведено комплексне дослідження та розробку системи генерації поведінки ігрового ворога для First-person shooter на основі скінчених автоматів. Основні результати роботи такі:

1. Проведено детальний аналіз існуючих моделей та методів створення поведінки неігрових персонажів, зокрема дерев прийняття рішень, нейронних мереж та скінченних автоматів. Обґрунтовано вибір скінченних автоматів як базової моделі завдяки їхній простоті, стабільності та високій продуктивності в реальному часі.
2. Запропонована система використовує скінченні автомати для моделювання поведінки ворогів, що охоплює такі стани, як патрулювання, переслідування, атака та діалог. Кожен стан реалізовано як окремий модуль із чіткими умовами переходу між ними. Це забезпечує гнучкість і зручність масштабування системи.
3. Розроблено структуру State Machine, яка включає базовий клас стану, механізм переходів і обробку контексту ворога. Система дозволяє легко розширювати набір станів і конфігурацій для адаптації до різних сценаріїв ігрового середовища.
4. Розроблено алгоритм для управління переходами між станами, який враховує ключові умови ігрового середовища, такі як положення ворога, наявність перешкод, стан гравця та дистанція до цілі. Алгоритм забезпечує швидку реакцію на події та стабільну роботу навіть у великих сценах.
5. Система була реалізована в середовищі Unity із використанням C#. Тестування проводилося на сценах із 100 ворогами, що дозволило оцінити її продуктивність у реальних умовах.
6. Проведене тестування підтвердило ефективність системи за двома основними метриками. Зокрема, час реакції переходу між станами становив у середньому 0.16 мс на одного ворога. Максимальне споживання пам'яті системою склало 258.1 кілобайт, що є значно меншим, ніж у інших моделей.

7. Скінченні автомати продемонстрували низьке споживання ресурсів та високу стабільність, забезпечивши узгоджену поведінку ворогів навіть за змінюваних умов ігрового середовища.
8. Результати тестування підтвердили, що запропонована система забезпечує високу ефективність, узгодженість дій ворогів із заданими умовами та стабільність роботи в умовах обмежених ресурсів.
9. Запропонований підхід поєднує алгоритмічну простоту та гнучкість у створенні систем поведінки ворогів, що робить його оптимальним вибором для first-person shooter. Це дозволяє не лише покращити якість ігрового досвіду, а й мінімізувати витрати ресурсів.

Результати дослідження апробовано та опубліковано у наступних тезах доповіді на конференціях:

1. Левчук М.П. Огляд методів провадження штучного інтелекту в гру на ігровому рушії Unity. // Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії". – Київ: ДУІКТ, 2024. С.194-197.
2. Левчук М.П. Роль штучного інтелекту у розвитку сучасних відеоігор. // Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії". – Київ: ДУІКТ, 2024. С.169-197.

ПЕРЕЛІК ПОСИЛАНЬ

1. Level up your code with game programming patterns. Unity, 2021, p. 1-99 .
2. DOU: Як створити цікавого ворога в грі [Електронний ресурс]. URL: https://gamedev.dou.ua/articles/creating-enemies-in-the-game/
3. Unity learn: Artifical Intelligence for Beginners [Електронний ресурс]. URL: https://learn.unity.com/course/artificial-intelligence-for-beginners
4. About ML-Agents package [Електронний ресурс]. URL: https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/manual/index.html
5. Toptal: Unity AI development. [Електронний ресурс]. URL: https://www.toptal.com/unity-unity3d/unity-ai-development-finite-state-machine-tutorial
6. Unity Learn: Finite state machines. [Електронний ресурс]. URL: https://learn.unity.com/tutorial/finite-state-machines
7. What are 2023’s top game genres? [Електронний ресурс]. URL: https://newzoo.com/resources/blog/top-game-genres-2023?utm_campaign=2023-09-GMRF-top%20video%20game%20genres&utm_content=263330567&utm_medium=social&utm_source=linkedin&hss_channel=lcp-1710460
8. The NetBSD Foundation, "Portability and supported hardware platforms," [Електронний ресурс]. URL: http://netbsd.org/about/portability.html.
9. Microsoft, Windows NT Hardware Abstraction Layer (HAL), [Електронний ресурс]. URL: http://support.microsoft.com/kb/99588.
10. A. Nareyek, N. Combs, B. Karlsson, S. Mesdaghi, and I. Wilson, "The 2003 report of the IGDA's artificial intelligence interface standards committee," Tech. Rep., International Game Developers Association, 2003 [Електронний ресурс]. URL: http://www.igda.org/ai/report- 2003/report-2003.html.
11. B. F. F. Karlsson, "Issues and approaches in artificial intelligence middleware development for digital games and entertainment products" [Електронний ресурс]. URL: CEP 50740:540, 2003.
12. B. Yue and P. de Byl, "The state of the art in game AI standard¬isation," in Proceedings of the 2006 International Conference on Game Research and Development,

pp. 41-46, Murdoch University., 2006.

13. Unity Technologies, "Unity - Game Engine," [Електронний ресурс]. URL: https://unity.com/
14. Epic Games, Unreal Engine Technology - Home, [Електронний ресурс]. URL: https://www.unrealengine.com/
15. Godot Engine [Електронний ресурс]. URL: https://godotengine.org/
16. S. Bjork, L. Sus, and H. Jussi, "Game design patterns," in Pro¬ceedings of the Level Up-lst International Digital Games Research Conference,Utrecht, TheNetherlands, November2003.
17. C. M. Olsson, S. Bjork, and S. Dahlskog, "The conceptual rela¬tionship model: understanding patterns and mechanics in game design," in Proceedings ofthe DiGRA International Conference (DiGRA 14),2014.
18. A B Loyall and J Bates, "Hap: a reactive, adaptive architecture for agents," Tech. Rep. CMU-CS-97-123, Carnegie Mellon University, School of Computer Science, 1991.
19. M. Mateas and A. Stern, "A behavior language: joint action and behavioral idioms," in Life-Like Characters, Cognitive Technolo¬gies, pp. 135-161, Springer, Berlin, Germany, 2004. International Journal of Computer Games Technology
20. J. D. Funge, "Making them behave: cognitive models for com¬puter animation," 1998.
21. J. Orkin, "Agent architecture considerations for real-time plan¬ning in games," in Proceedings of the Artificial Intelligence and Interactive Digital Entertainment (AIIDE '05), pp. 105-110,2005.
22. E. F. Anderson, "Scripting behaviour—towards a new language for making NPCs act intelligently," in Proceedings of the zfx-CON05 2nd Conference on Game Development,2005.
23. D. C. Cheng and R. Thawonmas, "Case-based plan recognition for real-time strategy games," in Proceedings ofthe 5th Inter¬national Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE 04),pp. 36-40, 2004.

24. S. Ontanon, K. Mishra, N. Sugandh, andA. Ram, "Case-based planning and execution for real-time strategy games," in Case-Based Reasoning Research and Development: 7th International Conference on Case-Based Reasoning, ICCBR 2007 Belfast, Northern Ireland, UK, August13-16,2007 Proceedings, vol. 4626, pp. 164-178, Springer, Berlin, Germany, 2007.

25. B. G. Weber and M. Mateas, "Case-based reasoning for build order in real-time strategy games," in Proceedings of the 5th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '09), pp. 106-111, October 2009.

26. Unity Technologies - Finite State Machines. [Електронний ресурс]. URL: https://learn.unity.com/project/finite-state-machines-1

27. Artificial Intelligence for Beginners, 2023. [Електронний ресурс]. URL: https://learn.unity.com/course/artificial-intelligence-for-beginners

28. Ashwin Ram, Santiago Ontaon, Manish Mehta - *Artificial Intelligence for Adaptive Computer Games, 2021.* [Електронний ресурс]. URL: https://www.researchgate.net/publication/221439041_Artificial_Intelligence_for_Adaptive_Computer_Games

29. *How do you use finite state machines in game AI programming?, 2023.* [Електронний ресурс]. URL: https://www.linkedin.com/advice/3/how-do-you-use-finite-state-machines-game-ai-programming

30. Matteo Iovino, Edvards Scukins, Jonathan Styru, Petter Ogren - *A Survey of Behavior Trees in Robotics and AI, 2020.* [Електронний ресурс]. URL: *https://www.researchgate.net/publication/341341939_A_Survey_of_Behavior_Trees_in_Robotics_and_AI*

31. State Machine for Games: Summarizing and Simplifying, 2021. [Електронний ресурс]. URL: https://medium.com/al-game-code/state-machine-for-games-summarizing-and-simplifying-d08d64705258

32. Building a rule-based system in C#, 2024. [Електронний ресурс]. URL: https://medium.com/@kohouri/learning-from-saga-frontier-building-a-rule-based-game-system-in-c-eb338bf6eb9b

33. Designing Intelligent Enemies in Games, 2024. [Електронний ресурс]. URL: https://www.toolify.ai/ai-news/designing-intelligent-enemies-in-games-a-realistic-approach-1911887

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Система генердля First-person shooter на основі скінчених автоматівації поведінки ігрового ворога»

Виконав: студент групи ПДМ-62 Левчук Микола Петрович

Керівник: к.т.н., доц., доцент кафедри ІПЗ Довженко Тимур Павлович

Київ - 2025

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищити реалістичність та ефективність взаємодії ігрових ворогів з гравцем у First-person shooter за рахунок використання системи генерації поведінки, заснованої на скінчених автоматах.

Об'єкт дослідження: процес генерації поведінки ігрових ворогів у жанрі First-person shooter.

Предмет дослідження: методи та технології генерації поведінки ігрових ворогів у жанрі First-person shooter на основі скінчених автоматів.

АКТУАЛЬНІСТЬ РОБОТИ

Розробка ефективних систем штучного інтелекту для ігор жанру First-person shooter є важливою через потребу в динамічній і передбачуваній поведінці ворогів. Традиційні методи, як Rule-based Systems і Behavior Trees, обмежені адаптивністю, а нейронні мережі (ML agents) вимагають значних ресурсів і часу на навчання.
Скінченні автомати (FSM) забезпечують високу швидкість реакції, наприклад, час прийняття рішення ~0.15 мс порівняно з ~3 мс у нейронних мереж (в Rule-Based системах та поведінкових дерев це 1,5 та 0,6 мс відповідно), витрачаючи часу приблизно у 20 разів менше. Також, в порівнянні із ML agents, скінчені автомати демонструють значно менше споживання пам’яті. Для управління 100 ігровими ворогами системі скінчених автоматів потрібно лише 258 KB пам’яті, тоді як система ML-Agents із такою ж кількістю агентів може споживати до 1536 KB (в Rule-Based системах та поведінкових дерев це 750 та 500 KB відповідно)

3

ПОРІВНЯННЯ ІСНУЮЧИХ МЕТОДІВ

Метод/модель	Опис	Ключові недоліки
1. Нейронні мережі (ML agents)	Система, що навчається на основі вхідних даних	- Висока обчислювальна складність під час навчання та прогнозування в реальному часі - Потреба у великих наборах даних для навчання - Непередбачуваність поведінки персонажа в умовах, відмінних від навчальних даних
2. Системи на основі правил (Rule-based Systems)	Набір предвизначених правил "якщо-то" для визначення поведінки	- Обмежена гнучкість через суворі прив’язки до умов - Складність розширювання зі збільшенням гри, що ускладнює керування системою, робить її важчою для читання, налагодження та оновлення
3. Дерева поведінки (Behavior Trees)	Ієрархічна структура вузлів, що визначає послідовність дій та прийняття рішень ігрових персонажів	- Важко відлагоджувати великі дерева - Високі витрати пам'яті при створенні великих дерев з багатьма вузлами
4. Скінченні автомати (FSM)	Система станів та переходів між ними на основі умов. Кожен стан представляє конкретну поведінку (атака, патрулювання, відступ тощо)	- стає важко керованою, якщо кількість станів перевищує десятки або сотні.

4

МАТЕМАТИЧНА МОДЕЛЬ СКІНЧЕНОГО АВТОМАТУ

Формула для визначення ймовірності переходу до кожного стану залежно від поточного стану

$$P(s_j|s_i, x) = \frac{w_{ij} \cdot f(x)}{\sum_k w_{ik} \cdot f(x)}$$

w_{ij}— вага переходу зі стану s_i до s_j
f(x) — функція умовних факторів (відстань, доброзичливість, шанс)

Функція умовних операторів

$$f(x) = f_d(d) \cdot f_\alpha(\alpha) \cdot f_p(p_{\text{interaction}})$$

$$f_d(d) = \begin{cases} 1, & \text{якщо } d \le d_{\text{shoot}}, \\ \frac{d_{\text{chase}} - d}{d_{\text{chase}} - d_{\text{shoot}}}, & \text{якщо } d_{\text{shoot}} < d \le d_{\text{chase}}, \\ 0, & \text{якщо } d > d_{\text{chase}}. \end{cases}$$

$$f_\alpha(\alpha) = \begin{cases} 1 - \frac{\alpha}{100}, & \text{якщо } \alpha \ge 0, \\ 1 + \frac{\alpha}{100}, & \text{якщо } \alpha < 0. \end{cases}$$

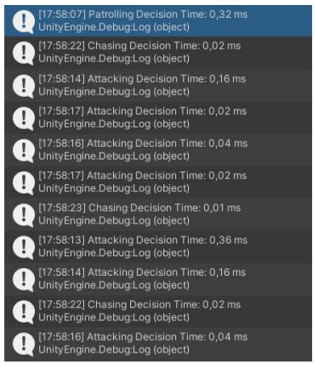
$$f_p(p_{\text{interaction}}) = p_{\text{interaction}}$$

S={S_{idle} , $S_{patroling}$, $S_{chasing}$, S_{attack} , $S_{dialogue}$ } — множина станів.
d — відстань до гравця, dshoot, dchase — порогові значення для стрільби та переслідування.
α∈[−100,100]— рівень доброзичливості ворога.
pinteraction — шанс на взаємодію (0≤ pinteraction ≤1).
δ(s,x)— функція переходу між станами, де x — набір вхідних параметрів (наприклад d,α,pinteraction).

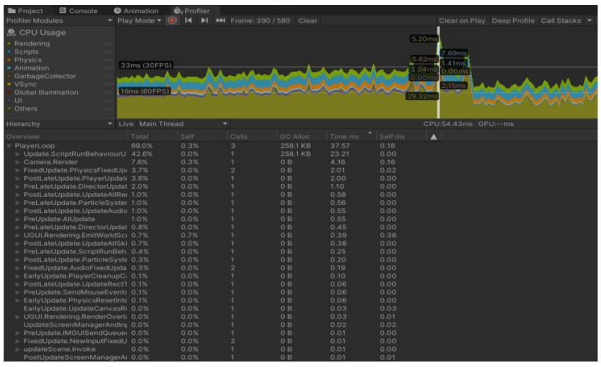
5

МЕТРИКИ ОЦІНЮВАННЯ

DecisionTimeLogger

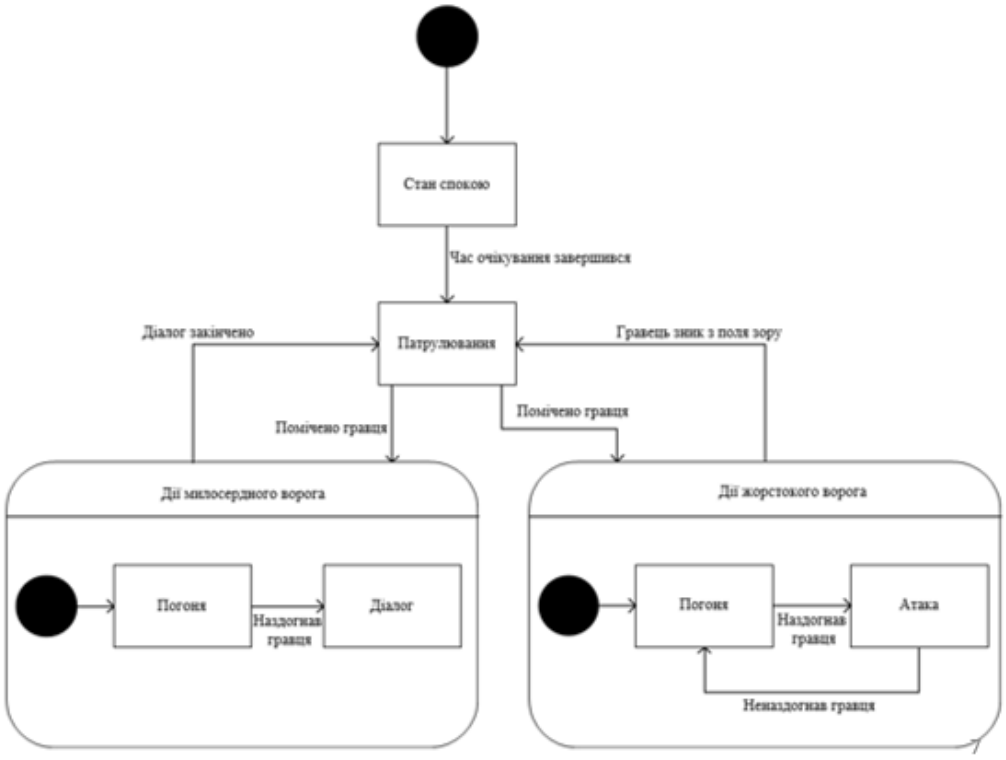


Unity profiler

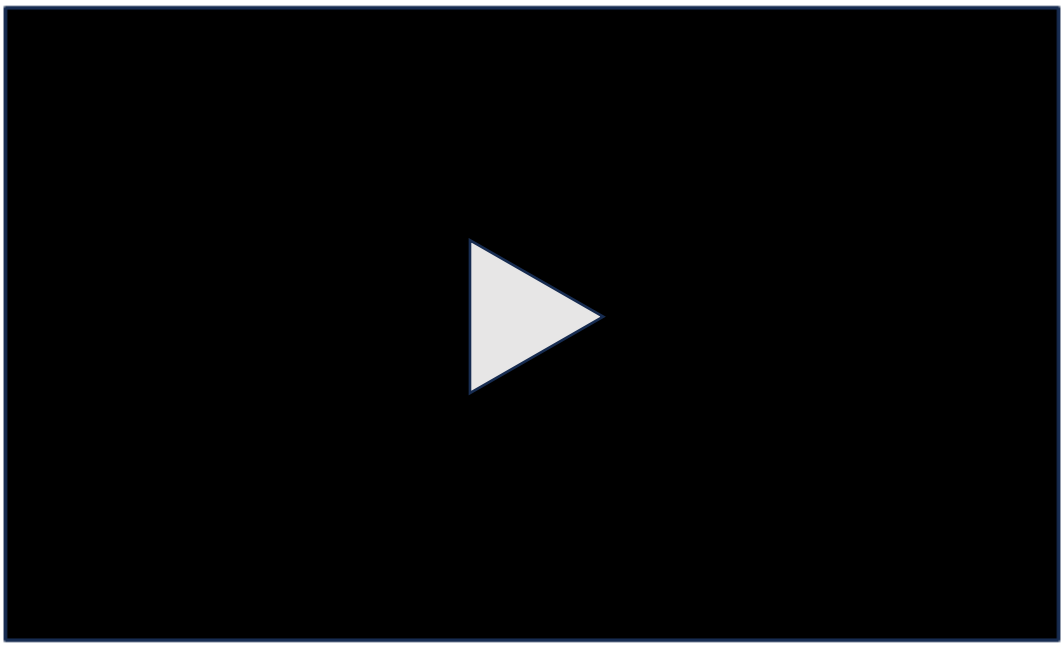


6

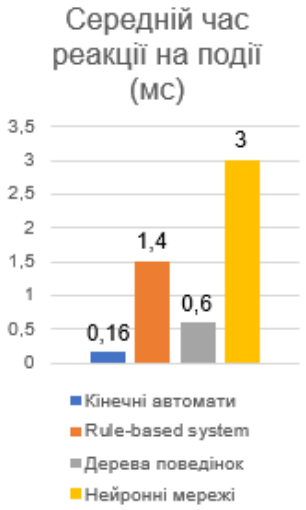
UML діаграма скінчених станів



ЕКРАННІ ФОРМИ



ПОРІВНЯЛЬНИЙ АНАЛІЗ



9

ВИСНОВКИ

1. Проведено аналіз існуючих підходів до генерації поведінки NPC у FPS-іграх, таких як скінчені автомати, дерева рішень та нейронні мережі. Обґрунтовано вибір FSM як основи для реалізації системи.
2. Розроблено структуру State Machine, яка включає базовий клас стану, механізм переходів і обробку контексту ворога. Система дозволяє легко розширювати набір станів і конфігурацій для адаптації до різних сценаріїв ігрового середовища.
3. Система була реалізована в середовищі Unity із використанням C#. Тестування проводилося на сценах із 100 ворогами, що дозволило оцінити її продуктивність у реальних умовах.
4. Проведене тестування підтвердило ефективність системи за двома основними метриками. Зокрема, час реакції переходу між станами становив у середньому 0.16 мс на одного ворога. Максимальне споживання пам'яті системою склало 258.1 кілобайт, що є значно меншим, ніж у інших моделей.

10

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Левчук М.П. Огляд методів провадження штучного інтелекту в гру на ігровому рушії Unity. // Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії". – Київ: ДУТ, 2024. *Робота подана до друку*
2. Левчук М.П. Роль штучного інтелекту у розвитку сучасних відеоігор. // Всеукраїнська науково-технічна конференція "Виклики та рішення в програмній інженерії". – Київ: ДУТ, 2024. *Робота подана до друку*

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

TransitionEvaluator

```
using System;
using System.Collections.Generic;
using UnityEngine;

public static class TransitionEvaluator
{
   public static string GetNextState(string currentState, Dictionary<string, float> externalFactors, Dictionary<string, Dictionary<string, float>> transitionProbabilities)
   {
      if (!transitionProbabilities.ContainsKey(currentState))
      {
         Debug.LogWarning($"Transition probabilities for state {currentState} not found.");
         return currentState;
      }

      var stateProbabilities = transitionProbabilities[currentState];
      string nextState = null;
      float maxProbability = float.MinValue;

      foreach (var state in stateProbabilities)
      {
         float adjustedProbability = state.Value;

         if (externalFactors != null && externalFactors.TryGetValue(state.Key, out float factor))
         {
            adjustedProbability *= factor;
         }

         if (adjustedProbability > maxProbability)
         {
            maxProbability = adjustedProbability;
            nextState = state.Key;
         }
      }

      Debug.Log($"Transition from {currentState} to {nextState} with probability {maxProbability}");
      return nextState ?? currentState;
   }
}
```

BaseState

```
using UnityEngine;
using System.Collections.Generic;

public abstract class BaseState : IState
{
   protected IStateSwitcher StateSwitcher;
   protected StateMachineData Data;
   protected EnemyContext _enemyContext;

   public BaseState(IStateSwitcher switcher, StateMachineData data, EnemyContext enemyContext)
   {
      StateSwitcher = switcher;
      Data = data;
      _enemyContext = enemyContext;
   }

   protected Transform Transform => _enemyContext.Transform;
```

```
    protected NavMeshAgent Agent => _enemyContext.Agent;
    protected PlayerDetector PlayerDetector => _enemyContext.PlayerDetector;
    protected bool Grounded => _enemyContext.Grounded;
    protected EnemyView View => _enemyContext.View;

    public virtual void Enter() { }
    public virtual void Exit() { }
    public virtual void HandleConditions() { }
    public virtual void Update() { }

    // Виклик математичної моделі
    protected void TransitionBasedOnModel()
    {
        string nextState = TransitionEvaluator.GetNextState(
            GetType().Name,
            _enemyContext.ExternalFactors,
            _enemyContext.TransitionProbabilities
        );

        // Перемикаємо стан, якщо він відрізняється від поточного
        if (!string.IsNullOrEmpty(nextState) && nextState != GetType().Name)
        {
            StateSwitcher.SwitchState(GetStateTypeByName(nextState));
        }
    }

    private System.Type GetStateTypeByName(string stateName)
    {
        return stateName switch
        {
            "IdleState" => typeof(IdleState),
            "PatrollingState" => typeof(PatrollingState),
            "ChasingState" => typeof(ChasingState),
            "DialogueState" => typeof(DialogueState),
            "ShootingState" => typeof(ShootingState)
        };
    }
}
```

PatrollingState

```
public class PatrollingState : BaseState
{
    private PatrollingConfig _config;

    private Vector3 _walkPoint;
    private bool _isWalkPointSet;

    public PatrollingState(IStateSwitcher switcher, StateMachineData data, EnemyContext enemyContext) : base(switcher,
data, enemyContext)
    {
        _config = enemyContext.Config.PatrollingConfig;
    }

    private float _speed => _config.Speed;
    private float _walkPointRange => _config.WalkPointRange;

    public override void Enter()
    {
        base.Enter();
        Data.Speed = _speed;
        Agent.speed = Data.Speed;
        View.SetPatrollingAnimation();
```

```
    }

    public override void Exit()
    {
        base.Exit();
        View.UnsetPatrollingAnimation();
    }

    public override void HandleConditions()
    {
        base.HandleConditions();
        TransitionBasedOnModel(); // Використання математичної моделі
    }

    public override void Update()
    {
        base.Update();
        Patrolling();
    }

    private void Patrolling()
    {
        if (!_isWalkPointSet)
            SearchForWalkingPoint();

        if (_isWalkPointSet)
            Agent.SetDestination(_walkPoint);

        if (DefineDistanceToWalkPoint(_walkPoint))
            _isWalkPointSet = false;
    }

    private bool DefineDistanceToWalkPoint(Vector3 walkPoint)
    {
        float distance = Vector3.Distance(Transform.position, walkPoint);
        bool arrivedAtTheDestinationPoint = distance < 0.1f;
        return arrivedAtTheDestinationPoint;
    }

    private void SearchForWalkingPoint()
    {
        float randXPos = Random.Range(-_walkPointRange, _walkPointRange);
        float randZPos = Random.Range(-_walkPointRange, _walkPointRange);

        _walkPoint = new Vector3(randXPos, Transform.position.y, randZPos);

        if (Grounded)
            _isWalkPointSet = true;
    }
}
```

ChasingState

```
using Assets.Scripts;

public class ChasingState : BaseState
{
    private ChasingConfig _config;
    private StateTransitionHelper _transitionHelper;

    public ChasingState(IStateSwitcher switcher, StateMachineData data, EnemyContext enemyContext,
StateTransitionHelper transitionHelper)
```

```
        : base(switcher, data, enemyContext)
    {
        _config = enemyContext.Config.ChasingConfig;
        _transitionHelper = transitionHelper;
    }

    private float _speed => _config.Speed;

    public override void Enter()
    {
        base.Enter();
        Data.Speed = _speed;
        Agent.speed = Data.Speed;
        View.SetChasingState();
    }

    public override void Exit()
    {
        base.Exit();
        View.UnsetChasingState();
    }

    public override void HandleConditions()
    {
        base.HandleConditions();

        if (_transitionHelper.ShouldTransitionToShooting(PlayerDetector))
        {
            StateSwitcher.SwitchState<ShootingState>();
        }
        else if (_transitionHelper.ShouldTransitionToPatrolling(PlayerDetector))
        {
            StateSwitcher.SwitchState<PatrollingState>();
        }
    }

    public override void Update()
    {
        base.Update();
        ChasePlayer();
    }

    private void ChasePlayer()
    {
        Agent.SetDestination(PlayerDetector.Player.transform.position);
    }
}
```

ShootingState

```
using Assets.Scripts;
using System.Threading.Tasks;
using UnityEngine;

public class ShootingState : BaseState
{
    private ShootingConfig _config;
    private StateTransitionHelper _transitionHelper;
    private Transform _muzzle;
    private AudioSource _audioSource;
    private bool _alreadyAttacked;
```

```
    public ShootingState(IStateSwitcher switcher, StateMachineData data, EnemyContext enemyContext,
StateTransitionHelper transitionHelper)
        : base(switcher, data, enemyContext)
    {
        _config = enemyContext.Config.ShootingConfig;
        _audioSource = enemyContext.AudioSource;
        _muzzle = enemyContext.Muzzle;
        _transitionHelper = transitionHelper;
    }

    private float _speed => _config.Speed;
    private AudioClip _shootingSound => _config.ShootingSound;
    private ParticleSystem _shootingEffect => _config.ShootingEffect;
    private float _timeBetweenShoots => _config.TimeBetweenShoots;

    public override void Enter()
    {
        base.Enter();
        Data.Speed = _speed;
        Agent.speed = Data.Speed;
        View.SetShootingState();
    }

    public override void Exit()
    {
        base.Exit();
        View.UnsetShootingState();
    }

    public override void HandleConditions()
    {
        base.HandleConditions();

        if (_transitionHelper.ShouldTransitionToChasing(PlayerDetector))
        {
            StateSwitcher.SwitchState<ChasingState>();
        }
    }

    public override void Update()
    {
        base.Update();
        AttackPlayer();
    }

    private async void AttackPlayer()
    {
        Transform.LookAt(PlayerDetector.Player.transform);

        if (!_alreadyAttacked)
        {
            _audioSource.PlayOneShot(_shootingSound);
            ParticleSystem shootEffect = ParticleSystem.Instantiate(_shootingEffect, _muzzle.position, _muzzle.rotation);
            EnemysProjectile projectile = ObjectPool.instance.GetBullet();
            projectile.transform.position = _muzzle.position;
            projectile.ProjcetileFly(_muzzle.forward);

            _alreadyAttacked = true;

            await Task.Delay((int)_timeBetweenShoots * 1000);
            ResetAttack();
        }
```

```
    }

    private void ResetAttack()
    {
        _alreadyAttacked = false;
    }
}
```

DialogueState

```
public class DialogueState : BaseState
{
    private StateTransitionHelper _transitionHelper;

    public DialogueState(IStateSwitcher switcher, StateMachineData data, EnemyContext enemyContext,
StateTransitionHelper transitionHelper)
        : base(switcher, data, enemyContext)
    {
        _transitionHelper = transitionHelper;
    }

    public override void Enter()
    {
        base.Enter();
        View.SetDialogueState(); // Установка анімації або іншої візуалізації для діалогу
        Agent.isStopped = true;  // Зупинка ворога під час діалогу
    }

    public override void Exit()
    {
        base.Exit();
        View.UnsetDialogueState(); // Зняття анімації діалогу
        Agent.isStopped = false;  // Дозволити ворогу рухатись після виходу зі стану
    }

    public override void HandleConditions()
    {
        base.HandleConditions();

        // Якщо ворог більше не взаємодіє з гравцем, повертаємося до патрулювання
        if (_transitionHelper.ShouldTransitionToPatrolling(PlayerDetector))
        {
            StateSwitcher.SwitchState<PatrollingState>();
        }
        // Можна додати інші умови переходу
    }

    public override void Update()
    {
        base.Update();
    }
}
```

StateMachine

```
public class StateMachine : IStateSwitcher
{
    private IState _currentState;
    private List<IState> _states;
    private StateTransitionHelper _transitionHelper;

    public StateMachine(EnemyContext enemyContext)
    {
        StateMachineData data = new StateMachineData();
```

```
        _transitionHelper = new StateTransitionHelper();

        _states = new List<IState>
        {
            new PatrollingState(this, data, enemyContext, _transitionHelper),
            new ChasingState(this, data, enemyContext, _transitionHelper),
            new DialogueState(this, data, enemyContext, _transitionHelper),
            new ShootingState(this, data, enemyContext, _transitionHelper)
        };

        _currentState = _states[0];
        _currentState.Enter();
    }

    public void SwitchState<T>() where T : IState
    {
        IState newState = _states.FirstOrDefault(state => state is T);

        _currentState.Exit();
        _currentState = newState;
        _currentState.Enter();
    }

    public void HandleConditions() => _currentState.HandleConditions();
    public void Update() => _currentState.Update();
}
```

EnemyContext

```
using System.Collections.Generic;
using UnityEngine;

public class EnemyContext : MonoBehaviour
{
    [SerializeField] private EnemyConfig _config;
    [SerializeField] private EnemyView _view;
    [SerializeField] private NavMeshAgent _agent;

    [SerializeField] private PlayerDetector _playerDetector;
    [SerializeField] private GroundDetector _groundDetector;

    [SerializeField] private AudioSource _audioSource;
    [SerializeField] private AudioClip _shotSound;

    [SerializeField] private Transform _muzzle;

    private StateMachine _stateMachine;

    public bool Grounded => _groundDetector.Grounded;
    public Transform Transform => transform;
    public PlayerDetector PlayerDetector => _playerDetector;
    public EnemyView View => _view;
    public NavMeshAgent Agent => _agent;
    public EnemyConfig Config => _config;
    public AudioSource AudioSource => _audioSource;
    public AudioClip ShotSound => _shotSound;
    public Transform Muzzle => _muzzle;

    // Дані для математичної моделі
    public Dictionary<string, Dictionary<string, float>> TransitionProbabilities { get; private set; }
    public Dictionary<string, float> ExternalFactors { get; private set; }
```

```
    private void Awake()
    {
        InitializeTransitionData();
        _stateMachine = new StateMachine(this);
    }

    private void InitializeTransitionData()
    {
        TransitionProbabilities = new Dictionary<string, Dictionary<string, float>>
        {
            { "IdleState", new Dictionary<string, float> { { "PatrollingState", 0.6f }, { "DialogueState", 0.2f }, { "ChasingState",
0.2f } } },
            { "PatrollingState", new Dictionary<string, float> { { "IdleState", 0.4f }, { "ChasingState", 0.4f }, { "DialogueState",
0.2f } } },
            { "ChasingState", new Dictionary<string, float> { { "ShootingState", 0.7f }, { "PatrollingState", 0.3f } } },
            { "DialogueState", new Dictionary<string, float> { { "IdleState", 0.5f }, { "PatrollingState", 0.5f } } },
            { "ShootingState", new Dictionary<string, float> { { "IdleState", 0.3f }, { "ChasingState", 0.7f } } },
        };

        ExternalFactors = new Dictionary<string, float>
        {
            { "ChasingState", 1.5f },
            { "DialogueState", 0.8f },
            { "ShootingState", 1.2f },
        };
    }

    private void Update()
    {
        _stateMachine.HandleConditions();
        _stateMachine.Update();
    }
}
```