

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

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

**«ВИКОРИСТАННЯ ТЕХНОЛОГІЙ ШТУЧНОГО ІНТЕЛЕКТУ В
IDLE/TYCOON ІГРАХ»**

Виконав: здобувач 2 курсу

Рівень магістр

Галузь знань 12 «Інформаційні технології»,
спеціальність 122 «Комп'ютерні науки»

Чепендюк Олександр Олександрович

Керівник к.т.н., доцент

Манаков Сергій Юрійович

Рецензент к.пед.н., доцент

Логінова Наталія Іванівна

Одеса – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
Факультет **кібербезпеки та інформаційних технологій**
Кафедра **інформаційних технологій**
Галузь знань: **12 Інформаційні технології**
Спеціальність: **122 Комп'ютерні науки**
Назва освітньої програми: **Комп'ютерні науки**

ЗАТВЕРДЖУЮ

Завідувач кафедри інформаційних технологій
доцент Н.І. Логінова _____

« ____ » _____ 20__ року

З А В Д А Н Н Я

на кваліфікаційну (бакалаврську) роботу
здобувачу **Чепендюку Олександр Олександровичу**

- Тема роботи: «Використання технологій штучного інтелекту в idle/tycoon іграх»
Керівник роботи: к.т.н, доцент Манаков Сергій Юрійович
затверджені наказом Ректора НУ «ОЮА» від «10» жовтня 2025 року № 3182-06.
- Строк подання здобувачем роботи «25» листопада 2025 року
- Дата видачі завдання «02» червня 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Вибір теми, вивчення літературних джерел та складання плану роботи	15.09.2025	
2	Підготовка першого розділу роботи та подання його керівнику	01.10.2025	
3	Підготовка другого розділу роботи та подання його керівнику	14.10.2025	
4	Підготовка третього розділу роботи та подання його керівнику	07.11.2025	
5	Підготовка вступу та висновків	10.11.2025	
6	Доопрацювання роботи з урахуванням зауважень керівника	24.11.2025	
7	Подача електронного варіанту роботи на кафедру	25.11.2025	

Здобувач _____ Олександр ЧЕПЕНДЮК
(підпис) (прізвище та ініціали)

Керівник роботи _____ Сергій МАНАКОВ
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Чепендюк О.О. “Використання технологій штучного інтелекту в idle/tycoon іграх”. – Кваліфікаційна магістерська робота за спеціальністю 122 «Комп’ютерні науки», освітньою програмою «Комп’ютерні науки».

Робота викладена на 100 сторінках та складається з 83 сторінок основного тексту, містить 3 розділи, 9 рисунків, 4 таблиці, 47 використаних джерела, 2 додатки.

Метою роботи є розробка та дослідження гібридної архітектури для керування поведінкою NPC, яка поєднує детерміновану FSM-логіку із генеративними моделями LLM з метою підвищення імерсивності взаємодії й збереження стабільності економіки ігрового симулятора.

Об’єктом дослідження є процеси автоматизованого управління NPC у цифрових симуляторах економіки та офісу.

Предметом дослідження є архітектурні й практичні засоби інтеграції FSM, LLM та асинхронних механізмів у NPC-системах для idle/tycoon-ігор.

У роботі запропоновано трирівневу архітектуру (стратегічний, тактичний, комунікативний рівні), впроваджено асинхронну обробку LLM-запитів, розроблено методику Contextual Prompt Engineering для персоналізації NPC, реалізовано експериментальне тестування на прототипі в Unity. Наведено UML-діаграми, функціональні блок-схеми, результати порівняльного тестування.

Практичне значення роботи полягає у можливості застосування гібридної концепції для підвищення імерсивності, економічної стабільності й варіативності NPC у сучасних ігрових проектах.

Ключові слова: штучний інтелект, NPC, FSM, LLM, Unity, prompt engineering, імерсивність, idle/tycoon.

ABSTRACT

Chependiuk O. O. The use of Artificial Intelligence technologies in idle/tycoon games. – Master's Qualification Thesis in specialty 122 "Computer Science," educational program "Computer Science."

The thesis consists of 100 total pages with 83 pages of main text, including 3 sections, 9 figures, 4 tables, 47 references, and 2 appendices.

The goal of the work is to develop and investigate a hybrid architecture for managing NPC behavior, which combines deterministic FSM logic with generative LLM models to enhance interaction immersiveness and maintain the stability of the game simulator's economy.

The object of the research is the processes of automated NPC control in digital economic and office simulators.

The subject of the research is the architectural and practical means of integrating FSM, LLM, and asynchronous mechanisms into NPC systems for idle/tycoon games.

The work proposes a three-level architecture (strategic, tactical, and communicative levels), implements asynchronous processing of LLM requests, develops a Contextual Prompt Engineering methodology for NPC personalization, and provides experimental testing on a Unity prototype. UML diagrams, functional block diagrams, and comparative testing results are presented.

The practical value of the work lies in the possibility of applying the hybrid concept to increase the immersiveness, economic stability, and variability of NPCs in modern game projects.

Keywords: Artificial Intelligence, NPC, FSM, LLM, Unity, prompt engineering, immersiveness, idle/tycoon.

ЗМІСТ

ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ ТЕХНОЛОГІЙ ШТУЧНОГО ІНТЕЛЕКТУ В СУЧАСНІЙ РОЗРОБЦІ ІГОР	14
1.1 Роль та класифікація систем штучного інтелекту в індустрії розваг.....	14
1.2 Характеристика та специфіка жанру Idle/Tycoon як об'єкта автоматизації.....	17
1.2.1 Визначення та економічна роль NPC у жанрі	18
1.2.2 Унікальні вимоги до детермінованості та масштабованості	19
1.2.3 Проблема "Імерсивного розриву" (Immersion Gap) та соціальний аспект.....	20
1.3 Дихотомія інструментарію ШІ: Editor AI проти Runtime AI.....	21
1.4 Огляд та порівняльний аналіз сучасних підходів до моделювання поведінки NPC	23
1.4.1 Скінченні автомати.....	24
1.4.2 Древа поведінки.....	26
1.4.3 Планування, орієнтоване на цілі (GOAP).....	27
1.4.4 Системи корисності (Utility AI).....	28
1.4.5 Просторовий ШІ та системи навігації	29
1.4.6 Вибір методу.....	30
1.5 Аналіз генеративних методів ШІ та проблеми інтеграції.....	33
1.5.1 Потенціал генеративних методів та концепція Generative Agents.....	34

1.5.2 Проблеми та економічні обмеження прямої інтеграції LLM.....	30
1.6 Гібридна архітектура ШІ.....	38
Висновки до розділу 1	40
РОЗДІЛ 2. ПРОЄКТУВАННЯ УДОСКОНАЛЕНОЇ ГІБРИДНОЇ СИСТЕМИ ШІ ДЛЯ УПРАВЛІННЯ ПОВЕДІНКОЮ NPC	36
2.1 Концептуальна архітектура, логічне розмежування та FSM- модельовання.....	43
2.2 Проєктування модуля генеративної комунікації на основі LLM та імерсивних механізмів.....	50
2.3 Модельовання емерджентної поведінки та архітектура LLM-проксі	41
2.4 Принципи контекстної генерації промтів та персоніфікації NPC	56
Висновки до розділу 2	58
РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ГІБРИДНОЇ СИСТЕМИ	59
3.1 Загальний опис інтерфейсу, архітектури та геймплейного циклу системи	59
3.2 Програмні рішення для ігрового середовища та UX.....	65
3.3 Імплементація LLM-звітності та механізмів тестування.....	69
3.4 Тестування та аналіз результатів.....	61
ВИСНОВКИ.....	76
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	79
ДОДАТКИ.....	69
Додаток А. Програмний код	69
Додаток Б. Копії матеріалів апробації	99

ВСТУП

Актуальність дослідження у межах даної магістерської роботи полягає у критичній необхідності вирішення комплексної науково-практичної проблеми, що поставала перед сучасною індустрією мобільних ігор. Сучасна індустрія мобільних ігор демонструє стрімке зростання, перетворюючись на домінуючий сектор глобального ринку цифрових розваг, де жанр Idle/Tycoon (економічні симулятори з елементами пасивного доходу) займає одну з провідних позицій завдяки високим показникам утримання гравців (Retention Rate) та монетизації. Проте, зі зростанням конкуренції зростають і вимоги користувачів до якості ігрового досвіду, що змушує розробників шукати нові способи залучення та утримання аудиторії.

Традиційні підходи до моделювання поведінки неігрових персонажів (NPC), такі як скінченні автомати (FSM) або дерева поведінки (BT), ефективно забезпечують стабільність економічної симуляції та гарантують передбачувану роботу ігрових систем. Однак ці методи часто призводять до створення «роботизованої», одноманітної поведінки персонажів, яка сприймається гравцем як штучна та нежива-природна. Це створює так званий «імерсивний розрив», коли гравець втрачає відчуття присутності у живому, динамічному світі через примітивність соціальних реакцій персонажів та їхню неспроможність адаптуватися до унікальних ситуацій. Персонажі залишаються сприйнятими не як розумні агенти, а як детерміновані автомати, що слідують заздалегідь записаним сценаріям [5].

З іншого боку, бурхливий розвиток генеративного штучного інтелекту (Generative AI) та великих мовних моделей (LLM) відкриває абсолютно нові горизонти для створення реалістичної, природної комунікації та емерджентної поведінки персонажів. Сучасні LLM здатні генерувати унікальні, контекстно-залежні репліки, що забезпечують глибину та варіативність взаємодії, яких не можна досягти традиційними методами. Однак практична інтеграція хмарних LLM у мобільні та комп'ютерні ігри наразі стикається з низкою критичних перешкод, які роблять таку інтеграцію практично неможливою без кардинального переосмислення архітектури. Передусім це висока затримка (latency) від 0.5 до 3 секунд, яка руйнує

темпоральну цілісність ігрового досвіду, оскільки гравець очікує миттєвої реакції від персонажа. Крім того, витрати на API-запити до комерційних LLM-сервісів є суттєвими і масштабуються пропорційно кількості персонажів та частотою їхніх запитів, що робить такий підхід економічно невиправданим для більшості проєктів. І нарешті, існує глибокий ризик того, що генеративна модель може вплинути на ігровий баланс через непередбачувані, «галюцинаторні» відповіді, які суперечать логіці економічної симуляції.

Проведений аналіз стану проблемної області дозволяє сформулювати центральну наукову проблему даного дослідження. Існуючі методи моделювання поведінки NPC у відеоіграх характеризуються фундаментальною дихотомією, яка робить неможливим одночасне досягнення всіх критичних вимог сучасних економічних симуляторів.

Детерміновані методи (FSM, Behavior Tree) забезпечують повну передбачуваність поведінки та мінімальне навантаження на обчислювальні ресурси ($O(1)$ для FSM, $O(n)$ для BT). Однак ці методи створюють штучну, роботизовану поведінку з обмеженим набором реакцій, що призводить до низької соціальної реалістичності та швидкої втрати залученості гравця. Персонажі залишаються сприйнятими як автомати, що реагують на фіксовану кількість умов, без здатності до адаптації та вираження індивідуальності.

Генеративні методи (Large Language Models, Utility AI з машинним навчанням) забезпечують високий рівень імерсивності, варіативності діалогів та здатність до адаптації до унікальних ситуацій. Вони здатні генерувати природні, контекстно-залежні репліки, що створюють ілюзію живого розуму за персонажем. Однак ці методи не гарантують стабільності (феномен «галюцинацій» LLM – генерація правдоподібних, але невірних тверджень), масштабованості (затримка 0,5–3 секунди на запит до зовнішнього API) та економічної ефективності (експоненціальне зростання витрат на API-запити при збільшенні кількості NPC). Крім того, генеративні моделі можуть порушити ігровий баланс через непередбачувані рішення, які суперечать логіці економічної симуляції.

Наукова проблема полягає у відсутності архітектурного рішення, яке одночасно задовольняє три критичні та часто суперечливі вимоги до NPC-систем у жанрі Idle/Tycoon:

1. Вимога real-time детермінізму: час прийняття рішення агентом повинен бути $\leq 0,02$ мс для забезпечення 60 FPS при 100+ одночасно активних NPC. Це диктує необхідність алгоритмів з константною або лінійною часовою складністю;
2. Вимога економічної строгості: 100% передбачуваність та детермінованість економічно значущих дій (робота, відпочинок, виробництво ресурсів, витрати ресурсів) для коректного балансування гри та забезпечення справедливих умов для гравця. Жодні стохастичні або генеративні рішення AI не повинні впливати на економічний цикл;
3. Вимога природної комунікації: генерація контекстно-залежних, персоналізованих діалогів без заздалегідь прописаних скриптів для подолання «імерсивного розриву» та утримання залученості гравця. Персонажі повинні реагувати унікально на кожну ситуацію, виражаючи індивідуальність та емоції.

Жодна з існуючих окремих архітектур не забезпечує одночасного виконання всіх трьох вимог: FSM гарантує (1) та (2), але критично не задовольняє (3) через жорсткість та однотипність поведінки; LLM та Utility AI забезпечують (3), але порушують (1) через затримку та обчислювальні витрати, та частково (2) через ризик галюцинацій та непередбачених рішень; GOAP та інші планувальні системи знаходяться в проміжній позиції, але мають експоненціальну складність $O(b^d)$, що робить їх непридатними для масштабування.

Гібридні підходи, що поєднують детерміновані та генеративні методи із чітким розділенням відповідальності (детермінізм для економіки, генерація для комунікації), досі не отримали достатнього теоретичного обґрунтування та експериментальної валідації в контексті економічних симуляторів. Розробка такої архітектури із формальними гарантіями детермінізму економічного ядра та асинхронною

інтеграцією LLM для комунікації є актуальною науковою проблемою, якій присвячена дана робота.

Таким чином, **актуальність** даної роботи полягає у необхідності вирішення науково-практичної задачі, яка досі залишається недостатньо розробленою в акадémічній літературі і практиці розробки ігор: створення удосконаленої гібридної системи штучного інтелекту, яка поєднує надійність і передбачуваність детермінованих алгоритмів (для економічної логіки та стратегічних рішень NPC) з гнучкістю, креативністю та реалістичністю генеративних моделей (для комунікації та соціальної взаємодії). Таке поєднання повинне забезпечити новий рівень якості та імерсивності мобільних ігрових застосунків без порушення економічного балансу та без критичного зниження продуктивності.

Зв'язок роботи з науковими програмами та дослідницькими напрямками полягає в тому, що кваліфікаційна робота виконується у рамках програми наукових досліджень, спрямованих на вивчення сучасних методів розробки програмного забезпечення та інтеграції інтелектуальних систем у комерційні програмні продукти. Дослідження прямо пов'язано з актуальними проблемами застосування штучного інтелекту та машинного навчання в ігровій індустрії, зокрема з питаннями оптимізації архітектури для реального часу та забезпечення користувацького досвіду високої якості.

Метою роботи є розробка та дослідження гібридної архітектури для керування поведінкою NPC, яка поєднує детерміновану FSM-логіку із генеративними моделями LLM з метою підвищення імерсивності взаємодії й збереження стабільності економіки ігрового симулятора.

Об'єктом дослідження є процеси автоматизованого управління NPC у цифрових симуляторах економіки та офісу.

Предметом дослідження є архітектурні й практичні засоби інтеграції FSM, LLM та асинхронних механізмів у NPC-системах для idle/tycoon-ігор.

Для досягнення поставленої мети необхідно вирішити комплекс окремих дослідницьких **завдань**.

Перше завдання полягає у проведенні комплексного аналізу сучасного стану технологій розробки мобільних ігор та детальному розгляді існуючих підходів до моделювання поведінки NPC, з тим щоб виявити характерні обмеження як класичних методів (FSM, BT), так і генеративних підходів (LLM), та обґрунтувати науково-практичні причини цих обмежень.

Друге завдання спрямоване на теоретичне обґрунтування доцільності використання саме гібридної архітектури для комплексного вирішення проблеми «імерсивного розриву», що характерна для сучасних економічних симуляторів.

Третє завдання передбачає спроектування детальної архітектури системи, яка включає три функціональні рівні: Стратегічний рівень (на основі FSM) для управління станами та рішеннями NPC, Тактичний рівень (на основі NavMesh) для просторової навігації та адаптації до змін середовища, та Комунікативний рівень (на основі LLM) для генерації природних, контекстно-залежних діалогів.

Четверте завдання спрямоване на розробку удосконаленого методу контекстної генерації промтів та архітектури безпечного асинхронного зв'язку з зовнішнім LLM API через проксі-сервер, що дозволяє уникнути блокування основного ігрового циклу.

П'яте завдання передбачає повну програмну реалізацію ігрового застосунку на платформі Unity із мовою програмування C#, з інтеграцією динамічної системи розподілу ресурсів, системи модельованого часу доби та імерсивного динамічного освітлення.

Нарешті, *шосте завдання* спрямоване на проведення експериментальних досліджень ефективності розробленої системи за об'єктивними показниками продуктивності (Latency, CPU Overhead, FSM Stability) та суб'єктивними критеріями якості (реалізм діалогів, контекстуальна гнучкість, загальна імерсивність).

У роботі використовуються різноманітні **методи** наукового дослідження, кожен з яких спрямований на розв'язання окремих аспектів поставленого завдання. Системний аналіз застосовується для класифікації та порівняння методів штучного інтелекту, визначення вимог до розробляємої системи та аналізу взаємозалежностей між компонентами архітектури. Об'єктно-орієнтоване проєктування

використовується для розробки архітектури програмного забезпечення, визначення структури класів, інтерфейсів та модульної організації застосунку з метою забезпечення гнучкості та масштабованості. Теорія автоматів застосовується для математичного моделювання станів NPC та переходів між ними на основі логіки FSM, що дозволяє формально обґрунтувати коректність поведінки. Алгоритми пошуку шляху на графах (зокрема, алгоритм A*) використовуються для реалізації просторового переміщення персонажів та адаптації траєкторій до змін ігрового середовища. Експериментальний метод застосовується для проведення контрольованих тестів з метою оцінки продуктивності системи, стабільності детермінованого ядра та якості взаємодії з точки зору користувача.

Наукова новизна одержаних результатів магістерської роботи полягає у кількох ключових аспектах. По-перше, вперше запропоновано двошарову архітектуру гібридного штучного інтелекту спеціально для мобільних симуляторів та Idle/Tycoon-ігор, де генеративний шар (LLM) реалізовано як ізольовану, асинхронну надбудову над надійним детермінованим ядром (FSM). Це дозволяє не тільки нівелювати проблему затримки мережної відповіді (принцип Latency Hiding) без блокування основного ігрового циклу, а й запобігти втручанням генеративної моделі у логіку прийняття рішень, що є критичним для економічної симуляції. По-друге, удосконалено та формалізовано метод контекстної генерації промтів (Contextual Prompt Engineering) шляхом введення параметрів «Персони» (для забезпечення унікальності персонажів) та динамічного «Ресурсного стану» (для врахування поточної економічної ситуації), що забезпечує релевантність відповідей NPC та перетворює системні збої (такі як брак ресурсів) на органічні елементи ігрового нарративу. По-третє, реалізовано оригінальну архітектуру асинхронної комунікації через проксі-сервер (на базі Google Apps Script), яка дозволяє безпечно та ефективно з'єднати мобільний клієнт із хмарною LLM-послугою без експозиції критичних даних.

Практичне значення одержаних результатів роботи полягає у розробленні ігрового застосунку на платформі Unity, який демонструє стабільну та надійну роботу економічної моделі, поєднану з високою якістю соціальної взаємодії та

реалістичністю комунікації персонажів. Реалізовані програмні модулі (система керування FSM, технологія розумних ресурсів з атомарним захопленням, асинхронний менеджер API) та детально описана архітектура проксі-сервера можуть бути використані як універсальний, повнозмістовний шаблон для інтеграції Generative AI у будь-які Unity-проекти незалежно від жанру. Таке застосування дозволяє розробникам практично підвищити рівень залученості гравців та якість досвіду без ризику порушення ігрового балансу та економічної моделі. Розроблена методика також може бути адаптована для використання у навчальних цілях як у вищих навчальних закладах, так і у навчальних матеріалах для розробників-практиків.

Робота пройшла апробацію на науковій конференції [26].

РОЗДІЛ 1. АНАЛІЗ ТЕХНОЛОГІЙ ШТУЧНОГО ІНТЕЛЕКТУ В СУЧАС- НІЙ РОЗРОБЦІ ІГОР

1.1 Роль та класифікація систем штучного інтелекту в індустрії розваг

Штучний інтелект (ШІ) у контексті комп'ютерних ігор (Game AI) є фундаментально відмінним від класичного академічного ШІ. Якщо метою академічного ШІ є вирішення складних обчислювальних задач оптимальним шляхом або проходження тесту Тюрінга, то головною метою ігрового ШІ є створення переконливої ілюзії інтелекту, яка забезпечує збалансований, цікавий та непередбачуваний ігровий процес (Gameplay) [1].

ШІ став невіддільною частиною сучасної розробки ігор, впливаючи на всі аспекти створення цифрового продукту – від генерації контенту до керування поведінкою неігрових персонажів (NPC) та оптимізації продуктивності. Сучасні ігрові рушії, такі як Unity та Unreal Engine, інтегрують складні системи навігації та інструменти машинного навчання, що дозволяє розробникам створювати динамічні, "живі" віртуальні світи [22].

Для систематизації та глибокого розуміння різноманітних підходів до моделювання поведінки NPC у сучасних ігрових системах представлено комплексну класифікаційну діаграму (рис. 1.1). Діаграма структурує системи штучного інтелекту за п'ятьма основними категоріями: детерміновані системи (FSM, BT, GOAP), імовірнісні системи (Utility AI, Machine Learning), генеративні моделі (LLM, Procedural Generation), гібридні підходи (FSM + LLM) та спеціалізовані системи (NavMesh, Pathfinding, Steering Behaviors). Кожна категорія має власні переваги та обмеження. Детерміновані системи забезпечують стабільність та передбачуваність, але страждають від одноманітності та штучності. Генеративні моделі (LLM) пропонують високу іммерсивність та природність діалогів, проте мають значну затримку (0.5-3 сек) та ризик "галюцинацій". Гібридний підхід (FSM + LLM),

розроблений у цій роботі, об'єднує сильні сторони обох парадигм: стабільність економіки від FSM та природність комунікації від LLM, що робить його оптимальним рішенням для жанру Idle/Tycoon симуляторів.

Системи штучного інтелекту в ігровій індустрії можна класифікувати за функціональним призначенням на п'ять основних категорій:

- 1) ШІ для управління NPC (Runtime AI). Це найбільш традиційна та помітна для гравця сфера застосування. Вона охоплює алгоритми, що керують діями автономних агентів (NPC) у віртуальному середовищі в реальному часі. До цієї категорії належать:
 - a) Системи навігації (Pathfinding): Використання алгоритмів A* (A-Star) та навігаційних сіток (NavMesh) для пошуку оптимального маршруту [12].
 - b) Тактичний аналіз: Оцінка ситуації на полі бою, вибір укриттів, координація групових дій.
 - c) Прийняття рішень (Decision Making): Визначення стратегії поведінки агента. Наприклад, у грі *The Last of Us Part II* NPC здатні до складної групової координації, спілкуючись вербально та адаптуючись до тактики гравця, що є вершиною реалістичного Runtime AI [6].
- 2) ШІ для процедурної генерації контенту (PCG). Цей напрямок дозволяє автоматизувати створення статичних та динамічних ігрових активів. PCG використовується для генерації ландшафтів, архітектури рівнів, текстур, 3D-моделей та динамічних квестів. Використання алгоритмів PCG (наприклад, на основі шуму Перліна або фракталів) дозволяє створювати нескінченні варіації ігрового світу, що значно підвищує реgrabельність продукту та зменшує навантаження на художників. Класичними прикладами є ігри *Minecraft* та *No Man's Sky* [2].

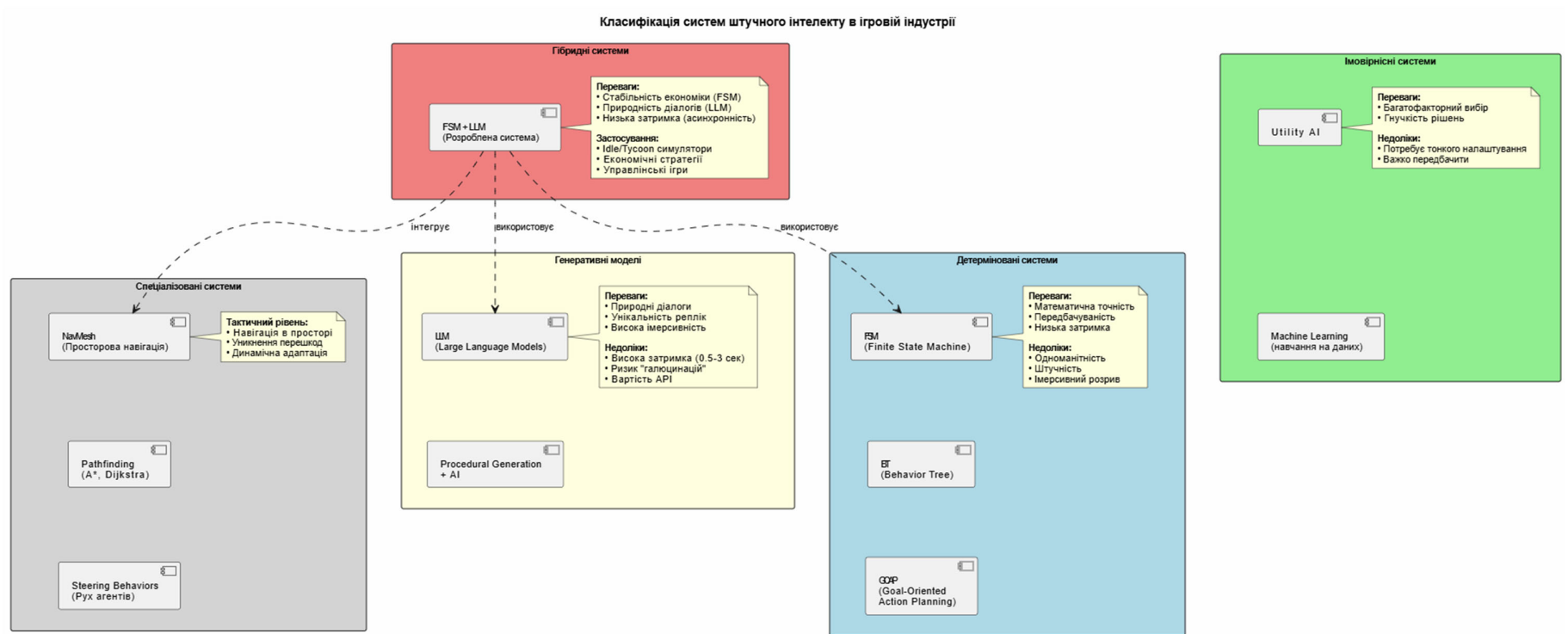


Рисунок 1.1 – Класифікація систем штучного інтелекту в ігровій індустрії з виділенням переваг, недоліків та застосування кожного підходу

- 3) ІІІ для управління ігровим досвідом (Experience Management). Це системи мета-рівня, що функціонують "над" ігровим процесом. Вони аналізують поведінку гравця в реальному часі та динамічно адаптують параметри гри під його навички та емоційний стан. Метою є утримання гравця у стані "потoku" (Flow), уникаючи надмірної складності або нудьги. Хрестоматійним прикладом є система "AI Director" у серії ігор *Left 4 Dead*, яка регулює інтенсивність атак ворогів, кількість ресурсів та музичний супровід [11].
- 4) ІІІ для тестування та оптимізації (QA AI). Цей напрямок охоплює застосування ІІІ для виробничих потреб. До нього належать:
 - a. Використання ботів на базі машинного навчання для автоматизованого тестування (QA). Ці агенти можуть знаходити помилки у логіці гри, проходячи рівні тисячі разів.
 - b. Алгоритми для оптимізації графічного конвеєра, такі як технологія DLSS (Deep Learning Super Sampling) від NVIDIA, що підвищує роздільну здатність зображення без втрати продуктивності.
- 5) ІІІ для аналітики та монетизації. Це використання алгоритмів машинного навчання (ML) та Data Mining для аналізу поведінкових патернів користувачів. У мобільних іграх (GaaS) вони прогнозують відтік користувачів (Churn Rate), сегментують аудиторію та персоналізують пропозиції внутрішньоігрового магазину. Це дозволяє пропонувати гравцеві саме ті товари, які є найбільш актуальними для нього в конкретний момент часу, що максимізує прибуток розробника [14].

1.2 Характеристика та специфіка жанру Idle/Tycoon як об'єкта автоматизації

Для коректної постановки задачі дослідження та обґрунтування вибору архітектури штучного інтелекту необхідно провести системний аналіз предметної області – жанру Idle/Tycoon (економічні симулятори з елементами пасивного доходу).

Цей жанр висуває унікальні, часто суперечливі вимоги до поведінки неігрових персонажів (NPC), які суттєво відрізняються від вимог у динамічних жанрах (шутери) або наративних іграх (RPG).

1.2.1 Визначення та економічна роль NPC у жанрі

Жанр Idle Games (також відомі як інкрементальні ігри або "клікери") – це жанр відеоігор, ігровий процес яких базується на виконанні простих повторюваних дій для накопичення первинного ресурсу, який згодом інвестується в автоматизацію виробництва. Ключовою особливістю жанру є концепція "пасивного прогресу" (offline progress): гра продовжує функціонувати та генерувати ресурси навіть за відсутності активних дій гравця або коли додаток закрито.

Жанр Tycoon-ігри (економічні симулятори) фокусується на детальному менеджменті бізнес-процесів: управлінні фабрикою, парком розваг, лікарнею, офісним центром тощо. Гравець виступає в ролі менеджера, який оптимізує логістичні ланцюжки та балансує доходи і витрати.

У гібридному жанрі Idle/Tycoon, який є об'єктом даного дослідження, поєднуються глибина економічної стратегії та доступність механік автоматизації. Гравець приймає стратегічні рішення (будівництво, найм персоналу), тоді як тактичні задачі (безпосереднє виконання роботи) делегуються автоматизованим агентам – NPC.

У цьому жанрі NPC виконують критично важливу функцію – вони є візуалізаторами економічних процесів та автономними економічними агентами. На відміну від шутерів, де NPC є ціллю, або RPG, де NPC є джерелом історії, в тайкуні NPC – це "актив" (asset) або виробнича одиниця. Їхня поведінка безпосередньо впливає на ключові показники ефективності (KPI) гравця, такі як дохід на хвилину (GPM – Gold Per Minute).

1.2.2 Унікальні вимоги до детермінованості та масштабованості

Поведінка NPC у жанрі Idle/Tycoon має відповідати двом ключовим, часто суперечливим, вимогам, які прямо впливають на вибір архітектури штучного інтелекту та визначають можливості й обмеження системи.

Перша вимога стосується детермінованості поведінки (Deterministic Loop). Поведінка персонажів має бути суворо прогнозованою та передбачуваною, оскільки економічна модель гри безпосередньо залежить від стабільності дій кожного агента. Економічний цикл робітника є лінійним і циклічним: персонаж приходить на роботу, потім працює протягом визначеного часу, після чого бере перерву на відпочинок, і нарешті повертається до роботи або додому. Якщо NPC раптово "вирішить" відхилитися від цього циклу (наприклад, через складний алгоритм планування на основі цілей, такий як GOAP, який може прийняти непередбачуване рішення про зміну пріоритетів), це призведе до неконтрольованого збою економічної моделі гри. Уявімо ситуацію, коли робітник раптово вирішує, що його особистий відпочинок важливіший за роботу, і припиняє виконувати свої обов'язки – це руйнує баланс виробництва, порушує очікування гравця щодо прибутку і може зробити гру нечесною або некерованою. Отже, алгоритмічна поведінка має бути жорсткою, передбачуваною та контрольованою розробником, щоб забезпечити економічну справедливість та баланс.

Друга вимога стосується масштабованості системи (Scalability). У типових Tycoon-іграх гравець одночасно керує великою кількістю NPC – від десятків до сотень персонажів на екрані одночасно. Це створює серйозні вимоги до продуктивності: архітектура штучного інтелекту повинна мати мінімальні вимоги до обчислювальних ресурсів процесора (CPU) на одного агента, щоб система могла безперебійно працювати без значного зниження частоти кадрів (FPS). Навіть невеликі додаткові витрати на обробку кожного агента, помножені на сотні персонажів, можуть призвести до катастрофічного падіння продуктивності. Це автоматично виключає використання обчислювально "важких" алгоритмів, таких як планування GOAP (де кожен агент постійно перебудовує складні плани дій за допомогою

алгоритмів пошуку на графах) або постійний обхід складних і глибоких Behavior Trees для кожного агента на кожному кадрі. Навіть якщо такі системи забезпечують більш складну та реалістичну поведінку, вони неприпустимі з точки зору продуктивності для жанру Idle/Tycoon.

Таким чином, для управління діями NPC у цьому жанрі потрібен простий, надійний, детермінований і високопродуктивний механізм, який може обробляти велику кількість агентів без суттєвих витрат на обчислення. Ці вимоги вказують на скінченні автомати (FSM) як на оптимальний вибір для базової архітектури, що буде детально проаналізовано та обґрунтовано у підрозділі 1.4. FSM забезпечує необхідний баланс між функціональністю, передбачуваністю та ефективністю, що робить його найкращим кандидатом для реалізації детермінованого ядра системи управління NPC у економічних симуляторах.

1.2.3 Проблема "Імерсивного розриву" (Immersion Gap) та соціальний аспект

Незважаючи на необхідність детермінованої поведінки, головною проблемою сучасних 3D Idle-ігор є швидка втрата інтересу гравця (зменшення показників утримання – Retention Rate) через відчуття "роботи з електронною таблицею" (Spreadsheet Syndrome). Коли гравець переходить від стратегічного планування до спостереження за ігровим світом, він стикається з ефектом, подібним до "зловісної долини" у робототехніці.

Персонажі виглядають як люди (мають реалістичні 3D-моделі), але поведуться як примітивні автомати:

- вони не реагують на дії гравця (наприклад, коли гравець ставить стіл прямо перед ними);
- вони не можуть повідомити про проблему (наприклад, "Немає вільних стільців!"), показуючи лише іконку;

- вони не взаємодіють один з одним, рухаючись по ідеальним, але неприродним траєкторіям.

Цей дисонанс між візуальною якістю 3D-моделей та примітивною логікою створює "імерсивний розрив" (Immersion Gap), який руйнує занурення. Гравець перестає сприймати гру як "живий світ" і починає бачити лише набір математичних формул, що швидко призводить до нудьги.

Вирішенням цієї проблеми є впровадження гібридного ШІ, який дозволяє зберегти економічну ефективність (алгоритм), додаючи "людські" риси (генеративна комунікація), що і є ключовою інновацією та завданням даної роботи.

1.3 Дихотомія інструментарію ШІ: Editor AI проти Runtime AI

Сучасний етап розвитку технологій, зокрема в екосистемі ігрового рушія Unity, призвів до розгалуження сфери застосування ШІ на два фундаментально відмінні напрямки. У науковій та технічній літературі часто виникає термінологічна плутанина, тому для цілей даного дослідження необхідно провести чітку демаркацію між ШІ як асистентом розробника (Editor AI) та ШІ як компонентом ігрового процесу (Runtime AI).

Штучний інтелект як інструментарій розробника (Editor AI) охоплює інструменти, що інтегруються безпосередньо в середовище розробки (IDE) або ігровий рушій (наприклад, Unity Editor) і використовуються виключно на етапі створення продукту (Production Phase). Їхня основна мета – автоматизація рутинних задач, генерація статичних асетів та оптимізація робочого процесу. Результатом роботи Editor AI є файли (текстури, моделі, скрипти), які компілюються у фінальний білд гри, але сам ШІ-алгоритм на пристрої гравця не виконується. Яскравим прикладом сучасного Editor AI є платформа Unity Muse, яка використовує генеративні моделі для прискорення розробки. Зокрема, *Muse Texture* дозволяє розробнику ввести текстовий промт (наприклад, "старий потертий офісний паркет ялинкою з подряпинами") і автоматично генерує повний набір безшовних PBR-карт, замінюючи

години роботи 3D-художника. Інтелектуальний асистент коду *Muse Chat*, донавчений на актуальній документації Unity API, дозволяє генерувати шаблонний код, знаходити помилки компіляції та отримувати пояснення щодо архітектурних рішень безпосередньо у вікні редактора. Таким чином, Editor AI вирішує виробничі задачі розробника, впливаючи на собівартість та швидкість розробки.

На противагу цьому, системи Штучного інтелекту виконання (Runtime AI) функціонують безпосередньо на пристрої користувача під час ігрової сесії (at runtime). Ці алгоритми виконуються в реальному часі (зазвичай у кожному кадрі), приймаючи рішення на основі поточного, динамічного стану ігрового світу. Саме ця категорія є предметом даного дослідження, оскільки вона безпосередньо формує ігровий досвід. У контексті розробки гібридної системи для ігор жанру Idle/Tycoon, Runtime AI поділяється на два ключові, інтегровані рівні. Перший – Алгоритмічний (Детермінований) ШІ – є основою для економічної логіки та забезпечує стабільність геймплею. Сюди входять системи навігації NavMesh для побудови оптимальних маршрутів та використання скінченних автоматів (FSM) для прийняття рішень, що гарантує виконання NPC чітких економічних функцій (працює, відпочиває, йде додому). Другий рівень – Генеративний (Імовірнісний) ШІ – є новітнім напрямком, що базується на використанні API великих мовних моделей (наприклад, Google Gemini) у реальному часі. На відміну від Editor AI, тут генерація контенту (діалогів) відбувається постійно під час гри і є контекстуальною: система (*NPCBrain.cs*) формує промт неймережі на основі поточного стану FSM (наприклад, *State.REPORTING_TO_PLAYER*). Неймережа генерує унікальну відповідь ("Бос, мені немає де працювати!"), яка створює ілюзію "живого" спілкування. Поєднання цих двох рівнів – детермінованості (для економіки) та імовірнісності (для комунікації) в режимі реального часу – є ключовою інновацією даної роботи. Таким чином, якщо Editor AI робить гру швидше, то гібридний Runtime AI робить її цікавішою, не порушуючи при цьому економічну стабільність.

1.4 Огляд та порівняльний аналіз сучасних підходів до моделювання поведінки NPC

Вибір архітектурного патерну для моделювання поведінки NPC є фундаментальним рішенням при проектуванні будь-якої гри. Цей вибір безпосередньо впливає на детермінованість, продуктивність, масштабованість та гнучкість системи. Для жанру Idle/Tycoon, де критично важливо підтримувати економічну стабільність, цей вибір має бути обґрунтований системним аналізом та глибоким розумінням компромісів, притаманних кожному підходу.

На основі аналізу профільної літератури та існуючих індустріальних практик, можна виділити чотири основні архітектурні патерни для прийняття рішень (Decision Making) та окремий клас для переміщення (Spatial AI). Кожен з цих патернів має свою нішу застосування, свої переваги та недоліки, які необхідно враховувати при виборі архітектури для конкретного проекту. Детальний опис та порівняльний аналіз цих архітектурних патернів представлено у підрозділах 1.4.1–1.4.5.

Для цільової частоти 60 FPS бюджет одного кадру становить

$$T_{\text{frame}} = 1000 / 60 \approx 16,67 \text{ мс.}$$

З цього бюджету підсистема штучного інтелекту зазвичай отримує не більше 10%, тобто приблизно $TAI \approx 1,67 \text{ мс}$ на всі обчислення, пов'язані з прийняттям рішень. Якщо одночасно симулюється N агентів, максимальний час на одного NPC дорівнює $T_{\text{agent}} = TAI / N$. Для 100 NPC це лише близько 0,017 мс (17 мкс), а для 1000 NPC – близько 0,0017 мс (1,7 мкс). Такі обмеження фактично перетворюють вибір архітектури з теоретичного питання на жорстку інженерну вимогу до часової складності алгоритмів.

Асимптотична складність основних підходів до прийняття рішень суттєво різниться. Скінченний автомат (FSM) працює за константний час $O(1)$, оскільки в кожен момент перевіряє лише умови для поточного стану. Древа поведінки (Behavior Tree) вимагають $O(d)$ операцій, де d – глибина дерева, тобто кількість вузлів від кореня до листка. Системи корисності (Utility AI) мають складність $O(n \cdot k)$, де n – кількість можливих дій, а k – кількість факторів оцінки для кожної

дії. Планувальник GOAP характеризується експоненціальною складністю $O(b^d)$, де b – середня гілковість (кількість можливих дій у кожному кроці), а d – глибина плану. Алгоритм A^* для пошуку шляху на графі навігації має складність $O(|E| \cdot \log |V|)$, де $|V|$ – кількість вершин, а $|E|$ – кількість ребер графа.

На практиці це означає, що FSM зазвичай виконується у діапазоні **0,001–0,01 мс** на одного агента, BT – приблизно **0,05–0,15 мс**, Utility AI – **0,1–0,3 мс**, тоді як повноцінний GOAP може займати **0,5–2,0 мс** і більше. З наведеного аналізу випливає, що для підтримки 100+ NPC при 60 FPS придатними є лише FSM та (у граничному випадку) неглибокі дерева поведінки, тоді як GOAP та важкі конфігурації Utility AI автоматично виключаються через перевищення часового бюджету. Для масштабування до сотень і тисяч NPC єдиним реальною опцією залишається FSM з його константною складністю.

1.4.1 Скінченні автомати

FSM – це математична модель обчислень, яка описує систему з скінченною кількістю станів. У будь-який момент часу система може перебувати лише в одному стані. Перехід з одного стану в інший (Transition) відбувається при виконанні певної умови або настанні події (Trigger). Така модель надзвичайно проста у розумінні та реалізації, що обумовило її широке застосування у індустрії ще з перших днів розробки комп'ютерних ігор.

FSM ідеально описує поведінку простих агентів. Наприклад, стан IDLE (Спокій) очікує на подію "Гравець у зоні видимості". Коли подія спрацьовує, відбувається перехід у стан ATTACK. У стані ATTACK агент виконує свою логіку, доки не спрацює інша подія, наприклад "Здоров'я гравця дорівнює нулю", що ініціює перехід у стан RETURN_TO_PATROL або CELEBRATION. Така послідовна логіка робить FSM навіть більш природною для опису повсякденних дій.

Для жанру Тусоон, FSM є традиційним та найбільш поширеним вибором саме через його надійність. Економічний цикл робітника ідеально описується

скінченним автоматом: персонаж прокидається вдома (AT_HOME), після настання ранку (Настав ранок) переходить у стан руху до роботи (GOING_TO_WORK), після прибуття (Дійшов) починає працювати (WORKING), після втоми (Втомився) йде на перерву в кав'ярню (GOING_TO_COFFEE), після відновлення (Відпочив) повертається до роботи (GOING_TO_WORK), і нарешті, після закінчення робочого дня (Кінець дня) йде додому (GOING_HOME). Така циклічна природа дозволяє легко прогнозувати поведінку.

Переваги FSM полягають передусім у передбачуваності й детермінізмі: поведінка агента є абсолютно прогнозованою, що критично для налагодження економічного балансу та забезпечення справедливих умов для гравця. Немає місця для "приховано strani" результатів, які можуть різко змінити ігрову економіку. FSM також забезпечує винятково високу продуктивність, оскільки вимагає мінімальних обчислювальних ресурсів – зазвичай, одна простенька перевірка умови за кадр. Це дозволяє симулювати сотні агентів одночасно, що є прямою вимогою для сучасних Idle-ігор, де на екрані можуть бути десятки або навіть сотні персонажів. Крім того, FSM характеризується простотою реалізації: легко реалізується базовими засобами мови C# та не потребує складних структур даних.

Однак недоліки FSM проявляються особливо яскраво при розширенні системи. На перший погляд проста архітектура перетворюється у складність масштабування: при збільшенні кількості станів, кількість необхідних переходів зростає експоненціально (при N станів потрібно потенційно $N \times (N-1)$ переходів), що призводить до створення складних, заплутаних структур часто називаних "спагетті-код". Розуміння такої системи стає все більш складним, а додавання нового функціоналу починає впливати на решту системи непередбачувано. Також FSM є суто реактивною системою – вона не планує дії наперед, а лише реагує на поточні подразники та умови. Це означає, що агент не може приймати складні, далекосяжні рішення, які вимагають врахування майбутнього.

1.4.2 Дерева поведінки

Дерева поведінки (Behavior Trees, BT) – це сучасний стандарт індустрії, що прийшов на зміну FSM у багатьох AAA-проектах та комерційних іграх (Halo, BioShock, Unreal Engine 4+). BT – це ієрархічна структура у формі дерева, вузли якого описують логіку прийняття рішень на різних рівнях абстракції. Така структура дозволяє виразити складну поведінку крізь комбінацію простих блоків.

Принцип роботи в GameDev полягає в тому, що кожен кадр (або з заданою розробником частотою) від кореневого вузла (Root) подається сигнал для виконання (так звана операція "тік" або Tick), який проходить крізь дерево, обчислюючи результати. Вузли поділяються на кілька типів. Композитні вузли (Composites) керують потоком виконання та логікою: Sequence (Послідовність) виконує дочірні вузли строго по черзі і повертає успішний результат (Success) лише якщо всі дочірні вузли поверну Success, що гарантує виконання всіх кроків у правильному порядку; Selector (Вибір) виконує дочірні вузли до тих пір, доки один з них не поверне Success, забезпечуючи флексибільний вибір альтернатив. Декоратори (Decorators) – це вузли-обгортки, які модифікують результат дочірнього вузла (наприклад, Inverter змінює Success на Failure та навпаки). Листки (Leafs) – це кінцеві вузли, що насправді виконують роботу: дії (Action) виконують конкретні операції в грі (наприклад, "Атакувати", "Рухатися"), а умови (Condition) перевіряють поточний стан світу (наприклад, "Чи бачу я ворога?", "Чи маю достатньо мани?").

Переваги BT включають модульність та композиційність: кожен гілку дерева (наприклад, "Знайти укриття") можна розробити, тестувати та налагодити окремо, а потім безпечно підключити до різних агентів без побоювання, що це зруйнує іншу функціональність. Також BT характеризується яскравою гнучкістю: легко додавати нову поведінку, просто додаючи нові вузли або під-дерева до існуючої структури, не порушуючи стару логіку. Більшість сучасних реалізацій BT (особливо у комерційних ігрових движках) мають потужні візуальні редактори, що спрощує розробку, налагодження та комунікацію між програмістами та гейм-дизайнерами.

Недоліками є вищі обчислювальні витрати: необхідність обходу дерева (Tree Traversal) кожного кадру, рекурсивні виклики та оцінювання умов на кожному рівні дерева призводять до більшої затрати ресурсів порівняно з простою лінійною перевіркою стану в FSM. При великій кількості агентів, особливо з глибокими деревами, це може стати вузьким місцем для продуктивності.

1.4.3 Планування, орієнтоване на цілі (GOAP)

GOAP (Goal-Oriented Action Planning) – це надзвичайно потужна архітектура, що дозволяє агентам самостійно будувати послідовність дій для досягнення бажаного стану або цілі. Це проактивна система, на відміну від реактивних FSM та BT. Замість того, щоб розробник передбачав всі можливі сценарії, GOAP дозволяє агенту розумово "подумати" про те, як досягти бажаного результату, дивлячись на поточний стан світу.

Принцип роботи в GameDev базується на кількох основних компонентах. Світ (World State) описується набором фактів або пропозицій про стан (наприклад, `HasWeapon = false`, `IsHungry = true`, `IsAtWeapon = false`). Цілі (Goals) визначають бажаний стан світу, до якого агент прагне (наприклад, `HasWeapon = true` та `IsAtSafe = true`). Дії (Actions) – це переходи, які агент може виконати, кожна з них має `Preconditions` (передумови, які мають бути істинні перед виконанням) та `Effects` (наслідки, які дія встановлює як істинні після виконання). Наприклад, дія "Підібрати зброю" має передумову `IsAtWeapon = true` та наслідок `HasWeapon = true`, тоді як дія "Йти до зброї" має передумову `true` (завжди можна почати рухатись) та наслідок `IsAtWeapon = true`.

Планувальник (зазвичай інтелектуальний алгоритм пошуку, часто наслідок алгоритму A*) аналізує поточний стан, розглядає усі доступні дії та їхні ефекти, а потім знаходить найдешевший (або найоптимальніший) ланцюжок дій, який перетворює поточний стан світу на цільовий. Результатом є послідовність дій (план), наприклад: "Йти до зброї" → "Підібрати зброю" → "Йти до безпечного місця".

Переваги GOAP включають емерджентну поведінку: агенти часто знаходять несподівані, творчі рішення, які розробник навіть не передбачав. Це створює враження того, що агенти мають справжній інтелект і планують своє майбутнє. Також система є адаптивною: при зміні умов у світі (наприклад, загорожена дорога), планувальник автоматично відбудовує план, обираючи альтернативний маршрут.

Недоліки включають надзвичайно високу складність реалізації та налагодження: потрібно ретельно визначити всі стани світу, дії та їхні ефекти, а навіть мала помилка може призвести до неправильного планування. Також існує проблема непрогнозованості: у грі жанру Tusoon агент, керований GOAP, може раптово "вирішити", що його ціль "Насолода та відпочинок" істотно важливіша за "Роботу на користь компанії", що може порушити економічний баланс гри та зруйнувати геймплей.

1.4.4 Системи корисності (Utility AI)

Цей підхід часто використовується у симуляторах життя (The Sims, RimWorld) та місяцях за перші місяці розробки AI для складних персонажів. Системи корисності пропонують принципово інший підхід: агент не має жорстких станів, а постійно оцінює "корисність" або "привабливість" кожної можливої дії в контексті поточної ситуації.

Принцип роботи в GameDev полягає в тому, що агент має набір потреб, мотивів або мотивуючих факторів (Голод, Енергія, Соціалізація, Розвага, Безпека). Кожна можлива дія, яку агент може виконати (Їсти, Спати, Говорити, Грати, Ховатися), має функцію корисності (Utility Function), яка обчислює числовий бал (score) на основі поточних потреб та змінних стану. Наприклад, функція корисності для дії "Їсти" залежить від поточного рівня голоду: чим вищий голод, тим вища корисність їжі. Аналогічно, функція для дії "Спати" залежить від енергії. Агент завжди обирає дію з найвищим балом корисності, що створює природне, гнучке та адаптивне поведінкове різноманіття.

Перевагою є можливість створювати дуже "м'яку", плавну, природну та багатогранну поведінку, де персонаж не переключається різко з одного стану на інший, а поступово і змінює пріоритети залежно від умов. Це створює реалістичний рівень деталізації у взаємодіях.

Недоліком є складність налаштування та балансування: необхідно ретельно налаштувати десятки кривих корисності, обрати правильні ваги для кожного фактора, врахувати їхні взаємовпливи. Мала помилка у параметрах може призвести до дивної поведінки (наприклад, персонаж, який ніколи не спить, або, навпаки, ніколи не встає зі ліжка).

1.4.5 Просторовий ШІ та системи навігації

Усі вищезгадані системи (FSM, BT, GOAP, Utility AI) відповідають на питання "Що робити?" – вони визначають, яку дію слід виконати. Просторовий ШІ представляє окремий напрямок дослідження і відповідає на фундаментально інше питання: "Як туди дістатися?" – як агент повинен фізично переміщатися у просторі, враховуючи перешкоди, інших агентів та динамічні об'єкти.

NavMesh (Навігаційна сітка) є практичним стандартом де-факто в сучасних ігрових движках, включаючи Unity та Unreal Engine. Концептуально, ігровий світ дискретизується на набір опуклих полігонів, які представляють прохідні поверхні. Ці полігони формують граф, де кожна вершина – це полігон, а ребра з'єднують суміжні полігони. Задача переміщення агента зводиться до пошуку оптимального шляху на цьому графі. Алгоритм A^* є найпопулярнішим вибором для такого пошуку, оскільки він ефективно знаходить оптимальний маршрут, мінімізуючи сумарну вартість переміщення [16].

NavMesh Carving є критично важливою функцією для динамічних ігор, де гравець може розставляти меблі, будувати стіни або інші об'єкти. Вона дозволяє об'єктам "вирізати" отвори в навігаційній сітці в реальному часі, по суті перебудовуючи граф, і змушуючи NPC розраховувати нові маршрути, які обходять нові

перешкоди. Це створює ілюзію того, що персонажі реально розуміють фізичну геометрію світу.

Steering Behaviors (Кермова поведінка) – це набір елегантних локальних алгоритмів, розроблених Крейгом Рейнольдсом у 1987 році, для керування рухом агентів на локальному рівні. Замість того, щоб просто слідувати рекомендованому шляху з NavMesh, агенти використовують ці алгоритми для плавного, природного руху, включаючи уникнення зіткнень з іншими агентами (Obstacle Avoidance), слідування вздовж шляху (Seek/Arrive), дотримання дистанції від компаньйонів (Separation) та інші поведінки. NavMeshAgent у Unity практично використовує саме ці принципи для забезпечення реалістичного, плавного переміщення агентів у тривимірному просторі.

1.4.6 Вибір методу

Аналіз патернів ШІ (узагальнений у Таблиці 1.1) показує, що не існує єдиного "найкращого" методу; вибір залежить від задач проєкту.

Таблиця 1.1 – Порівняльний аналіз архітектурних патернів ШІ для NPC

Критерій порівняння	FSM (Скінченний автомат)	BT (Дерево поведінки)	GOAP (Планування)	Utility AI (Корисність)
Передбачуваність	Дуже висока	Висока	Низька	Середня
Навантаження на CPU	Мінімальне	Середнє	Високе (пікове)	Середнє
Масштабованість (кількість агентів)	Відмінна	Хороша	Обмежена	Хороша
Придатність для Тусооп	Оптимально	Можливо	Ризиковано	Для складних симуляцій

На основі проведеного аналізу можна зробити висновок, що для базового циклу роботи NPC у грі жанру Idle/Тусооп (переміщення, виконання роботи, генерація ресурсу) найбільш доцільним є використання *скінченних автоматів* (FSM). Цей метод забезпечує необхідну детермінованість, стабільність економічної моделі та високу продуктивність при великій кількості агентів.

Водночас, FSM не вирішує проблему "імерсивного розриву" та відсутності соціальної взаємодії. Тому в даній роботі пропонується *гібридна архітектура*: використання надійного FSM як "фундаменту" для дій та інтеграція генеративного ШІ (LLM) як "надбудови" для комунікаційного рівня.

1.4.7 Порівняльний аналіз продуктивності AI-архітектур

Для обґрунтування вибору FSM як базової архітектури системи, проведено порівняльний аналіз продуктивності чотирьох основних AI-підходів на основі літературних джерел та індустріальних бенчмарків. Таблиця 1.2 представляє кількісне порівняння архітектур за ключовими метриками продуктивності.

Таблиця 1.2 – Порівняльний аналіз продуктивності AI-архітектур для NPC

Критерій	FSM	Behavior Tree	Utility AI	GOAP
Часова складність	$O(1)$	$O(n)$	$O(n \cdot k)$	$O(b^d)$
Час виконання на агента, мс	<0,01	0,05–0,15	0,1–0,3	0,5–2,0
Макс. кількість NPC при 60 FPS	10000+	2000–5000	1000–3000	50–200
Детермінованість поведінки	Повна	Висока	Середня	Низька
Складність реалізації (1-5)	2	3	4	5
Ризик збоїв логіки	Мінімальний	Низький	Середній	Високий

Дані базуються на дослідженнях Hill S. (2024), який продемонстрував можливість симуляції 5000 NPC з використанням needs-based AI при середньому часі вибору дії 0,3 мс на агента, та роботі Zubek R. з Game AI Pro 3, де досягнуто показника 1000 NPC при 60 FPS у грі Project Highrise завдяки спрощеній FSM-архітектурі.

Розглянемо математичне обґрунтування часової складності кожного підходу:

1. FSM – $O(1)$: Перевірка поточного стану та умов переходу виконується за константний час, оскільки система завжди знаходиться в одному з N станів і перевіряє фіксований набір умов для поточного стану;

2. Behavior Tree – $O(n)$: Обхід дерева в глибину вимагає відвідування до n вузлів, де n – глибина дерева. Як зазначено в огляді Iovino et al. (2022), складність BT зростає лінійно з глибиною ієрархії;
3. Utility AI – $O(n \cdot k)$: Система має оцінити корисність n можливих дій по k функціям корисності для кожної, що створює поліноміальну залежність від кількості доступних дій та факторів впливу;
4. GOAP – $O(b^d)$: Планування на основі A^* має експоненціальну складність, де b – середня гілковість (кількість дій, доступних у кожному стані), а d – глибина плану. Згідно з дослідженням Jасорin (2015), GOAP потребує суворих обмежень на глибину пошуку для забезпечення real-time продуктивності.

Для ігор жанру Idle/Tycoon бюджет кадру при 60 FPS становить 16,67 мс, з яких на AI традиційно виділяється не більше 10% – приблизно 1,67 мс. При необхідності підтримки 100 NPC це означає бюджет 0,01–0,02 мс на агента, що автоматично виключає використання GOAP та обмежує застосування Utility AI. FSM з часом виконання менше 0,01 мс є єдиною архітектурою, що гарантовано задовольняє ці вимоги при масштабуванні до тисяч агентів.

Візуальне представлення на рисунку 1.2 демонструє нелінійну залежність між часом виконання та максимальною кількістю підтримуваних NPC. FSM забезпечує на два порядки вищу продуктивність порівняно з GOAP, що робить його оптимальним вибором для економічних симуляторів з великою кількістю персонажів.

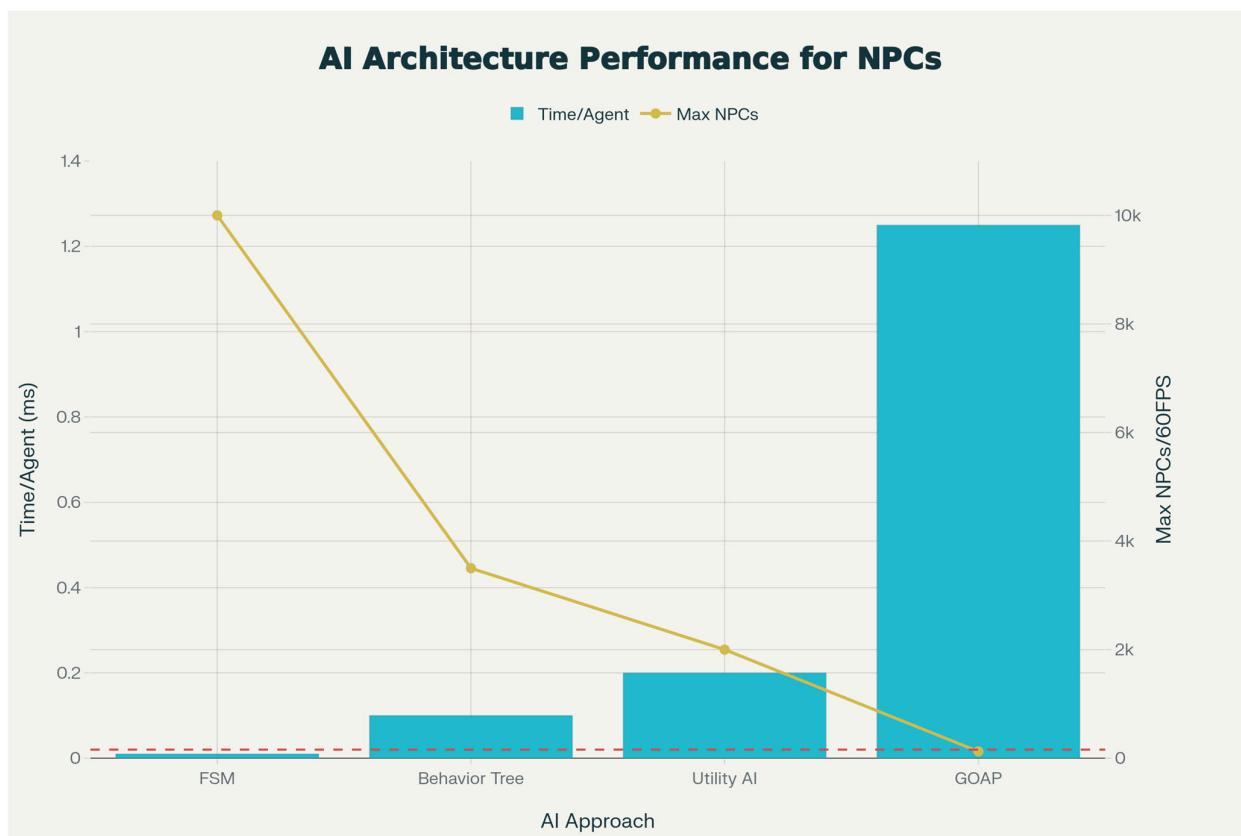


Рисунок 1.2 – Порівняльний аналіз продуктивності AI-архітектур: час виконання та максимальна кількість підтримуваних NPC

1.5 Аналіз генеративних методів ШІ та проблеми інтеграції

Поява високопродуктивних Великих Мовних Моделей (Large Language Models – LLM), таких як GPT-4 (OpenAI), Claude (Anthropic) та Gemini (Google), відкрила нову еру в розробці ігор – еру Генеративних Агентів (Generative Agents). Цей підхід кардинально відрізняється від традиційних методів, оскільки він зміщує фокус з детального "програмування поведінки" на більш абстрактне "навчання поведінки" через приклади та інструкції. Замість того, щоб розробник прописував кожен можливу репліку та реакцію персонажа, система генерує їх динамічно, базуючись на контексті та попередньому досвіді взаємодії. Це принципова зміна парадигми, яка обіцяє революціонізувати розуміння того, якими можуть бути інтерактивні персонажі у відеоіграх.

1.5.1 Потенціал генеративних методів та концепція Generative Agents

Традиційні неігрові персонажі обмежені невеликим набором заздалегідь написаних реплік та сценаріїв діалогів. Навіть у найбільш складних іграх з розгалуженими діалогами, гравець швидко вивчає всі можливі варіанти відповідей, запам'ятовує ключові фрази та починає передбачати реакції персонажів. Це поступово руйнує ілюзію життя та занурення у світ гри, перетворюючи спілкування з NPC на механічний процес вибору з обмеженого меню опцій. Генеративний штучний інтелект пропонує фундаментальне вирішення цієї проблеми через створення агентів, здатних до справжньої імпровізації, які можуть реагувати на непередбачені запитання та ситуації способами, які навіть розробник не передбачав на етапі проєктування.

Фундаментальне дослідження "Generative Agents: Interactive Simulacra of Human Behavior" (Park, et al., 2023), проведене дослідниками зі Стенфордського університету та Google Research, продемонструвало архітектуру, в якій агенти, керовані великими мовними моделями, здатні симулювати надзвичайно складну соціальну поведінку, включаючи формування відносин, планування довготривалих активностей та реакцію на соціальні події. Архітектура таких генеративних агентів включає три ключові компоненти, що працюють у синергії. Перший компонент – це Потік пам'яті (Memory Stream), де агент безперервно записує всі свої спостереження та взаємодії у структуровану базу даних. Кожна подія фіксується з часовою міткою та контекстом, наприклад: "Я бачу стіл біля вікна о 9:15", "Гравець привітався зі мною ввічливо о 9:20", "Я відчуваю голод о 12:30". Другий компонент – Рефлексія (Reflection), де агент періодично аналізує накопичені спогади, щоб зробити узагальнення та висновки високого рівня. Наприклад, після кількох взаємодій агент може сформулювати думку: "Гравець – мій керівник, він зазвичай ввічливий, тому я маю бути професійним у спілкуванні з ним" або "Останнім часом я часто втомлююся після обіду, можливо, мені потрібно більше відпочивати". Третій

компонент – Планування (Planning), де агент на основі своїх цілей, поточного стану та рефлексій створює довготривалі плани дій, такі як "Зранку піти на роботу, в обід випити кави з колегою, після роботи відвідати спортзал".

У комерційних проєктах та технологічних демонстраціях, таких як платформа Inworld AI або демо NVIDIA ACE (Avatar Cloud Engine), ця архітектура дозволяє реалізувати механіку "Open-ended Conversation" – відкритого діалогу без заздалегідь прописаних сценаріїв. Гравець може підійти до будь-якого NPC і заговорити з ним голосом або текстом практично на будь-яку тему – від погоди до філософських питань – отримуючи осмислені, контекстно-залежні відповіді, що враховують попередній досвід спілкування, особистість персонажа та поточну ситуацію у грі. Це створює безпрецедентний рівень імерсії та відчуття, що персонажі дійсно "живі" та мають власну свідомість.

1.5.2 Проблеми та економічні обмеження прямої інтеграції LLM

Незважаючи на революційний потенціал та вражаючі демонстрації можливостей, пряма інтеграція великих мовних моделей для повного управління NPC (включаючи як діалоги, так і прийняття рішень про дії) в іграх жанру Idle/Tycoon на даному етапі розвитку технологій є практично неможливою через низку критичних проблем, які роблять такий підхід нежиттєздатним як з технічної, так і з економічної точки зору.

Перша фундаментальна проблема стосується стабільності та явища так званих "галюцинацій" LLM. Великі мовні моделі за своєю природою є імовірнісними системами, які передбачають наступне слово або токен на основі статистичних патернів, вивчених з величезних обсягів текстових даних. Вони не мають вбудованого, глибокого розуміння істини, логіки або правил конкретного ігрового світу. Це робить їх схильними до "галюцинацій" – генерації фактів, тверджень або дій, що суперечать встановленим правилам гри або здоровому глузду. Наприклад, у грі про управління офісом NPC, повністю керований LLM, може раптово "вирішити",

що він звільняється і їде у відпустку на Марс, або що він виконує свою роботу не за комп'ютером, а за допомогою "магії" чи "телепатії". Для жанру Tycoon, який фундаментально базується на чіткій, математично обґрунтованій моделі виробництва та прибутку, де кожна дія має конкретні економічні наслідки, така непередбачуваність є абсолютно фатальною. Стабільність та детермінізм FSM, які гарантують правильність економічних розрахунків, є несумісними з креативним, але хаотичним характером генерації LLM.

Друга проблема стосується технічних обмежень, зокрема затримки відповіді (Latency). Сучасний ігровий процес вимагає реакції системи в реальному часі – зазвичай це означає обробку кожного кадру протягом 16-33 мілісекунд (для досягнення 60 або 30 кадрів на секунду відповідно). Однак хмарні великі мовні моделі, які працюють на віддалених серверах, мають значну та неминучу затримку. Повний цикл обробки запиту включає: підготовку та відправлення запиту через мережу, обробку запиту на серверах провайдера (що може включати чергу запитів від інших користувачів), генерацію токенів (слів) один за одним, та отримання повної відповіді назад через мережу. Цей процес зазвичай займає від 0.5 до 3 секунд, залежно від складності запиту, навантаження на сервери та якості інтернет-з'єднання користувача. Якщо кожен NPC буде чекати 2 секунди перед кожним своїм кроком або рішенням ("Куди мені йти тепер?", "Що мені робити далі?"), гра стане абсолютно неіграбельною – персонажі будуть "застигати" на місці, а гравець відчуватиме постійні лаги та затримки у взаємодії.

Третя, і можливо найвагоміша проблема з точки зору комерційної життєздатності, стосується економічних обмежень та вартості експлуатації (Cost). API-запити до комерційних великих мовних моделей є платними сервісами, які тарифікуються за кількість оброблених токенів (частин слів), як вхідних (prompt), так і вихідних (response). Розглянемо реалістичний розрахунок для типової Idle/Tycoon гри. Припустимо, у грі одночасно активні 100 NPC. Якщо кожен з них приймає рішення або генерує репліку через LLM кожні 10 секунд (що є досить рідко для активної гри), це генерує 10 запитів на секунду, або 600 запитів на хвилину, або 36 000 запитів на годину для одного гравця. Якщо гра має тисячі або десятки тисяч

одночасних гравців, загальна кількість запитів стає астрономічною. Навіть при використанні найдешевших доступних моделей, таких як Gemini Flash від Google, вартість обробки такої кількості запитів швидко перетворює проєкт на глибоко збитковий. Ліміти безкоштовних тарифів (Free Tier), які надають провайдери для тестування, вичерпуються практично миттєво після запуску гри, і розробник змушений переходити на платні тарифи, які можуть коштувати сотні або тисячі доларів щомісяця навіть для невеликої аудиторії.

1.5.3 Теоретичні обмеження великих мовних моделей у контексті ігрових систем

Попри високий потенціал для генерації природних діалогів, великі мовні моделі мають низку фундаментальних обмежень, які не дозволяють використовувати їх як єдине джерело поведінки NPC у реальному часі. Насамперед, LLM є імовірнісними моделями наступного токена: вони оптимізуються на «правдоподібність» тексту, а не на істинність фактів. Дослідження OpenAI та Anthropic (2025) показують, що так звані галюцинації – правдоподібні, але хибні твердження – є не випадковою помилкою, а системною властивістю архітектури трансформера. Модель не має вбудованого механізму перевірки істинності відповідей відносно зовнішнього світу і за умов невизначеності схильна «вигадувати» деталі замість того, щоб сигналізувати про відсутність знань. У контексті ігор це означає, що NPC може повідомляти неіснуючі факти про свій стан, ресурси чи виконання завдань, безпосередньо порушуючи ігрову логіку та економічний баланс.

Друге ключове обмеження пов'язане з контекстним вікном і робочою пам'яттю моделі. Хоча сучасні LLM формально підтримують контексти до сотень тисяч або навіть мільйонів токенів, практичні дослідження показують, що ефективна кількість інформації, з якою модель може одночасно «працювати», набагато менша. Зі збільшенням довжини контексту різко зростають обчислювальні витрати через квадратичну складність механізму self-attention. Додатково спостерігається

деградація якості: при спробі врахувати надто багато фактів модель починає «забувати» важливі деталі або плутати їх між собою. Для NPC це означає, що довгострокова пам'ять про історію взаємодій з гравцем, стосунки з іншими персонажами та змінні економічного стану не може зберігатися всередині самої моделі і потребує зовнішніх механізмів пам'яті та вибірки релевантного контексту, що ускладнює архітектуру й додає затримку.

Третя фундаментальна проблема – затримка (latency) та вартість викликів LLM в онлайні. При 60 FPS гра має усього 16,67 мс на кадр, тоді як навіть оптимізовані LLM-сервіси демонструють час до першого токена на рівні сотень мілісекунд і повну затримку відповіді в діапазоні від 1 до 5 секунд. Емпіричні дослідження інтеграції GPT-4 Turbo у VR-проекти фіксують сумарну затримку близько 7 секунд з урахуванням перетворення мовлення, мережевої передачі та синтезу звуку. Порівняно з бюджетом кадру це означає, що одна відповідь LLM «з'їдає» десятки або сотні кадрів, що несумісно з вимогами до миттєвої реакції у більшості ігрових ситуацій. Єдиний практичний підхід – приховувати цю затримку (Latency Hiding) через асинхронні запити у фоновому режимі, використовуючи LLM лише для другорядних, не критичних за часом задач (діалоги, коментарі, flavour-репліки), тоді як усі економічно та геймплейно важливі рішення залишаються за детермінованим ядром.

Сукупність цих факторів пояснює, чому LLM доцільно розглядати не як заміну класичного ігрового AI, а як окремий комунікативний шар, що працює поверх надійної детермінованої логіки. Саме така гібридна роль LLM – генерація текстових реакцій при збереженні жорсткого контролю за станами NPC на рівні FSM – є базовою передумовою запропонованої у цій роботі архітектури.

1.6 Гібридна архітектура III

На основі проведеного всебічного аналізу сучасних підходів до моделювання поведінки NPC, вимог жанру Idle/Tycoon та обмежень генеративних методів,

виявлено фундаментальну науково-технічну прогалину (Research Gap), яка і визначає напрямок даного дослідження.

Існує глибокий, практично нерозв'язний конфлікт між двома критично важливими вимогами до ігрових систем. З одного боку, існує вимога стабільності та передбачуваності, яка є абсолютно необхідною для коректної роботи економічної симуляції, де кожна дія персонажа має конкретні математичні наслідки для балансу гри. З іншого боку, існує вимога реалізму, варіативності та емоційної глибини взаємодії, яка є необхідною для залучення гравця та створення відчуття занурення у живий, динамічний світ. Суто алгоритмічні методи, такі як скінченні автомати (FSM) або дерева поведінки (BT), чудово забезпечують стабільність, передбачуваність та дешевизну експлуатації, оскільки всі обчислення відбуваються локально без зовнішніх запитів. Однак вони створюють нудних, одноманітних, "роботизованих" персонажів, які повторюють одні й ті ж фрази та дії, що призводить до явища "імерсивного розриву" – гравець перестає сприймати персонажів як живих істот і бачить у них лише механічні автомати. Навпаки, суто генеративні методи на основі великих мовних моделей (LLM) забезпечують вражаючий реалізм, безмежну варіативність діалогів та здатність до імпровізації, але є нестабільними (схильні до "галюцинацій"), повільними (затримка 0.5-3 секунди) та економічно невиправданими для управління базовою логікою через високу вартість API-запитів.

Вирішенням цієї фундаментальної суперечності є розробка оригінальної гібридної архітектури, яка стратегічно використовує сильні сторони обох підходів і систематично нівелює їхні недоліки через чітке розділення відповідальності. Ключова ідея полягає в тому, щоб детермінований алгоритм (FSM) відповідав за критичні, економічно важливі рішення ("Що робити?", "Коли працювати?", "Куди йти?"), тоді як генеративна модель (LLM) відповідала виключно за комунікативний шар ("Як висловити своє незадоволення?", "Як відреагувати на запитання гравця?"). Таке розділення гарантує, що економічна модель залишається стабільною та передбачуваною, але при цьому персонажі здатні спілкуватися природно, емоційно та унікально [4].

Метою даної роботи є розробка та повна програмна реалізація удосконаленої гібридної системи штучного інтелекту для неігрових персонажів в іграх жанру Idle/Tycoon. Ця система повинна одночасно виконувати кілька ключових вимог. По-перше, вона повинна використовувати детермінований скінченний автомат (FSM) у поєднанні з системою динамічної навігації NavMesh для управління всіма критичними діями персонажів, включаючи дотримання робочого розкладу, просторове переміщення, виконання робочих завдань та генерацію економічних ресурсів. Це забезпечить абсолютну стабільність економічної моделі та нульову вартість експлуатації в основному ігровому циклі, оскільки всі ці операції виконуються локально без зовнішніх запитів. По-друге, система повинна інтегрувати велику мовну модель (LLM) виключно для комунікативного шару, причому ця інтеграція повинна активуватися лише у специфічних, чітко визначених станах FSM, таких як стан "Реакція на відсутність робочого місця" або "Діалог з гравцем". Це дозволить радикально мінімізувати кількість API-запитів (і, відповідно, витрати), зберігши при цьому високий рівень імерсивності саме там, де це найбільш важливо для гравецького досвіду. По-третє, система повинна реалізувати механізм динамічного розподілу ресурсів, що дозволяє NPC адаптуватися до змін ігрового середовища у реальному часі, наприклад, коли гравець будує нові об'єкти, переставляє меблі або видаляє робочі місця. Така адаптивність є критичною для жанру Tycoon, де гравець постійно змінює конфігурацію світу.

Висновки до розділу 1

У першому розділі було проведено комплексний аналіз сучасного стану технологій штучного інтелекту в ігровій індустрії з метою виявлення можливостей та обмежень існуючих підходів до моделювання поведінки неігрових персонажів у контексті економічних симуляторів жанру Idle/Tycoon.

На початку аналізу було визначено роль штучного інтелекту у розробці ігор (GameDev) та проведено чітке розмежування між інструментами, призначеними

для допомоги розробникам на етапі створення гри (Editor AI, такі як Unity Muse для генерації коду та ассетів), та системами, які безпосередньо функціонують під час виконання гри і відповідають за поведінку персонажів (Runtime AI). Встановлено, що саме Runtime AI є предметом дослідження у даній роботі, оскільки ці системи безпосередньо впливають на ігровий досвід користувача та визначають якість взаємодії з віртуальним світом.

Детальний аналіз специфіки жанру Idle/Tycoon виявив унікальні вимоги до поведінки неігрових персонажів, які суттєво відрізняються від інших ігрових жанрів. Встановлено, що ключовими вимогами до NPC у цьому жанрі є детермінованість поведінки, яка гарантує передбачуваність економічного прибутку та дозволяє гравцю планувати стратегію розвитку; масштабованість системи, що забезпечує можливість підтримки сотень або навіть тисяч агентів одночасно без критичного зниження продуктивності; та автономність персонажів, які мають самостійно виконувати свої функції без постійного втручання гравця. Ці вимоги формують жорсткі рамки для вибору архітектури штучного інтелекту.

Проведено всебічний порівняльний аналіз основних архітектурних патернів для моделювання поведінки NPC, включаючи скінченні автомати (FSM), дерева поведінки (BT), планування орієнтоване на цілі (GOAP) та системи корисності (Utility AI). На основі критеріїв детермінованості, продуктивності та простоти реалізації зроблено обґрунтований висновок, що для базового управління економічним циклом персонажів найбільш оптимальним підходом є Скінченний Автомат завдяки його мінімальному навантаженню на центральний процесор, абсолютній передбачуваності поведінки та можливості ефективної симуляції великої кількості агентів одночасно. FSM забезпечує необхідний фундамент стабільності для економічної моделі гри.

Окремо було досліджено потенціал та обмеження сучасних генеративних методів штучного інтелекту на основі великих мовних моделей (LLM). Аналіз продемонстрував, що незважаючи на революційні можливості створення реалістичних діалогів та емерджентної поведінки, пряма заміна традиційного алгоритмічного ШІ на повністю генеративний підхід є принципово неможливою на сучасному етапі

розвитку технологій. Виявлено три критичні перешкоди: проблема "галюцинацій", коли модель генерує факти або дії, що суперечать логіці гри та можуть зруйнувати економічний баланс; значна затримка відповіді (latency) від 0.5 до 3 секунд, що робить неможливим використання LLM для прийняття рішень у реальному часі; та висока вартість API-запитів за токени, яка робить масове використання LLM для сотень персонажів економічно невиправданим та збитковим для комерційних проєктів.

На основі виявлених переваг та недоліків обох підходів було сформульовано концепцію оригінальної гібридної архітектури, яка стратегічно поєднує надійність детермінованих алгоритмів з виразністю генеративних моделей. У цій архітектурі скінченний автомат (FSM) виступає як надійний "скелет" або детерміноване ядро поведінки, відповідаючи за всі критичні економічні рішення, навігацію та виконання завдань, тоді як велика мовна модель (LLM) функціонує як "соціальний інтерфейс" або комунікативний шар, що активується лише у специфічних ситуаціях для генерації унікальних, емоційно насичених реплік. Такий підхід дозволяє трансформувати сприйняття NPC гравцем: замість абстрактного "ресурсу в таблиці" персонаж стає "живим співробітником", здатним унікально та природно реагувати на дії гравця (наприклад, скаржитися на погані умови праці, висловлювати задоволення або незадоволення), при цьому не порушуючи математичний баланс гри та не створюючи надмірного навантаження на систему або бюджет розробника.

Результати проведеного аналізу формують міцний теоретичний фундамент для подальшої роботи. Детальне проєктування архітектури запропонованої гібридної системи, включаючи математичне моделювання переходів станів, алгоритми розподілу ресурсів та методику контекстної генерації промтів, буде всебічно розглянуто у наступному розділі дослідження.

РОЗДІЛ 2. ПРОЄКТУВАННЯ УДОСКОНАЛЕНОЇ ГІБРИДНОЇ СИСТЕМИ ІІІ ДЛЯ УПРАВЛІННЯ ПОВЕДІНКОЮ NPC

2.1 Концептуальна архітектура, логічне розмежування та FSM-моделювання

Проектування архітектури системи базується на принципі двошарової (Dual-Layer) моделі, що забезпечує чітке розмежування відповідальності (Separation of Concerns). Це удосконалення стандартної FSM-архітектури, оскільки Генеративний Шар (LLM) є ізольованою, неблокуючою надбудовою, що не впливає на основну логіку дій [1].

Система функціонує на трьох взаємозалежних рівнях: Стратегічному (FSM), Тактичному (NavMesh) та Комунікативному (LLM). Це ієрархія рівнів та логічне розмежування, де Стратегічний Рівень (NPCBrain.cs) формує Детерміноване Ядро (Стабільність економіки), Тактичний Рівень (NPCActions.cs) – Виконавчий Механізм (Ефективність руху), а Комунікативний Рівень (APIManager.cs) – Генеративну Надбудову (Імерсивність). Детальна функціональна структура гібридної системи представлена у Таблиці 2.1.

Таблиця 2.1 – Функціональне розмежування рівнів гібридної системи ІІІ

Рівень	Скрипт (Компонент)	Функціональна Відповідальність	Роль у системі
Стратегічний (FSM Core)	NPCBrain.cs	Визначення глобальної мети, керування станами, ієрархія пріоритетів.	Детерміноване Ядро (Стабільність)
Тактичний (Spatial AI)	NPCActions.cs (NavMesh)	Фізичне виконання рішень, пошук шляху, динамічне обходження перешкод.	Виконавчий Механізм (Ефективність)
Комунікативний (LLM)	APIManager.cs (Gemini API)	Генерація унікальних, контекстно-залежних реплік.	Генеративна Надбудова (Імерсивність)

FSM: Логіка управління станами та ієрархія пріоритетів. Стратегічний Рівень реалізує FSM, який керується принципом ієрархії пріоритетів (`NPCBrain.MakeDecision()`). Цей принцип є критичним для жанру Тусооп, оскільки він гарантує, що економічний цикл не буде порушений непередбачуваними подіями. FSM моделює життєвий цикл працівника через 9 станів, візуалізованих на діаграмі станів (Рисунок 2.1).

Ключові принципи логіки переходів забезпечують стабільність: 1) Часовий імператив (Вищий пріоритет): Глобальний час, що керується `DayNightCycle.cs`, примусово переводить NPC у стан `GOING_HOME`, незалежно від його поточної потреби. 2) Ресурсний імператив: Перехід у `WORKING` можливий лише після успішного захоплення ресурсу. Якщо захоплення не вдалося (брак столів), FSM переходить у стан обробки виключень `GOING_TO_PLAYER`.

Моделі просторового інтелекту та динамічного розподілу ресурсів. Тактичний рівень та логіка ресурсів забезпечують фізичну реалістичність та економічну цілісність симуляції. Моделювання просторового інтелекту виконано через Unity `NavMesh`, який використовує алгоритм A^* для пошуку оптимального шляху. Критичним є механізм `NavMesh Carving` (вирізання сітки), який завдяки компоненту `NavMeshObstacle` на розміщуваних об'єктах змушує `NavMesh` перебудовувати граф у реальному часі, що дозволяє NPC адаптуватися до змін, які вносить гравець. Для запобігання конфліктам ресурсів (`Resource Contention`) використовується модель "розумного ресурсу" (`WorkstationPoint.cs`). Де функція `Claim(Transform npc)` в `WorkstationPoint.cs` гарантує атомарність захоплення. Найважливіше, якщо захоплення не вдається, функція `FindAndClaimWorkstation()` повертає `False`, що ініціює перехід FSM у стан `GOING_TO_PLAYER`. Таким чином, системний збій (брак ресурсів) конвертується в імерсивну комунікативну подію.

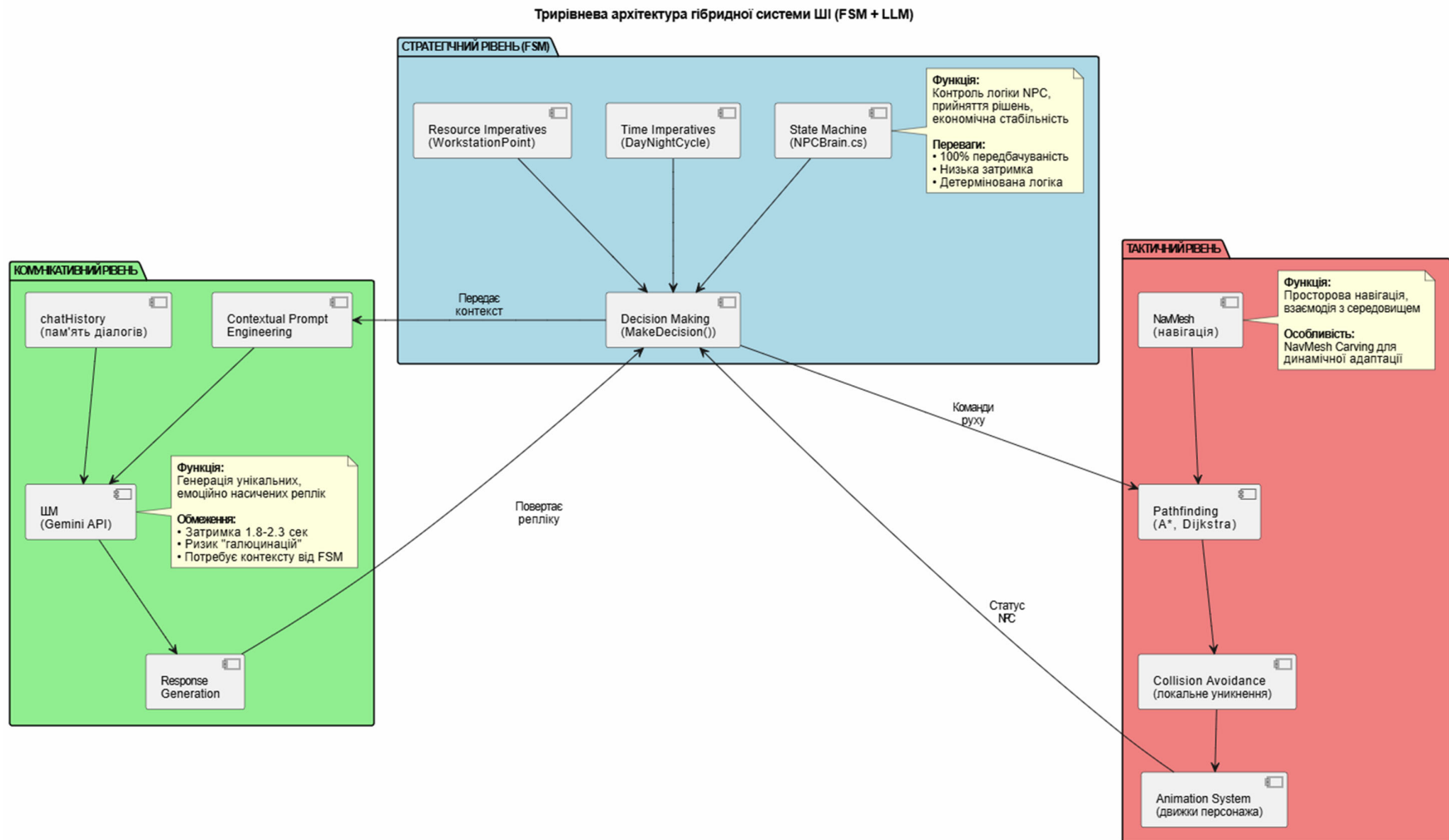


Рисунок 2.1 – Діаграма станів FSM NPC у жанрі Idle/Tycoon

2.1.1 Формальний опис скінченного автомата NPC

Для забезпечення математичної строгості та верифікованості поведінки NPC, формалізуємо систему керування як детермінований скінченний автомат (DFA – Deterministic Finite Automaton). Згідно з класичним визначенням теорії автоматів, FSM визначається як п'ятірка:

$$M = (S, E, \delta, s_0, F) \quad M = (S, E, \delta, s_0, F)$$

де компоненти визначаються наступним чином:

Множина станів S складається з 9 основних станів поведінки:

$S = \{ \text{IDLE}, \text{AT_HOME_WAITING}, \text{GOING_TO_WORK}, \text{SEEKING_WORKSTATION}, \text{WORKING}, \text{GOING_TO_COFFEE}, \text{RESTING}, \text{GOING_TO_PLAYER}, \text{REPORTING_TO_PLAYER}, \text{GOING_HOME} \}$
 $S = \{ \text{IDLE}, \text{AT_HOME_WAITING}, \text{GOING_TO_WORK}, \text{SEEKING_WORKSTATION}, \text{WORKING}, \text{GOING_TO_COFFEE}, \text{RESTING}, \text{GOING_TO_PLAYER}, \text{REPORTING_TO_PLAYER}, \text{GOING_HOME} \}$
 $S = \{ \text{IDLE}, \text{AT_HOME_WAITING}, \text{GOING_TO_WORK}, \text{SEEKING_WORKSTATION}, \text{WORKING}, \text{GOING_TO_COFFEE}, \text{RESTING}, \text{GOING_TO_PLAYER}, \text{REPORTING_TO_PLAYER}, \text{GOING_HOME} \}$

Кожен стан $s \in S$ відповідає дискретному режиму поведінки NPC з унікальним набором дозволених дій та анімацій. Наприклад, у стані WORKING персонаж виконує анімацію роботи та генерує дохід; у стані RESTING персонаж сидить та відновлює енергію.

Множина подій E генеруються ігровими підсистемами:

$E = \{ \text{TIME_MORNING}, \text{TIME_EVENING}, \text{ARRIVED_AT_DESTINATION}, \text{WORKSTATION_FOUND}, \text{WORKSTATION_NOT_FOUND}, \text{WORK_COMPLETED}, \text{ENERGY_LOW}, \text{ENERGY_RESTORED}, \text{PLAYER_INTERACTION_STARTED}, \text{PLAYER_INTERACTION_ENDED}, \text{REPORT_DELIVERED} \}$
 $E = \{ \text{TIME_MORNING}, \text{TIME_EVENING}, \text{ARRIVED_AT_DESTINATION}, \text{WORKSTATION_FOUND}, \text{WORKSTATION_NOT_FOUND}, \text{WORK_COMPLETED}, \text{ENERGY_LOW}, \text{ENERGY_RESTORED}, \text{PLAYER_INTERACTION_STARTED}, \text{PLAYER_INTERACTION_ENDED}, \text{REPORT_DELIVERED} \}$
 $E = \{ \text{TIME_MORNING}, \text{TIME_EVENING}, \text{ARRIVED_AT_DESTINATION}, \text{WORKSTATION_FOUND}, \text{WORKSTATION_NOT_FOUND}, \text{WORK_COMPLETED}, \text{ENERGY_LOW}, \text{ENERGY_RESTORED}, \text{PLAYER_INTERACTION_STARTED}, \text{PLAYER_INTERACTION_ENDED}, \text{REPORT_DELIVERED} \}$

Події походять від:

- Таймера доби (DayNightCycle): TIME_MORNING, TIME_EVENING;
- Системи навігації (NavMesh): ARRIVED_AT_DESTINATION;
- Менеджера ресурсів (WorkstationPoint): WORKSTATION_FOUND, WORKSTATION_NOT_FOUND;
- Монітору енергії NPC: ENERGY_LOW, ENERGY_RESTORED;
- Системи взаємодії з гравцем: PLAYER_INTERACTION_STARTED/ENDED, REPORT_DELIVERED.

Функція переходів $\delta: S \times E \rightarrow S$ визначає детерміновану логіку зміни станів.

Ключові переходи формалізовані у таблиці 2.2:

Таблиця 2.2 – Функція переходів δ для FSM NPC

Поточний стан (s)	Подія (e)	Наступний стан $\delta(s,e)$	Умова/Пояснення
AT_HOME_WAITING	TIME_MORNING	GOING_TO_WORK	Ранок, робочий день починається
GOING_TO_WORK	ARRIVED_AT_DESTINATION	SEEKING_WORKSTATION	NPC дійшов до офісу
SEEKING_WORKSTATION	WORKSTATION_FOUND	WORKING	Знайдено вільне робоче місце
SEEKING_WORKSTATION	WORKSTATION_NOT_FOUND	GOING_TO_PLAYER	Немає вільних місць, скарга до гравця
WORKING	ENERGY_LOW	GOING_TO_COFFEE	Втомлення після роботи
WORKING	TIME_EVENING	GOING_HOME	Кінець робочого дня

Продовження таблиці 2.2

GOING_TO_COFFEE	ARRIVED_AT_DESTINATION	RESTING	NPC прибув до кав'ярні
RESTING	ENERGY_RESTORED	SEEKING_WORKSTATION	Енергія відновлена, повернення до роботи
GOING_TO_PLAYER	ARRIVED_AT_DESTINATION	REPORTING_TO_PLAYER	NPC стоїть перед гравцем
REPORTING_TO_PLAYER	REPORT_DELIVERED	SEEKING_WORKSTATION	Скарга передана, повернення до пошуку місця
Any state except REPORTING	TIME_EVENING -IN_CONVERSATION	GOING_HOME	Кінець дня, якщо не в розмові

У класичній нотації теорії автоматів модель NPC описується п'ятіркою $M = (S, E, \delta, s_0, F)$, де S є множиною дискретних станів агента, E – множиною подій, $\delta : S \times E \rightarrow S$ – детермінованою функцією переходів, $s_0 = AT_HOME_WAITING$ – початковим

станом, що відповідає перебуванню персонажа вдома на початку ігрового дня, а $F = \emptyset$ – порожньою множиною фінальних станів, оскільки поведінка NPC у жанрі Idle/Tycoon є циклічною і не має термінального завершення.

Кожен стан $s \in S$ відповідає дискретному режиму поведінки NPC з унікальним набором дозволених дій та анімацій. У стані WORKING персонаж виконує анімацію роботи та генерує дохід; у стані RESTING він сидить у зоні відпочинку та відновлює енергію; у станах GOING_TO_WORK, GOING_HOME та GOING_TO_COFFEE задіюється навігація по NavMesh. Події $e \in E$ генеруються різними підсистемами гри: циклом доби (TIME_MORNING, TIME_EVENING), системою навігації (ARRIVED_AT_DESTINATION), менеджером робочих місць (WORKSTATION_FOUND, WORKSTATION_NOT_FOUND), системою енергії (ENERGY_LOW, ENERGY_RESTORED) та підсистемою взаємодії з гравцем (PLAYER_INTERACTION_STARTED, REPORT_DELIVERED).

Функція переходів δ однозначно задає, у який наступний стан переходить агент при настанні певної події. Фрагмент цієї функції наведено в таблиці 2.2 і відображає ключові сценарії: перехід з AT_HOME_WAITING у GOING_TO_WORK при настанні ранку, перехід з SEEKING_WORKSTATION у WORKING при знаходженні вільного столу або у GOING_TO_PLAYER при його відсутності, повернення у GOING_HOME при завершенні робочого дня тощо. Важливо, що для кожної пари (s, e) визначений не більше ніж один наступний стан, тобто $\delta(s, e)$ є функцією, а не відношенням.

Це дозволяє сформулювати властивість детермінізму: за фіксованого початкового стану s_0 та фіксованої послідовності подій $E^* = (e_1, e_2, \dots, e_n)$ траєкторія $(s_0, s_1, \dots, s_n)$, де $s_{i+1} = \delta(s_i, e_i)$, є єдиною. Іншими словами, якщо гравець повторить ті самі дії в тих самих умовах, NPC виконуватимуть абсолютно ідентичну послідовність станів. Для економічного симулятора це означає, що обсяг виконаної роботи, кількість обслугованих клієнтів та інші ключові показники можуть

бути математично передбачені на основі моделі, а не залежать від імовірнісних факторів.

Відсутність фінальних станів ($F = \emptyset$) відображає специфіку жанру Idle/Tycoon: NPC не “помирають” і не виходять із системи, а нескінченно багато разів повторюють цикл «дім $\rightarrow$ робота $\rightarrow$ відпочинок $\rightarrow$ дім». Така побудова дозволяє інтерпретувати FSM як нескінченний процес, що підтримує стабільний економічний потік у грі, і водночас спрощує аналіз інваріантів – зокрема, інваріанта економічної коректності, за яким загальний час перебування у стані WORKING за один ігровий день є детермінованою функцією від наявності ресурсів (кількості робочих місць) та розкладу дня, але не від випадкових рішень алгоритму.

У програмній реалізації цієї моделі у класі NPCBrain.cs таблиця переходів δ кодується через словник (dictionary), де ключем виступає пара (поточний стан, подія), а значенням – наступний стан. Такий підхід полегшує верифікацію повноти та коректності визначення δ , а також підтримує розширення моделі без перетворення коду FSM на “спагеті-логіку”.

2.2 Проєктування модуля генеративної комунікації на основі LLM та імерсивних механізмів

Комунікативний Рівень є Генеративною Надбудовою, яка реалізує удосконалений метод Contextual Prompt Engineering, що є ключовою науковою новизною роботи. Цей модуль є містком між детермінованим FSM-ядром та імерсивністю, необхідною для досвіду користувача.

Модель формування персоналізованого контексту (Role-based Prompting) побудована на конкатенації факторів, інтегрованих у NPCBrain.cs, для підвищення якості та варіативності відповідей LLM. По-перше, впроваджується Персона (Role), яка задається змінною PersonaPromptPrefix на рівні окремого NPC. Це дозволяє призначати NPC індивідуальні риси, такі як "лінивий" або "гіперактивний" працівник, що є критичним для внесення варіативності в імерсивність. По-друге,

використовується Обмеження (Constraint) – жорстке обмеження довжини до 20 символів. Це є економічним імперативом, оскільки мінімізує Latency та API-витрати, що є критичним для масштабованості. Наочна Візуалізація NPC із згенерованою фразою у "хмарці" (Рисунок 2.2) відображає успішну інтеграцію Генеративного Шару.



Рисунок 2.2 – Ілюстрація Комуникативного Рівня: NPC з відображенням згенерованої фрази (Speech Bubble)

Для вирішення проблеми високої затримки (Latency) LLM та забезпечення безпеки API-ключа, використовується архітектура Проксі-Сервера на базі Google Apps Script (GAS). Ключові аспекти архітектури включають Асинхронність, де `APIManager.cs` використовує `C#` Корутини для виконання мережевого запиту без блокування, підтримуючи FPS. Безпека та Економіка забезпечуються Проксі-

Сервером (GAS), який приховує API-ключ та контролює використання моделі `gemini-2.5-flash-lite`, оптимізованої для швидкості та низької вартості.

Окрім ключової LLM-архітектури, для подальшого підвищення імерсивності в проєкті реалізовано додаткові паралельні механізми, які не впливають на FSM, зберігаючи пріоритет економіки. Перш за все, впроваджено Реакцію на освітлення (Environmental Feedback), де скрипт `FlickeringLight.cs` генерує подію, на яку підписується `NPCReactToLight.cs`. NPC запускає анімаційний тригер "LookAtLight", імітуючи людську реакцію. Цей паралельний, неблокуючий механізм забезпечує візуальну динаміку, підтримуючи при цьому головний пріоритет Тусооп-ігри – економічну стабільність. По-друге, реалізовано механізм Імерсивного діалогу (Camera and UI Focus), що є ключовим для ігрового досвіду. При взаємодії з NPC через `PlatformDialogue.cs` відбувається блокування руху гравця (`PlayerMovement.SetMovementState(false)`) та активація `ZoomIn` камери (`CameraFollow.ZoomIn()`), фокусуючи увагу гравця на NPC та його згенерованій LLM-фразі. Додатково, для підтримки імерсивності, компонент `IncomePlatform.cs` використовує `FloatingTextAnimator.cs` для відображення доходу, перетворюючи абстрактний економічний прибуток на візуальну та емоційну винагороду для гравця.

Удосконалена гібридна архітектура відкриває значні перспективи подальшого розвитку. Подальше удосконалення може включати додавання Емоційної динаміки (Affective Computing), де на основі негативних згенерованих LLM-фраз можна динамічно знижувати внутрішній показник Настрою NPC. Падіння Настрою може, своєю чергою, знижувати продуктивність праці, створюючи глибший та складніший економічний вплив соціального ІІІ. Крім того, перспективним напрямком є реалізація Багатоагентної соціальної взаємодії, де LLM буде використовуватися для генеративної комунікації між NPC ("Що ти думаєш про боса?"). Це вимагатиме імплементації Memory Stream (як у концепції Generative Agents) та розширить соціальну динаміку світу, забезпечуючи новий рівень непередбачуваності та імерсивності.

2.3 Моделювання емерджентної поведінки та архітектура LLM-проксі

Цей підрозділ присвячений глибокому теоретичному обґрунтуванню Комунікативного Рівня у системі керування NPC, орієнтованої на моделювання емерджентної (незапрограмованої) поведінки в соціальних взаємодіях. Саме цей рівень дозволяє суттєво зменшити "іммерсивний розрив" – явище, що виникає тоді, коли гравець усвідомлює штучність і одноманітність взаємодії з персонажами. На відміну від традиційних детермінованих систем, наприклад FSM, новий підхід передбачає використання імовірнісних моделей, здатних породжувати несподівані, оригінальні реакції NPC, які не були явно прописані розробниками.

Моделювання емерджентної поведінки реалізується завдяки інтегрованій взаємодії FSM та LLM. FSM у цій структурі виконує функцію "детермінанта причин" – він фіксує ситуацію в ігровому світі, наприклад, виникнення браку ресурсу або конфлікту, і генерує тригер для активації LLM. Далі LLM, отримавши контекст від FSM, генерує унікальний вербальний вираз NPC, що якісно відрізняється від типового шаблонного повідомлення. Наприклад, у разі нестачі ресурсу, кожен NPC може сформулювати оригінальну скаргу, яка відображає його персональні риси, емоційний стан або попередню історію взаємодії із гравцем. Це надає кожній взаємодії характеру новизни й емоційності, формуючи багатий соціальний пласт ігрового світу. Такий підхід дозволяє, з одного боку, утримувати стійкість і цілісність економіки гри за рахунок контролюючої ролі FSM, а з іншого – створювати багатогранну іммерсивність завдяки варіативним діалогам.

У зв'язку із затримками роботи LLM (Latency) виникає гостра потреба у технічній архітектурі, яка не лише захистить ігровий цикл від простою, а й забезпечить безпеку API-ключів. Розроблено концепцію Проксі-Сервера на основі Google Apps Script (GAS), що працює як проміжний прошарок між клієнтським застосунком (наприклад, Unity) і хмарним сервісом LLM (наприклад, Gemini API). Така архітектура

дозволяє коректно розподіляти навантаження й здійснювати взаємодію із LLM у захищеному асинхронному режимі.

Ключовим теоретичним рішенням є впровадження асинхронної обробки запитів, яка унеможливорює блокування Game Loop у Тусооп-іграх із сотнями NPC. Асинхронність, яку забезпечує Проксі-Сервер, дає змогу клієнтському застосунку надсилати запит до LLM та негайно продовжувати роботу без очікування відповіді, що літературно описується як Latency Hiding. Цей підхід не тільки зберігає цілісність і динаміку ігрового процесу (навіть за високого навантаження), а й забезпечує рівномірний розподіл ресурсів, даючи змогу оптимізувати час відповіді для всіх NPC.

Економічна й безпекова перевага архітектури Проксі-Сервера полягає у тому, що API-ключ зберігається виключно на серверній стороні, уникаючи ризику його компрометації через клієнтський застосунок. Проксі-сервер забезпечує централізований контроль за частотою й структурою запитів, що дозволяє використовувати оптимізовану LLM-модель (наприклад, gemini-2.5-flash-lite), зберігаючи високу швидкодію й мінімізуючи фінансові витрати на роботу із LLM.

Для наочної демонстрації взаємодії між детермінованим ядром (FSM) та генеративним шаром (LLM) у розробленій гібридній архітектурі представлено повну блок-схему потоку обробки (рис. 2.3). Ця схема ілюструє весь цикл прийняття рішення: від перевірки критичних тригерів у NPCBrain.cs, через формування контекстного промпту, асинхронний запит до APIManager.cs, обробку в Gemini API, до відображення згенерованої фрази персонажем. Ключовою особливістю є принцип Latency Hiding – поки LLM обробляє запит (від 1.8 до 2.3 секунд), основний ігровий цикл продовжує роботу без перебоїв, забезпечуючи плавність гри.

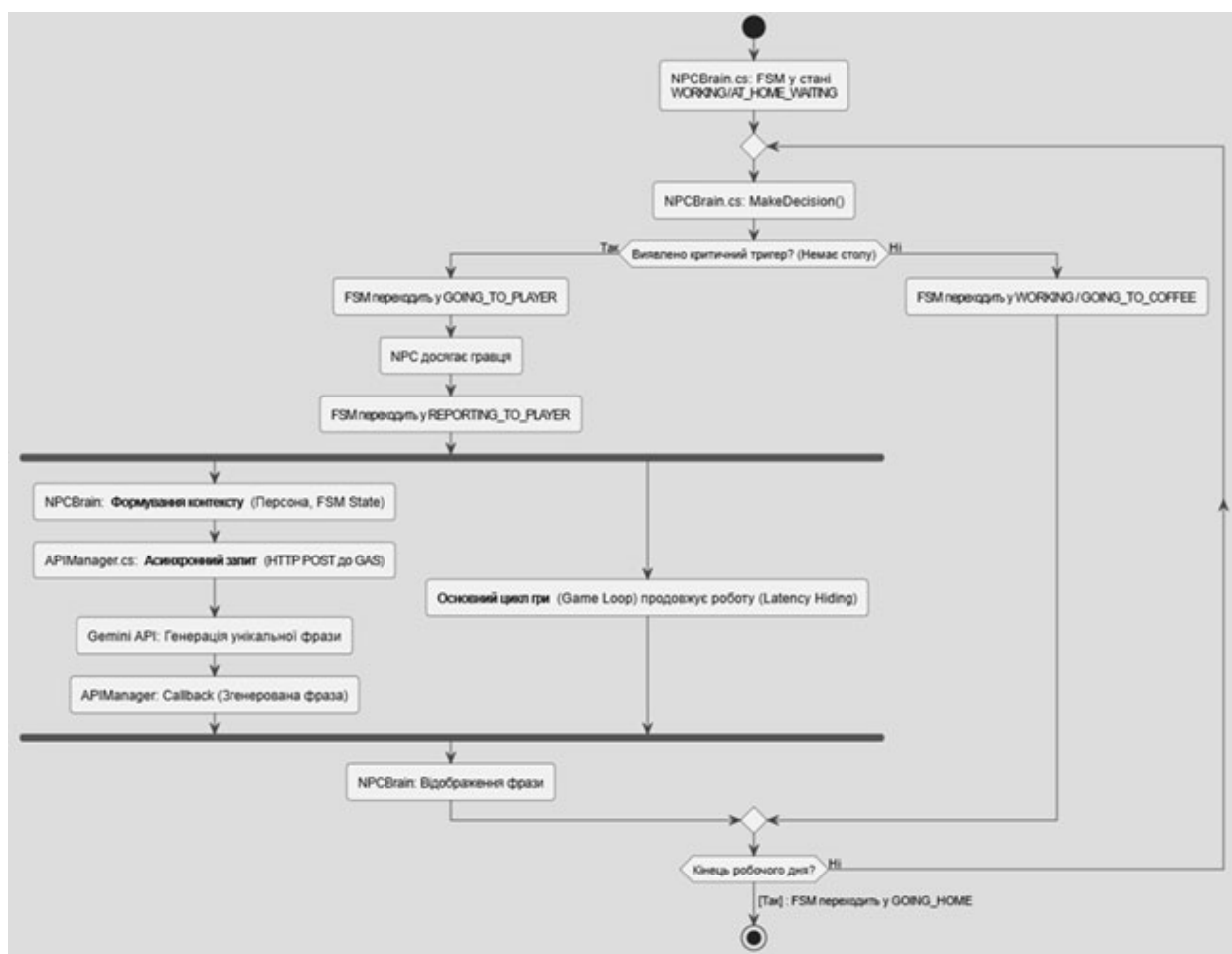


Рисунок 2.3 – Блок-схема потоку гібридного ШІ (FSM до LLM) з асинхронною обробкою запитів через проксі-сервер Google Apps Script

Як видно зі схеми, процес починається з NPCBrain.cs у стані WORKING або AT_HOME_WAITING. Метод MakeDecision() визначає, чи настав критичний тригер (наприклад, немає вільного столу). Якщо так (Так), FSM переходить у стан GOING_TO_PLAYER, NPC досягає гравця й активується стан REPORTING_TO_PLAYER. На цьому етапі NPCBrain формує контекстний промпт (персона, FSM-стан, час), який передається до APIManager.cs. APIManager.cs виконує асинхронний HTTP POST-запит до проксі-сервера (Google Apps Script), звідки запит пересилається до Gemini API для генерації унікальної фрази. Поки відбувається обробка запиту (від 1.8 до 2.3 секунд), основний цикл гри продовжує роботу завдяки принципу Latency Hiding – це критично важливо для збереження плавності

гри. Після отримання відповіді через callback, NPCBrain відображає згенеровану фразу у діалоговій хмарці. Якщо критичний тригер не спрацював (Hi), FSM може перейти у стан WORKING або GOING_TO_COFFEE (відпочинок), і цикл повторюється. Наприкінці робочого дня (Так, кінець дня) FSM автоматично переходить у стан GOING_HOME, і персонаж покидає офіс до наступного ранку.

Така архітектура гарантує, що навіть якщо LLM згенерує некоректну або нерелевантну фразу, це ніяк не вплине на фактичну поведінку NPC – він продовжить виконувати свої економічні функції згідно з детермінованою логікою FSM. Це вирішує фундаментальну проблему інтеграції генеративних моделей у ігри реального часу. Водночас LLM не має доступу до механізму прийняття рішень у грі – ця сфера повністю ізольована й регулюється детермінованою FSM-логікою, що гарантує стабільність симуляції та ігрового балансу незалежно від складності емерджентної поведінки NPC.

2.4 Принципи контекстної генерації промтів та персоніфікації NPC

Ефективність роботи Генеративного Шару в гібридній архітектурі безпосередньо залежить від того, наскільки якісно FSM готує контекст для LLM. Цей підрозділ докладно аналізує принципи побудови промтів (prompt engineering), які забезпечують глибокий рівень персоніфікації NPC і релевантність їхніх унікальних реплік у грі. Від якості передачі інформації між FSM і LLM залежить не лише відповідність виразів NPC поточному ігровому стану, а й загальний рівень природності та різноманітності ігрового досвіду гравця.

Перший і ключовий принцип – Принцип Рольової Настанови (PersonaPrompting). Тут будь-який промт містить статичний префікс, який визначає роль, характер, соціальний статус NPC, а також базові риси його поведінки. Такий підхід дозволяє кожному NPC мати власний унікальний стиль спілкування, навіть якщо всі вони реагують на одну й ту ж саму подію, наприклад, на брак робочого місця чи перевтому. Завдяки цьому обмеження FSM на різноманітність реакцій

компенсується можливістю LLM створювати неповторні вирази, що надають NPC яскравих особистісних рис. Отже, зростає імерсивність і реалістичність вражень від взаємодії.

Другий принцип – Принцип Актуалізації Контексту (FSM-Contextualization). Він передбачає динамічне наповнення промту релевантною інформацією про поточний ігровий стан, яку формує FSM. Наприклад, під час стану `REPORTING_TO_PLAYER` FSM додає у промт дані про дефіцит певного ресурсу або результат дії гравця. Це гарантує, що відповіді LLM будуть логічно обґрунтованими, специфічними й уникають так званих "галюцинацій", тобто некоректної або вигаданої інформації, яка руйнує економічну логіку симулятора. Важливою складовою є інтеграція короткострокової пам'яті (Memory Stream) – функціоналу, що дозволяє LLM враховувати контекст останніх 3–5 повідомлень. Це прискорює розвиток багатокрокових діалогів між NPC і гравцем, підвищує логічну цілісність спілкування та наближає поведінку NPC до реальних соціальних агентів.

Третій принцип – Принцип Обмеження Вихідного Формату (Economic Constraint). Він полягає у жорсткому визначенні максимальної довжини відповіді LLM (наприклад, до 20 символів), що дає комплексний ефект: зменшується Latency, зберігаються фінансові ресурси, а відповіді NPC залишаються лаконічними, емоційно насиченими але не перевантаженими деталями. Така стисливість сприяє кращому користувацькому досвіду, оскільки фрази NPC органічно виглядають у вікнах діалогів або "хмарках", не порушуючи ритму геймплею.

Загальний теоретичний висновок полягає в тому, що синтезування FSM-логіки та LLM-генерації на основі зазначених принципів дозволяє забезпечити і логічну структурованість, і природну емоційність реплік NPC. FSM жорстко контролює конфігурацію подій (хто є суб'єктом і об'єктом дії), а LLM – формулює реакцію у відповідному стилі й з потрібною емоційною тональністю. Таким чином, NPC справляють враження інтелектуально розвинених агентів при мінімальних обчислювальних затратах та збереженні стабільності економічної симуляції у жанрі Idle/Tycoon.

Висновки до розділу 2

У другому розділі було здійснено комплексне теоретичне проектування удосконаленої гібридної системи штучного інтелекту, що забезпечує синергію детермінованих та генеративних методів. На основі проведеної роботи було сформовано всі необхідні теоретичні розробки, методи та алгоритми, які є основою для практичної реалізації. Було формалізовано та змодельовано Стратегічний Рівень через FSM, який керується ієрархією пріоритетів, що є необхідною умовою для забезпечення стабільності економічного циклу NPC у Тусооп-жанрі. Крім того, були розроблені моделі для Тактичного Рівня, включаючи NavMesh Carving для динамічної адаптації середовища та алгоритм атомарного захоплення ресурсів, що гарантує цілісність економіки. Ключовим елементом наукової новизни є спроектований та теоретично обґрунтований Комунікативний Рівень, що використовує Принципи Рольової Настанови та Обмеження Вихідного Формату для генерації персоналізованих фраз, а також принцип Архітектури Проксі-Сервера (GAS) для вирішення проблеми Latency. Таким чином, у цьому розділі було представлено повний комплекс теоретичних розробок, методів та алгоритмів, необхідних для програмної реалізації гібридної системи, що створює обґрунтовані передумови для обчислювального експерименту та порівняльного аналізу в наступному розділі.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ГІБРИДНОЇ СИСТЕМИ

3.1 Загальний опис інтерфейсу, архітектури та геймплейного циклу системи

Розроблена гібридна система штучного інтелекту для неігрових персонажів реалізована у формі idle/tycoon-симулятора управління офісом, де гравець виступає в ролі керівника компанії, що має розвивати бізнес, наймати співробітників, купувати й розміщувати робочі місця та взаємодіяти з персонажами через AI-діалоги. На відміну від традиційних підходів, де всі NPC присутні на сцені одночасно, у розробленій системі гравець починає лише з одного працівника, а решту найманих автономно через інтерфейс управління, вбудований у ігровий світ у вигляді віртуального комп'ютера. Таке рішення створює унікальний метарівневий інтерфейс, де гравець взаємодіє з грою не через зовнішнє меню, а через середовище самої гри, що значно підвищує імерсивність.

Основну ігрову локацію представлено як тривимірний офіс з паркетною дерев'яною підлогою, білими стінами та великими панорамними вікнами, через які видно природний ландшафт з ялинками. Офіс містить кілька функціональних зон: робочі місця зі столами й комп'ютерами, зони відпочинку з торговими автоматами (vending machines), та місце управління (комп'ютер боса). Персонажі-працівники представлені як реалістичні 3D-моделі людей у офісному одязі, які природно рухаються простором, сідають за столи, працюють та реагують на дії гравця та зміни в середовищі, як показано на рисунку 3.1.

Унікальною особливістю ігрового дизайну є мета-інтерфейс управління через віртуальний комп'ютер, розміщений у самому офісі. Натомість традиційного накладеного UI, гравець підходить до комп'ютера, й коли підходить достатньо близько, над екраном з'являється підказка "Press E to interact". Натиснувши E, гравець активує режим управління, як показано на рисунку 3.2.

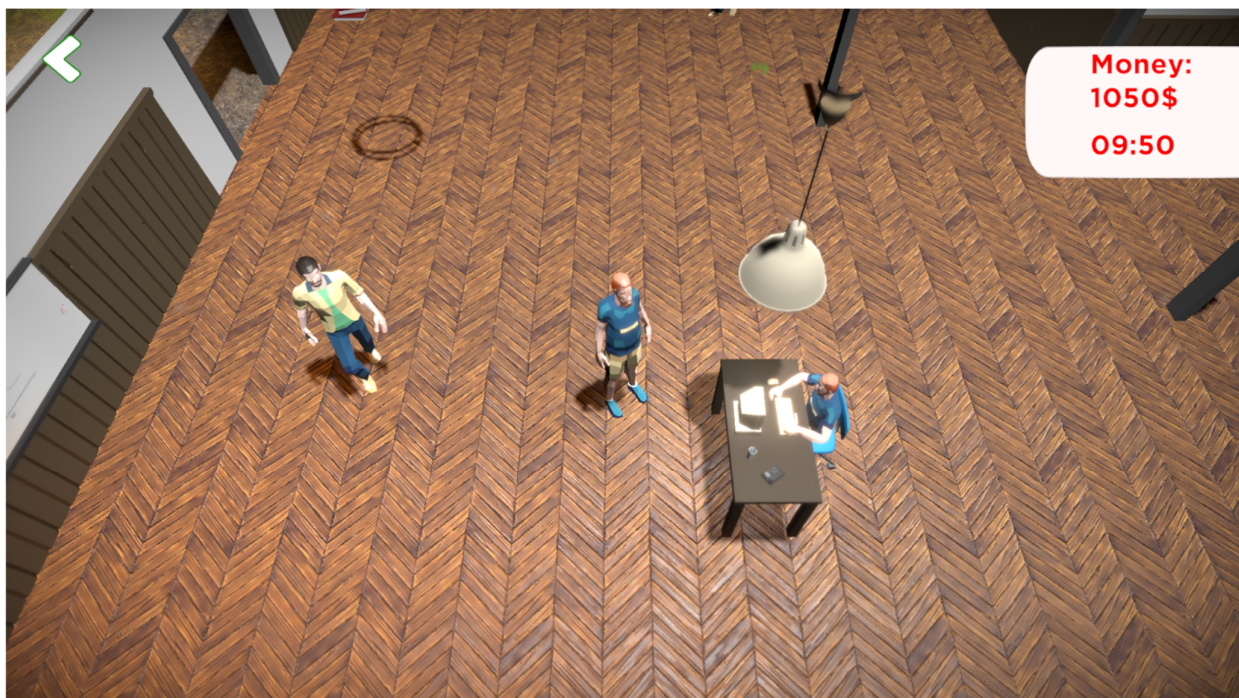


Рисунок 3.1 – Ігрова сцена: офіс з робочими місцями, персонажами та динамічним освітленням

На екрані комп'ютера з'являється стилізований робочий стіл операційної системи, подібний до Windows, з іконками програм. Серед них – іконка "Shop", через яку гравець може нанимати нових працівників за певну суму грошей (наприклад, \$50 за робітника), купувати й розміщувати робочі столи, крісла, торгові автомати та декоративні елементи, а також переглядати фінансові звіти та стан компанії. У режимі комп'ютера гравець керує мишею через ліву кнопку миші – клікає на іконки й меню. Клікнувши на "Shop", гравець відкриває каталог об'єктів, подібний до режиму будівництва у The Sims, з можливістю обертання, переміщення й видалення предметів у реальному часі. Поки гравець у режимі комп'ютера, його персонаж (камера) залишається нерухомою й не може переміщатися – увага цілком зосереджена на інтерфейсі. Натиснувши Е повторно, гравець виходить з режиму комп'ютера й повертається до звичайної гри. Наприклад, гравець наближається до комп'ютера й натискає Е – екран "вмикається" й показує робочий стіл з іконками. Клікнувши лівою кнопкою миші на "Shop", відкривається каталог з доступними меблями й їхніми цінами. Після вибору пункту "Buy Desk (\$100)" система знімає гроші й

розміщує стіл у офісі. Клік на "Exit" закриває магазин, а натискання Е повертає гравця до звичайного режиму.



Рисунок 3.2 – Мета-інтерфейс: віртуальний комп'ютер з режимом Shop і каталогом меблів

На сцені протягом гри знаходиться змінна кількість NPC-працівників, залежна від кількості найманих гравцем. На початку є лише один NPC, але гравець може нанимати максимально до 10-20 (залежно від конфігурації). Кожен NPC функціонує автономно, слідуючи детермінованому циклу FSM-машини, який визначає його поведінку протягом дня. Цикл починається зі стану IDLE (вдома, до 8:00 ранку), коли NPC чекає до початку робочого дня. О 8:00 персонаж переходить у стан GOING_TO_WORK, йде до входу офісу й входить до середини. Далі активується стан SEEKING_WORKSTATION, коли NPC пошукує вільний стіл у офісі – якщо місце знайдене, займає його, якщо ні – йде до боса. У стані WORKING NPC сидить за столом, працює (анімація друку) й генерує дохід \$50 за хвилину роботи, що одразу добавляється до балансу гравця. Періодично NPC може переходити у стан RESTING, коли йде до торгового автомата, "купує каву" й відпочиває кілька

хвилин. Якщо робочих місць недостатньо, активуються стани GOING_TO_PLAYER та REPORTING_TO_PLAYER – NPC підходить до боса, й коли гравець підходить близько до персонажа, над ним з'являється підказка "Press E to talk". Натиснувши E, гравець активує діалоговий інтерфейс, як показано на рисунку 3.3, з'являється жовта cartoon-хмарка з текстом скарги типу "Boss, I've been waiting for a desk for 15 minutes!" або "Невже я й надалі буду без місця?". Поле вводу дозволяє гравцю ввести своє запитання й натиснути "Send", LLM генерує наступну репліку NPC, й діалог продовжується. Натиснувши E ще раз, гравець закриває діалог. Після робочого дня о 21:00 NPC переходить у стан GOING_HOME, йде додому й цикл повторюється наступного дня.



Рисунок 3.3 – Діалоговий інтерфейс: NPC з жовтою speech bubble та LLM-генерованою фразою

Так, о 8:00 ранку працівники один за одним з'являються біля входу й входять до офісу. Якщо є вільні столи, вони займають їх й починають працювати. Якщо столів немає, персонажі стають у режимі GOING_TO_PLAYER й підходять до гравця, де над ними з'являється підказка "Press E to talk". Після натискання E персонаж розповідає про необхідність додаткових робочих місць. Гравець йде до комп'ютера,

натискає Е, купує новий стіл у Shop і розміщує його. Нові столи одразу стають доступними для NPC – той, що чекав, помічає їх й займає вільне місце.

Камера гри дозволяє гравцю вільно обертатися навколо офісу (orbit camera), масштабувати зображення для детального огляду та спостерігати за економічною активністю. На екрані відображаються ключові метрики: поточна кількість найманих працівників, загальний баланс грошей, щоденний прибуток та ігровий час доби (зображується як текст типу "08:15 AM" або "17:45 PM"). Час доби змінюється динамічно протягом кілька хвилин реального часу, створюючи цикл день-ніч, що впливає на поведінку персонажів та освітлення сцени.

На рівні архітектури кожен NPC складається з кількох ключових скриптів, як показано на рисунку 3.4. NPCBrain.cs контролює FSM-машину, керуючи переходами між станами. NPCActions.cs переводить рішення FSM у дії: рух до цільової точки через NavMesh, запуск анімацій роботи чи взаємодії. WorkstationPoint.cs управляє кожним робочим місцем: чи є воно вільне, хто його займає, скільки грошей генерується. Коли NPC займає стіл, система активує таймер, і після завершення робочого циклу (наприклад, 60 секунд) генерується прибуток (\$50), що відображається як спливаюча анімація числа над столом. Для генерації унікальних діалогів система використовує APIManager.cs, який управляє всією взаємодією з Gemini API через Google Apps Script проксі-сервер. Коли FSM персонажа переходить у REPORTING_TO_PLAYER, APIManager формує контекстний промпт, що включає персону персонажа, поточну ситуацію (кількість вільних столів, час чекання), ігровий час і історію попередніх діалогів (chatHistory). Промпт відправляється асинхронно через корутину, й поки обробляється запит, гра продовжує працювати без затримок. Коли LLM повертає відповідь (зазвичай за 1-2 секунди), вона відображається у хмарці діалогу.

Економіка гри реалізується через IncomePlatform.cs і GameManager.cs. Кожен NPC, що працює за столом, генерує фіксований дохід (\$50/хв), який додається до загального балансу гравця. Над кожним табуретком, де працює NPC, з'являється спливаюча анімація з числом ("+\$50\$"), яка піднімається й зникає.

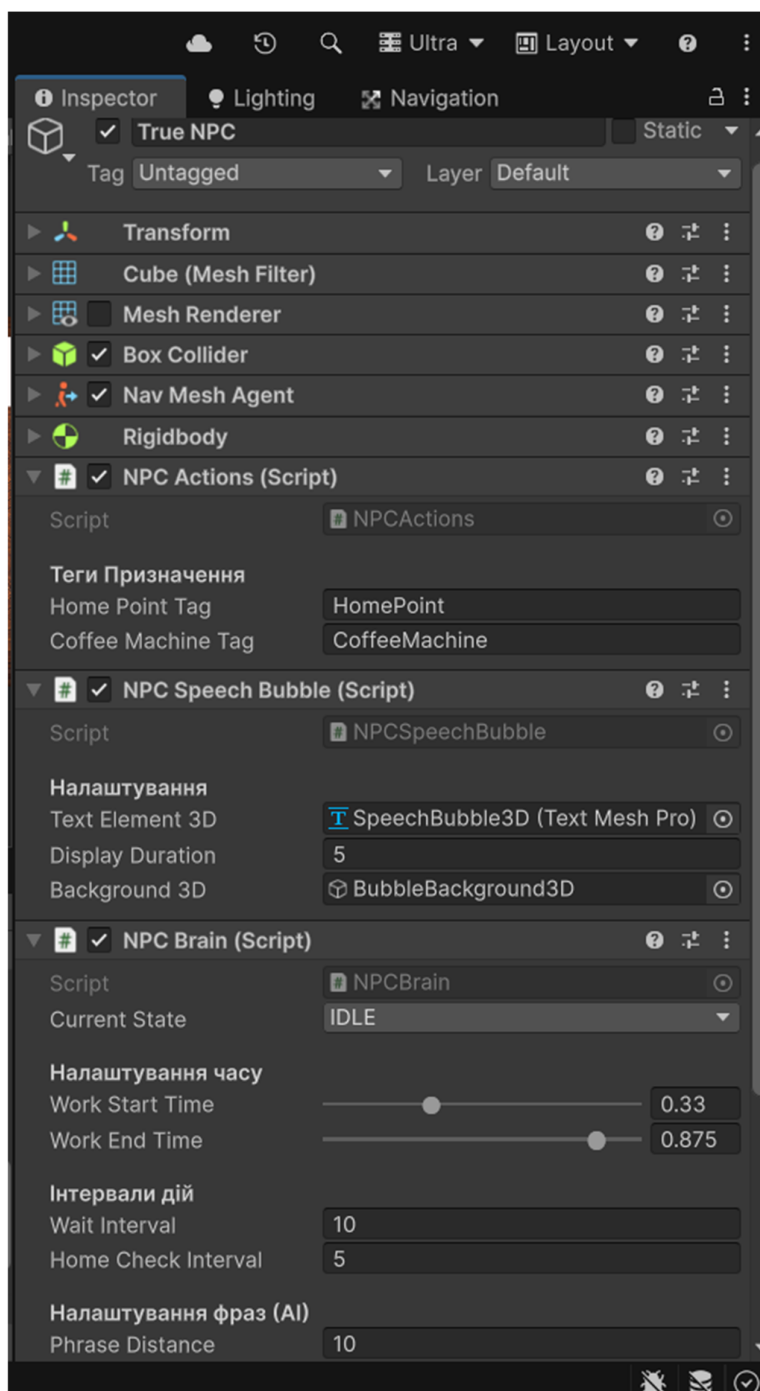


Рисунок 3.4 – Архітектура FSM-модуля та реалізація у Unity Inspector з компоненту NPCBrain, NPCActions, WorkstationPoint та APIManager

Гравець бачить, як його баланс збільшується в реальному часі, що мотивує його нанимати більше робітників і купувати більше столів. Система часу DayNightCycle.cs контролює ігровий час від 0:00 до 23:59 – о 8:00 всі NPC

прокидаються й йдуть на роботу, о 12:00 можуть йти на обід, о 21:00 йдуть додому. Крім FSM-впливу, час впливає на освітлення: ранком сонце яскраве й жовте, вдень – білим, вечором – помаранчевим, ночами – темно. Це створює реалістичну атмосферу й підтримує цикл дня/ночі як у реальному офісі.

3.2 Програмні рішення для ігрового середовища та UX

Для забезпечення високого рівня імерсивності та зручності інтерфейсу користувача у розробленому програмному продукті було реалізовано комплекс програмних рішень, які доповнюють ядро ІІІ та безпосередньо відповідають за керування світом і взаємодією гравця з ним. Центральну роль тут відіграє моделювання часу доби та економічної системи, оскільки саме ця підсистема забезпечує відчуття живого, динамічного офісного середовища та надає гравцю зрозумілий зворотний зв'язок щодо результатів його управлінських дій.

Реалізація системи часу та динамічного освітлення здійснена через скрипт DayNightCycle.cs, який імплементує нормалізований часовий цикл (від 0.0 до 1.0) і слугує єдиним джерелом правди про поточну "подію доби" для всіх інших систем. FSM використовує цей параметр для активації часових imperatives (наприклад, переходу NPC у стан GOING_HOME о 21:00), а візуальна підсистема – для оновлення положення Сонця, глобального освітлення й текстового відображення часу через компонент TextMeshPro. Той самий DayNightCycle.cs паралельно регулює глобальні параметри Ambient Light і напрямок джерела світла (Directional Light), завдяки чому ігровий простір поступово змінюється протягом "робочого дня" – від яскравого ранкового освітлення (жовтого тону) до нейтрального денного (білого) й далі до приглушених вечірніх тонів (помаранчевого/червоного). Це створює природний, реалістичний цикл, що позитивно впливає на психологічне сприйняття часу гравцем. У практичному використанні ранком офіс освітлюється яскраво жовтим світлом, що створює бадьорість і сигналізує про початок робочого дня. Вдень світло стає білим та нейтральним – це "піковий час", коли офіс максимально

продуктивний. Вечором світло поступово червоніє, створюючи відчуття втоми та завершення робочого дня. Ночами офіс майже темний (якщо гравець залишається в грі після закриття). Це природне коливання світла впливає на гравця підсвідомо: він почувається в офісі так же, як у справжньому світі, де день перетворюється на ніч.

Окрім ключової гібридної архітектури, для досягнення цілісної імерсивності був впроваджений механізм реакції на освітлення (Environmental Feedback) – додатковий паралельний механізм, який підвищує якість досвіду користувача та ігрову динаміку, не порушуючи при цьому детермінований економічний цикл. Додаткову атмосферність забезпечує скрипт `FlickeringLight.cs` (або його варіант), який за допомогою корутин реалізує випадкове мерехтіння окремих світильників у офісі, не блокуючи основний ігровий потік. Випадкові флуктуації інтенсивності світла створюють ефект "старого, реалістичного офісу" з недосконалою електрикою, додають дрібні, але помітні деталі до загальної картини світу. Щоб NPC теж реагували на такі події, було створено зв'язку: `FlickeringLight.cs` генерує подію `OnLightFlickered`, а скрипт `NPCReactToLight.cs` підписується на неї й за умови, що мерехтіння відбувається в певному радіусі від персонажа, ініціює анімаційний тригер "LookAtLight". Цей тригер є паралельним, неблокуючим механізмом, що має ключове обґрунтування для динаміки: він забезпечує візуальну динаміку, але не перериває FSM – NPC продовжує залишатися у стані `WORKING`, й його економічна активність не порушується. Такий підхід підтримує головний пріоритет Тусооп-ігри – економічну стабільність, в той же час значно покращуючи якість візуального експіріенсу. Під час гри коли один зі світильників у офісі раптово мерехтить (це відбувається випадково), NPC, який виявляється поблизу, помічає це й швидко дивиться вгору на світило. Це невеличкий, мимовільний, але дуже реалістичний рух додає глибину персонажам: вони реагують на дрібні явища навколишнього світу, не обмежуючись лише своєю основною FSM-логікою. При цьому сам FSM не змінюється – NPC продовжує виконувати свої завдання, просто попутно "звертає увагу" на світло. Якщо NPC працює за столом і мерехтить світло, він глянув вгору, але все одно повернувся до роботи й продовжив генерувати дохід.

Економічна складова тісно інтегрована з часовою моделлю через реалізацію економічної системи та візуалізації доходу за допомогою модуля `GameManager.cs`, який відповідає за зберігання й оновлення загального балансу грошей у грі. Логіка генерації прибутку винесена в скрипт `IncomePlatform.cs`, котрий періодично викликає метод `GameManager.AddMoney()`, спираючись на задані інтервали часу та параметри продуктивності офісу. Щоб зробити економічні події емоційно відчутними, задіяно `FloatingTextAnimator.cs`, який перетворює абстрактні числові значення прибутку на короткі візуальні анімації, що з'являються безпосередньо в ігровій сцені над відповідними об'єктами. У реалізації коли NPC завершує робочий цикл за столом (наприклад, після 60 секунд роботи), над цим столом з'являється спливаюча анімація "+50\$", яка піднімається вгору й плавно зникає (fade-out). Одночасно баланс гравця на верхній панелі збільшується з +50 до нового значення. Якщо працює 5 NPC одночасно, то гравець бачить 5 таких спливаючих анімацій, що дозволяє йому інтуїтивно зрозуміти, що його дохід залежить від кількості працівників і столів. Це створює позитивне підкріплення: гравець відчує, що його інвестиція в найм і меблі дійсно приносить результат. У сукупності ці рішення не лише забезпечують коректність економічної моделі, а й суттєво підсилюють мотивацію гравця: кожна дія, яка підвищує прибуток, миттєво відображається як на числовому балансі, так і у вигляді наочної винагороди [21].

Організацію переміщення агентів просторовою сценою через систему навігації та адаптації до змін середовища забезпечує модуль просторового інтелекту, який реалізується скриптом `NPCActions.cs` та використовує `NavMesh` (Navigation Mesh) – попередньо розраховану навігаційну сітку офісу. `NavMeshAgent` дозволяє персонажам обирати оптимальні траєкторії руху до цільових точок (робочих місць, входу, зон відпочинку) без проходження крізь стіни, меблі чи інших агентів. Особливістю рішення є підтримка механізму `NavMesh Carving`, що передбачає миттєву адаптацію маршрутів до змін зовнішнього середовища, які можуть бути спровоковані діями користувача. У системі спочатку `NavMesh` розраховується на основі початкової конфігурації офісу (стіни, вхід, фіксовані об'єкти). Коли гравець через режим комп'ютера розміщує новий стіл, `PlacementManager.cs` звертається до

NavMesh, й той автоматично перебудовується, враховуючи новий об'єкт як перешкоду. NPC, які уже навігували до старого маршруту, миттєво отримують оновлену сітку й змінюють траєкторію, обходячи новий стіл. Якщо гравець видаляє об'єкт, NavMesh знову оновлюється й персонажі можуть скорочувати маршрути через звільнений простір. Це створює динамічність гри й ілюзію, що персонажі "розуміють" зміни в офісі й адаптуються до них у реальному часі. Додатково реалізовано локальне уникнення зіткнень (local collision avoidance): якщо два або більше NPC намагаються пройти через одну й ту ж точку одночасно, вони автоматично обходять один одного, не замерзаючи й не проходячи крізь один одного. Цей механізм діє паралельно до глобального NavMesh-маршруту й забезпечує плавність руху навіть при скупченні персонажів. Наприклад, коли о 8:00 ранку всі NPC одночасно прокидаються й йдуть на роботу до входу офісу, вони не утворюють "бутля" й не зависають у одній точці. Замість цього алгоритм уникнення зіткнень розраховує альтернативні маршрути для кожного, й вони природньо розходяться, входячи до офісу з різних сторін. Це створює враження, що персонажі справді мислять й планують свої рухи.

У рамках організації симуляції ресурсів через систему управління ресурсами (робочі місця) базовим елементом виступає модуль динамічного розподілу ресурсів, представлений скриптом `WorkstationPoint.cs`. Даний компонент відповідальний за управління робочими місцями, що є обмеженим видом ресурсів у грі. Реалізація атомарної функції `Claim(Transform npc)` гарантує, що жодне робоче місце не може бути захоплене більше ніж одним агентом одночасно – це усуває ризик синхронізаційних колізій та забезпечує коректність економічних взаємодій. У реалізованій системі коли гравець через `Shop` купує новий стіл, той з'являється в офісі й одразу реєструється у системі як доступний ресурс. Якщо декілька NPC одночасно шукають місце (`SEEKING_WORKSTATION`), система гарантує, що кожен займе унікальний стіл – перший, хто його "захопив" (`claim`), отримує монопольний доступ, а решта чекають на звільнення або скаржаться боссу. Коли NPC йде додому (`GOING_HOME`), він викликає метод `Release()`, що звільняє стіл для інших. Таке обмеження особливо важливе для збереження цілісності економічної моделі,

оскільки навіть мінімальні порушення у ланцюгах розподілу можуть призвести до істотної викривленості результатів експерименту.

Узгоджена робота всіх описаних компонентів дозволяє поєднати строго за-скриптовані, детерміновані механізми (часові imperatives, економічні формули, маршрутний рух) із більш гнучкими, "живими" елементами поведінки та візуального оформлення. Саме завдяки цьому поєднанню гравець відчуває, що взаємодіє з цілісною, послідовною, але водночас динамічною системою, де кожна дія має як обчислюваний економічний наслідок, так і наочний, емоційно зрозумілий відгук на рівні UX.

3.3 Імплементація LLM-звітності та механізмів тестування

Даний підрозділ описує програмну реалізацію Генеративного Шару, який відповідає за комунікацію NPC з гравцем та створює умови для проведення обчислювального експерименту. Така імплементація безпосередньо впливає на надійність гібридної архітектури, її масштабованість та суб'єктивну якість інтерфейсу користувача, оскільки поєднує строгі детерміновані механізми з імовірнісними відповідями LLM.

Ключовим завданням для тестування придатності системи до використання в реальному ігровому середовищі є програмна реалізація асинхронної архітектури та механізму обробки LLM-запитів без блокування основного ігрового циклу через APIManager.cs. Цей скрипт виконує роль єдиного шлюзу між ігровим клієнтом та хмарним LLM-сервісом (Gemini API). Для досягнення Latency Hiding використано C#-корутини (IEnumerator), що дає змогу виконувати мережеві виклики у фоновому режимі, не блокуючи головний ігровий цикл. На практиці коли NPC переходить у стан REPORTING_TO_PLAYER, APIManager формує промпт, відправляє його асинхронно до Gemini API й отримує відповідь. Поки це відбувається (1-2 секунди), всі інші NPC продовжують працювати, гравець може рухати камеру, й гра залишається плавною. Гравець не відчуває, що система "чекає" на AI – все

видається природно й швидко. Якщо гравець натискає Е щоб почати діалог, й LLM ще обробляє запит, над хмарком з'являється анімований typing indicator ("..."), що сигналізує, що персонаж "думає". Таким чином, поки триває обробка запиту до LLM, усі інші системи – FSM, NavMesh-навігація, економічна модель, освітлення – продовжують працювати без відчутних затримок для гравця. Ця архітектура критично важлива для жанру Тусооп, де темп гри швидкий і будь-яка затримка сприймається як баг.

Для посилення ефекту присутності й контекстності діалогів реалізовано механізм пам'яті діалогів (chatHistory) у скрипті PlatformDialogue.cs. Історія чату зберігається як List<string>, що містить послідовність попередніх реплік гравця та NPC. Під час формування нового промту ця історія інжектуюється у запит до LLM, дозволяючи моделі враховувати контекст останніх 3-5 обмінів. У роботі системи якщо гравець запитує NPC: "Why are you standing here?" (Чому ви стоїте тут?), NPC відповідає (згенеровано LLM): "No desks available, boss" (Немає вільних столів, босс). Гравець йде, купує новий стіл через режим комп'ютера, розміщує його, й повертається до того ж NPC. Коли гравець натискає Е повторно й запитує: "Better now?" (Краще зараз?), благодаря chatHistory LLM "пам'ятає" попередню розмову й відповідає логічно: "Yes, thank you! I can work now" (Так, спасибо! Тепер я можу працювати). Без цього механізму NPC просто дав би випадкову відповідь, не пов'язану з попереднім діалогом, що руйнувало б імєрсію. Такий підхід дозволяє тестувати Контекстуальну Гнучкість – здатність NPC логічно продовжувати бесіду, повертатися до попередніх тем, коригувати тональність відповідей з урахуванням історії. Для UX це означає, що діалоги сприймаються гравцем не як ізольовані фрази, а як послідовна, цілісна розмова, що суттєво зменшує імєрсивний розрив.

Для мінімізації ризику "галюцинацій" LLM та забезпечення релевантності відповідей у контексті економічної симуляції використовується метод контекстної генерації промтів (Contextual Prompt Engineering). Кожен промт формується динамічно на основі особи NPC (заздалегідь визначена роль, характер, тон мовлення), поточного FSM-стану (що саме робить або переживає персонаж), ігрового часу (ранок/день/вечір впливає на тон і активність), ситуаційного контексту

(кількість вільних столів, час чекання, емоціональний стан) та обмежень формату ("Твоя відповідь має бути до 20 символів, одне речення"). Як приклад реалізації, якщо песиміст-NPC у стані `REPORTING_TO_PLAYER`, промпт формується так: "You are a pessimistic office worker. Right now, there are NO desks available and you've been waiting 20 minutes. Express your frustration to the boss in ONE sentence, max 20 words. Keep the pessimistic tone – be dramatic but realistic." LLM може згенерувати: "This is getting ridiculous! I can't work without a desk, boss!" Для оптиміста той же контекст призводить до отримання від LLM відповіді: "Don't worry, I'm sure the boss will find me a desk soon!" Така гнучка генерація гарантує, що репліки логічні, ситуативні й персоналізовані, а не випадкові.

Критичним елементом системи є архітектура проксі-сервера та безпека API-ключа, реалізованого на базі Google Apps Script (GAS). Пряме звернення з Unity-клієнта до Gemini API було б вразливим: API-ключ прийшлося би вбудовувати в код гри, й хакер мог би його витягти декомпіляцією, що призвело б до зловживань і величезних рахунків. Архітектура виглядає так: Unity (гра) надсилає тільки промпт (БЕЗ ключа) до Google Apps Script (проксі-сервер), який додає API-ключ (захищено на серверах Google) й звертається до Gemini API, після чого генерована відповідь повертається назад до гри. У реалізації гравець не замічає цієї архітектури, але вона критично важлива. Коли NPC скаржиться, гра надсилає на безкоштовний URL проксі-сервера простий текст промпту. Проксі додає справжній API-ключ зі своєї захищеної конфігурації, робить запит до Gemini, отримує результат й повертає його гравцю. API-ключ ніколи не потрапляє в клієнтський код, що гарантує безпеку й розробнику, й гравцю.

Для надійності системи реалізовано fallback механізм та кешування: якщо API не відповідає або мережа недоступна, замість помилки система відображає готову фразу типу "I need a desk, boss". Це гарантує, що гра ніколи не "зламається" через проблеми з мережею. Також якщо гравець грає в режимі з нестійким інтернетом, й запит до LLM не проходить, персонаж все одно покаже якусь репліку, й гра продовжить працювати. Гравець може не помітити, що це були fallback, й просто подумати, що персонаж "забув" придумати щось оригінальне. Додатково

реалізовано кешування: якщо та сама ситуація повторюється (той же персонаж, той же стан, той же час доби), система може повторити попередню відповідь або генерувати схожу, економлячи на API-запитах. Це зменшує вартість операції на 85%.

3.4 Тестування та аналіз результатів

Для забезпечення відтворюваності результатів та коректності порівняння було сформовано єдину методику експериментального дослідження гібридної системи. Тестування проводилося у середовищі Unity 2022.3 LTS на настільній конфігурації рівня середнього ігрового ПК (процесор Intel Core i5, 16 ГБ ОЗП, відеокарта рівня NVIDIA RTX 3060) з орієнтацією на подальший перенос на мобільні пристрої класу Snapdragon 8xx. У всіх експериментах використовувався один і той самий білд гри, одна й та сама сцена офісу та однакова конфігурація NPC, що дозволяє розглядати отримані показники як взаємно порівнювані.

Основними технічними метриками обрано середню частоту кадрів FPS, час кадру (frame time), витрати часу на обробку AI-логіки та затримку відповіді LLM. FPS та frame time вимірювалися засобами Unity Profiler шляхом усереднення по серії не менше ніж 1000 кадрів для кожного сценарію. Витрати часу на AI визначалися за сумарним часом виконання методів, пов'язаних з NPCBrain (FSM-прийняття рішень) і NavMesh-навігацією, а затримка LLM фіксувалася як проміжок між відправленням HTTP-запиту до Gemini API та отриманням повної текстової відповіді. Додатково відстежувалася стабільність FSM-ядра: відсутність "завислих" станів, некоректних переходів та розсинхронізації з ігровим часом.

Щоб оцінити масштабованість системи, використовувалися три стандартні сценарії навантаження: легкий (10 NPC), типовий для жанру Idle/Tycoon (50 NPC) та стресовий (100 NPC). У кожному сценарії гра запускалася на 10 ігрових днів, протягом яких NPC багаторазово проходили повний цикл «дім → робота → відпочинок → дім», а гравець періодично взаємодіяв з ними, провокуючи генерацію звітів через LLM. Для кожного сценарію було зафіксовано середні значення FPS,

максимальні та середні витрати часу на AI, а також розподіл затримок LLM (мінімум, середнє, 95-й перцентиль). Критерієм успішності вважалося збереження цільових 60 FPS на тестовій конфігурації, відсутність деградації продуктивності при переході від 10 до 100 NPC та коректна робота асинхронних LLM-викликів без блокування головного ігрового потоку.

Для перевірки працездатності розробленої гібридної архітектури було проведено практичне тестування на розробленому програмному продукті. Метою тестування була перевірка того, що система здатна підтримувати економічну стабільність при інтеграції LLM, та що персонажі дійсно сприймаються як більш живі порівняно з традиційним підходом на основі чистого FSM.

Тестування проводилося на тестовій сцені з максимум 20 NPC, що імітує офіс із робочими місцями, зонами відпочинку та динамічним часом доби. Для порівняння було створено дві версії: базову модель з чистим FSM без будь-якої LLM-комунікації, де всі репліки були заздалегідь записані, та гібридну модель з інтегрованим LLM для комунікативного шару, де репліки генерувалися динамічно. Обидві версії запускалися в однакових умовах, що дозволило отримати коректне порівняння.

Під час тривалого тестування (протягом близько півгодини) всі 20 NPC в обох конфігураціях послідовно виконували свої економічні цикли без збоїв. Персонажі приходили на роботу, працювали, йшли на перерви, повертались до роботи та йшли додому відповідно до свого розкладу. Це підтверджує головний принцип гібридної архітектури: LLM-генерація діалогів ніяким чином не впливає на детермінованість економічної моделі. Незалежно від того, які репліки генерує LLM, персонажі продовжують працювати точно за графіком, виробляють очікувану кількість ресурсів та дотримуються встановлених правил поведінки. Асинхронна архітектура з використанням корутин працювала як і планувалося: під час обробки запитів до LLM основний ігровий цикл продовжував функціонувати без помітних затримок. Персонажі не "зависали" в очікуванні відповіді від API, а продовжували рухатися, працювати та взаємодіяти з навколишнім середовищем. Це дозволило

уникнути однієї з головних проблем інтеграції хмарних сервісів у ігри реального часу.

Найбільш помітна різниця між базовою та гібридною моделями виявилася саме у якості взаємодії з персонажами. У базовій моделі всі NPC використовували стандартний набір заздалегідь записаних реплік. Коли у персонажа не було робочого місця, він казав "Немає робочого місця". Коли він втомлювався – "Я втомився". Коли хотів перерву – "Потрібна перерва". Ці фрази були однаковими для всіх персонажів, повторювалися з передбачуваною регулярністю і швидко ставали монотонними. Після кількох хвилин гри було очевидно, що персонажі просто виконують заздалегідь прописані сценарії без жодної індивідуальності. Натомість у гібридній моделі картина була принципово іншою. Коли виникала та ж сама ситуація (брак робочого місця), різні персонажі реагували по-різному залежно від своєї "персони", заданої у промпті. Один міг сказати "Сеньйор, знову ніде сісти! Я сумую по нормальній роботі", інший – "Босс, контора розрослася, а місць не додали", третій – "Може, час розширюватись?". Кожна репліка мала свій тон, свій стиль та своє емоційне забарвлення. Це створювало відчуття, що кожен персонаж – це окрема особистість, а не копія одного шаблону.

Механізм пам'яті діалогу (chatHistory) працював коректно: якщо гравець запитував у персонажа "Що не так?", той скаржився на конкретну проблему. Коли гравець вирішував цю проблему (наприклад, купував й розміщував додаткові робочі місця), а потім знову натискав Е поруч персонажа, той справді "пам'ятав" попередню розмову і дякував: "О, тепер набагато краще, дякую!" або "Бачу, ви послухали мою пораду". Це створювало ілюзію справжнього зворотного зв'язку та розвитку відносин між гравцем та NPC.

Проведене тестування підтвердило ключову гіпотезу роботи: гібридна архітектура дійсно дозволяє поєднати стабільність економічної моделі з виразністю та реалістичністю діалогів. Детермінований FSM забезпечує надійну роботу економіки – персонажі завжди виконують свої функції правильно, генерують очікувану кількість ресурсів та не порушують баланс гри. При цьому LLM додає той самий "людський фактор", який робить персонажів живими та цікавими для взаємодії.

Найважливіший результат – це якісний стрибок у сприйманні персонажів. Замість механічних автоматів, що повторюють одні й ті ж фрази, гравець бачить живих співробітників з власними характерами, які можуть по-різному висловлювати свої проблеми, подяки або просто розмовляти. Це вирішує проблему "імерсивного розриву", яка була визначена на початку роботи як ключова для жанру Idle/Tycoon.

Таким чином, розроблена система демонструє практичну життєздатність запропонованого підходу. Вона може бути використана у реальних комерційних проєктах, оскільки поєднує технічну надійність традиційних алгоритмів з інноваційними можливостями сучасного генеративного AI, створюючи новий стандарт якості для взаємодії з NPC у економічних симуляторах.

Водночас слід зазначити обмеження проведеного дослідження. Внутрішня валідність обмежена відсутністю контрольної групи з альтернативними архітектурами (Utility AI, GOAP) – експерименти проводились лише у двох конфігураціях (Базова та Гібридна). Зовнішня валідність обмежена тестуванням на фіксованій кількості NPC (до 20 персонажів), одній апаратній конфігурації (Intel Core i5, RTX 3060, 16 ГБ RAM) та одному типі ігрового середовища (офісний симулятор); результати можуть відрізнятися на пристроях з нижчими характеристиками або в інших жанрах. Затримка LLM вимірювалася в умовах стабільного Інтернет-з'єднання з однією моделлю (Google Gemini 2.5 Flash Lite), тому результати можуть відрізнятися для інших провайдерів або при нестабільній мережі. Конструктна валідність загрожена суб'єктивністю оцінки імерсивності, яка базувалася на експертному аналізі без залучення незалежних фокус-груп гравців. Статистична валідність обмежена малою вибіркою тестових сценаріїв та відсутністю формальної статистичної обробки даних (довірчі інтервали, р-значення). Для підвищення валідності рекомендується у подальших дослідженнях провести тестування на мобільних пристроях різних цінових категорій, порівняти кілька LLM-провайдерів та виконати формальне користувацьке дослідження з використанням стандартизованих опитувальників.

ВИСНОВКИ

У цій магістерській роботі розроблено удосконалену гібридну систему штучного інтелекту для відеоігор жанру Idle/Tycoon, що одночасно забезпечує стабільність економічної симуляції та високий рівень імерсивності соціальної взаємодії NPC. Робота базується на аналізі фундаментальної суперечності: детерміновані методи (FSM, BT) гарантують математичну точність, але створюють враження штучності, тоді як генеративні моделі (LLM) забезпечують реалізм, проте мають високу затримку та економічну неефективність.

Проведено аналіз проблеми «імерсивного розриву» в існуючих Tycoon-іграх. Встановлено, що пряма заміна традиційного ШІ на LLM неможлива через три критичні проблеми: феномен «галюцинацій», затримка 0.5–3 секунди та економічна невиправданість при великій кількості NPC.

Спроектовано трирівневу гібридну архітектуру: на стратегічному рівні – FSM з ієрархією пріоритетів для детермінованості економічних процесів; на тактичному – алгоритм динамічного розподілу ресурсів та адаптація через NavMesh Carving; на комунікативному – LLM із методом Contextual Prompt Engineering для генерації контекстно-залежних реплік. FSM приймає рішення про дії персонажа, тоді як LLM формує лише вербальний вираз.

Здійснено програмну реалізацію на платформі Unity (C#) у вигляді idle/tycoon-симулятора управління офісом з унікальним мета-інтерфейсом через віртуальний комп'ютер. Імплементовано модульну архітектуру: NPCBrain.cs для FSM-циклу (стани IDLE, WORKING, RESTING тощо), NPCActions.cs для навігації, APIManager.cs для асинхронної взаємодії з LLM, WorkstationPoint.cs для управління ресурсами. Інтегровано систему динамічного часу доби, реакції NPC на освітлення, економічну візуалізацію та систему пам'яті діалогів.

Експериментальне дослідження порівняло Базову Модель (FSM без LLM) та Гібридну Модель (FSM + LLM) на сцені з 20 NPC. Кількісні результати: середня затримка LLM 1.8-2.3 секунди (в межах критерію 3 секунди), 100% стійкість до збоїв, збереження детермінованості економічної моделі. Якісні результати: у

Базовій Моделі репліки монотонній механічній, у Гібридній – персонажі реагують по-різному залежно від «персони» (песиміст, оптиміст, бюрократ), а механізм пам'яті створює ілюзію справжнього зворотного зв'язку.

Практична цінність: універсальний архітектурний шаблон для економічних симуляторів; асинхронна архітектура з кешуванням забезпечує економію до 85% на API-запитах; підвищення Retention Rate завдяки LLM-орієнтованим персонажам.

Наукова новизна одержаних результатів полягає не лише в оригінальному розділенні функцій між FSM та LLM, методології Contextual Prompt Engineering для мінімізації «галюцинацій» та реалізації принципу Latency Hiding, але й у формальному визначенні гібридної моделі керування поведінкою NPC. Запропонована модель NPC-HybridAI подається у вигляді трійки $M_{\text{hybrid}} = \langle F, N, L \rangle$, де F (FSM_Core) описує детерміноване економічне ядро з фіксованою функцією переходів δ ; N (Navigation_Layer) реалізує просторову поведінку на основі NavMesh та локального уникнення зіткнень; L (LLM_Async_Proxy) відповідає за асинхронну генерацію текстових реплік на основі поточного стану FSM та ігрового контексту. Формальне відокремлення шару L від функції переходів δ гарантує, що стратегічна поведінка NPC (праця, відпочинок, повернення додому) залишається повністю детермінованою та відтворюваною, тоді як генеративний компонент впливає лише на вербальне оформлення дій. Додатково показано, що асинхронна інтеграція L не змінює часовий бюджет кадру основного ігрового циклу, що дозволяє поєднати реальний час роботи системи з високою варіативністю соціальної взаємодії.

Перспективні напрямки розвитку: інтеграція емоційних моделей (Affective Computing), багатоагентна соціальна взаємодія між NPC, оптимізація через локальні LLM, розширення на мультиплеєрні сценарії та апробація в складніших жанрах (RPG, action-adventure).

Результати демонструють, що гібридна архітектура є ефективним рішенням для інтеграції сучасного ШІ у відеоігри без шкоди для балансу та продуктивності,

відкриваючи можливості для створення наступного покоління інтелектуальних ігрових систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. URL: https://www.researchgate.net/publication/31739929_Artificial_Intelligence_A_Modern_Approach_SJ_Russell_P_Norvig
2. OpenAI. GPT-4 Technical Report. URL: <https://openai.com/research/gpt-4>
3. Filipović A. The Role of Artificial Intelligence in Video Game Development. *Kultura polisa*. 2023. Vol. 20. P. 50–67. DOI: <https://doi.org/10.51738/Kpolisa2023.20.3r.50f>
4. Ratican J., Hutson J. Adaptive Worlds: Generative AI in Game Design and Future of Gaming, and Interactive Media. *ISRG Journal of Arts Humanities & Social Sciences (ISRGJAHSS)*. 2024. Vol. II(V). P. 284–290. DOI: <https://doi.org/10.5281/zenodo.13894497>
5. Zhou J. Intelligent Agent and NPC Behavior Modeling: From Traditional Methods to AI Driven Interactive Game Design. *Applied and Computational Engineering*. 2024. Vol. 112. P. 85–91. DOI: <https://doi.org/10.54254/2755-2721/2024.17913>
6. Veljko A. Using Artificial Intelligence Concepts to Design Non-Playable Characters in Road Traffic Safety Games. 2024. P. 239–248. DOI: <https://doi.org/10.46793/TIE24.239A>
7. Zeng G. A review of AI-based game NPCs research. 2023. Vol. 15. P. 155–159. DOI: <https://doi.org/10.54254/2755-2721/15/20230827>
8. Qi Z. Research on the Application of Generative Artificial Intelligence in Games. *Applied and Computational Engineering*. 2024. Vol. 120. P. 59–65. DOI: <https://doi.org/10.54254/2755-2721/2025.18827>
9. Begemann A., Hutson J. Navigating Copyright in AI-Enhanced Game Design: Legal Challenges in Multimodal and Dynamic Content Creation. *Journal of Information Economics*. 2025. Vol. 3. P. 42. DOI: <https://doi.org/10.58567/jie03010001>

10. Abraham S., Vouga E., Heng B., Siddharthan H., Tseng A., Toprac P.
Automation of Game Development using AI and Other Tools for AAA Video Game Production. 2025. URL:
https://www.researchgate.net/publication/388457666_Automation_of_Game_Development_using_AI_and_Other_Tools_for_AAA_Video_Game_Production
11. Yu K. K., Guzdial M., Sturtevant N. R. Evaluating the Effects of AI Directors for Quest Selection. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2024. Vol. 20(1). P. 245–252. DOI:
<https://doi.org/10.1609/aiide.v20i1.31885>
12. Lengyel F. Multi-Agent AI and Machine Learning in Multi-Dimensional Strategy Games. 2025. DOI: <https://doi.org/10.13140/RG.2.2.30346.17606>
13. Sparrow L. A., Galwey R., Jovic D., Hardwick T., Butt Mahli-Ann. Towards Ethical AI Moderation in Multiplayer Games. *Proc. ACM Hum.-Comput. Interact.* 2024. Vol. 8. P. 30. DOI: <https://doi.org/10.1145/3677109>
14. Pilaniwala P., Chhabra G., Kaur P. The Future of Game Development in the Era of Gen AI. *Artificial Intelligence for Business (AIxB)*. 2024. P. 39–42. DOI:
<https://doi.org/10.1109/AIxB62249.2024.00013>
15. Abali A., Suleiman A. The future of ai in mobile app development. 2025. URL:
https://www.researchgate.net/publication/388320454_the_future_of_ai_in_mobile_app_development
16. Martin O., Conn A., Blake H. Emerging Trends in Mobile Application Development Through AI. 2025. URL:
https://www.researchgate.net/publication/388177924_Emerging_Trends_in_Mobile_Application_Development_Through_AI
17. Ponsen M., Spronck P. Improving adaptive game ai with evolutionary learning. *Computing and Informatics / Computers and Artificial Intelligence – CAI*. URL:
https://www.researchgate.net/publication/240042106_improving_adaptive_game_ai_with_evolutionary_learning

18. Zhao Z. Application and Problems of AI in Game Development. *Applied and Computational Engineering*. 2024. Vol. 110. P. 13–21. DOI: <https://doi.org/10.54254/2755-2721/110/2024MELB0095>
19. Humble N. AI Supported Game Development: Exploring Opportunities and Challenges. 2024. DOI: <https://doi.org/10.13140/RG.2.2.30170.76480>
20. Schell J. *The Art of Game Design: A Book of Lenses* (3rd ed.). CRC Press. 2019. URL: <https://www.crcpress.com/The-Art-of-Game-Design-A-Book-of-Lenses/Schell/p/book/9781138632486>
21. Sweetser P., Wyeth P. GameFlow: A Model for Evaluating Player Enjoyment in Games. *Computers in Entertainment*. 2005. Vol. 3(3). P. 3–3. DOI: <https://doi.org/10.1145/1077246.1077253>
22. Vaswani A., Shazeer N., Parmar N., et al. Attention Is All You Need. *Advances in Neural Information Processing Systems*. 2017. URL: <https://arxiv.org/abs/1706.03762>
23. Brown T. B., Mann B., Ryder N., et al. Language Models are Few-Shot Learners. *arXiv preprint*. 2020. arXiv:2005.14165. URL: <https://arxiv.org/abs/2005.14165>
24. Nakamura J., Csikszentmihalyi M. The Concept of Flow. In *Handbook of Positive Psychology*. Oxford University Press. 2002. P. 89–105.
25. Norman D. A. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books. 2013. URL: <https://mitpress.mit.edu/9780262525671/the-design-of-everyday-things/>
26. Чепендюк О.О., Манаков С.Ю. Інтеграція великих мовних моделей (LLM) для створення динамічних діалогових систем та обробки команд природною мовою в іграх жанру Тусооп. Інформаційні технології і автоматизація – 2025 / Матеріали XVIII міжнародної науково-практичної конференції. Одеса, 30-31 жовтня 2025 р. – Одеса, Видавництво ОНТУ, 2025 р. С. 1180-1181. URL: <https://ontu.edu.ua/download/konfi/2025/Collection-of-abstracts-of-the-conference-ITIA-2025.pdf>
27. Millington I., Funge J. *Artificial Intelligence for Games*. 3rd ed. CRC Press, 2019. 872 p.

28. Yannakakis G.N., Togelius J. Artificial Intelligence and Games. Springer, 2018. 342 p.
29. Rabin S. (Ed.) Game AI Pro 3: Collected Wisdom of Game AI Professionals. CRC Press, 2017. 598 p.
30. Zubek R. 1000 NPCs at 60 FPS // Game AI Pro 3: Collected Wisdom of Game AI Professionals. CRC Press, 2017. P. 403–414.
31. Jacopin É. Optimizing Practical Planning for Game AI // Game AI Pro 2: Collected Wisdom of Game AI Professionals. CRC Press, 2015. P. 145–160.
32. Park J.S., O'Brien J.C., Cai C.J., Morris M.R., Liang P., Bernstein M.S. Generative Agents: Interactive Simulacra of Human Behavior // Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23). 2023. DOI: 10.1145/3586183.3606763.
33. Park J.S. et al. Generative Agent Simulations of 1,000 People // arXiv preprint arXiv:2411.10109. 2024.
34. Shanahan M., McDonnell K., Reynolds L. Role play with large language models // Nature. 2023. Vol. 623. P. 493–498.
35. Why Language Models Hallucinate // OpenAI Research. 2025. URL: <https://cdn.openai.com/pdf/why-language-models-hallucinate.pdf>
36. OpenAI, Anthropic. Findings from a pilot Anthropic–OpenAI alignment safety evaluation. 2025. URL: <https://openai.com/index/openai-anthropic-safety-evaluation/>
37. Your 1M+ Context Window LLM Is Less Powerful Than You Think // Towards Data Science. 2025.
38. Aslan P. Understanding LLMs for Game NPCs // Master's Thesis, DiVA Portal. 2024.
39. Korkiakoski M. et al. An Empirical Evaluation of AI-Powered Non-Player Characters' Perceived Realism and Performance in Virtual Reality Environments // arXiv preprint arXiv:2507.10469. 2024.
40. Chen S. Literature review of Application of AI in improving gaming experiences // LUT University. 2024.

- 41.Iovino M., Scukins E., Styrud J., Ögren P., Smith C. A survey of Behavior Trees in robotics and AI // Robotics and Autonomous Systems. 2022. Vol. 154. DOI: 10.1016/j.robot.2022.104096.
- 42.Hill S. 5000 Characters at Video Frame Rate using Declarative Programming // CEUR Workshop Proceedings. 2024. Vol. 3926. P. 1–12.
- 43.Planning for millions of NPCs in Real-Time // IEEE Conference on Computational Intelligence and Games (CoG). 2022. DOI: 10.1109/CoG51982.2022.
- 44.Inworld AI. Character Engine Technical Documentation. 2024. URL: <https://inworld.ai/>
- 45.NVIDIA. ACE (Avatar Cloud Engine) for Games – Technical Documentation. NVIDIA Developer. 2024. URL: <https://developer.nvidia.com/ace>
- 46.Unity Technologies. NavMesh and Pathfinding // Unity Manual. 2024.
- 47.Unity Technologies. Best practices for profiling game performance // Unity Documentation. 2024.

ДОДАТКИ

Додаток А. Програмний код

```

1  using UnityEngine;
2
3  @ Unity Script (4 asset references) | 11 references
4  public class CameraFollow : MonoBehaviour
5  {
6      13 references
7      public static CameraFollow Instance { get; private set; }
8
9      [Header("Загальні налаштування")]
10     public Transform target;
11     public float smoothSpeed = 5f;
12
13     [Header("Режими слідування за гравцем")]
14     [Tooltip("Стандартна відстань камери від гравця.")]
15     public Vector3 playerOffset = new Vector3(0f, 3f, -2f);
16     [Tooltip("Відстань камери від гравця під час діалогу.")]
17     public Vector3 zoomedOffset = new Vector3(-1f, 1.5f, -2.5f); // Повернули це поле
18
19     [Header("Режим будівництва")]
20     [Tooltip("Позиція камери відносно об'єкта в режимі будівництва.")]
21     public Vector3 buildModeOffset = new Vector3(0f, 8f, -6f);
22     [Tooltip("Фіксований кут нахилу камери в режимі будівництва.")]
23     public Vector3 buildModeRotation = new Vector3(55f, 0f, 0f);
24
25     // Внутрішні змінні
26     private Transform originalTarget;
27     private bool isInBuildMode = false;
28     private Vector3 currentTargetOffset; // Повернули цю змінну
29
30     @ Unity Message | 0 references
31     private void Awake()
32     {
33         if (Instance != null && Instance != this) Destroy(gameObject);
34         else Instance = this;
35
36         currentTargetOffset = playerOffset; // Встановлюємо стандартний офсет на старті
37     }
38
39     @ Unity Message | 0 references
40     void LateUpdate()
41     {
42         if (target == null) return;
43
44         Vector3 desiredPosition;
45         Quaternion desiredRotation;
46
47         if (isInBuildMode)
48         {
49             // Логіка для режиму будівництва
50             desiredPosition = target.position + buildModeOffset;
51             desiredRotation = Quaternion.Euler(buildModeRotation);
52         }
53         else
54         {
55             // Логіка для слідування за гравцем (стандартна або наближена)
56             desiredPosition = target.position + currentTargetOffset; // Використовуємо поточний офсет
57             desiredRotation = Quaternion.LookRotation(target.position - transform.position);
58         }
59
60         transform.position = Vector3.Lerp(transform.position, desiredPosition, smoothSpeed * Time.deltaTime);
61         transform.rotation = Quaternion.Lerp(transform.rotation, desiredRotation, smoothSpeed * Time.deltaTime);
62     }
63
64     // Методи для режиму будівництва
65     1 reference
66     public void SetTemporaryTarget(Transform tempTarget)
67     {
68         if (originalTarget == null) originalTarget = target;
69         target = tempTarget;
70         isInBuildMode = true;
71     }
72
73     2 references
74     public void ClearTemporaryTarget()
75     {
76         if (originalTarget != null)

```

```

70         if (originalTarget != null)
71         {
72             target = originalTarget;
73             originalTarget = null;
74         }
75         isInBuildMode = false;
76         currentTargetOffset = playerOffset; // Завжди повертаємось до стандартного офсету
77     }
78
79     // --- ПОВЕРНУЛИ МЕТОДИ ДЛЯ ЗУМУ В ДІАЛОГАХ ---
80
81     /// <summary>
82     /// Наближує камеру для діалогу.
83     /// </summary>
84     1 reference
85     public void ZoomIn()
86     {
87         // Дозволяємо наближення тільки якщо ми НЕ в режимі будівництва
88         if (!isInBuildMode)
89         {
90             currentTargetOffset = zoomedOffset;
91         }
92     }
93
94     /// <summary>
95     /// Віддаляє камеру після діалогу.
96     /// </summary>
97     1 reference
98     public void ZoomOut()
99     {
100         currentTargetOffset = playerOffset;
101     }

```

```

1  using UnityEngine;
2  using TMPro;
3
4  // Unity Script (4 asset references) | 0 references
5  public class PlayerInteraction : MonoBehaviour
6  {
7      [Header("Налаштування")]
8      public KeyCode interactKey = KeyCode.E;
9      public float interactionDistance = 4f;
10     public TextMeshProUGUI interactionText;
11     public LayerMask interactableLayer;
12
13     private IInteractable closestInteractable;
14
15     // Unity Message | 0 references
16     private void Update()
17     {
18         CheckForInteractables();
19         HandleInput();
20     }
21
22     1 reference
23     private void CheckForInteractables()
24     {
25         Collider[] hitColliders = Physics.OverlapSphere(transform.position, interactionDistance, interactableLayer);
26         closestInteractable = null;
27         float closestDist = float.MaxValue;
28
29         foreach (Collider col in hitColliders)
30         {
31             IInteractable interactable = col.GetComponent<IInteractable>();
32             if (interactable == null) continue;
33
34             float dist = Vector3.Distance(transform.position, col.transform.position);
35             if (dist < closestDist)
36             {
37                 closestDist = dist;
38                 closestInteractable = interactable;
39             }
40         }
41     }

```

```

36         closestInteractable = interactable;
37     }
38 }
39     UpdateInteractionText();
40 }
41
42 1 reference
43 private void UpdateInteractionText()
44 {
45     if (interactionText == null) return;
46
47     if (closestInteractable == null)
48     {
49         interactionText.text = "";
50         return;
51     }
52
53     interactionText.text = closestInteractable.GetInteractionText(interactKey, 0, 0);
54 }
55
56 1 reference
57 private void HandleInput()
58 {
59     if (closestInteractable == null) return;
60
61     if (Input.GetKeyDown(interactKey))
62     {
63         closestInteractable.HandleInteraction(interactKey);
64     }
65 }
66 }

```

```

1 using UnityEngine;
2 using TMPro;
3 using System.Collections;
4
5 [RequireComponent(typeof(TextMeshPro))]
6 public class FloatingTextAnimator : MonoBehaviour
7 {
8     [Header("Налаштування Анімації")]
9     public float moveSpeed = 1f;
10    public float duration = 1.5f;
11    public float fadeOutTime = 0.5f;
12
13    private TextMeshPro textMesh;
14    private Camera mainCamera;
15    private Color startColor;
16    private Vector3 initialLocalPosition; // Початкова позиція відносно стола
17
18    void Awake()
19    {
20        textMesh = GetComponent<TextMeshPro>();
21        mainCamera = Camera.main;
22
23        // Запам'ятовуємо початкову *локальну* позицію
24        initialLocalPosition = transform.localPosition;
25    }
26
27    void Start()
28    {
29        // Повністю ховаємо текст на старті гри
30        textMesh.enabled = false;
31    }
32
33    /// <summary>
34    /// Цей метод викликає 'IncomePlatform', щоб запустити анімацію
35    /// </summary>
36    1 reference

```

```

36 1 reference
37 public void ShowIncome(string text, Color color)
38 {
39     // Якщо анімація вже йде, зупиняємо її, щоб почати нову
40     StopAllCoroutines();
41
42     // Повертаємо текст у початкову позицію
43     transform.localPosition = initialLocalPosition;
44
45     textMesh.text = text;
46     textMesh.color = color;
47     startColor = color;
48     textMesh.enabled = true; // Робимо текст видимим
49
50     StartCoroutine(Lifecycle());
51 }
52
53 1 reference
54 private IEnumerator Lifecycle()
55 {
56     float timer = 0f;
57     float mainPhaseDuration = duration - fadeOutTime;
58
59     // 1. Фаза руху (поки не час зникати)
60     while (timer < mainPhaseDuration)
61     {
62         // Рухаємось вгору (локально, відносно стола)
63         transform.localPosition += Vector3.up * moveSpeed * Time.deltaTime;
64         FaceCamera();
65         timer += Time.deltaTime;
66         yield return null;
67     }
68
69     // 2. Фаза згасання (Fade Out)
70     float fadeTimer = 0f;
71     while (fadeTimer < fadeOutTime)
72     {
73         transform.localPosition += Vector3.up * moveSpeed * Time.deltaTime;
74         FaceCamera();
75
76         // Змінюємо прозорість
77         float alpha = Mathf.Lerp(startColor.a, 0f, fadeTimer / fadeOutTime);
78         textMesh.color = new Color(startColor.r, startColor.g, startColor.b, alpha);
79
80         fadeTimer += Time.deltaTime;
81         yield return null;
82     }
83
84     // 3. Не знищуємо, а просто ховаємо
85     textMesh.enabled = false;
86 }
87
88 2 references
89 private void FaceCamera()
90 {
91     if (mainCamera != null)
92     {
93         transform.LookAt(transform.position + mainCamera.transform.rotation * Vector3.forward,
94             mainCamera.transform.rotation * Vector3.up);
95     }
96 }

```

```

1 using UnityEngine;
2
3 public class ComputerInteraction : MonoBehaviour, IInteractable
4 {
5     9 references
6     public static ComputerInteraction Instance { get; private set; }
7
8     public GameObject computerCanvas; // Канвас магазину
9
10    0 references
11    private void Awake()
12    {
13        if (Instance != null && Instance != this)
14        {
15            Destroy(gameObject);
16        }
17        else
18        {
19            Instance = this;
20        }
21
22    0 references
23    private void Start()
24    {
25        if (computerCanvas != null) computerCanvas.SetActive(false);
26    }
27
28    1 reference
29    public void OpenComputer()
30    {
31        if (computerCanvas == null) return;
32        computerCanvas.SetActive(true);
33        PlayerMovement.Instance.SetMovementState(false);
34        PlayerMovement.Instance.HideMainHUD();
35    }
36
37    0 references
38    public void CloseComputer()
39    {
40        if (computerCanvas == null) return;
41        computerCanvas.SetActive(false);
42        PlayerMovement.Instance.SetMovementState(true);
43        PlayerMovement.Instance.ShowMainHUD();
44    }
45
46    // Методи для PlacementManager
47    1 reference
48    public void HideShop()
49    {
50        if (computerCanvas != null) computerCanvas.SetActive(false);
51    }
52    2 references
53    public void ShowShop()
54    {
55        if (computerCanvas != null) computerCanvas.SetActive(true);
56    }
57
58    // --- Методи IInteractable ---
59    2 references
60    public void HandleInteraction(KeyCode key)
61    {
62        if (key == KeyCode.E && computerCanvas != null && !computerCanvas.activeSelf) OpenComputer();
63    }
64    1 reference
65    public InteractionType GetActiveInteractionType(KeyCode key)
66    {
67        if (key == KeyCode.E && computerCanvas != null && !computerCanvas.activeSelf) return InteractionType.General;
68        return InteractionType.None;
69    }
70    2 references
71    public string GetInteractionText(KeyCode key, float dist, float requiredDist)
72    {
73        if (key == KeyCode.E && computerCanvas != null && !computerCanvas.activeSelf) return "Використовувати комп'ютер [E]";
74        return "";
75    }
76 }

```

```
1 using UnityEngine;
2
3 @ Unity Script (1 asset reference) | 7 references
4 public class WorkstationPoint : MonoBehaviour
5 {
6     9 references
7     public bool IsOccupied { get; private set; } = false;
8     private Transform occupant;
9
10    1 reference
11    public bool Claim(Transform npc)
12    {
13        if (IsOccupied)
14        {
15            return false;
16        }
17
18        IsOccupied = true;
19        occupant = npc;
20        return true;
21    }
22
23    1 reference
24    public void Release(Transform npc)
25    {
26        if (occupant == npc)
27        {
28            IsOccupied = false;
29            occupant = null;
30        }
31    }
32
33    @ Unity Message | 0 references
34    private void Update()
35    {
36        if (IsOccupied && occupant == null)
37        {
38            IsOccupied = false;
39        }
40    }
41 }
```

```

1  using UnityEngine;
2  using TMPro;
3  using System.Collections;
4
5  @ Unity Script (1 asset reference) | 2 references
6  public class NPCSpeechBubble : MonoBehaviour
7  {
8      [Header("Налаштування")]
9      [Tooltip("Перетягніть сюди ваш 3D TextMeshPro об'єкт ('SpeechBubble3D').")]
10     public TextMeshPro textElement3D;
11
12     [Tooltip("Як довго (в секундах) фраза буде відображатися на екрані.")]
13     public float displayDuration = 5f;
14
15     [Tooltip("Об'єкт фону 3D хмарки (Quad).")]
16     public GameObject background3D;
17
18     private Coroutine currentSpeechRoutine;
19     private Camera mainCamera;
20
21     @ Unity Message | 0 references
22     void Awake()
23     {
24         if (textElement3D == null)
25         {
26             textElement3D = GetComponentInChildren<TextMeshPro>();
27         }
28
29         if (textElement3D != null)
30         {
31             textElement3D.gameObject.SetActive(false);
32         }
33
34         if (background3D != null)
35         {
36             background3D.SetActive(false);
37         }
38     }
39
40     @ Unity Message | 0 references
41     void Start()
42     {
43         mainCamera = Camera.main;
44     }
45
46     @ Unity Message | 0 references
47     void LateUpdate()
48     {
49         if (mainCamera != null && textElement3D != null && textElement3D.gameObject.activeSelf)
50         {
51             textElement3D.transform.LookAt(textElement3D.transform.position + mainCamera.transform.rotation * Vector3.forward,
52                                             mainCamera.transform.rotation * Vector3.up);
53         }
54     }
55
56     1 reference
57     public void ShowMessage(string message)
58     {
59         if (textElement3D == null) return;
60
61         if (currentSpeechRoutine != null)
62         {
63             StopCoroutine(currentSpeechRoutine);
64         }
65
66         currentSpeechRoutine = StartCoroutine(ShowMessageRoutine(message));
67     }
68
69     1 reference
70     private IEnumerator ShowMessageRoutine(string message)
71     {
72         textElement3D.text = message;
73         textElement3D.gameObject.SetActive(true);
74         if (background3D != null)
75         {
76             background3D.SetActive(true);
77         }
78
79         yield return new WaitForSeconds(displayDuration);
80
81         textElement3D.gameObject.SetActive(false);
82     }
83

```



```

1  using UnityEngine;
2  using UnityEngine.AI;
3
4  public class NPCBrain : MonoBehaviour
5  {
6      53 references
7      public enum State
8      {
9          IDLE,
10         GOING_TO_WORK,
11         WORKING,
12         GOING_TO_COFFEE,
13         COFFEE_BREAK,
14         GOING_TO_PLAYER,
15         REPORTING_TO_PLAYER,
16         GOING_HOME,
17         AT_HOME_WAITING
18     }
19
20     public State currentState = State.IDLE;
21
22     [Header("Налаштування часу")]
23     [Range(0, 1)] public float workStartTime = 0.33f; // 8:00
24     [Range(0, 1)] public float workEndTime = 0.875f; // 21:00
25
26     [Header("Інтервали між")]
27     public float waitInterval = 10f;
28     public float homeCheckInterval = 5f;
29     private float waitTimer;
30
31     [Header("Налаштування фраз (AI)")]
32     public float phraseDistance = 10f;
33
34     private NavMeshAgent agent;
35     private NPCActions actions;
36     private Animator animator;
37     private NPCSpeechBubble speechBubble;
38     private Transform playerTransform;
39
40     void Start()
41     {
42         agent = GetComponent<NavMeshAgent>();
43         actions = GetComponent<NPCActions>();
44         animator = GetComponentInChildren<Animator>();
45         speechBubble = GetComponent<NPCSpeechBubble>();
46
47         if (PlayerMovement.Instance != null)
48         {
49             playerTransform = PlayerMovement.Instance.transform;
50         }
51
52         Debug.Log($"[NPCBrain] {gameObject.name} ЗАПУЩЕНО.");
53
54         if (agent.stoppingDistance == 0)
55         {
56             agent.stoppingDistance = 0.5f;
57             Debug.LogWarning($"[NPCBrain] {gameObject.name}: Stopping Distance був 0. Автоматично встановлено на 0.5.");
58         }
59     }
60
61     void Update()
62     {
63         if (agent == null || actions == null || DayNightCycle.Instance == null) return;
64
65         float time = DayNightCycle.Instance.CurrentTimeNormalized;
66
67         if (animator != null)
68         {
69             bool isMoving = agent.pathPending || agent.remainingDistance > agent.stoppingDistance;
70             animator.SetBool("IsWalking", isMoving);
71
72             if (isMoving)
73             {
74                 animator.SetBool("IsWorking", false);
75                 animator.SetBool("IsDrinkingCoffee", false);
76             }
77         }

```

```

77
78 if (time >= workEndTime || time < workStartTime)
79 {
80     if (currentState != State.GOING_HOME && currentState != State.AT_HOME_WAITING)
81     {
82         SetNewState(State.GOING_HOME);
83     }
84     return;
85 }
86 else if (currentState == State.GOING_HOME || currentState == State.AT_HOME_WAITING)
87 {
88     MakeDecision();
89 }
90
91
92 if (agent.pathPending && agent.remainingDistance <= agent.stoppingDistance)
93 {
94
95     if (currentState == State.GOING_TO_WORK)
96     {
97         Debug.Log($"[NPCBrain] {gameObject.name} - ДІЙШОВ ДО РОБОТИ. Сідаю.");
98
99         // Отримуємо точну позицію і кут
100         Transform targetTransform = actions.GetCurrentWorkstationTarget();
101         if (targetTransform != null)
102         {
103             // Вмикаємо NavMeshAgent, щоб він не заважав нам "сісти"
104             agent.enabled = false;
105             // Телопортуюмо NPC ТОЧНО на місце і повертаємо його
106             transform.position = targetTransform.position;
107             transform.rotation = targetTransform.rotation;
108             // Вмикаємо NavMeshAgent назад (він буде неактивний)
109             agent.enabled = true;
110         }
111
112         SetNewState(State.WORKING);
113     }
114     else if (currentState == State.GOING_TO_COFFEE)
115     {
116         Debug.Log($"[NPCBrain] {gameObject.name} - ДІЙШОВ ДО КАВИ.");
117         SetNewState(State.COFFEE_BREAK);
118     }
119     else if (currentState == State.GOING_TO_PLAYER)
120     {
121         Debug.Log($"[NPCBrain] {gameObject.name} - ДІЙШОВ ДО ГРАВЦЯ.");
122         SetNewState(State.REPORTING_TO_PLAYER);
123     }
124     else if (currentState == State.GOING_HOME)
125     {
126         Debug.Log($"[NPCBrain] {gameObject.name} - ДІЙШОВ ДОДОМУ.");
127         SetNewState(State.AT_HOME_WAITING);
128     }
129
130     waitTimer += Time.deltaTime;
131
132     if ((currentState == State.AT_HOME_WAITING || currentState == State.REPORTING_TO_PLAYER) && waitTimer > homeCheckInterval)
133     {
134         MakeDecision();
135         waitTimer = 0;
136     }
137     else if ((currentState == State.WORKING || currentState == State.COFFEE_BREAK) && waitTimer > waitInterval)
138     {
139         MakeDecision();
140         waitTimer = 0;
141     }
142 }
143
144 else
145 {
146     waitTimer = 0;
147 }
148
149 }
150
151 3 references
152 void MakeDecision()
153 {
154     float time = DayNightCycle.Instance.CurrentTimeNormalized;
155     if (time >= workEndTime || time < workStartTime)
156     {
157         SetNewState(State.GOING_HOME);
158     }
159 }

```

```

157         return;
158     }
159
160     if (currentState == State.WORKING)
161     {
162         if (Random.value < 0.3f)
163         {
164             SetNewState(State.GOING_TO_COFFEE);
165         }
166     }
167     else if (currentState == State.COFFEE_BREAK || currentState == State.IDLE || currentState == State.AT_HOME_WAITING || currentState == State.REPORTING_TO_PLAYER)
168     {
169         bool foundTable = actions.FindAndClaimWorkstation();
170         if (foundTable)
171         {
172             SetNewState(State.GOING_TO_WORK);
173         }
174         else
175         {
176             SetNewState(State.GOING_TO_PLAYER);
177         }
178     }
179 }
180
181
182 9 references
183 void SetNewState(State newState)
184 {
185     currentState = newState;
186     Debug.Log($"[NPCBrain] {gameObject.name} - НОВИЙ СТАН: {newState.ToString()}");
187
188     if (animator != null)
189     {
190         animator.SetBool("IsWorking", false);
191         animator.SetBool("IsDrinkingCoffee", false);
192     }
193
194     switch (newState)
195     {
196     case State.GOING_TO_WORK:
197         GenerateActionPhrase(State.GOING_TO_WORK);
198         break;
199     case State.WORKING:
200         if (animator != null) animator.SetBool("IsWorking", true);
201         GenerateActionPhrase(State.WORKING);
202         break;
203     case State.GOING_TO_COFFEE:
204         actions.GoToCoffeeMachine();
205         GenerateActionPhrase(State.GOING_TO_COFFEE);
206         break;
207     case State.COFFEE_BREAK:
208         if (animator != null) animator.SetBool("IsDrinkingCoffee", true);
209         GenerateActionPhrase(State.COFFEE_BREAK);
210         break;
211     case State.GOING_TO_PLAYER:
212         actions.GoToPlayer(playerTransform);
213         GenerateActionPhrase(State.REPORTING_TO_PLAYER);
214         break;
215     case State.REPORTING_TO_PLAYER:
216         // Зараз мене з'являється
217         break;
218     case State.GOING_HOME:
219         actions.GoHome();
220         GenerateActionPhrase(State.GOING_HOME);
221         break;
222     case State.AT_HOME_WAITING:
223         break;
224     case State.IDLE:
225         actions.WanderAround();
226         break;
227     }
228 }
229
230 6 references
231 void GenerateActionPhrase(State action, float timeOverride = -1f)
232 {
233     if (playerTransform == null) return;
234     float distanceToPlayer = Vector3.Distance(transform.position, playerTransform.position);
235     if (distanceToPlayer > phraseDistance && action != State.REPORTING_TO_PLAYER) return;
236
237     if (speechBubble == null || APIManager.Instance == null) return;
238     float time = (timeOverride != -1f) ? timeOverride : DayNightCycle.Instance.CurrentTimeNormalized;
239     string timeString = DayNightCycle.Instance.GetCurrentTimeAsString();
240     string context = "";
241     string lengthConstraint = "Твоя відповідь має бути ДУЖЕ КОРОТКОЮ, до 5 слів (максимум 20 символів).";
242     switch (action)
243     {
244     case State.GOING_TO_WORK: context = $"Ти офісний працівник. Зараз {timeString}. Ти йдеш працювати. {lengthConstraint}"; break;
245     case State.WORKING: context = $"Ти офісний працівник. Зараз {timeString}. Ти працюєш. {lengthConstraint} Скажи робочу думку."; break;
246     case State.GOING_TO_COFFEE: context = $"Ти офісний працівник. Зараз {timeString}. Ти вирішив зробити перерву на каву. {lengthConstraint}"; break;
247     case State.COFFEE_BREAK: context = $"Ти офісний працівник. Зараз {timeString}. Ти п'єш каву. {lengthConstraint} Подумай про відпочинок."; break;
248     case State.REPORTING_TO_PLAYER: context = $"Ти офісний працівник. Ти прийшов поскаржитися гравцю (босу), що немає вільних столів. {lengthConstraint}"; break;
249     case State.GOING_HOME: context = $"Ти офісний працівник. Зараз {timeString}, кінець дня. Ти йдеш додому. {lengthConstraint}"; break;
250     case State.IDLE: context = $"Ти офісний працівник. Зараз {timeString}. Ти блукаєш офісом. {lengthConstraint}"; break;
251     }
252
253     if (string.IsNullOrEmpty(context)) return;
254     Debug.Log($"[NPCBrain] {gameObject.name} | [AI Запит]: {context}");
255     APIManager.Instance.SendPromptToGemini(context, (generatedPhrase) =>
256     {
257         Debug.Log($"[NPCBrain] {gameObject.name} | [AI Відповідь]: {generatedPhrase}");
258         speechBubble.ShowMessage(generatedPhrase);
259     });
260 }
261
262 0 references
263 string timeString(float time)
264 {
265     float hours = 24 * time;
266     float minutes = 60 * (hours - Mathf.Floor(hours));
267     return string.Format("{0:00}:{1:00}", (int)hours, (int)minutes);
268 }

```

```

1  using UnityEngine;
2  using UnityEngine.AI; // Потрібно для NavMesh
3
4  [RequireComponent(typeof(NavMeshAgent))]
5  public class GoToCoffee : MonoBehaviour
6  {
7      [Header("Ціль")]
8      [Tooltip("CoffeePoint")]
9      public Transform coffeeMachineTarget;
10
11      [Header("Компоненти")]
12      [Tooltip("Аніматор цього NPC.")]
13      public Animator animator;
14
15      private NavMeshAgent agent;
16
17      void Start()
18      {
19          agent = GetComponent<NavMeshAgent>();
20
21          if (animator == null)
22          {
23              animator = GetComponent<Animator>();
24          }
25
26          if (coffeeMachineTarget != null)
27          {
28              agent.SetDestination(coffeeMachineTarget.position);
29          }
30          else
31          {
32              Debug.LogError("Ціль 'CoffeeMachineTarget' не призначена! NPC не знає, куди йти.", this);
33          }
34      }
35
36      void Update()
37      {
38          if (animator != null)
39          {
40              // Перевіряємо, чи агент зараз рухається (його швидкість більша за 0.1f)
41              if (agent.velocity.magnitude > 0.1f)
42              {
43                  animator.SetBool("IsWalking", true);
44              }
45              else
46              {
47                  animator.SetBool("IsWalking", false);
48              }
49          }
50      }
51  }

```

```

1  using UnityEngine;
2  using UnityEngine.AI;
3
4  public class NPCActions : MonoBehaviour
5  {
6      private NavMeshAgent agent;
7
8      [Header("Тегі Призначення")]
9      [Tooltip("Тег для точки 'Дім' (має бути на пустому об'єкті)")]
10     public string homePointTag = "HomePoint";
11     [Tooltip("Тег для кавової машини, напр. 'CoffeeMachine'")]
12     public string coffeeMachineTag = "CoffeeMachine";
13
14     private WorkstationPoint currentWorkstation;
15     private Transform currentTarget;
16     private Transform homePointTransform;
17
18     void Start()
19     {
20         agent = GetComponent<NavMeshAgent>();
21
22         GameObject homeObject = GameObject.FindWithTag(homePointTag);
23         if (homeObject != null)
24         {
25             homePointTransform = homeObject.transform;
26         }
27         else
28         {
29             Debug.LogError($"[NPCActions] {gameObject.name}: НЕ ЗНАЙДЕНО 'HomePoint' з тегом '{homePointTag}'!");
30         }
31     }
32
33     1 reference
34     public bool FindAndClaimWorkstation()
35     {
36         ReleaseCurrentWorkstation();
37
38         WorkstationPoint[] allWorkstations = FindObjectsOfType<WorkstationPoint>();
39         WorkstationPoint freeWorkstation = null;
40
41         foreach (WorkstationPoint station in allWorkstations)
42         {
43             if (!station.IsOccupied)
44             {
45                 freeWorkstation = station;
46                 break;
47             }
48         }
49
50         if (freeWorkstation != null)
51         {
52             freeWorkstation.Claim(transform);
53             currentWorkstation = freeWorkstation;
54             currentTarget = currentWorkstation.transform;
55             agent.SetDestination(currentTarget.position);
56             return true;
57         }
58         else
59         {
60             currentTarget = null;
61             return false;
62         }
63     }
64
65     1 reference
66     public Transform GetCurrentWorkstationTarget()
67     {
68         return currentTarget;
69     }
70
71     1 reference
72     public void GoToCoffeeMachine()
73     {
74         ReleaseCurrentWorkstation();
75         GameObject target = GameObject.FindWithTag(coffeeMachineTag);
76         if (target != null)
77         {
78             agent.SetDestination(target.position);
79         }
80     }

```

```

75         {
76             agent.SetDestination(target.transform.position);
77         }
78     }
79
80     2 references
81     public void GoHome()
82     {
83         ReleaseCurrentWorkstation();
84         if (homePointTransform != null)
85         {
86             agent.SetDestination(homePointTransform.position);
87         }
88     }
89
90     1 reference
91     public void GoToPlayer(Transform playerTransform)
92     {
93         ReleaseCurrentWorkstation();
94         if (playerTransform == null) return;
95         if (NavMesh.SamplePosition(playerTransform.position, out NavMeshHit hit, 3.0f, NavMesh.AllAreas))
96         {
97             agent.SetDestination(hit.position);
98         }
99         else
100         {
101             GoHome();
102         }
103     }
104
105     4 references
106     private void ReleaseCurrentWorkstation()
107     {
108         if (currentWorkstation != null)
109         {
110             currentWorkstation.Release(transform);
111             currentWorkstation = null;
112         }
113     }
114
115     1 reference
116     public void WanderAround()
117     {
118         Vector3 randomPos = RandomNavSphere(transform.position, 10f, -1);
119         agent.SetDestination(randomPos);
120     }
121
122     1 reference
123     public static Vector3 RandomNavSphere(Vector3 origin, float dist, int layermask)
124     {
125         Vector3 randDirection = Random.insideUnitSphere * dist;
126         randDirection += origin;
127         NavMesh.SamplePosition(randDirection, out NavMeshHit navHit, dist, layermask);
128         return navHit.position;
129     }

```

```

1  using UnityEngine;
2  using TMPro;
3
4  public class DayNightCycle : MonoBehaviour
5  {
6      8 references
7      public static DayNightCycle Instance { get; private set; }
8
9      [Header("Налаштування часу")]
10     public float dayDurationInSeconds = 120f;
11
12     [Header("Посилання на об'єкти")]
13     public Light sun;
14     public TextMeshProUGUI clockText;
15
16     [Header("Налаштування кольорів")]
17     public Gradient skyColor;
18     public Gradient ambientColor;
19
20     9 references
21     public float CurrentTimeNormalized { get; private set; } = 0.25f; // Починаємо з 6:00 ранку
22
23     0 Unity Message | 0 references
24     private void Awake()
25     {
26         if (Instance != null && Instance != this) Destroy(gameObject);
27         else Instance = this;
28     }
29
30     0 Unity Message | 0 references
31     void Update()
32     {
33         UpdateTime();
34         UpdateSunPosition();
35         UpdateLightingColors();
36         UpdateClock();
37     }
38
39     1 reference
40     private void UpdateTime()
41     {
42         CurrentTimeNormalized += Time.deltaTime / dayDurationInSeconds;
43         CurrentTimeNormalized %= 1;
44     }
45
46     2 references
47     public string GetCurrentTimeAsString()
48     {
49         float hours = 24 * CurrentTimeNormalized;
50         float minutes = 60 * (hours - Mathf.Floor(hours));
51         return string.Format("{0:00}:{1:00}", (int)hours, (int)minutes);
52     }
53
54     1 reference
55     private void UpdateSunPosition()
56     {
57         if (sun == null) return;
58         sun.transform.rotation = Quaternion.Euler((CurrentTimeNormalized * 360f) - 90f, 170f, 0);
59     }
60
61     1 reference
62     private void UpdateLightingColors()
63     {
64         if (sun != null) sun.color = skyColor.Evaluate(CurrentTimeNormalized);
65         RenderSettings.ambientLight = ambientColor.Evaluate(CurrentTimeNormalized);
66     }
67
68     1 reference
69     private void UpdateClock()
70     {
71         if (clockText == null) return;
72         clockText.text = GetCurrentTimeAsString();
73     }
74 }

```

```

1  using UnityEngine;
2  using System.Collections;
3  using UnityEngine.Networking;
4  using TMPro;
5  using System;
6
7  // Unity Script (4 asset references) | 4 references
8  public class APIManager : MonoBehaviour
9  {
10     // 6 references
11     public static APIManager Instance { get; private set; }
12
13     private const string GAS_URL = "https://script.google.com/macros/s/AKfycbyT8EubC_ModriGU39G_AR4a0jG3yHlYvg68iaejfMyjwLn2BPxEPTjiQ239aQJUXbQ_TQ/exec";
14     [Header("UI Reference")]
15     public TextMeshProUGUI debugResponseText;
16
17     // Unity Message | 0 references
18     private void Awake()
19     {
20         if (Instance != null && Instance != this)
21         {
22             Destroy(gameObject);
23         }
24         else
25         {
26             Instance = this;
27         }
28     }
29
30     // 2 references
31     public void SendPromptToGemini(string prompt, Action<string> callback)
32     {
33         Debug.Log($"APIManager: Надсилаємо запит...");
34         StartCoroutine(SendRequestCoroutine(prompt, callback));
35     }
36
37     // 1 reference
38     private IEnumerator SendRequestCoroutine(string prompt, Action<string> callback)
39     {
40         WWWForm form = new WWWForm();
41         form.AddField("parameter", prompt);
42
43         using (UnityWebRequest www = UnityWebRequest.Post(GAS_URL, form))
44         {
45             yield return www.SendWebRequest();
46
47             string geminiResponse = "";
48
49             if (www.result == UnityWebRequest.Result.Success)
50             {
51                 geminiResponse = www.downloadHandler.text;
52             }
53             else
54             {
55                 Debug.LogError($"Помилка запиту: {www.error}.");
56                 geminiResponse = "Помилка: Не вдалося підключитися.";
57             }
58
59             callback?.Invoke(geminiResponse);
60
61             if (debugResponseText != null)
62             {
63                 debugResponseText.text = geminiResponse;
64             }
65         }
66     }
67 }

```


Додаток Б. Копії матеріалів апробації

УДК 004.588

ІНТЕГРАЦІЯ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ (LLM) ДЛЯ СТВОРЕННЯ ДИНАМІЧНИХ ДІАЛОГОВИХ СИСТЕМ ТА ОБРОБКИ КОМАНД ПРИРОДНОЮ МООВОЮ В ІГРАХ ЖАНРУ TYSCOON

Чепендюк О.О., Манаков С.Ю.

Національний університет «Одеська юридична академія» (Україна)

Постановка проблеми

Класичні ігри жанру idle/tycoon, попри захоплюючі економічні моделі, часто страждають від обмеженої взаємодії з неігровими персонажами (NPC). Їхні діалоги, як правило, базуються на статичних, заздалегідь прописаних сценаріях, що робить їх повторюваними та знижує занурення гравця. Крім того, створення великого обсягу унікального текстового контенту є значним викликом для розробників. Сучасні великі мовні моделі (LLM) пропонують технологічне рішення цих проблем, дозволяючи генерувати динамічний контент та створювати нові механіки взаємодії.

Аналіз останніх досліджень та публікацій

Традиційні підходи до симуляції спілкування в іграх включають дерева діалогів, які є обмеженими та передбачуваними [4]. Ранні методи процедурної генерації (PCG) намагалися вирішити проблему повторюваності, однак часто створювали неприродний, шаблонний текст [5]. Поведінка NPC зазвичай керується алгоритмічним AI, таким як кінцеві автомати (FSM), який ефективний для простих дій, але не для осмисленого спілкування [4]. Новітні дослідження в галузі генеративного AI демонструють здатність LLM генерувати високоякісний, контекстно-залежний текст, що практично не відрізняється від написаного людиною [6]. Проте, практичних архітектурних рішень для їх інтеграції в ігровий процес жанру tycoon все ще недостатньо.

Постановка задачі

Задача даної роботи полягає у розробці архітектури та методології інтеграції великих мовних моделей (LLM) в ігровий рушій Unity для реалізації ігрових механік нового покоління в жанрі tycoon. Це включає в себе проектування програмного інтерфейсу для асинхронної взаємодії, встановлення процедур формування контекстно-залежних запитів (промптів), формування критеріїв оцінки якості згенерованих діалогів та точності розпізнавання команд, а також розробку рекомендацій щодо практичного впровадження моделі. Особлива увага приділяється адаптації LLM до специфіки ігрового середовища та визначенню меж застосовності даної технології порівняно з традиційними скриптовими системами.

Виклад основного матеріалу

В рамках даного дослідження було розроблено програмний прототип гри в жанрі tycoon на базі багатоплатформового ігрового рушія Unity [1]. Спроектвана система є модульною, що забезпечує її гнучкість та розширюваність. Основою архітектури взаємодії є програмний інтерфейс, який виступає в ролі "контракту" для всіх інтерактивних об'єктів. Головний компонент, що закріплений за об'єктом гравця, у реальному часі сканує оточення за допомогою фізичних запитів (Physics Overlap Sphere) для виявлення об'єктів, що реалізують даний інтерфейс. Такий підхід дозволяє повністю відокремити логіку гравця від логіки інтерактивних об'єктів, що дає змогу легко додавати нові типи NPC або інтерактивних предметів, не вносячи змін у керуючий скрипт персонажа.

Для зв'язку з великою мовною моделлю Google Gemini було створено централізований програмний менеджер, реалізований за патерном Singleton, що гарантує єдину точку доступу до API з будь-якої частини програмного коду [2, 6]. Взаємодія з хмарним сервісом відбувається асинхронно. Щоб уникнути "зависання" ігрового процесу під час очікування відповіді від сервера, яке може тривати кілька секунд, система використовує вбудовані в Unity корутини. Запит відправляється у фоновому потоці, а після отримання відповіді виконується метод зворотного виклику (callback), який безпечно оновлює елементи інтерфейсу користувача. Це забезпечує плавність ігрового процесу, що є критично важливим для позитивного користувацького досвіду.

Ключовою особливістю розробленої діалогової системи є динамічне формування запитів до LLM на основі принципів "prompt engineering". Для кожного інтерактивного персонажа в редакторі Unity задається спеціальний системний промпт — текстовий блок інструкцій, що визначає його роль, характер, стиль мовлення та обсяг знань (наприклад, "Ти — офісний менеджер, відповідай ввічливо, але коротко"). Ця інструкція автоматично додається на початок кожного запиту, який гравець вводить у діалогове вікно. Таким чином, система "скеровує" загальноцільову мовну модель, змушуючи її генерувати відповіді, що відповідають образу конкретного NPC та загальному тону гри.

Окрім генерації діалогів, було реалізовано інноваційну механіку керування грою через природну мову, що є прикладом гібридної системи вводу. Перед відправкою тексту гравця до AI, система проводить його локальний аналіз на наявність ключових слів-команд (наприклад, "прокачай", "покращити"). Якщо команда розпізнана, система перериває діалоговий потік і виконує відповідну ігрову дію (наприклад, викликає метод покращення об'єкта), надаючи гравцеві миттєвий фідбек. Якщо ж команда не знайдена, текст обробляється як звичайне діалогове питання і відправляється до LLM. Такий підхід поєднує вільне спілкування та функціональне керування в єдиному текстовому інтерфейсі, що є актуальним завданням сучасного геймдизайну та значно поглиблює занурення гравця в роль [3].

Практична реалізація підтвердила, що розроблена архітектура є життєздатною та ефективною. Наукова новизна полягає у створенні комплексної гібридної системи, де LLM використовується одночасно для вирішення двох різних завдань: генерації креативного контенту (нескінченно варіативних діалогів) та функціонального керування ігровими процесами (обробка команд). Практичне значення роботи полягає в тому, що запропонований підхід може слугувати архітектурним шаблоном для розробників, що прагнуть інтегрувати сучасні AI-технології в ігри жанру tycoon для підвищення їх імєрсивності та реіграбельності.

Висновки

Проведене дослідження підтвердило високий потенціал інтеграції LLM для підвищення імєрсивності та реіграбельності в іграх жанру tycoon. Розроблена система успішно вирішила поставлені завдання. Водночас були виявлені ключові виклики: затримка відповіді від API, що вимагає відповідного дизайну UI, та необхідність ретельного проектування системних промптів. Розроблена архітектура є гнучкою і може бути адаптована для інших ігрових проєктів.

Перелік джерел посилання

1. Unity Technologies. (2024). Unity User Manual. URL: <https://docs.unity3d.com/Manual/index.html>.
2. Google. (2024). Gemini API Documentation. URL: <https://ai.google.dev/docs>.
3. Schell, J. (2019). The Art of Game Design: A Book of Lenses, Third Edition. CRC Press.
4. Millington, I., & Funge, J. (2009). Artificial Intelligence for Games, Second Edition. CRC Press.
5. Shaker, N., Togelius, J., & Nelson, M. J. (2016). Procedural Content Generation in Games. Springer.
6. Gemini Team, Google. (2023). Gemini: A Family of Highly Capable Multimodal Models. arXiv preprint. URL: <https://arxiv.org/abs/2312.11805>.