ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

Пояснювальна записка

до кваліфікаційної роботи
магістра
(освітній рівень)

на тему: «Дослідження ефективності методів автоматизованого тестування комп'ютерних ігор»

Виконав: студент групи 6ПР2
спеціальності
121 - «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

Мацалковський Ярослав Вікторович_____
(прізвище та ініціали)
Керівник д.т.н., проф. Шерстюк В.Г.___
(прізвище та ініціали)
Рецензент к.т.н. доцент Козел В.М.
(прізвище та ініціали)

Хмельницький - 2025

Херсонський національний технічний університет
(повне найменування вищого навчального закладу)

Факультет, відділення Інформаційних технологій та дизайну
Кафедра Програмних засобів і технологій
Освітній рівень магістр
(шифр і назва)
Напрям підготовки ОПП - Програмне забезпечення систем
Спеціальність 121 – Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗіТ
к.т.н. доц. О.Є. Огнєва
“____” _____ 2025 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Мацалковському Ярославу Вікторовичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження ефективності методів автоматизованого тестування комп’ютерних ігор»
керівник роботи д.т.н., проф. Шерстюк В.Г.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджена наказом вищого навчального закладу від 15.09.2025 р. №417-С
2. Строк подання студентом роботи 15.12.2025
3. Вихідні дані до роботи літературні та періодичні джерела, методи автоматизації тестування, мова C++, Unreal Engine 5
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): 1) вивчення доступних алгоритмів автоматизації; 2) аналіз переваг і недоліків; 3) класифікація поточних методів; 4) розроблення підходу, який буде поєднувати ефективні методи для підвищення ефективності процесу; 5) розроблення програмного забезпечення, що буде покривати ці методи.
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
1) Техніки тестування, вхідні дані та модель; 2) Порівняння розглянутих аналогів; 3) Запропонована схема GDLC; 4) Піраміда функціонального тестування з урахуванням часу; 5) Кореляція якісних критеріїв і основних етапів розробки по Кейну; 6) Схема роботи алгоритму навчання з підкріпленням; 7) Порівняння методів відтворення

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 29.09.2025

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів виконання роботи	Термін виконання етапів роботи	Примітки
1.	Отримання завдання	29.09.2025	Виконано
2.	Підбір літератури	05.10.2025	Виконано
3.	Аналіз предметної області	19.10.2025	Виконано
4.	Розробка та обґрунтування завдання	26.10.2025	Виконано
5.	Розробка концептуальної моделі	05.11.2025	Виконано
6.	Розробка алгоритму	12.11.2025	Виконано
7.	Проектування програми	26.11.2025	Виконано
8.	Розробка інтерфейсу програми	30.11.2025	Виконано
9.	Тестування програми	05.12.2025	Виконано
10.	Оформлення пояснювальної записки	10.12.2025	Виконано
11.	Захист кваліфікаційної роботи	15.12.2025	Виконано

Студент _____ Мацалковський Я.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Шерстюк В.Г.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 116 сторінок, 46 рисунків, 14 таблиць, 1 додаток, 53 джерела.

Об'єкт дослідження: процес розроблення і тестування відеоігор.

Предмет дослідження: метод автоматизації тестування з розширенням рівнів тестування.

Мета дослідження: пришвидшити процес розробки та підвищити його ефективність, представивши підхід до автоматизації тестування для використання в розробці ігор, оскільки ця сфера є унікальною та вимагає окремого підходу до тестування.

Новизна отриманих результатів полягає в тому, що модифіковано метод автоматизації тестування відеоігор, який за рахунок розширення піраміди тестування двома додатковими рівнями дозволяє зменшити кількість мануальної роботи та спростити процес розробки і тестування програмного забезпечення відеоігор. Метод враховує необхідність швидких змін, зберігаючи при цьому високу функціональну якість.

Практична цінність результатів роботи полягає у тому, що запропоновано модифікацію методу тестування та розроблено програмне забезпечення для вдосконалення методики тестування програмного забезпечення. Розроблене рішення має зменшити час, затрачений на мануальне тестування тих частин ПЗ, які до цього не могли бути автоматизовані.

Перелік ключових слів: автоматизація тестування, піраміда тестування, assert, нефункціональне тестування, end-to-end тестування, життєвий цикл розробки відеоігор.

АНОТАЦІЯ

Кваліфікаційна робота магістра складається зі вступу, чотирьох розділів, висновку, переліку використаних джерел та додатків.

Роботу присвячено питанням автоматизації процесів тестування, що може значно прискорити і масштабувати процес розроблення, де основним методом досі залишалось мануальне тестування. Більшість технік автоматизації тестування можуть бути використані для розробки симуляторів для тренування військових, ще є актуальним питанням сьогодення.

В роботі було вивчено доступні алгоритми автоматизації; проаналізовано їх переваги і недоліки; досліджено класифікацію поточних методів; розроблено підхід, який поєднує методи для підвищення ефективності процесу розробки та розроблення програмного забезпечення, що реалізує деякі з цих методів.

Запропоновано спосіб тестування програмного забезпечення відеоігор на рівні End-to-End, що відрізняється від існуючих комбінуванням декількох механізмів тестування з фіксуванням точок проходження, що дозволяє підвищити стійкість відтворення дій гравця. В майбутньому можливо враховувати мультидисциплінарну природу домену за допомогою навчання з підкріпленням, що закладено в модифікованому методі, яке підвищить нефункціональну якість програмного забезпечення відеоігор.

В результаті виконання роботи запропоновано модифікацію методу тестування та розроблено програмне забезпечення для вдосконалення методики тестування програмного забезпечення. Розроблене рішення має зменшити час, затрачений на мануальне тестування тих частин ПЗ, які до цього не могли бути автоматизовані.

ABSTRACT

The master's thesis consists of an introduction, four chapters, a conclusion, a list of sources and appendices.

The thesis is devoted to the issues of automation of testing processes, which can significantly accelerate and scale the development process, where manual testing has remained the main method. Most of the testing automation techniques can be used to develop simulators for military training, which is still a relevant issue today.

The thesis studied the available automation algorithms; analyzed their advantages and disadvantages; investigated the classification of current methods; developed an approach that combines methods to improve the efficiency of the development process and software development that implements some of these methods.

A method for testing video game software at the End-to-End level is proposed, which differs from existing ones by combining several testing mechanisms with fixed points of passage, which allows to increase the stability of the reproduction of player actions. In the future, it is possible to take into account the multidisciplinary nature of the domain using reinforcement learning, which is inherent in the modified method, which will increase the non-functional quality of video game software.

As a result of the thesis, a modification of the testing method was proposed and software was developed to improve the software testing methodology. The developed solution should reduce the time spent on manual testing of those parts of the software that could not previously be automated.

ЗМІСТ

ВСТУП

Серед інших сфер розробки програмного забезпечення відеоігри є однією з найменш протестованих і однією з найскладніших. У сучасних реаліях виправлення сучасної гри після випуску до стану без помилок займає роки, а сама розробка до релізу потребує ще більше часу та ресурсів.

Ринок праці переповнений фахівцями з контролю якості, але кількість багів у випущених іграх не зменшується. Навпаки, більш сучасні, більші по розміру відеоігри виходять з більшою кількістю багів, навіть якщо вони мають більш ніж достатньо QA. Для їх випуску потрібно набагато більше часу та набагато більше грошей.

Однією з причин цього є недостатній рівень автоматизації тестування під час розробки гри. На рис. 1 можна побачити порівняння двох часових шкал розробки. Жовта лінія показує розробку з автоматизацією тестування. І ми бачимо, що кількість помилок набагато менша.

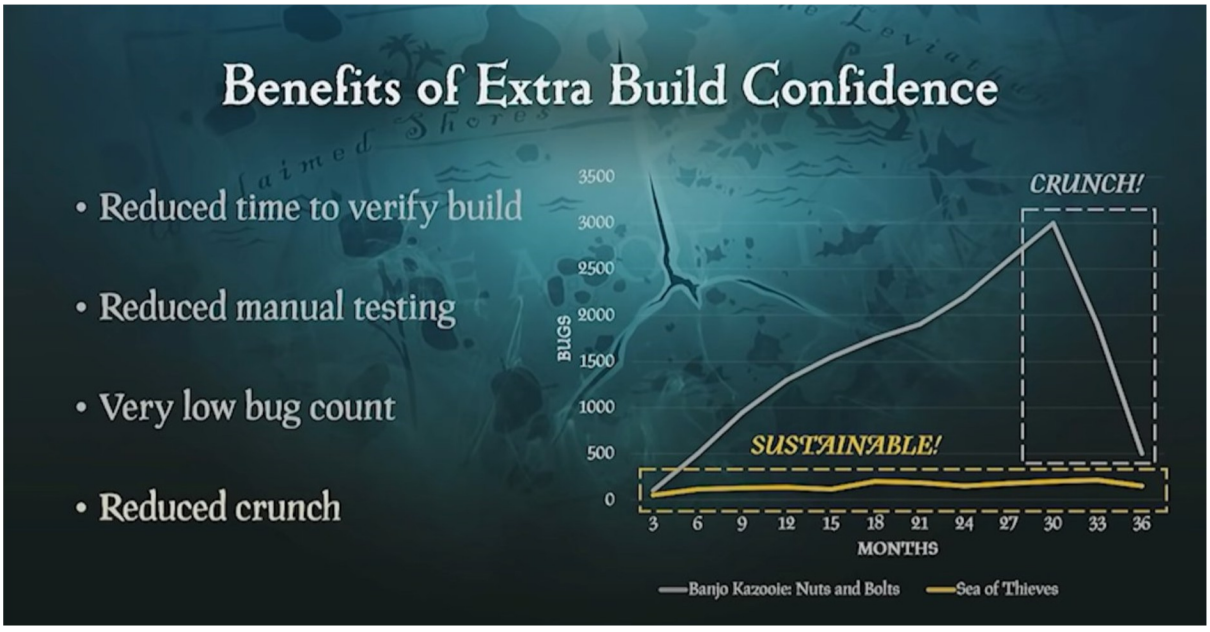

Рис. 1. Результати впровадження автоматизації тестування [18]

Автоматизація тестування допомагає пришвидшити процес розробки та ефективніше розробляти відеоігри. Але темпи автоматизації тестування в галузі низькі. Класичні засоби автоматизації тестування не пристосовані до сфери розробки відеоігор, до того ж вони не охоплюють нефункціональні

рівні розробки. Це викликано характером галузі зі швидкими змінами в дизайні та вимогах і її мультидисциплінарністю, де коректність виконання не є достатнім критерієм для оцінки якості програмного забезпечення.

Мануальне тестування, на відміну від автоматизованого погано масштабується. Автоматизація процесів може значно прискорити і масштабувати процеси розробки, де основним методом залишалось мануальне тестування. Багато з технік автоматизації також можуть бути використані для розробки симуляторів для тренування військових, тож це ще додатково підвищує актуальність даної тематики у період війни в Україні.

Мета дослідження полягає в тому, щоб пришвидшити процес розробки та підвищити його ефективність, представивши підхід до автоматизації тестування для використання саме в розробці ігор, оскільки ця сфера є унікальною та вимагає окремого підходу до тестування.

Для досягнення цієї мети було заявлено такі основні завдання: вивчення доступних алгоритмів автоматизації; аналіз їхніх переваг і недоліків; дослідження класифікації поточних методів; розроблення підходу, який буде поєднувати методи для підвищення ефективності процесу розроблення та розробку програмного забезпечення, що буде реалізувати деякі з цих методів.

Об'єктом дослідження є процес розробки і тестування відеоігор.

Предметом дослідження є метод автоматизації тестування з розширенням рівнів тестування.

Автоматизація тестування відеоігор зараз на досить низькому рівні через важкість її впровадження та через те, що автоматизовані тести не покривають нефункціональні рівні розробки. Дослідження є доцільним, оскільки у ньому розглядається новий підхід, що має вирішити ці проблеми.

Модифіковано метод автоматизації тестування відеоігор, який за рахунок розширення піраміди тестування двома додатковими рівнями, дозволяє зменшити кількість мануальної роботи та спростити процес розробки і тестування програмного забезпечення відеоігор. Метод враховує

необхідність швидких змін, зберігаючи при цьому високу функціональну якість.

Запропоновано спосіб тестування програмного забезпечення відеоігор на рівні End-to-End, який відрізняється від існуючих комбінуванням декількох механізмів тестування з фіксуванням точок проходження, що дозволяє підвищити стійкість відтворення дій гравця.

В майбутньому можливо враховувати мультидисциплінарну природу домену за допомогою навчання з підкріпленням, що закладено в модифікованому методі, яке підвищить нефункціональну якість програмного забезпечення відеоігор.

Практичні результати роботи полягають в тому, що запропоновано модифікацію методу тестування та розроблено програмне забезпечення для вдосконалення методики тестування програмного забезпечення.

Розроблене рішення має зменшити час, затрачений на мануальне тестування тих частин ПЗ, які до цього не могли бути автоматизовані.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Огляд існуючих рішень автоматизації End-to-End тестування відеоігор

End-to-End тестування програмного забезпечення – це не нова ідея. Але для ігор його реалізація є на порядок складнішою, ніж для інших типів програм, через складність і комплексність систем, що розглядаються.

Перші успішні рішення у цій області почали з’являтись у 2015 році. Ця область дуже перспективна, оскільки може допомогти сильно скоротити строки розробки завдяки оптимізації процесів тестування. Чим раніше буде знайдена проблема і помилка, тим дешевше і швидше її можна виправити.

Використовуючи автоматизацію тестування можна робити значно глибші тести значно частіше. За ніч можна зробити велику частину Smoke, Regression та Performance тестування для великих проектів, отримати дані, проаналізувати їх. І витрачати значно менше часу тестувальників на це.

1.1.1. The Division 2

Ubisoft при розробці доповнення до The Division 2 використовувала процедурну генерацію для деяких рівнів [1], і її за визначенням неможливо було відтестувати стандартними методами.

Для тестування The Division 2 крім стандартних методів використовувались ще 2 види ботів: клієнтські та серверні.

Серверні боти застосовуються для навантажувального тестування серверів. Боти симулюють багато дій, які відправляють на сервер, при цьому не слідуючи правилам гри. Їх ціль – навантажити сервери і протестувати як вони будуть себе під навантаженням гравців.

Клієнтські боти почали використовуватись для тестування процедурно згенерованого контенту, оскільки його неможливо відтестувати вручну. Цих ботів навчили проходити рівні, і використовували для Smoke, Performance та Regression тестування, що значно полегшило випуск гри. У матеріалі описані

варіанти використання таких ботів та нову методику тестування, яка прийшла з новим інструментом тестування, можна побачити на рисунку 1.1.

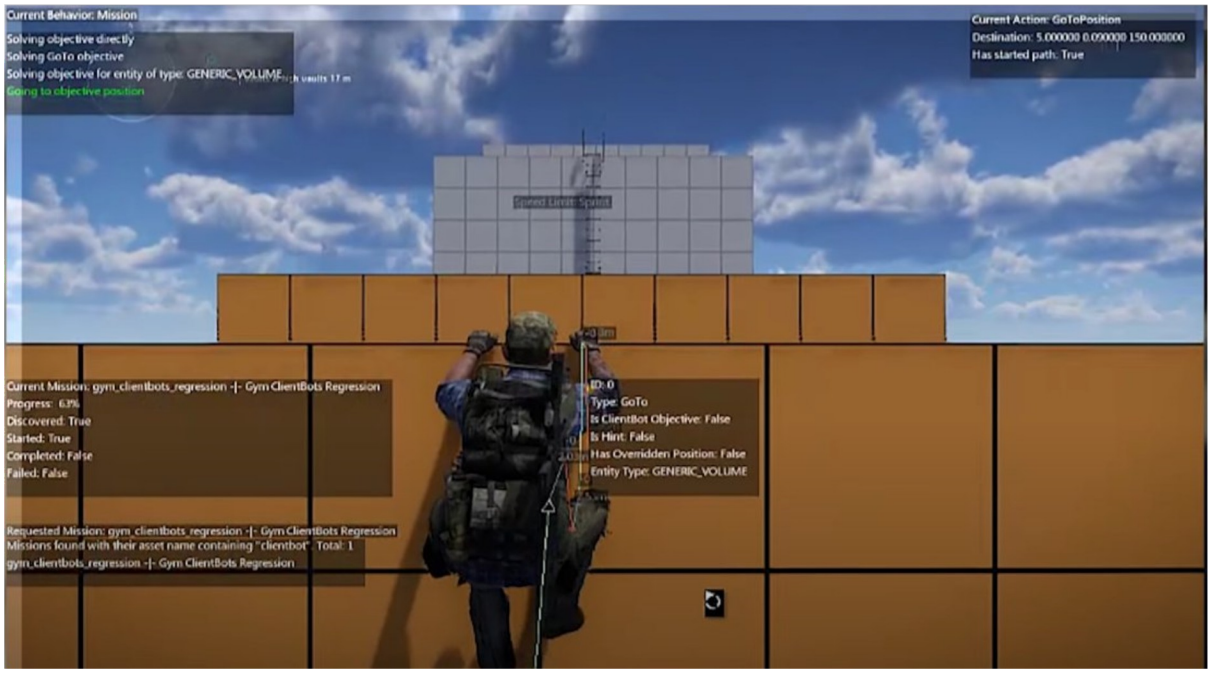

Рис. 1.1. Автоматичне тестування за допомогою клієнтських ботів [1]

Клієнтські боти були написані без використання машинного навчання, а просто алгоритмічно, бо від них не вимагалось виконання ніяких складних дій. Їх суть була у тому, щоб гра не могла відрізнити їх інтерфейс від реального гравця. Але поведінка могла бути зовсім іншою. На рівні була нанесена спеціальна розмітка не видима для людського ока. І за цією розміткою ШІ орієнтувався у просторі і виконував поставлені задачі.

Боти мали безлімітний набір очків здоров’я, та проходили сутички тільки за допомогою автоматичного повороту камери.

Також за допомогою таких ботів відловлювались проблеми зі штучним інтелектом, навігацією та багато іншим.

Перевірку навігації можна побачити на рисунку 1.2.

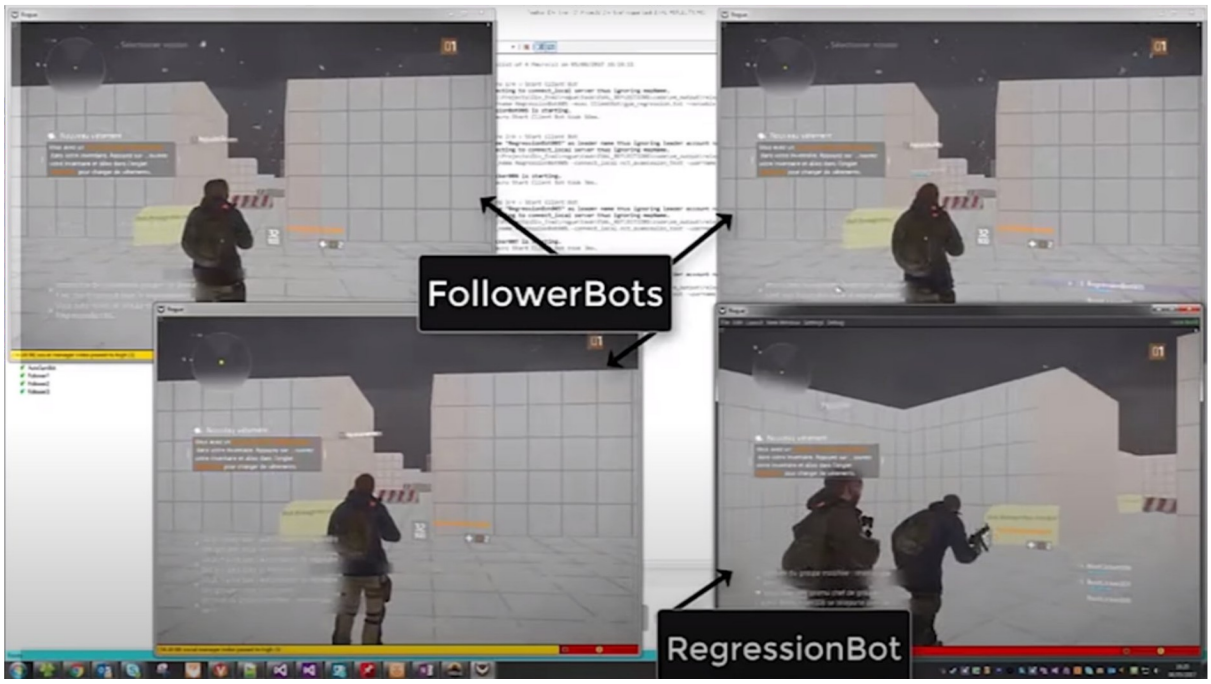

Рис. 1.2. Автоматичне тестування навігації [1]

Тож можна сказати, що даний інструмент виявився дуже корисним для розробки такої великої гри. У підсумку рішення вийшло масштабованим і дуже корисним для процесу розробки та фінального продукту.

Серед мінусів можна виділити те, що цей метод потребує додаткової розмітки на рівні для того, щоб ШІ орієнтувався, що потребує ще додаткового втручання контент-відділу, і підлаштування під це. Також неможливість використання даного методу під час активної фази розробки, коли навігаційна сітка може бути неактуальною, або нестабільною.

1.1.2. Reinforcement Learning

Reinforcement Learning (RL) як головний алгоритм контролю ботів є дуже перспективним напрямом досліджень, оскільки він може добре узагальнювати та автоматично проходити гру, проводячи самонавчання, без зовнішнього втручання, достатньо лише визначити правильну функцію нагороди.

Але у цього підходу є недоліки, оскільки Reinforcement learning довго тренується, особливо для складних оточень. І для виконання складних

завдань система має тренуватись дуже довго. Також при зміні оточення треба буде перетреновувати модель заново, бо вона не може сама пристосуватись до змін оточення.

1.1.2.1. Modl.ai

Modl.ai представляє собою плагін для ігрових рушіїв Unity та Unreal Engine [2]. Система бере на вхід дані про рівень та для роботи потребує інформацію про налаштування контролю персонажем.

Нижче, на рисунку 1.3 можна побачити графік дослідження локації штучним інтелектом-тестувальником.

Рис. 1.3. Карта гарячих точок при навчанні RL моделі [2]

Серед мінусів можна виділити пропрієтарність рішення і неможливість його розширення під потреби розробників.

1.1.2.2. Scalarr

На конференції Games Gathering 2021 було представлено систему RL [3], яка може автоматизувати тестування ігор, та бізнес-застосування машинного навчання, але на жаль зараз цей напрям не активний, і компанія

сфокусована на проти-шахрайських та захисних застосуваннях навчання з підкріпленням.

1.1.3. Переможці General Video Game AI

Розглянемо моделі переможців General Video Game AI [4] та алгоритми, які вони використовували для цього. Змагання з General Video Game AI частіше всього проходять для 2D ігор з невеликим простором рішень. Тож повністю брати за основу їх рішення не варто.

Переможець GECCO 2015: алгоритм Монте-Карло, пошук в ширину і таргетована евристика.

Даний алгоритм добре підходить для покрокових ігор, але не може вирішувати проблеми у реальному часі, оскільки пошук у дереві займає занадто багато часу, що можна побачити на рисунку 1.4.

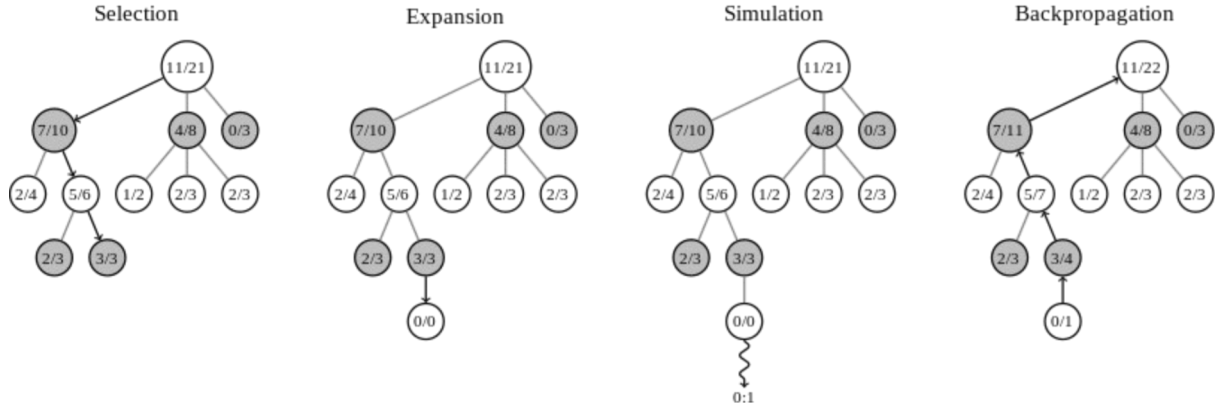

Рис. 1.4. Алгоритм Монте-Карло [4]

Переможець CIG 2015: мульти-евристичний контролер, базований на еволюційному алгоритмі, випадкові проходи та A*.

Цей підхід добре підходить для вирішення проблеми загального ігрового ШІ, оскільки підбирає найбільш підходящу під проблему модель, при цьому моделі проходять еволюційний відбір, що робить їх кращими у виконанні завдання. Але для вирішення проблеми тестування конкретного програмного забезпечення рішення, які спеціально розроблені під рішення цієї задачі будуть вести себе краще, ніж генералізований алгоритм, який

підбирає найкращу модель, оскільки він у більшості випадків буде обирати одну і ту саму модель, яка найкраще підходить під предметну область.

При цьому займає додатковий шар абстракції та займає багато ресурсів, через що показники продуктивності будуть не репрезентативними. Візуалізацію структури алгоритму можна побачити на рисунку1.5.

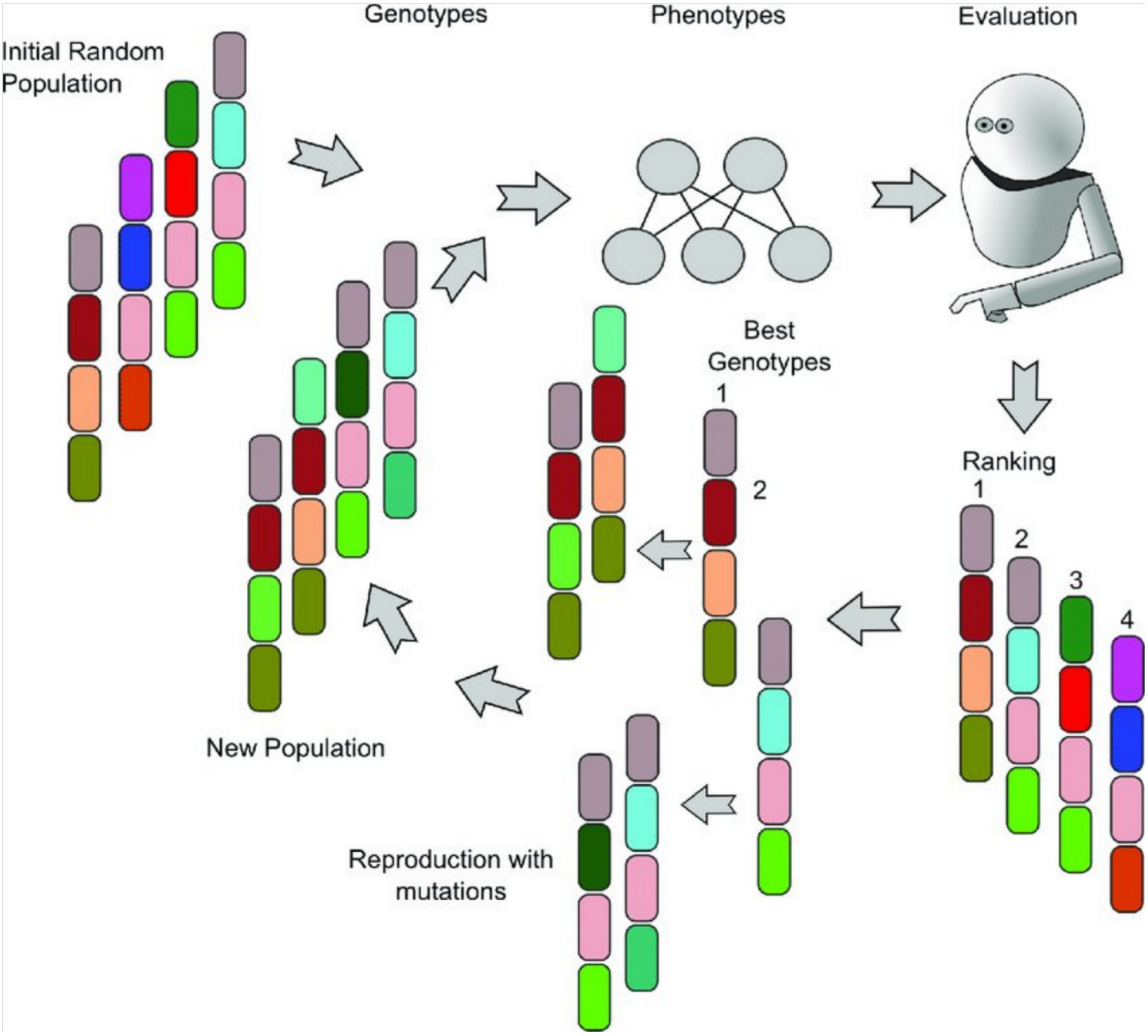

Рис. 1.5. Еволюційний алгоритм [4]

Переможець CEEC 2015: комбіноване реактивне уникнення загроз з ітеративним розширенням у їх алгоритмі пошуку в дереві. Цей алгоритм також займає забагато часу для того, щоб прорахувати оптимальну стратегію і не може бути використаний у застосунках реального часу, що можна побачити на рисунку 1.6.

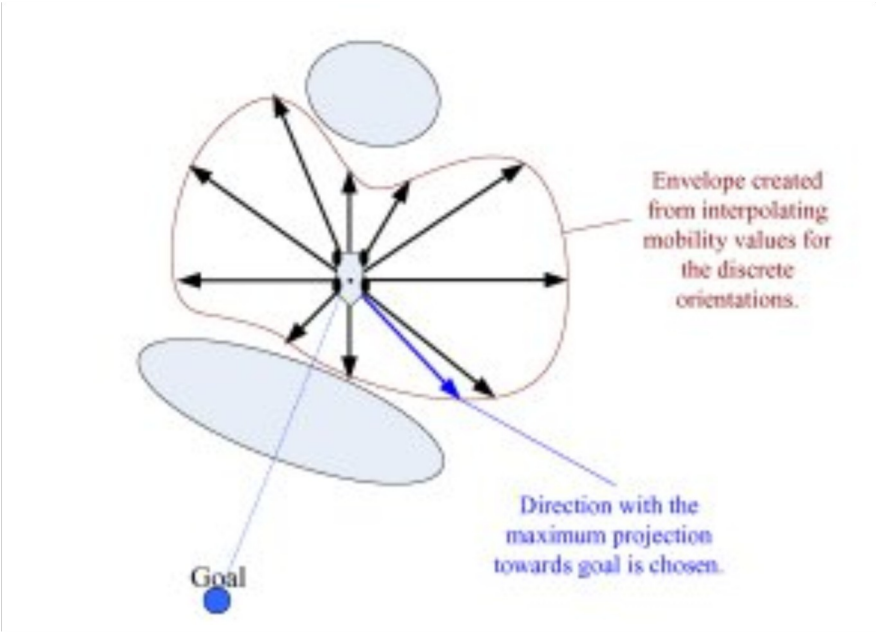

Рис. 1.6. Карта гарячих точок при навчанні RL моделі [4]

Із усіх цих алгоритмів можна винести ідею розділення алгоритмів у системі. Всі ці алгоритми комбіновані і це дозволяє їм вирішувати широкий спектр задач, тобто при розробленні ШІ, який буде симулювати дії гравця, різні задачі має вирішувати спеціалізована під це частина ШІ. Це дозволяє з більшою точністю вирішувати задачі, та легше вносити корективи у алгоритм, якщо змін потребуватиме тільки одна його частина, а не весь ШІ.

1.1.4. Drivatar

У цьому підрозділі розглянемо приклад моделі машинного навчання, який використовуються в ігровій індустрії для суперництва з гравцем, а не заради тестування програмного забезпечення [5].

Drivatar це модель машинного навчання, яка присутня у серії гоночних симуляторів Forza з 2005 року. І з того часу вона лише розвивалась. В іграх Forza Motorsport 1-4 модель машинного навчання була локальною: вона навчалась у гравця, а потім застосовувала знання у гонках проти нього.

Єдиним методом пограти з чужими Drivatar’ами було скопіювати значення з жорсткого диску консолі.

Починаючи з Forza Motorsport 5 моделі переїхали у хмару, і тепер всі можуть пограти з моделлю, яка навчилась на гонках. У останніх частинах серії немає жодної машини, крім гравця, яка не керується алгоритмом машинного навчання.

У серці серії Forza лежить дуже точний фізичний рушій, що оновлюється 360 разів на секунду, там дуже точна фізична симуляція, на яку впливає погода, вологість, характеристики машини, характеристики треку.

Тож ботам треба брати до уваги дуже багато змінних, частину з яких можна побачити на рисунку 1.7.

Рис. 1.7. Змінні, якими оперує Drivatar [5]

Ще одною складністю є те, що у цих іграх дуже багато треків та машин. Коли береться Drivatar людини, то він намагається брати ті карти і машини, де уже їздив гравець.

Але часто буває така ситуація, коли моделі потрібно пристосуватись до нових треків та машин, для яких у них немає даних, і тут дуже допомагає апроксимація і генералізація, щоб симулювати стиль водіння людини, чий це Drivatar. Для цього трек розбивається на багато сегментів: повороти і прямі між ними, і для кожного з них Drivatar вирішує окрему задачу.

Трек очима моделі можна побачити на рисунку нижче на рисунку 1.8.

Рис. 1.8. Трек у вигляді набору поворотів [5]

У перших частинах серії основою моделі слугувала звичайна нейронна мережа, яка навчалась у гравця, але через це поставала проблема одноманітності. Модель навчалась брати один поворот, і кожен наступний раз буде брати цей поворот так само, без змін. Це було б нудно для гри, тож з часом модель поміняли на Байєсівську нейронну мережу.

Із цими змінами вона могла приймати різні рішення в залежності від контексту, тому грати стало цікавіше. Більше різноманітних ситуацій і цікавіші суперники. Сучасний алгоритм уже побудований на основі глибинного навчання, але його деталі не розкриваються розробниками.

І ще однією особливістю сучасного Drivatar’у є те, що він дуже добре модифікується і змінює свою поведінку в залежності від обставин. Гравці можуть робити дії з різною девіацією від задуманої авторами: їхати задом наперед, робити кола на місці, врізатись у купу інших машин. Грати проти ботів, які роблять такі речі – невесело, тож у архітектурі Drivatar є додатковий рівень – Drivatar Adaptation, він модерує дії моделі. Вона не може їхати задом наперед, чи робити кола на місці, вона їздить як нормальний гонщик.

Ще цікавою особливістю є те, що ця модерація залежить від того чи знаєте ви цього Drivatar'а. Якщо це Drivatar людини, яка є у вас в друзях, то модель буде значно менше адаптуватись і модеруватись, і через це її поведінка буде більш схожа на істинну поведінку вашого друга чи подруги, які можуть вас таранити, підрізати, їхати поза гоночним треком.

Загальну архітектуру системи можна побачити на рисунку 1.9, де можна оцінити її загальний вигляд та взаємодію компонентів.

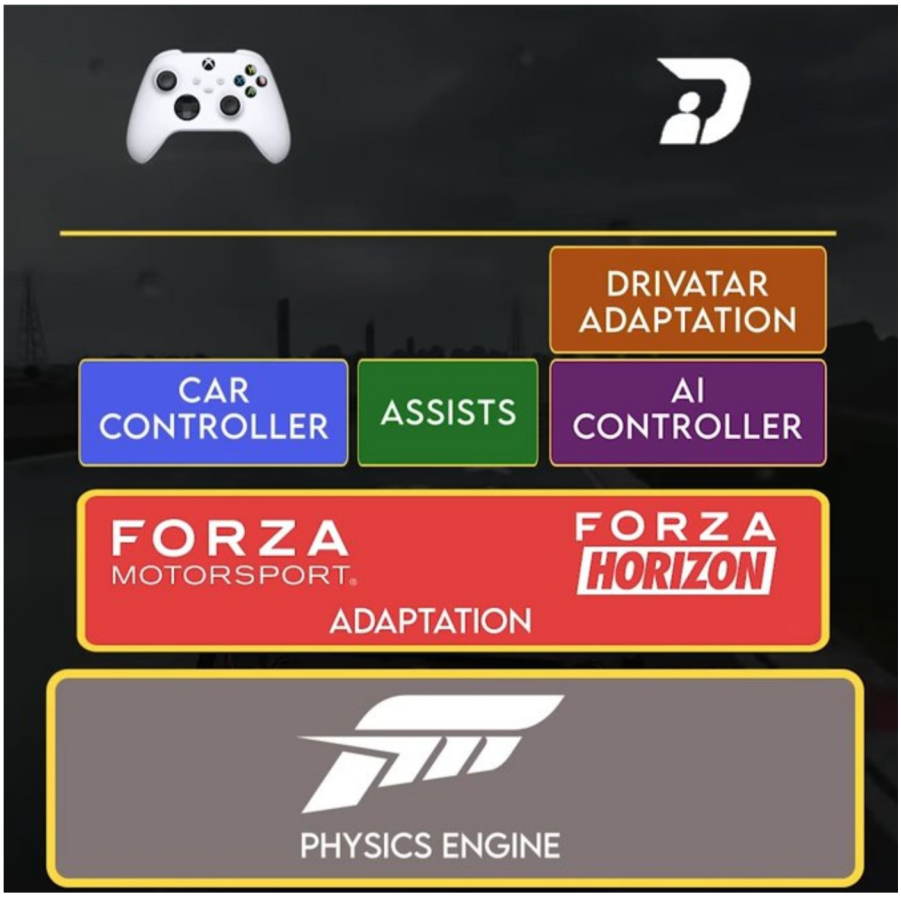

Рис. 1.9. Архітектура управління у Forza [5]

1.1.5. Відтворення вводу користувача у Retro City Rampage

Розглянемо метод автоматизації тестування програмного забезпечення за допомогою відтворення вводу користувача.

На конференції GDC 2015 автор Retro City Rampage розповів до досвід автоматизації тестування при роботі над цим проектом [6].

Зовнішній вигляд Retro City Rampage можна побачити на рисунку 1.10.

Рис. 1.10. Retro City Rampage [6]

Автор описує, що для цього він використовував метод запису і відтворення вводу користувача з контролера. Кожен раз, коли гравець починає гру, то всі його натискання на кнопки записуються, і потім за потреби можуть бути відтворені.

Під час запису даних вводу користувача він використовував спеціальний формат даних, за допомогою якого зберігав усі записані дані.

Він складався з таких частин:

- Заголовок – стандартні дані заголовка і поточний стан. Налаштування керування, наближення камери, одяг гравця, тощо;
- потоки кнопок. Один біт в кадр зі значенням чи натиснути кнопка;
- аналогові потоки. континуальні значення від -1 до 1 на кожен кадр;
- потоки подій. Миша натиснута, відпущена, у процесі перетягування, дотики;
- інформація для відлагодження програми. чексуми – для розробки кожен кадр, для гравців кожну секунду. Якщо відбувається десинхронізація чексуми і даних за цей кадр/секунду, то відбувається помилка.

Для даного формату даних він також застосував компресію, щоб зменшити розмір файлу. Під час відтворення цей файл зчитувався і покадрово

відтворювався ввід гравця, що дозволяло абсолютно точно відтворити його проходження.

Також автор реалізував можливість прискорення відтворення. І міг отримати за декілька хвилин повноцінне проходження гри, яка займала години за декілька хвилин. Це дозволило автоматизувати більшу частину тестування, і зберегти ресурси тестувальників для інших задач, що підвищило швидкість розробки і оптимізувало використання ресурсів тестувальників.

Повне регресійне тестування могло бути виконано за декілька годин повністю в автоматичному режимі, що дозволило пройти одночасно 9 сертифікацій від різних вендорів виробників консолей при портуванні гри на інші платформи.

Автор підкреслює, що для реалізації цього методу автоматизованого тестування потрібно мати повністю детерміністичну систему, щоб відтворення того самого вводу мало такий самий результат. Для отримання детерміністичної системи автор називає такі кроки:

- ініціалізація за замовчуванням даних для всіх елементів системи, оскільки в залежності від архітектури алокація і наповнення даних відбувається по-різному, якщо не вказати цього явно;
- детерміністичний генератор випадкових чисел;
- відсутність функцій зворотного виклику, і перевірка умови кожен кадр замість цього;
- уникання розділення задачі на декілька кадрів;
- детермінізм чисел з плаваючою точкою. У різних архітектурах операції над числами з плаваючою комою можуть бути реалізовані по різному, і для збереження детермінізму треба це явно контролювати.

Також додам, що детермінізм фізично не може бути досягнутий без використання фіксованої частоти кадрів, або повного контролю над частотою оновлення кадру під час відтворення, оскільки дані вводу не завжди можна агрегувати в один кадр без втрати точності.

Деякі з цих правил є дуже вимогливими, та протирічать частим архітектурним рішенням при побудові програмного забезпечення та технікам оптимізації, які для них використовуються.

За відсутності детермінізму система з плином часу буде все більше відхилятися від записаного стану і через деякий час відтворення запису перестане бути достовірною репрезентацією стану на момент запису.

1.2. Напрацювання у області автоматизації тестування за допомогою штучного інтелекту

У цьому підрозділі будуть розглянуті стандартизовані інструменти та алгоритми, які найбільше всього підходять під вирішення задачі контролю у змінних і динамічних середовищах, які мають відкритий вихідний код.

Алгоритми розглянуті до цього відкритого коду не мають, і мають бути реалізовані з нуля для їх використання.

За основу змінного і динамічного середовища візьмемо рушій Unreal Engine та множину програмного забезпечення, яка розроблена за його допомогою.

1.2.1. ML Adapter

Цей плагін підтримує player input контролери. За допомогою цього плагіна можна під’єднати python модель до рушія Unreal Engine[7].

Модель отримує інформацію про рівень як вхідні дані, і використовує його для навчання, прийняття рішень і симулювання вводу гравця.

Модель ніяк не обмежена. Можна навіть використовувати символічний ШІ, а не машинне навчання для цієї задачі.

Архітектурно це працює так, що python скрипт робить запит у рушій, щоб отримати інформацію і зробити якусь дію. Став доступним з версії 5.1 рушія.

1.2.2. Learning agents

Цей плагін [8] новіший за ML Adapter. З’явився з версії рушія 5.3.

Підтримує 2 варіанти моделей за замовчуванням: Reinforcement Learning та Behavior Cloning. Для нашої задачі найбільш підходящим є Behavior Cloning, оскільки найбільш підходящим вирішенням проблеми є клонування поведінки тестувальників. Також у ньому відрізняється архітектура агентів. Якщо у ML Agents агент робив запит у рушій, то у Learning Agents рушій робить запит до агента. Також значною перевагою цього методу є те, що він за замовчуванням підтримує збереження стану моделі у бінарному файлі, тобто можна натренувати, зберегти і надати у готовій версії кінцевому користувачу.

На рисунку 1.11 можна побачити результати роботи моделі.

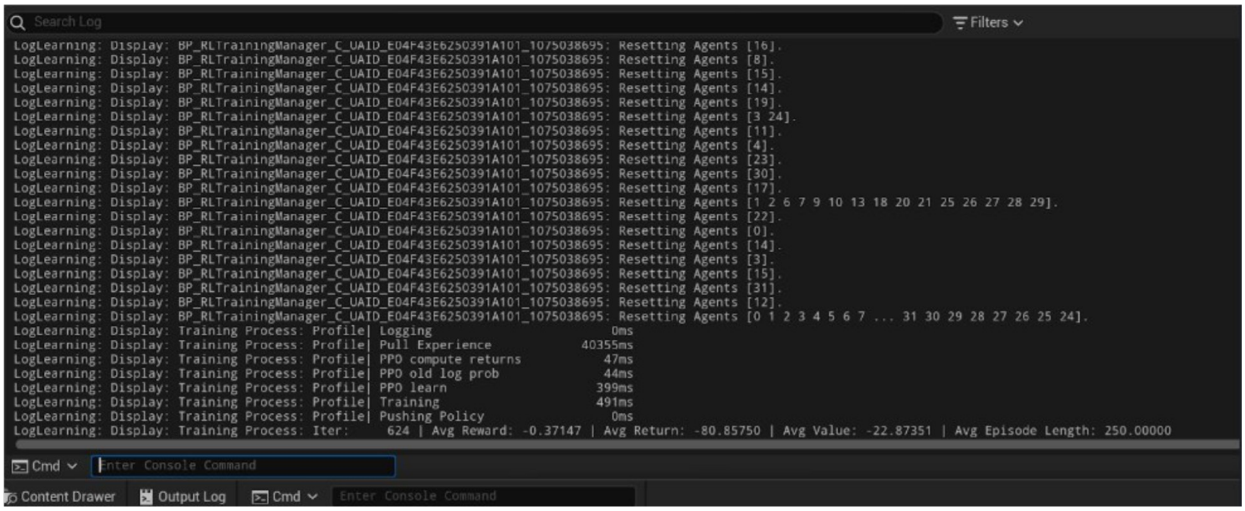

Рис. 1.11. Результат роботи Learning Agents [8]

1.2.3. Behavior transformers

Це новий метод для вирішення задачі Behavior Cloning, дочірньої задачі Imitation Learning. Суть полягає у тому, що модель навчається спостерігаючи за діями людини і потім клонує її поведінку для вирішення задач контролю.

На рисунку 1.12 можна побачити процес роботи з цією моделлю [9].

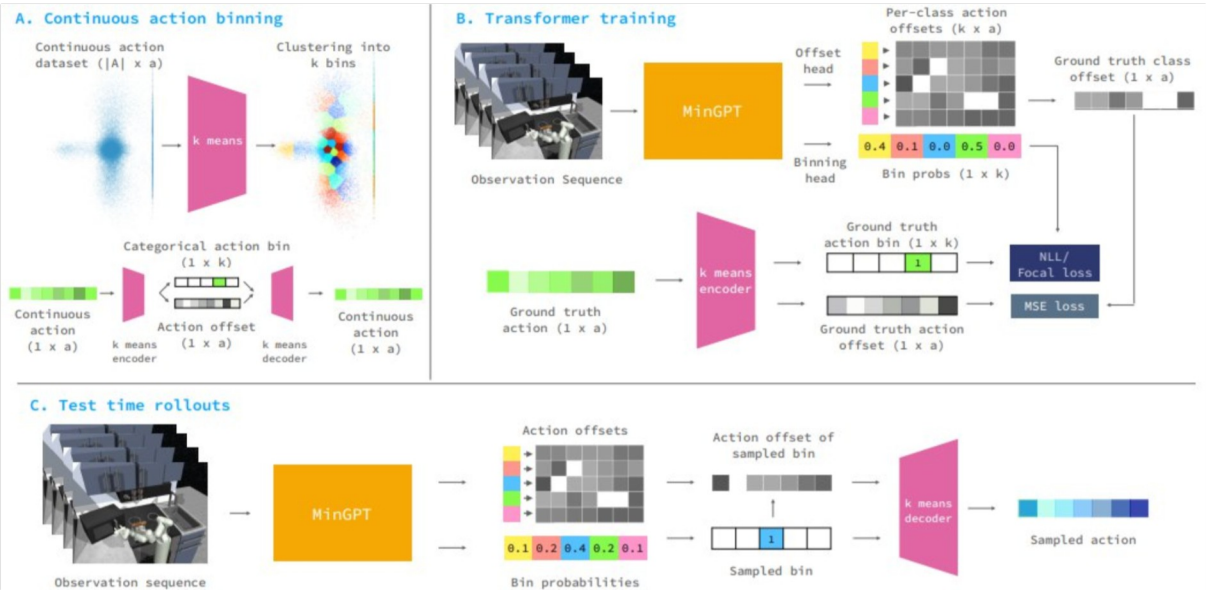

Рис. 1.12. Процес роботи з моделлю Behavior Transformer [9]

У її основі лежить архітектура трансформера, де ключовим поняттям слугує поняття уваги, на відміну від класичних нейромереж. Були представлені експерименти з цією моделлю з порівнянням її з іншими методами вирішення такої задачі: багатошаровий перцептрон з середньоквадратичною помилкою, метод найближчого сусіда, локально зважена регресія, варіаційні автоенкодери, нормалізуючий потік, неявне клонування поведінки.

Візуалізація експериментів у віртуальному середовищі представлена на рисунку 1.13. Були проведені експерименти: водіння, штовхання блоку, та різні симуляції пересування предметів на кухні.

На рисунку 1.14 представлені результати порівняння моделі з аналогами для вирішення. У порівнянні з аналогами ця модель показує чудові результати і у переважній більшості випадків набагато перевершує інші моделі.

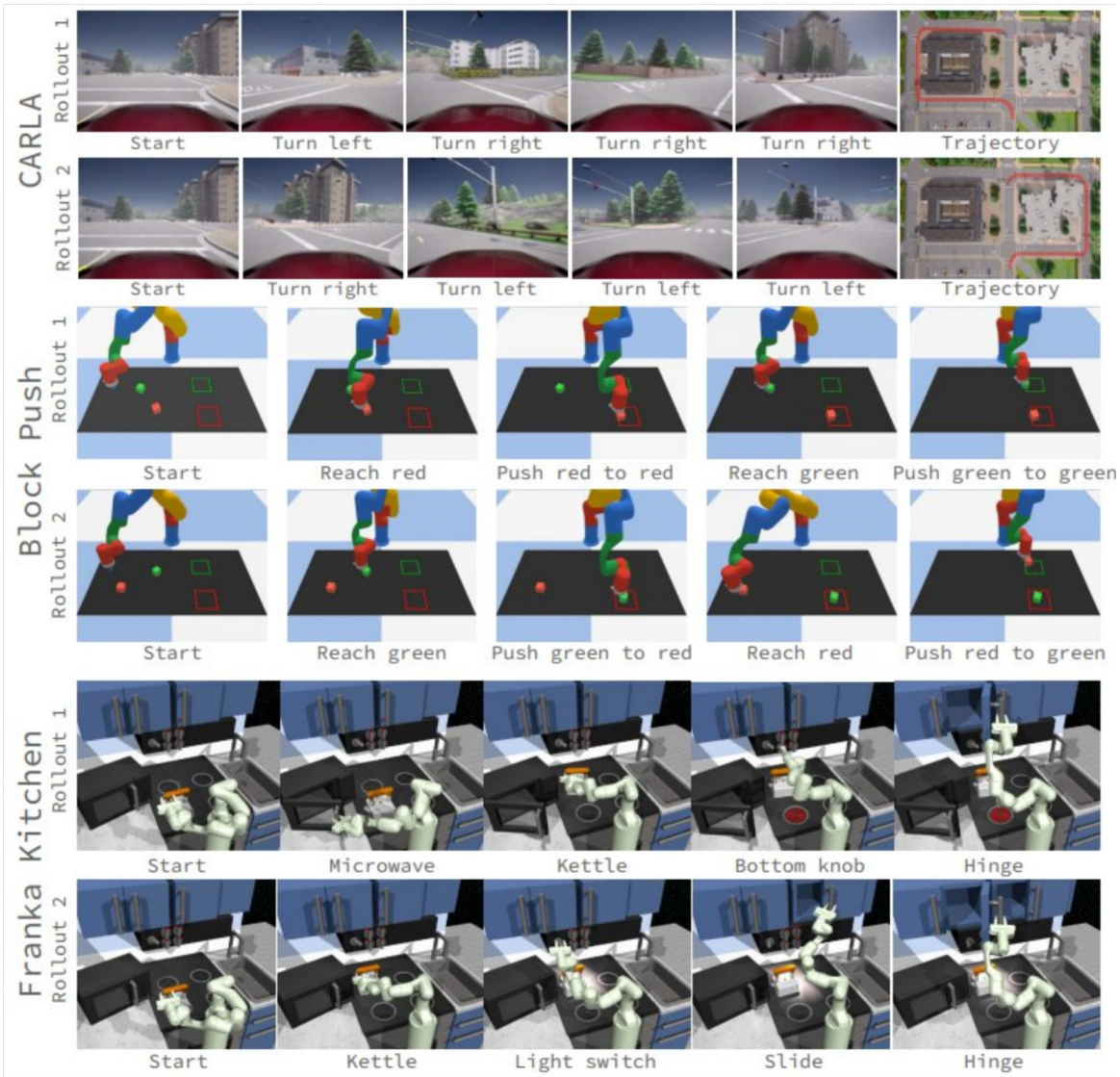

Рис. 1.13. Візуалізація експериментів [9]

	CARLA	Block push				Kitchen				
	Driving	Reach		Push		# Tasks completed				
Baselines	Success	R1	R2	P1	P2	1	2	3	4	5
RBC	0.98	0.67	0	0	0	0	0	0	0	0
1-NN	0.99	0.49	0.05	0.01	0	0.90	0.72	0.44	0.17	0
LWR	**1**	0.50	0.06	0	0	**1**	0.83	0.52	0.21	0
VAE	0	0.60	0.05	0	0	**1**	0	0	0	0
Flow	0.03	0.59	0.02	0	0	0.04	0	0	0	0
IBC	0.25	0.98	0.04	0.01	0	0.99	0.87	0.61	0.24	0
BeT (Ours)	0.98	**1**	**0.99**	**0.96**	**0.71**	0.99	**0.93**	**0.71**	**0.44**	**0.02**

Рис. 1.14. Результати проведення експериментів [9]

Також ця модель дозволяє робити мульти-модальні дії і не обмежена моделлю Марковських процесів, що є великим плюсом у задачі клонування поведінки людини, оскільки люди не ведуть себе згідно Марковським

процесам і одному і тому ж випадку можуть прийняти різні рішення, що дозволяє наблизитись до того, щоб краще симулювати поведінку людини.

На рисунку 1.15 нижче можна побачити результати роботи моделі у різних мультимодальних середовищах. Майже в усіх випадках модель перевершує всі інші з великим відривом.

	CARLA		Block: first block reached		Push: red block target		Push: green block target		Kitchen
Baselines	Left	Right	Red	Green	Red	Green	Red	Green	Task entropy
RBC	0	0.98	0.41	0.25	0	0	0	0	0
1-NN	0	0.99	0.24	0.25	0	0	0	0.01	2.12
LWR	0	1	0.26	0.26	0.01	0	0.01	0.01	2.29
VAE	0	0	0.27	0.33	0	0	0	0	0.72
Flow	0	0	0.31	0.29	0	0	0	0	0.08
IBC	0.12	0.13	**0.48**	**0.50**	0	0	0.01	0.01	2.41
BeT (Ours)	**0.34**	**0.64**	0.54	0.46	**0.43**	**0.44**	**0.41**	**0.40**	**2.47**
Demonstrations	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	2.96

Рис. 1.15. Результати проведення експериментів [9]

1.3. Аналіз експериментів з методами машинного навчання для задач тестування

Алгоритми машинного навчання (Machine Learning, ML, МН) [11] зараз використовуються в широкому спектрі областей програмування, таких як прогнозування і оцінка, генерація та синтез, трансформація та багато інших застосувань. Оскільки ці програмні системи на основі МН стають все більш популярними, помилки у цих програмних системах можуть викликати значні проблеми. Тестування програмного забезпечення – це широко використовуваний механізм для виявлення помилок у програмних системах. Тому, для запобігання критичних втрат через несправні програми, програмні застосунки, що базуються на МН, також повинні проходити перевірку за допомогою тестування програмного забезпечення.

Загалом, у тестуванні програмного забезпечення є дві основні проблеми: 1) пошук ефективних тестових вхідних даних, які дозволять виявляти помилки, та 2) надання способу визначення, чи працює система під

час тестування, як очікувалося (успішний тест) або ні (невдалий тест), що називається "проблемою тестового оракула" [12].

Чи то ми проводимо ручне тестування, чи автоматизоване, ці проблеми повинні бути вирішені, проте зазвичай вони важче вирішувати, коли генерація тестів та їх оцінка повністю автоматизовані, що є контекстом даної дисертації.

Ці проблеми потрібно вирішувати незалежно від типу застосування, технології, і застосованих алгоритмів.

Проте, програмні системи, що базуються на машинному навчанні, особливо ті, що використовують глибокі нейронні мережі, є більш складними і вимагають більше зусиль. Системи, реалізовані на основі глибоких нейронних мереж (DNN), відрізняються від традиційного програмного забезпечення.

У традиційних програмних системах розробники явно реалізовують логіку вимог (у вигляді послідовності програмних блоків: присвоєння, цикли, умови тощо) в методі або функції.

Однак в системах глибокого навчання логіка не кодується явно, а навчається з тренувального набору даних з використанням глибокої нейронної мережі, яка представляється як набір ваг, що подаються на нелінійні функції активації [13].

Загалом, завжди потрібно вирішувати ці складнощі щодо тестування програмного забезпечення незалежно від типу застосування, використаної технології та алгоритмів. Однак, програмні системи на основі машинного навчання, зокрема ті, які використовують глибокі нейронні мережі, є складнішими та більш викликають складності.

Системи програмного забезпечення на основі глибоких нейронних мереж (DNN) відрізняються від традиційного програмного забезпечення. У традиційних системах програмного забезпечення розробники експліцитно реалізовують логіку вимог (як послідовність програмних блоків, призначень, циклів та умов тощо) у вигляді методу або функції. Однак, в системах

глибокого навчання, логіка не реалізована явно, а навчається з навчального набору даних за допомогою глибокої нейронної мережі, яка представляє собою набір вагів, що подаються на неелементарні функції активації [13].

Отже, щодо автоматизованого тестування стратегії генерації вхідних тестів та оракулів можуть відрізнятися від традиційних підходів до автоматизованого тестування. Наприклад, у більшості традиційних систем програмного забезпечення є явний очікуваний результат, який відрізняє тестові випадки з відмінним результатом (відмінний від того, що очікується). Однак, системи програмного забезпечення на основі глибокого навчання відомі як проблеми без оракулу [14]. Тому виявлення неправильної поведінки програмного забезпечення на базі DNN є іншим у порівнянні з традиційним програмним забезпеченням. Наприклад, існуючі критерії відповідності тестів для традиційного програмного забезпечення, такі як покриття операцій, гілок та потоку даних, повністю не ефективні для тестування DNN.

Згідно з недавнім дослідженням тестування програмного забезпечення на основі машинного навчання [15], класифікували завдання тестування та відлагодження систем машинного навчання на шість груп:

1) Генерація тестового оракулу,

2) Оцінка достатності тестів,

3) Генерація тестових вхідних даних,

4) Пріоритізація та зменшення кількості тестів,

5) Аналіз звітів про помилки,

6) Відлагодження та виправлення.

В цій роботі розглядаються три основні категорії тестування:

1) генерація тестових оракулів,

2) оцінка адекватності тестів,

3) генерацію тестових вхідних даних.

Проводиться тестування моделей DNN за допомогою наведених вище методів тестування, організовуючи три експерименти.

По-перше, пропонується метод для створення тестових оракулів для виявлення помилок в програмному забезпеченні на основі машинного навчання.

По-друге, емпірично оцінюються існуючі критерії адекватності тестування для DNN.

Нарешті, пропонується новий метод Guided Mutation (GM) для генерації тестових вхідних даних для оцінки моделей DNN та їх подальшого перетренування за допомогою нових вхідних даних.

В першому експерименті застосовано техніку перевірки оракулу для перевірки моделі глибокого навчання - Варіаційний автоенкодер (VAE) [16], яка є генеративною моделлю для зменшення розміру вхідних даних.

Багатоімплементаційне тестування (MIT) [17] є одним з методів, який вирішує проблему "Відсутність оракула". MIT порівнює різні реалізації однієї і тієї ж моделі, використовуючи той самий тестовий ввід. Отримавши вихід кожної моделі, оракул вважається найбільш повторюваним виходом серед усіх реалізацій. Якщо будь-який вихід моделі відрізняється від тестового оракула (з допустимим порогом), то вважається, що програма містить помилку.

Другий експеримент зосереджений на оцінці адекватності тестів. Серед усіх критеріїв адекватності тестів у традиційному тестуванні програмного забезпечення, критерії "покриття коду" та "тестування мутацій" є двома популярними техніками, які також застосовуються у тестуванні машинного навчання і, зокрема, у тестуванні глибоких нейронних мереж, тому у цьому експерименті фокус був зосереджений на критеріях, пов'язаних з покриттям та мутаціями коду.

У традиційному тестуванні програмного забезпечення, покриття коду визначає відсоток елементів вихідного коду (наприклад, операторів, гілок, умов), які були виконані тестовим набором [18]. Вище покриття є приблизним значенням вищої ймовірності виявлення помилок. На відміну від традиційного програмного забезпечення, програма на основі DNN не реалізує

логіку програми через явні оператори, гілки та умови. Замість цього логіка вивчається через набір нейронів у нейронних мережах [19], тому останні роботи з тестування DNN вводять багато нових критеріїв покриття на основі "покриття нейронів", щоб оцінити, наскільки добре вхідні дані охопили DNN-модель [20].

В [19] було введено метрику покриття нейронів (Neuron Coverage, NC) як метрику тестування DNN. NC визначається як співвідношення активованих нейронів для заданих тестових вхідних даних до загальної кількості нейронів у DNN-моделі.

Після пропонування NC як нової метрики, багато дослідників проявили зацікавленість у дослідженні NC та ввели ще кращі метрики покриття. Наприклад, в [21] ввели нове поняття для тестування систем машинного навчання, яке називається метрикою "Likelihood Surprise Coverage" (LSC). Вона визначає, наскільки несподіваним є результат випробування в порівнянні з вхідними даними навчання моделі.

Мутаційне тестування є однією з форм тестування білого ящика, яка використовується в традиційному тестуванні програмного забезпечення та передбачає модифікацію програми шляхом внесення невеликих змін [22]. Тестові випадки виявляють та відхиляють кожну модифіковану версію (мутант), спричиняючи різницю у поведінці мутанта від оригіналу. У мутаційному тестуванні потрібна метрика, щоб з'ясувати, наскільки добре тестові набори виявляють дефекти. Тому бал мутацій описується як співвідношення виявлених мутантів до всіх засіяних мутантів.

У машинному навчанні [23] запропонували підхід, який називається DeepMutation. Він полягає у мутації моделей DNN на рівні джерела коду або моделі, щоб внести незначні зміни у границю прийняття рішень в DNN. На основі цієї роботи було визначено метрику покриття мутацій як відношення кількості тестових випадків, в яких результати змінилися на мутанті порівняно з оригінальною програмою, до загальної кількості тестових випадків.

Критерії адекватності тестування, такі як покриття коду або коефіцієнт мутацій, зазвичай є метриками, які люди використовують для оцінки своєї набору тестів або стратегії тестування. Однак у цьому дослідженні метою є оцінка самого критерію адекватності. У традиційному тестуванні програмного забезпечення це було б зроблено, оцінивши здатність метрик виявляти реальні помилки. У тестуванні на основі нейронних мереж, введення помилок та їх виявлення пов'язані з характеристиками моделі та її вимогами. Тому класичні метрики оцінки, такі як точність та повнота, можуть не бути підходящими для оцінки самого критерію адекватності на основі завдання моделі DNN.

При оцінці критеріїв адекватності на основі DNN-моделей можуть бути використані інші підходи для оцінки їх ефективності. Наприклад, можна використовувати метрики, які базуються на порівнянні результатів виконання оригінальної та мутованої моделей на тестових вибірках з реальними помилками. Також можна використовувати метрики, які враховують специфіку завдання моделі, наприклад, якість класифікації, визначення аномалій, передбачення часових рядів тощо. Крім того, можна проводити порівняльні аналізи різних критеріїв адекватності для визначення їх ефективності на конкретній моделі та в конкретному контексті.

В [21] та [25] DeepMutation використовуються для виявлення атак на DNN, тому другий експеримент порівнює ці два критерії щодо їх здатності виявляти атаки на DNN.

У третьому експерименті застосовано три різні методи генерації тестів (включаючи новий метод) до моделей DNN і порівняно їх результативність, якщо згенеровані тестові дані використовуються для повторної навчання моделей.

У експериментах 1 та 2 використовуються зображення як вхідні дані. Але в третьому експерименті використано текстові дані.

Однією з нових застосувань DNN є застосування їх на вихідному коді в межах області текстових даних. Останні роки були відзначені величезним

прогресом у глибокому навчанні для завдань, пов'язаних з вихідним кодом, таких як пошук коду та генерація коментарів.

Методи для вивчення розподілених представлень створюють векторні представлення низького розміру для об'єктів, які називаються вкладеннями [26]. Вкладення коду визначається як векторне представлення фрагментів коду. Навчання вкладень коду дозволяє застосовувати нейромережеві техніки в широкому спектрі задач програмування. У цьому дослідженні оцінюються мотивуючі завдання (пошук коду, підписання коду та передбачення назв методу), експериментуючи з трьома найбільш відомими та новітніми інструментами, такими як Code2Vec [23], Code2Seq [22] та CodeBert [24], в області інженерії програмного забезпечення.

Таблиця 1.1. Техніки тестування, вхідні дані та модель

Категорія тестування	модель	Вхідні дані	Тип вхідних даних
Генерація тестового оракулу	Варіаційний автоенкодер (VAE)	Набір даних MNIST	Зображення
Оцінка адекватності тесту	LeNet	Набір даних MNIST	Зображення
	Conv5	Набір даних MNIST	Зображення
	GoogleLeNet	Набір даних CIFAR10	Зображення
	Conv12	Набір даних CIFAR10	Зображення
Генерація тестових сценаріїв	Code2seq	Фрагменти коду на мові Java	Текст (фрагмент коду)
	Code2vec	Фрагменти коду на мові Java	Текст (фрагмент коду)
	CodeBERT	Фрагменти коду на мові Java	Текст (фрагмент коду)

Опираючись на обговорення, виявляється, що виявлення адверсальних прикладів є однією з основних тем у тестуванні.

Тому для тестування технік вбудовування коду потрібно визначити адверсальні приклади для коду. Хоча існують стратегії генерації текстових адверсальних прикладів, вони не є найкращим варіантом для джерела коду.

Основна ідея полягає у використанні рефакторингу джерела коду як способу зміни коду. Їхня мета полягала в тому, щоб показати, що вони можуть обдурити оригінальну DNN, використовуючи ці оператори рефакторингу. У третьому експерименті я запропонував новий метод генерації адверсальних прикладів для джерела коду та порівняв його з цими підходами щодо їх здатності обдурити моделі DNN. Крім того, перетреновано моделі, використовуючи адверсальні приклади, і показано, наскільки запропоновані підходи є кращими у порівнянні з альтернативними.

Запропонований генератор адверсарних зразків базується на алгоритмі Guided Mutation (GM). Він ітеративно змінює код, використовуючи оператори рефакторингу, щоб максимізувати функцію пристосованості (яка базується на оцінці коефіцієнта мутацій DeepMutation++).

Таблиця 1.1 узагальнює всі три експерименти щодо аспекту тестування, моделі DNN та специфікації набору даних.

1.4. Порівняльний аналіз розглянутих методів

Порівняємо розглянуті в підрозділі 1.2 методи з погляду на зміст експериментів, декларованих у підрозділі 1.3.

Результати порівняння можна побачити у таблиці 1.2.

З розглянутих методів найбільш багатообіцяючими є методи використані у The Division та Retro City Rampage, оскільки за мінімальних інвестицій часу вони можуть принести користі, підвищуючи рівень покриття тестами.

Таблиця 1.2. Порівняння розглянутих аналогів

Критерій	The Division	Reinforcement learning	General Video Game AI	Drivatar	Retro city rampage
Складність реалізації	Низька	Висока	Дуже висока	Дуже висока	Середня
Пристосованіст ь до зміни умов	Висока	Низька	Висока	Низька	Низька
Універсальність	Висока	Висока	Висока	Низька	Висока
Якість роботи в реальному часі	Висока	Висока	Дуже низька	Висока	Висока
Якість роботи у комплексних середовищах	Висока	Висока	Низька	Низька	Висока
Складність підтримки під час розробки	Середня	Середня	Низька	Низька	Низька

Висновок до розділу 1

У першому розділі були розглянуті рішення, які використовуються для вирішення задач тестування, такі як The Division 2 та RL методи. Обидва з них мають свої плюси, але вони обидва пропрієтарні, але при цьому обидва методи дуже ресурсомісткі, або місткі у плані контенту, або у плані тренування моделей.

Також були розглянуті методи, які використовувались для вирішення задач загального ігрового ШІ для покрокових ігор.

Найбільшим недоліком є неможливість адаптації під застосування в реальному часі, оскільки обрахування одного кроку займає великий проміжок часу, але підхід з комбінуванням підходів вартий більш глибокого дослідження.

Було також проілюстровано метод запису і відтворення вводу гравця, і реконструкції ігрової сесії, але цей метод важко реалізувати в поточних

умовах, де для повноцінної роботи цього методу треба реалізувати детерміністичність рушія і гри, що іноді суперечить архітектурним і оптимізаційним технікам.

Отже, жоден з цих методів повноцінно не підійшов, але підхід, який використаний для їх створення, можна використати при проектуванні нового рішення.

Показано використання методів машинного навчання, які присутні в інших іграх не для цілей тестування. З них можна почерпнути багато технік для використання машинного навчання у динамічних віртуальних середовищах, таких як відеоігри. В кінці були розглянуті інструменти для вирішення задач контролю у динамічних віртуальних середовищах.

Жоден з розглянутих алгоритмів не є універсальним підходом для автоматизації тестування, таким чином у цьому дослідженні доцільно розглянути покращення методів для автоматизації тестування відеоігор.

РОЗДІЛ 2. МЕТОД ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ВІДЕОІГОР

2.1. Життєвий цикл розробки відеоігор

Для того, щоб повноцінно досліджувати методи автоматизації тестування потрібно розглянути контекст їх використання та потреби у тестуванні впродовж життєвого циклу продуктів.

Життєвий цикл розробки відеоігор (GDLC) відрізняється від стандартного життєвого циклу розробки програмного забезпечення, оскільки крім інженерної частини там є ще і креативна частина, якою не можна нехтувати. Ні інженерною частиною, ні креативною частиною не можна нехтувати, оскільки успіх залежить від обох частин у рівній мірі.

2.1.1. Якісні критерії оцінки, по яким можна класифікувати готовність

Як теоретичну основу життєвого циклу розробки відеоігор розглянемо роботу [10].

За основу класифікації готовності продукту там була взята модель ігрового прототипа Фуллертона, який можна побачити у таблиці 2.1. [31]. Вона складається з 4х стадій. Кожна зі стадій співвідноситься з різними якісними критеріями готовності відеогри.

Таблиця 2.1. Модель ігрового прототипа [31]

Стадія прототипа	a	b	c	D	e
Foundation				+	
Structure	+			+	
Formal details	+	+	+		
Refinement				+	+

Чотири виділені стадії прототипа:

1. Foundation. Базовий прототип, що репрезентує базові концепції гри. Неповний, або низької якості.
2. Structure. Покращений базовий прототип, що має основну ігрову логіку, механіки та правила.
3. Formal details. Повністю функціонуюча гра.
4. Refinement. Поліровка, майже закінчена гра.

Кожна стадія відноситься до якісних критеріїв.

a. Функціонуючий. Значить, що всі механіки гри добре функціонують. Перевіряється проходженням всіх тест кейсів.

b. Внутрішньо завершений. Значить всі правила, розгалуження і умовивраховані, досвід повноцінний. Перевіряється плейтестами.

c. Збалансований. Значить складність не завелика і не замала. Перевіряється дискусією, або опитувальником.

d. Веселий. Значить у гру цікаво грати. Цікавість дуже суб'єктивна,тож тестується опитуваннями гравців, без об'єктивного методу тестування.

e. Доступний. Значить гра легка для розуміння, навчання механікам інтуїтивне. Перевіряється спостереженням за гравцем і те наскільки швидко він навчається механікам.

Щоб оцінювати якість продукту, і коригувати його рух під час розробки треба неперервно проводити тестування його якісних критеріїв. У контексті тестування, частіше за все єдиний якісний критерій, який піддається автоматизації це функціональність. Але у розділі 2.5 буде розглянута автоматизація тестування інших якісних критеріїв.

2.1.2. GDLC

У статті [10] також представлене порівняння різних класифікацій GDLC, яке можна побачити у таблиці 2.2.

Таблиця 2.2. Порівняння різних моделей GDLC [10]

Лінійний GDLC		Ітеративний GDLC		
Blitz Game Studios Pitching	Арнольд Хендрік	Doopler Interactive	Хізер Чендлер	Узагальнена фаза
Preproduction	Prototype	Design	Pre-production	Design and prototype
	Pre-production			
Main production	Production	Develop/Redevelop	Production	Production
		Evaluate		
Alpha	Beta	Test	Testing	Testing
Beta		Review release		
Master	Live	Release	Post-production	

У [10] пропонується даний підхід до розгляду GDLC, який у собі комбінує ітеративний і лінійний підходи, співставляючи його з стадіями прототипа по Фуллертону (рисунок 2.1).

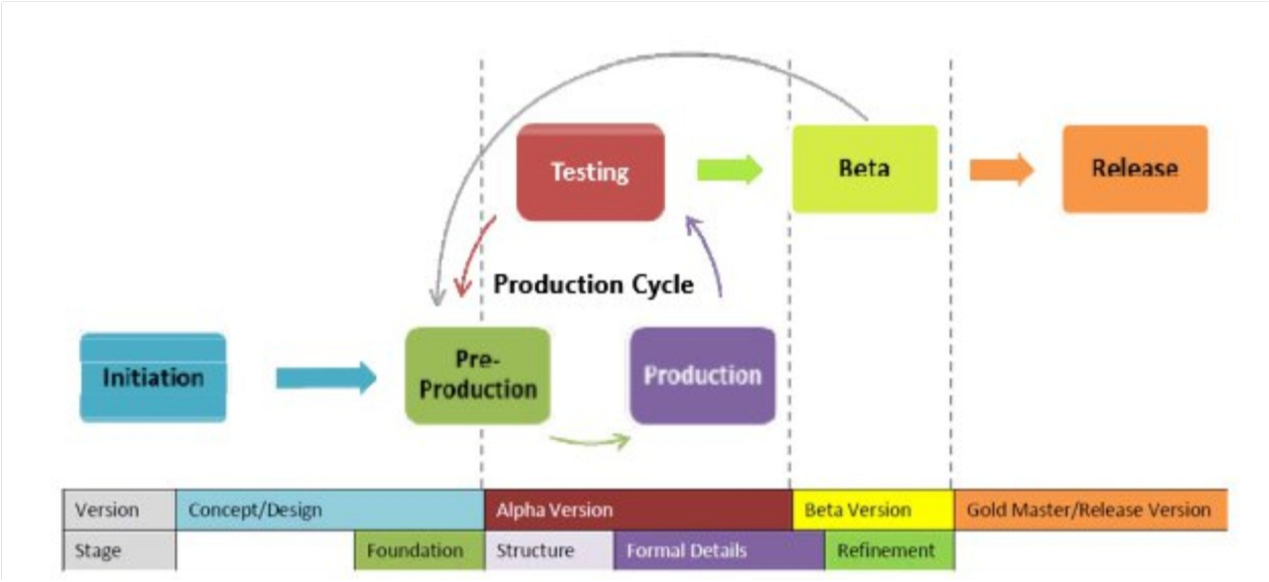

Рис. 2.1. Запропонована схема GDLC [10]

Дана схема поєднує в собі методології Waterfall та Agile. Кожне просування рухає ближче до релізу, і рухає гру у напрямку завершення по

стадіям розвитку прототипа. Але у той же час під час цього руху присутні ітерації, які співпадають з Agile принципами.

Кожна ітерація має в собі планування, розробку, і аналіз. Даний підхід дозволяє інкрементально рухатись до великої цілі, при цьому отримуючи і аналізуючи проміжні результати, фокусуючись на різних якісних критеріях у різні моменти розробки.

Але у даної схеми є вагомий недолік. Він не враховує нерівномірність процесів Pre-production, Production та Testing впродовж альфа версії, тобто перший етап Pre-production буде дуже довгим, щоб визначитись з повним обсягом роботи, який треба буде виконати впродовж подальшої розробки. А наступні цикли Pre-production будуть лише коригуванням та уточненням цілей і обсягу робіт, визначених під час першого циклу.

В той час як з часом тестування стане займати більше часу, оскільки система стала більшою і на її перевірку треба витрачати більше часу, щоб забезпечувати стабільність всіх її систем. Також у цій схемі важко зрозуміти готовність кінцевого продукту і з якого моменту є сенс починати активно тестувати.

2.1.3. GDLC по Кейну

Альтернативний підхід до стадій розробки [32] виділяє такі фази розробки:

- тестові кімнати. Збирання тестової кімнати для однієї механіки для визначення метрик, грубої оцінки механік;
- прототип. Збирання версії гри, яка грається. Щоб розуміти як механіки складаються докупи і як буде відчуватись гра;
- прекрасний куточок. Збір роботи художників для однієї сцени, щоб розуміти як буде виглядати фінальний результат в релізній версії;
- вертикальний зріз. Демонстраційна версія гри короткої довжини. Присутня переважна більшість механік та майже весь контент для даного

відрізку гри. Потрібен, щоб оцінити як буде відчуватись повноцінний фінальний варіант гри. В середньому десь готовий після ½-⅔ фази production.

- горизонтальний зріз. Повністю проходимий варіант гри від початку до кінця. Потрібен для оцінки загальної картини гри і її обсягу. Потрібен для визначення проблем між рівнями, або квестами, які були б непомітні при ізольованому огляді.

- альфа. Всю гру можливо пройти, майже весь контент є на місці, але відсутній баланс, є баги і можливо відсутні невеликі частини контенту.

- бета. Весь контент є на місці. Під час цього етапу іде фінальне налаштування балансу, виправлення багів та оптимізація.

- реліз. Стан гри, який випускається.

- патчі. Виправлення багів, балансу, мінорні додавання нового контенту, які не встигли, або не помітили до випуску.

- DLC. Повноцінне додавання нового контенту, можна розглядати як окремий цикл розробки.

У цьому розрізі можна також розглянути кореляцію до якісних критеріїв розглянутих раніше, у таблиці 2.3. До уваги взяті основні етапи. Інші етапи або занадто короткі, або їх можна розглянути як окремі процеси.

Де якісні критерії розставлені так:

a. Функціонуючий.
b. Внутрішньо завершений.
c. Збалансований.
d. Веселий.
e. Доступний.

Таблиця 2.3. Кореляція якісних критеріїв і основних етапів розробки по Кейну

Стадія розробки	a	b	c	d	e
Тестові кімнати				+	
Прототип	+			+	
Прекрасний куточок				+	
Вертикальний зріз	+	±		+	
Горизонтальний зріз	+	+			
Альфа	±	+	±	+	±
Бета	±		+		+

З цієї моделі можна виділити дві осі під час розробки:

- Вертикальна. Стосується завершеності механік, які формують загальний досвід гравця в поточний момент часу.
- Горизонтальна. Стосується завершеності всіх варіантів використання механік та контенту.

2.2. Піраміда тестування

Тестування ігор здебільшого складається з мануального тестування функціональної частини програми. Автоматизоване тестування складає лише невелику частину.

Це зумовлено тим, що вимоги до цього програмного забезпечення змінюються швидше, ніж в інших сферах розробки програмного забезпечення. Це зумовлено і тим, що це програмне забезпечення дуже тісно взаємодіє з користувачем і його успіх напряму залежить від того наскільки гравцю цікаво у це грати.

Через це проводяться регулярні плейтести, щоб оцінити наскільки гра відповідає очікуванням і що вона викликає саме ті емоції у гравця, які задумано, і в залежності від цього змінюються вимоги до програмного забезпечення. Також дуже висока потреба у оптимізації по швидкодії і використанні пам’яті.

Через це також код змінюється дуже часто і при покритті коду автотестами витрачається багато сил і часу на підтримку актуальності даних автотестів, бо код, який тестується часто змінюється через часту зміну вимог.

Тестування, як і етапи розробки по Кейну також можна розглядати у цих двох осях. Вертикальна вісь складає закінченість механік та окремих індивідуальних елементів. А горизонтальна вісь складає кількість і повноту механік та контенту.

При розгляді автоматизації тестування для цієї моделі розробки дуже зручно використовувати модифіковану систему піраміди тестування. Класична піраміда функціонального тестування зображена на рисунку 2.2 [33].

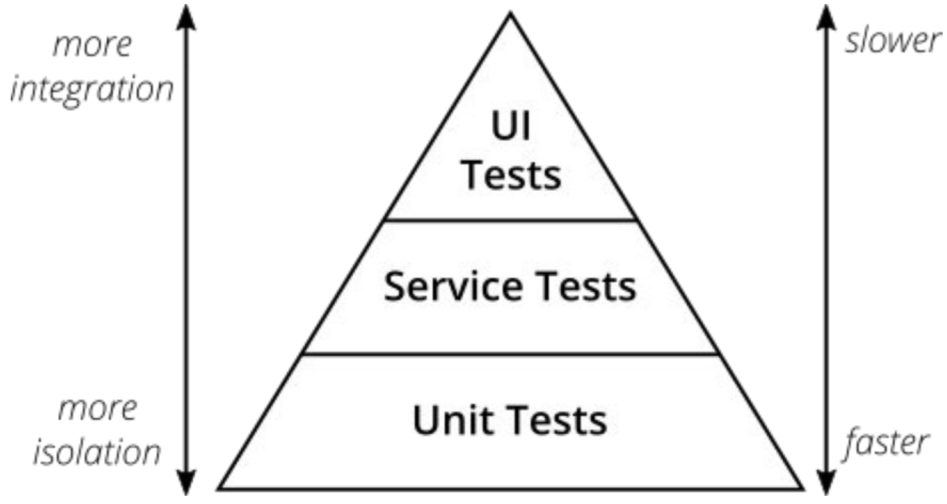

Рис. 2.2. Класична піраміда функціонального тестування [13]

Модифікована структура містить вісь часу, по якій піраміда росте вгору, додаючи тести різних типів. Модифіковану піраміду функціонального тестування можна побачити на рисунку 2.3.

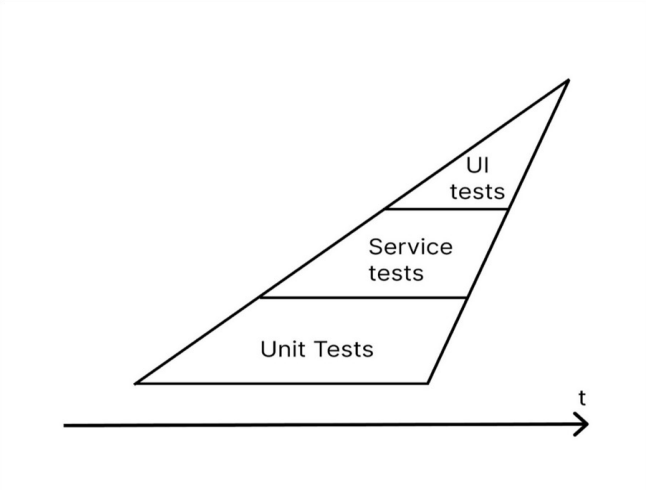

Рис. 2.3. Модифікована піраміда функціонального тестування з урахуванням часу

Недоліком класичної піраміди тестування можна назвати її статичність у часі. Вона показує результат покритого тестами продукту, але не показує порядок досягнення цього покриття.

Піраміда побудована у відповідності до GDLC Кейна, який вказує на те, що під час розробки вертикальний зріз з'являється раніше за горизонтальний зріз, тож вертикально індивідуально механіки стають готовими раніше, ніж всі варіанти використання цих механік.

Отже, можливість створення юніт- тестів з’являється раніше, ніж створення сервіс- і UI- тестів.

Після досягнення вертикального зрізу є можливість покрити більшість нижнього рівня піраміди.

Після досягнення горизонтального зрізу є можливість покрити більшу частину верхнього шару піраміди тестування.

До кінця альфи піраміда тестування має бути завершеною, оскільки весь контент на цей період готовий і не буде розширюватись. Плюс під час етапу бети відбувається більшість процесу оптимізації, що може призвести до значних регресій, тож дуже важливо покрити все тестами до початку оптимізації. Візуальне позначення з розділеннями на етапи розробки можна побачити на рисунку 2.4.

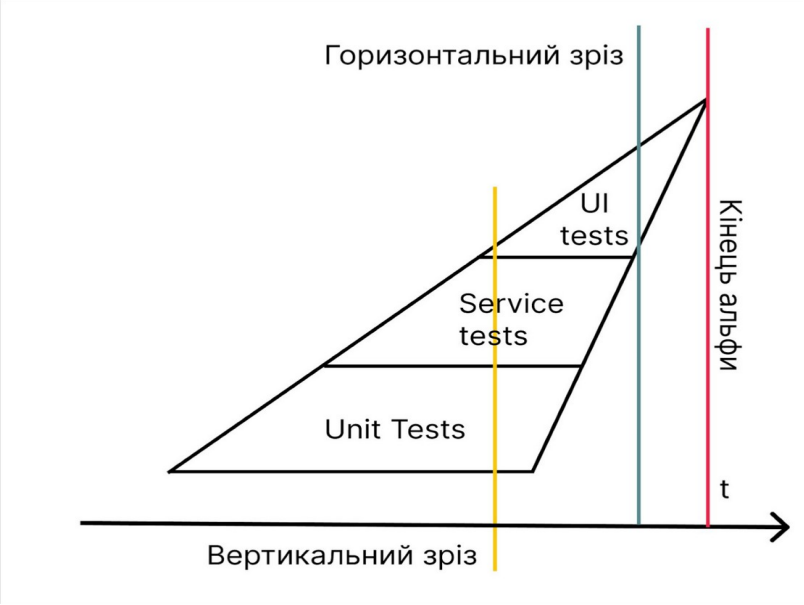

Рис. 2.4. Модифікована піраміда функціонального тестування з урахуванням етапів розробки

2.3. Нижні частини піраміди тестування

Кевін Ділл у своєму виступі на конференції Game Developers Conference [34] розібрав особливості покриття коду відеоігор автотестами. Він вводить альтернативне визначення юніт тестів, ніж представлене у книзі Art of unit testing [35].

Рішення Кевіна Ділла полягає у підвищенні гранулярності юніт тестів. В загальному розумінні такі тести б називались інтеграційними, або ж сервіс тестами. При цьому щоб зберігати визначеність системи на низькому рівні гранулярності він пропонує впроваджувати тести в сам код, який має тестуватись. Тобто робити строгі перевірки(assertions) на стан системи, стан вхідних і вихідних параметрів на низькому рівні в самому коді програми.

2.3.1. Підвищення гранулярності юніт-тестів

Визначення представлене у книзі: “Юніт тест – це автоматизований фрагмент коду, який викликає одиницю роботи в системі, а потім перевіряє єдине припущення про поведінку цієї одиниці роботи.

Хороший юніт тест зобов’язаний:

- бути повністю автоматизованим;
- мати можливість бути запущеним у будь-якому порядку, якщо є частиною набору тестів;
- консистентно повертати один і той самий результат;
- бути читаємим;
- бути легким у підтримці;
- достовірний;
- проходити швидко;
- запускатись у оперативній пам’яті (на приклад, без доступа до файлів, або БД);
- мати повний контроль над усіма частинами, які запущені (використовувати підміну частин системи щоб добитись ізоляції за потреби);
- тестувати одиничний логічний концепт в системі за раз.”

Кевін Ділл у своїй доповіді вказує на необхідність підняття рівня гранулярності тестів. І якщо повертатись до пунктів зазначених вище робить їх корекцію:

- проходити швидко. До якоїсь межі. Не є критичним атрибутом. Оскільки тести можна розділяти на рівні швидкодії. Швидкі запускати локально, а повільніші запускаються під час процесу CI/CD;
- запускатись у оперативній пам’яті. Не обов’язково. Основна аргументація цього пункту була у тому, щоб забезпечити швидкодію проходження тестів. Тож якщо треба буде завантажувати файли, чи БД, чи мережа, то цим можна знехтувати;
- тестувати одиничний логічний концепт в системі за раз і мати повний контроль над усіма частинами, які запущені. Кевін Ділл називає ці пункти явно шкідливим у контексті автоматизації тестування відео ігор. Оскільки вимоги дуже часто змінюються, через це змінюються реалізації. І кожну зміну реалізації треба змінювати код, який тестує цю реалізацію та підмінені частини цієї системи, які використовуються для інших систем. Саме це

спричиняло багато витраченого часу на підтримку юніт тестів і важкість їх впровадження у систему до цього.

2.3.2. Суворі перевірки у коді

За строгі перевірки в коді також виступає Tigerbeetle inc. Вони за 3 роки розробили надсучасну систему управління базами даних, яка оперує фінансовими транзакціями швидше і надійніше за аналоги [36].

Досягти цієї мети їм дозволив їх стиль написання коду в поєднанні з системою тестування [37].

Треба підкреслити один пункт з їх стилю написання коду про строгі перевірки (assertions). Вони перевіряють у кожній функції на вході і на виході чи є правильними дані, і всі перевірки відбуваються на версії для дистрибуції. Тобто, якщо виникає якась помилка у коді, то програма крашиться.

Оскільки для фінансової бази даних правильність розрахунків є критичною, то якщо правильність не була досягнута, тоді це критична помилка. Також це дозволяє окреслити позитивний і негативний простір роботи програми, що дозволяє писати більш надійний і стабільний код за коротший проміжок часу.

Разом з симуляційним тестуванням це допомагає їм тримати і надійність якість на дуже високому рівні. Їх система є повністю детерміністичною, що дозволяє повністю симулювати всю розподілену систему баз даних. Симуляційне тестування може прискорити час у багато разів та допомогти відловити дуже складні і приховані помилки. Ці методи дозволили їм добитись успіху за такий короткий проміжок часу. Демонстрацію симуляційного тестування можна побачити на рисунку 2.5.

Рис. 2.5. Демонстрація симуляційного тестування Tigerbeetle [36]

2.3.3. Інтеграційне тестування

На виступі на Game Developers Conference 2019 Роберт Маселла презентує автоматизацію тестування з використанням заготовлених сценарії в рушії [38]. Часто в ігрові рушії вбудовані інструменти для програмування сценаріїв візуально, або за використання спрощених інтерпретованих мов, які не потребують компіляції проекту, і виконуються в уже зібраному проекті.

Вони часто використовуються розробниками для написання конкретних сценаріїв проходження, так називаємі скрипти. Також скрипти можуть бути використані для автоматизації тестування. Коли скриптом задається поведінка, і якщо вона в результаті співпадає з очікуваною, то тоді тест вважається успішним.

Використовується в ролі інтеграційних тестів, оскільки тестуються лише окремі механіки, частіше за все одна за один сценарій тестування, та займають середню кількість часу. Ці скрипти часто вбудовуються у процеси CI/CD, де вони регулярно проводять валідацію написаного коду чи контенту.

Приклад такого скрипта можна побачити на рисунку 2.6. На ньому представлений інтеграційний тест системи керування.

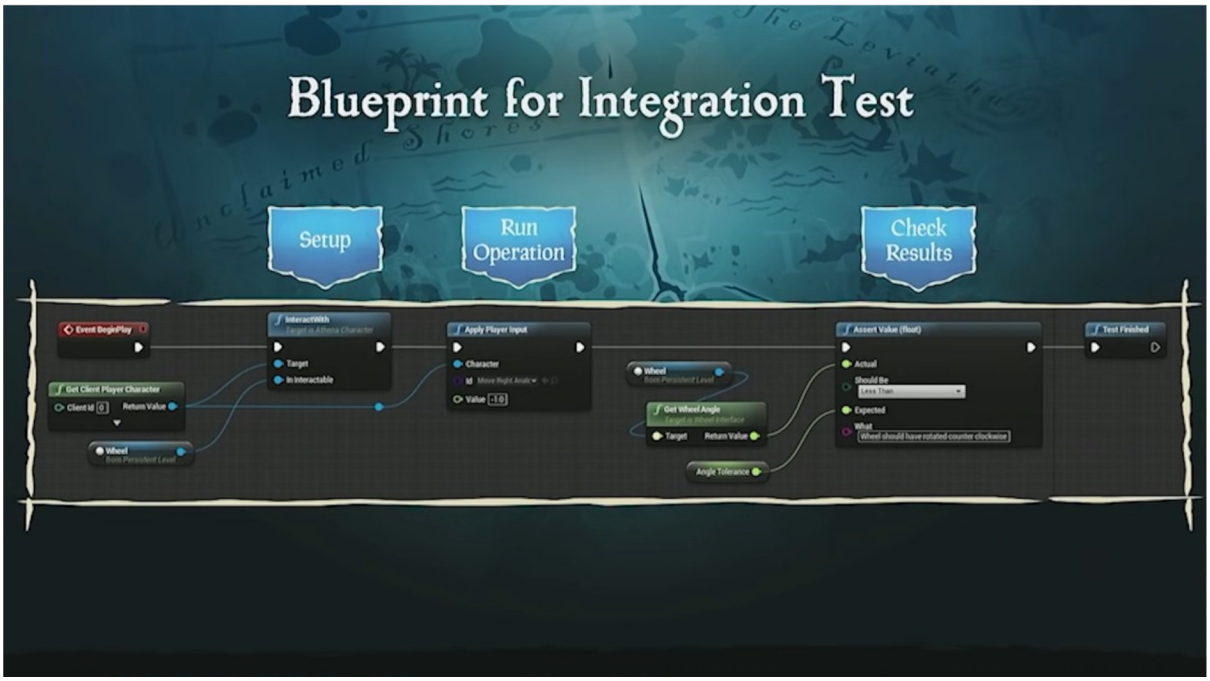

Рис. 2.6. Приклад скрипта для інтеграційного тестування [38]

2.4. End-to-End функціональне тестування

Для автоматизації End-to-End (E2E) тестування, тобто тестів на вершині піраміди тестування виникає потреба у збільшенні гранулярності тестування в порівнянні з інтеграційним тестуванням.

Тобто далі виникає потреба тестувати повноцінні частини гри, без участі людини. Саме у цьому можуть допомогти методи і рішення розглянуті в першому розділі.

Оскільки для повноцінного тестування взаємодії користувача з інтерфейсом і проходження повноцінних частин гри, треба підміняти ввід гравця, щоб усі системи працювали так, наче грає справжній гравець.

2.4.1. End-to-End функціональне тестування за допомогою клієнтських ботів

При порівнянні різних методів автоматизації тестування у першому розділі один з найбільш перспективних методів був запропонований Ubisoft. Клієнтські боти, які використовувались для тестування The Division [1]

показали себе надійним та якісним інструментом. Але у нього присутні 3 недоліки:

- прив’язаність до навігаційної сітки. Під час активної розробки та змін ландшафту навігаційна сітка може дуже швидко застарівати, бути неактуальною. Також не всі місця на карті є можливість покрити навігаційною сіткою. The Division 2 пощастило з тим, що там місцем подій виступає місто і всі поверхні, по яким ходить гравець дуже рівні. На рівних поверхнях дуже гарно будується навігаційна сітка, через що дане рішення там дуже добре підійшло. Але якщо масштабувати дане рішення на більш багаторівневі та криві поверхні, то навігаційна сітка дуже швидко почне обмежувати те, що зможе відтворити бот. Оскільки не на всіх поверхнях навігаційна сітка будується, не завжди є зв’язок між навігаційними сітками на двох різних поверхнях, або ж поверхня дуже нерівна і навігаційна сітка там дуже кострубата і заплутана;

- прив’язаність до того як саме розмічений рівень розробником. Для кожного нового рівня треба буде робити нову розмітку, і для кожної зміни на рівні розробнику потрібно буде міняти також і саму розмітку для тестування. І якщо на рівні присутні часті ітерації, то тоді кожну ітерацію потрібно змінювати дану розмітку

- складність росте лінійно зі складністю тест сценарію. Тобто чим більша щільність подій та взаємодій у сценарії, тим довше це налаштовувати та складнішим стає налаштування тест кейсу. І складність налаштування тест-кейсу росте лінійно зі складністю, але нелінійно з часом. Тобто у деяких випадках, щоб перевірити декілька секунд проходження можна буде витратити багато часу. Через що попередні недоліки будуть ще більш яскраво виражені у складних сценаріях.

В результаті даний метод найбільше всього підходить для автоматизації тестування стабілізованих сегментів гри, де присутня навігаційна сітка протягом всього проходження, та де не дуже велика щільність подій у сценарії.

2.4.2. End-to-End функціональне тестування за допомогою відтворення вводів

Також дуже перспективним методом себе показав метод відтворення вводів гравця, оскільки він дозволяє інтуїтивно і дуже швидко зібрати ввід гравця, і потім його використати для відтворення сценарію.

Але у цього метода є дані недоліки:

- впровадити в існуючі рушії повноцінно є важкою задачею, оскільки для повноцінного симуляційного тестування з прискоренням відтворення потрібно мати повний контроль над довжиною кадра при відтворенні. Фіксація кадрової частоти може частково вирішити цю проблему, але це не гарантує нам страхування від просадок кадрової частоти. Коли програма не є оптимізованою, то кадрова частота може значно просідати в деяких місцях і не бути стабільною.

- Зазвичай стабілізація кадрової частоти відбувається дуже пізно у процесі розробки, в районі стадії бети;

- також прискорене відтворення упирається у ресурси, які потребує гра. Якщо для прорахунку одного кадру гри потрібно 16 мілісекунд складовим комп’ютера, то тоді його не можна ще пришвидшити у 2 рази, оскільки комп’ютер просто не буде встигати прораховувати кадри. Тож пришвидше відтворення можливе у стільки разів, скільки система дозволяє при розрахунку одного кадру;

- розділення реального часу кадру і симульованого часу кадру є дуже нетривіальною задачею для існуючих рішень у рушіях, і потребуватиме переписування ключового функціоналу рушія. Простіше може бути реалізовано у власних рушіях, оскільки там є повний контроль над архітектурою системи.

- без повноцінної інтеграції цього метода виникає проблема з недетермінованістю часу. Оскільки при записі та відтворенні кадрова частота може бути варіативною, то тоді репрезентація вводів гравця буде неточною. Гравець міг натискати кнопку 3 кадри, але буде відтворюватись 2 кадри. Або

частота кадру при відтворенні більша і замість повороту на 0.075, він буде гратись кожен кадр до наступного кадру відтворення. Різниця в 1-2 кадри буде кумулятивною протягом усього часу відтворення, що буде змушувати відходити відтворення все далі від запису.

Тож в результаті, без повної детермінізації потоку програми даний метод можна надійно застосовувати тільки на невеликих сегментах, або не дуже складних сценаріях, де відхилення вводів у часі при відтворенні не буде суттєвим і не буде заважати проводити тестування.

2.4.3. Комбінований метод End-to-End тестування

Якщо застосовувати розглянуті методи тестування разом, то можна досягти кращих результатів, оскільки недоліки використання одного методу можна покрити за допомогою іншого методу.

Більшість випадків має покривати метод клієнтських ботів, оскільки на абсолютній більшості поверхонь є можливість будувати навігаційну сітку і пересуватись по ній. Ця система має під собою детерміністичні алгоритми навігації, які не залежать від частоти кадрів при записі чи відтворенні. Тож вона є більш стійкою до невеликих змін, оскільки навігаційна сітка може дати можливість обійти нову перешкоду, якої не було на попередній версії.

Для виправлення проблеми з неможливістю проходження по місцям без навігаційної сітки пропонується використання запису і відтворення вводів гравця. Оскільки це є лише допоміжним методом, то він має використовуватись лише в короткі проміжки часу, що дає змогу оминути найбільшу проблему цього методу. Оскільки при використанні цього методу під час коротких проміжків часу недетерміністичність часу не має такого сильного впливу. Бо помилка не встигає накопичитись до критичних значень і відхилитись від записаного маршруту.

Також метод запису і відтворення вводів гравця дає змогу складати тест кейси з складністю, яка буде лінійна до часу, і повністю незалежною від

складності цього сегменту. Для складних моментів можна буде переключитись на метод відтворення вводу.

Для збільшення надійності методу відтворення вводів користувача, пропонується альтернативний метод відтворення записаних даних. Основна ідея полягає у тому, щоб відтворювати специфічні вводи гравця, пов’язані з пересуванням базуючись на позиції. Тобто ставити умову не “натискати клавішу руху вперед 4 секунди, під час яких було 240 кадрів”, а “натискати клавішу руху вперед, поки не буде досягнута бажана точка”. Це дозволить збільшити точність для деяких випадків, наприклад пересування на велику дистанцію з монотонним вводом від гравця. Особливості реалізації комбінованого методу End-to-End тестування будуть розглянуті у 3 розділі.

2.5. Автоматизація тестування інших якісних критеріїв.

У доповіді на Games Gathering 2021 представлено використання алгоритмів навчання з підкріпленням (Reinforcement Learning) для автоматизації тестування казуальних та гіпер-казуальних відеоігор [33]. Крім класичних застосувань для функціонального тестування також виділено можливості Reinforcement Learning агентів для тестування інших якісних критеріїв по яким можна оцінити якість гри, таким як збалансованість і доступність. Алгоритми навчання з підкріпленням за своїм дизайном мають 2 властивості, які дозволяють їм оцінювати збалансованість та доступність:

- дослідження. Reinforcement Learning агент має імовірнісну природу. Він завжди має імовірність зробити випадкову дію, щоб дослідити новий варіант дії у даний момент часу. Тобто він завжди буде шукати нові методи покращення своєї політики, не заходячи в локальні оптимуми. Це дозволяє опановувати комплексні середовища, та знаходити нові стратегії, які можуть допомогти отримати винагороду, тобто агент може самостійно прийти до нового рішення, без участі розробника, і якщо з’являється якийсь дисбаланс, то він буде знайдений.

- експлуатація. Ціллю Reinforcement Learning агента є максимізація функції винагороди, без потреби уточнення методів її досягнення. Тобто, якщо будуть знайдені якісь незбалансовані кроки, чи механіки, то агент на них вийде, і буде використовувати для максимізації своєї функції винагороди. Це дозволяє знаходити незбалансовані сценарії.

Схему роботи алгоритму навчання з підкріпленням можна побачити на рисунку 2.7.

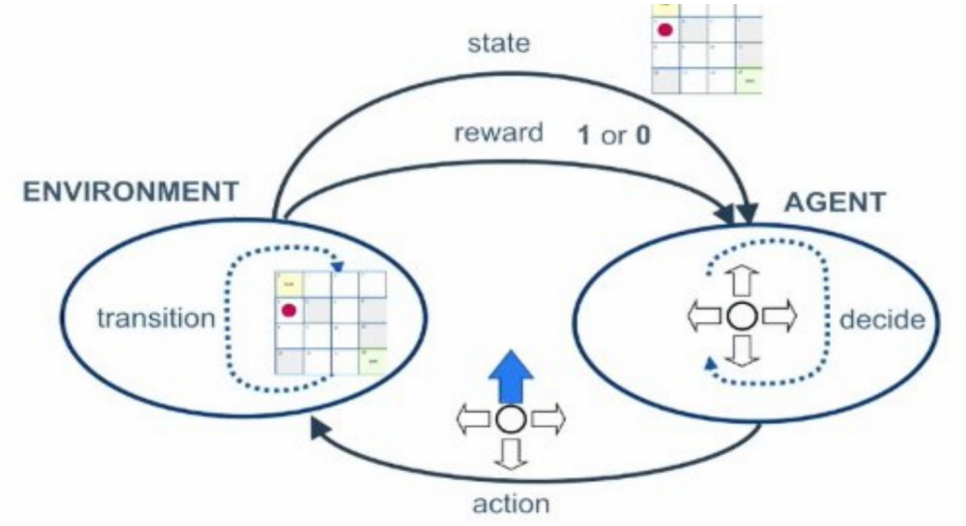

Рис. 2.7. Схема роботи алгоритму навчання з підкріпленням [33]

Оцінку балансу гри можна проводити за допомогою не тільки повністю натренованих Reinforcement Learning агентів, а і за допомогою агентів, які ще не повністю зійшлися, які будуть репрезентувати різні рівні умінь гравців.

Можна заморозити агента на умовних десяти мільйонах ітерацій, і він буде виконувати дії як гравець-новачок, на умовних 30 мільйонах як середній гравець і коли повністю зійшовся, то репрезентувати професіонала.

Для початкового навчання можна використати дані реальних гравців, щоб підігнати під них рівень гри Reinforcement Learning агентів. Приклад такого можна побачити на рисунку 2.7.

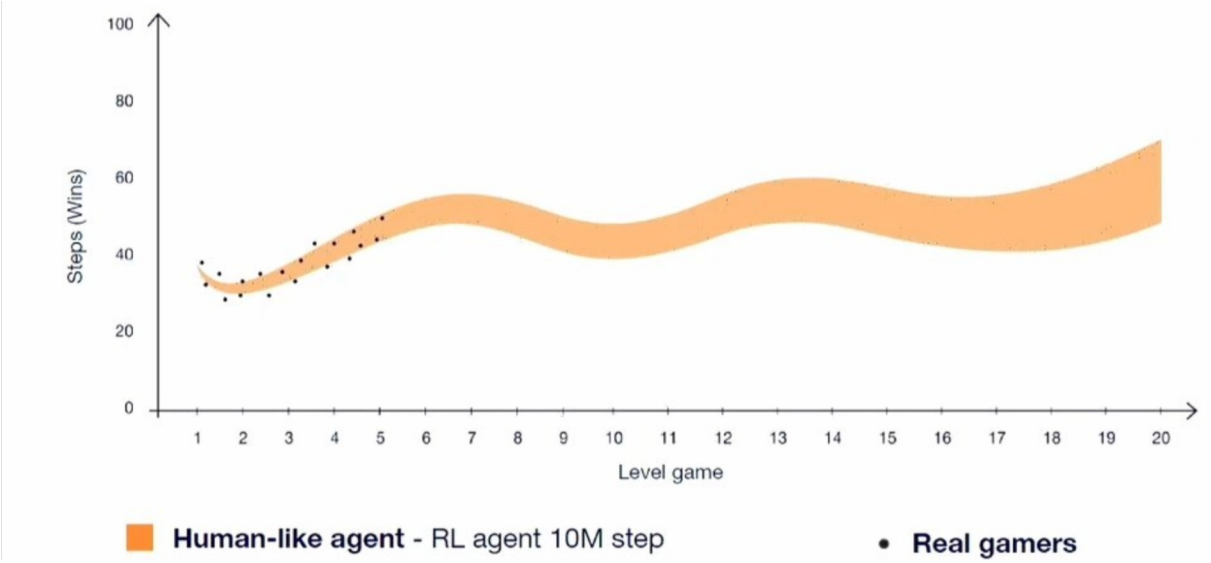

Рис. 2.8. Співвідношення умінь гравців і умінь ботів [33]

Оцінкою доступності може слугувати швидкість сходження моделі навчання з підкріпленням. Зазвичай у приклад доступності і швидкості опанування приводять у приклад Half-life 2, яка була розроблена у 2004 році. У своєму відео-есе “Half-Life 2 Invisible Tutorial” Марк Браун розбирає[19] чому саме гра є такою доступною. Він приходить висновку, що переважна більшість механік спочатку демонструється у мініатюрі, на спрощеному прикладі, що дозволяє користувачу побудувати ментальну модель цієї механіки, і зрозуміти основу її застосування. І потім подальші використання цієї механіки з кожним разом ускладнювались.

Як приклад можна побачити кадр з гри на рисунку 2.9, як саме проходить навчання механіці кидків предметів у ворогів.

Спочатку гравець діставав предмет, який застряг у стіні, і у момент, коли предмет опинявся у нього у руках, одразу випускався ворог, і гравець рефлекторно кидав предмет у нього.

Рис. 2.9. Навчання механіці кидків предметів у ворогів [19]

За схожою схемою працює і навчання у Reinforcement Learning агента. Він спочатку пробує механіку у мініатюрі, отримує зворотній зв’язок, що це правильна дія, отримує винагороду, і швидше сходиться до правильного рішення, коли треба виконувати повноцінні задачі. Тобто чим швидше сходиться агент, тим органічніше було представлене навчання новій механіці, і тим легше її опанує справжній гравець, що власне і робить гру більш доступною.

Також виділено можливість використання Reinforcement Learning для синтезу нових рівнів складності. Для цього Reinforcement Learning агенти навчаються до різних рівнів умінь гравців. Використовуючи гіперпараметри для конструювання рівнів можна генерувати рівні різної складності за допомогою цього метода.

Приклад використання гіперпараметрів для генерації рівнів можна побачити на рисунку 2.10. На ньому представлена генерація рівнів по складності, де складність регулюється гіперпараметрами.

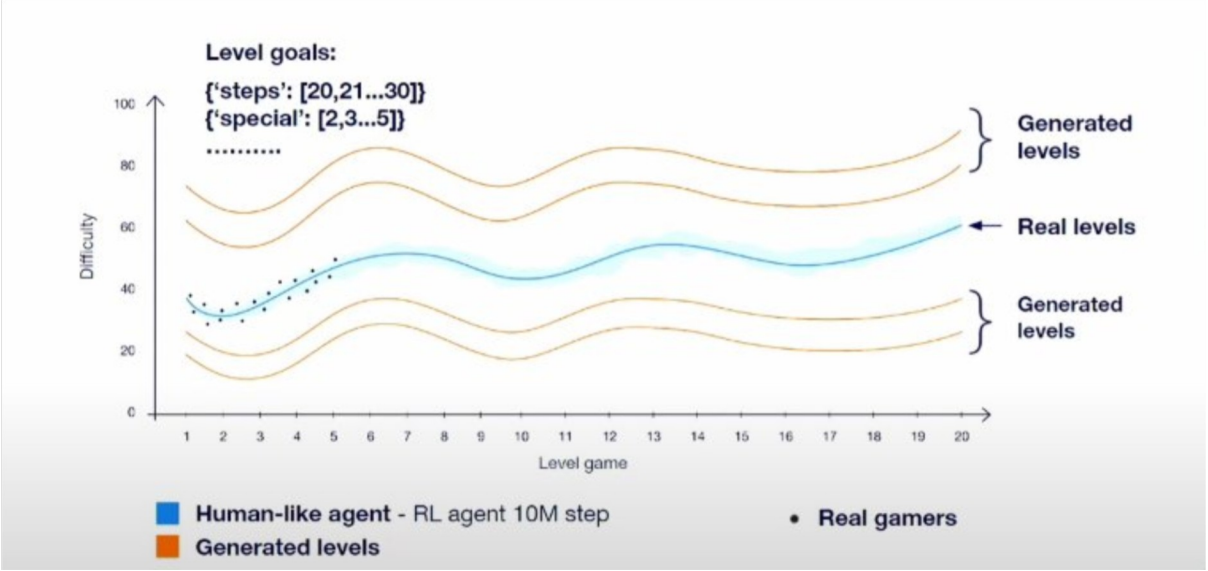

Рис. 2.10. Генерація рівнів за допомогою Reinforcement Learning агентів [33]

2.6. Модифікована піраміда тестування у розрізі автоматизації тестування відеоігор

З розглянутих розділів ми можемо прийти висновку, що класична піраміда тестування не відповідає потребам та стандартам індустрії. І для виправлення цього була складена нова схема, щоб систематизувати підхід до тестування. Вона складається з п'яти шарів, на відміну від звичайної трьохшарової піраміди. У неї є додатковий шар зверху і знизу. Нижній шар відповідає за строгі перевірки у коді на найменшому рівні гранулярності. Найвищий шар відповідає за автоматизацію нефункціональних якісних критеріїв. Саму оновлену піраміду можна побачити на рисунку 2.11.

Як приклад використання цього підходу можна розглянути автоматизацію тестування багатокористувацьких шутерів.

Рівень тверджень (assertion) реалізований у самому коді. Шутери значною мірою покладаються на правильну фізику стрільби, тому перевірка поглиблених фізичних розрахунків має дати правильний результат. Твердження допомагають визначити точну причину помилки.

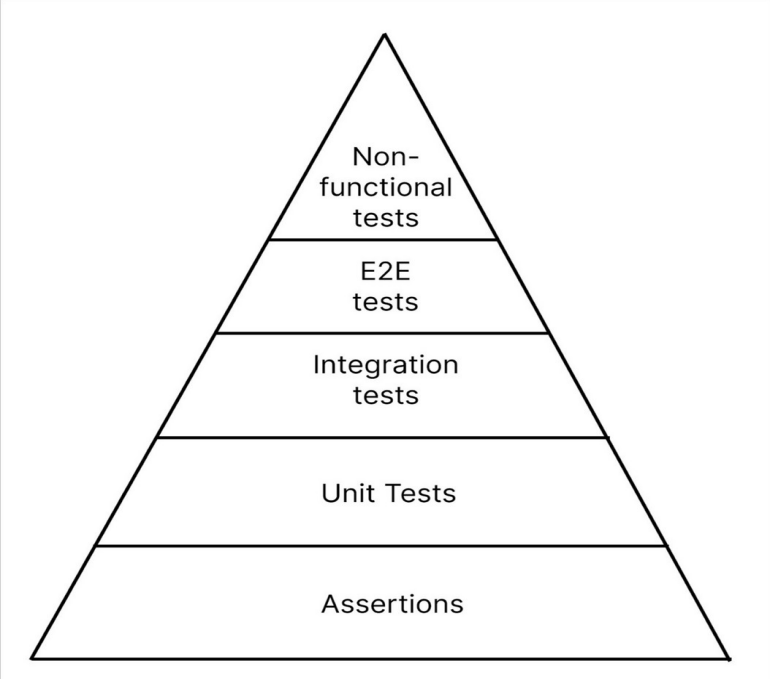

Рис. 2.11. Модифікована піраміда тестування для використання у ігровій індустрії

Рівень юніт тестів охоплює виклики окремих функцій, дещо вищий рівень деталізації, ніж звичайні модульні тести. Тож замість того, щоб налаштовувати модульні тести для кожного виклику фізики, ми повинні охопити більш складні та повні сценарії, наприклад, тестовий сценарій, де розглядається постріл кулі, від початку до кінця, його фізичні розрахунки, зіткнення тощо. З одного боку, це показує нам реалістичний основний сценарій гри, тому він загалом ніколи не повинен змінюватися, тож витрати на його підтримку мають бути мінімальними. З іншого боку, він достатньо детальний, щоб знайти причину проблеми, якщо цей тест провалиться.

Інтеграційні тести охоплюють повні змодельовані внутрішньо ігрові сценарії. Наприклад, бігати, стрибати, стріляти великою кількістю куль у різних умовах. Це знову піднімає рівень гранулярності та висвітлює проблеми, що виникають під час взаємодії систем.

Для E2E тестування найкраще підходять алгоритмічні боти. Запускається повний сервер з ботами, і вони грають повний матч. Це допомагає з тестуванням швидкодії, тестуванням навантаження та тепловими

картами. Також охоплює мережеву частину, де всі боти можуть бути підключені з різними рівнями стабільності мережі.

Для нефункціонального тестування найкраще підійдуть RL-агенти, які знайдуть різноманітні стратегії експлуатації гри та незбалансовані частини гри. Вони могли б охопити едж кейси балансу карт, зброї, стратегій.

Чому б тоді не поєднати нефункціональне тестування та тестування E2E?

Оскільки агенти RL не є детермінованими, їх потрібно регулярно перенавчати, а їхня навчена стратегія може пропустити деякі функціональні проблеми. Алгоритмічні боти показують більш детерміновані результати, і ви можете мати гарантію, що охоплено конкретний тестовий випадок.

Наприклад, на карті є велика подія, де, швидше за все, братимуть участь усі гравці-люди.

Алгоритмічні боти можуть гарантовано взяти у ній участь, але якщо ця подія погано збалансована, і винагорода недостатня, RL-боти ніколи не братимуть у ній участь, як це було задумано.

Алгоритмічних ботів можна розглядати як еквівалент смоук тестів, тоді як нефункціональне тестування на основі RL охоплює едж кейси та допомагає знайти дисбаланс.

Висновок до розділу 2

У даному розділі була розглянута теоретична частина роботи. Були розглянуті особливості процесів розробки, методи тестування на різних шарах та представлена модифікована піраміда тестування. Була систематизована інформація щодо теорії тестування у відеоіграх.

Проілюстровано життєвий цикл розробки відеоігор. Він має свої особливості, які також впливають на тестування. Найбільш вагомими відмінностями є часта зміна умов, ітеративність розробки та мультидисциплінарність продукту.

Була розглянута загальна теорія тестування, та скоригована з урахуванням особливостей даної галузі.

Продемонстровано практичне застосування цього підходу до тестування, де порівнево розібрані різні методи: строгі перевірки, юніт тести, інтеграційні тести, E2E та нефункціональні тести.

Була розглянута модифікація і покращення методів E2E тестування, які були розглянуті у першому розділі роботи. Розглянута модифікація має підвищити стійкість тестів до змін і можливість адаптації до нових умов. Також полегшення їх установки і підтримки.

На основі всього цього був складений модифікований підхід до автоматизації тестування на основі піраміди тестування, який відповідає потребам і більш чітко репрезентує підхід до тестування у індустрії розробки відеоігор.

Також був розглянутий приклад запропонованого підходу до тестування для багатокористувацьких шутерів де був оглянутий кожен шар модифікованої піраміди та його застосування у конкретному випадку.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ МЕТОДІВ E2E ТЕСТУВАННЯ

3.1. Вибір технологій та опис формату даних

Для реалізації алгоритмів End-to-End тестування був обраний ігровий рушій Unreal Engine 5. Оскільки він є досить популярним на ринку та вбудовано має в собі просунутий інструментарій для автоматизації тестування на всіх рівнях піраміди тестування, нижче End-to-End [40]. Також присутній і інший функціонал, який дозволяє полегшити реалізацію алгоритмів End-to-End тестування.

Реалізація написана на мові C++ і складається з таких логічних частин: формат даних читання/запису, запис інформації для проведення тестування, алгоритм відтворення інформації з файлу.

Для реалізації обраний комбінований метод навігаційної сітки та відтворення вводу гравця.

Цей інструмент – framework для E2E у UE5: він керує грою через наявні API (спавнить, рухає Pawn, викликає інтерфейси взаємодії, читає стан світу).

Більш складні дії (симуляція натискання клавіш із GUI, порівняння зображень) легко додаються як плагіни до цього фреймворку.

Архітектура є такою.

UGameInstanceSubsystem UEndToEndTestSubsystem – життєвий цикл сесії тестів. Приймає список TestScenario і виконує їх послідовно.

FTestStep – атомарний крок: команда (MoveTo, Wait, CallFunction, Check), параметри та таймаут.

FTestScenario – набір кроків + метадані (ім'я, опис).

Тестовий раннер виконує кроки асинхронно, переходи по таймеру, робить перевірки.

Результати логуються в UE_LOG і в JSON звіт (файл).

API для віддаленого запуску: консольна команда або HTTP endpoint (можна додати Remote Control) – неблокувальна.

3.1.2. Опис формату даних

Як формат зберігання даних був обраний формат даних csv, оскільки він зберігає дані у читаємому форматі та може бути зручно представлений у вигляді таблиці. Також багато мов мають вбудований інструментарій для роботи з цим форматом даних, що дозволить використовувати даний формат також для інших мов програмування і технологій. Також він може бути використаний для збору і аналізу телеметрії, оскільки цей формат також сумісний з найпопулярнішими бібліотеками для аналізу даних, такими як pandas та numpy. Також даний варіант є простий у реалізації.

Дані містять такі стовпці: Timestamp, Location, Actor Rotation, Camera Location, Inputs.

Опис кожного елемента детально:

- Timestamp — часова позначка у форматі дельта-часу. Означає кількість часу, яка пройшла з минулого оновлення кадру до поточного оновлення кадру. Формат дельта-часу дозволяє продовжувати відтворення запису з будь-якого моменту часу, не прив’язуючись до стартового моменту відтворення. Це дозволяє динамічно переключатись між різними методами виконання тестування;

- Location — позиція актора гравця у даний момент часу. Дозволяє задати стартову точку для тестування, куди актор телепортується для старту сценарію. Також використовується для того, щоб робити проекції на навігаційну сітку і за допомогою навігаційної сітки орієнтуватись у просторі і доходити до бажаної точки;

- Actor Rotation — поворот актора гравця у даний момент часу. Дозволяє зробити точну телепортацію для відтворення стану актора гравця у момент запису. Оскільки переміщення і керування персонажем також залежать від орієнтації актора гравця у просторі;

- Camera Location — позиція камери гравця у світі. У багатьох форматах керування і позиції камери ввід гравця може варіюватись в залежності від позиції камери, тож це теж варто запису;

- Inputs — масив вводів, зроблений гравцем за проміжок часу між попереднім кадром і поточним кадром. Одиниця вводу може бути представлена чотирма варіантами: 3д-вектор, 2д-вектор, число з плаваючою комою, булеве значення.

Особливістю вводів є те, що їх можна поділити на 2 категорії:

- адитивні. Не залежить від того коли вони були програні впродовж кадру, частіше за все аналогові значення з інтервалом -1…1. Наприклад рух камерою. Можна програти 10 рухів камерою в один кадр і рухи будуть програватись так само, як і були записані.

- неадитивні. Мають гратись лише 1 раз доступності вікна для вводу.

Наприклад стрибки та рух на клавіатурі.

Були розглянуті різні варіації оптимізацій місця, яке займає даний формат даних, але для використання комбінованого підходу відтворення оптимізації місця застосовувати не вдалось, оскільки при використанні різних методів потрібна різна інформація, для загального збільшення стійкості методів відтворення і адаптації під нові умови, які можливі при зміні навколишнього середовища.

Результуючий формат даних можна побачити на рисунку 3.1.

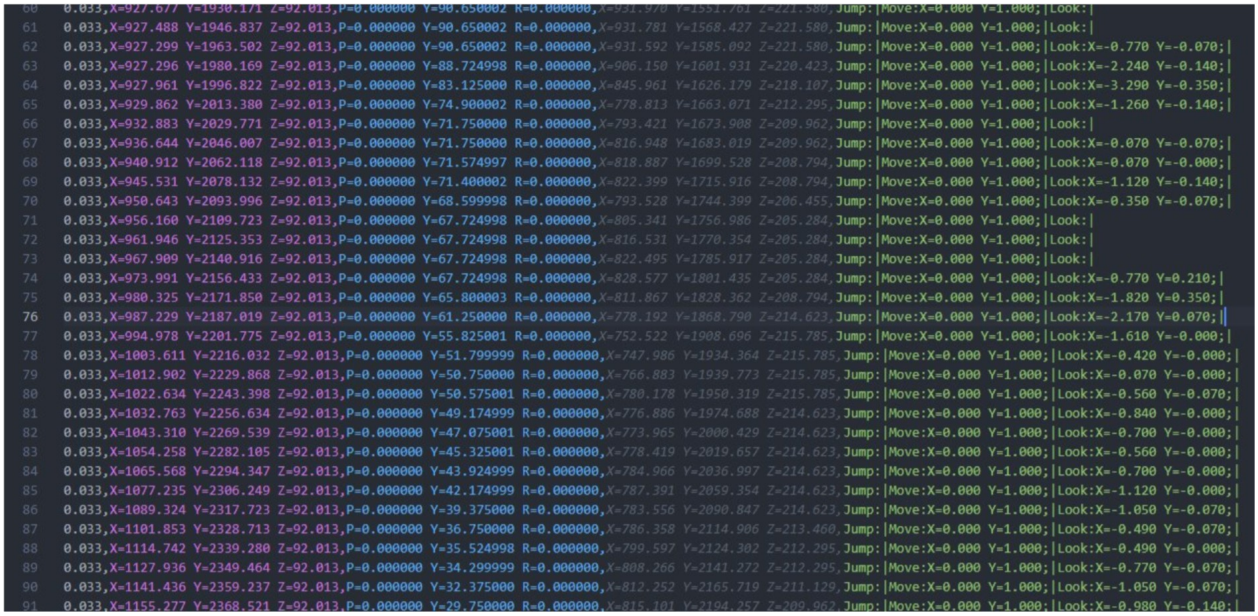

Рис. 3.1. Дані запису гравця

Запис даних проходить за достатньо прямолінійним алгоритмом. Кожен ввід користувача посилає подію, і записується у масив вводів, які виконав користувач. Наприкінці кадру фіксується позиція актору гравця, позиція камери, поворот гравця і записується у масив даних разом з масивом вводів і дельтачасом. Потім на кінці запису ці дані записуються у csv файл, який потім використовується для відтворення сценарія.

3.2. Опис методу відтворення вводів гравця

В основі реалізації цього метода лежить механізм підміни вводів гравця, програма відтворює вводи гравця у такій же послідовності та такому ж часовому проміжку.

На початку відтворення персонаж гравця телепортується у початкову точку маршруту з початковим поворотом та початковим поворотом камери, оскільки все це може вплинути на подальше відтворення.

Особливістю симуляції вводів гравця є те, що цей метод може працювати відносно надійно тільки з зафіксованим показником відмальовки кадрів на секунду. Без цього детермінізму починаються великі розходження між записом і відтворенням, в особливості з неадитивними вводами.

Це стається через особливість оновлення системи вводів. Система вводів частіше за все оновлюється двічі на кадр заради плавності і хорошого відчуття від керування. Тож при реалізації алгоритму відтворення було прийнято до уваги цю особливість і записані вводи відтворюються через рівні проміжки в залежності від кількості їх на поточний кадр, щоб зберігати гнучкість і пристосованість до інших налаштувань системи вводу та вікон обробки вводів.

Всі вводи, які не потрапили у поточне часове вікно обробки вводів можуть перейти на наступне вікно. Через це можуть ставатись розриви у відтворенні вводу, які можуть призвести до втрати інерції та подальшим розходженням з записом.

Зворотною стороною є те, що якщо відтворювати декілька неадитивних подій у одному вікні вводу замість потрібної кількості, то вводи після першого просто втратяться, оскільки перший ввод і так є максимальним значенням, до якого неможливо додати ще додаткові значення. Це призводить до втрати вводів і неточного відтворення.

Фіксована кількість кадрів за секунду дозволяє детерміновано співвіднести вікна вводу при записі та відтворенні. Оскільки при відтворенні вводів нам потрібно знати яку кількість часу займе наступний кадр, який ще не стався, для того, щоб точно відтворити вводи, співвідносячи їх з записом.

Були розглянуті альтернативи до розглянутого методу. Найбільш цікавою альтернативою було запам’ятовувати скільки часу була активна та чи інша дія, але це працює тільки для неаналогових вводів. Наприклад у випадку руху за допомогою аналогового стіку у нас виникає випадок аналогового вводу, який може змінюватись по декілька разів на кадр. І ми не можемо сказати скільки часу був активний рух, бо рух у нас репрезентований континуальним значенням від 1 до 1.

Цей метод може працювати тільки для дискретних вводів, які приймають значення 0 або 1. Бо кожен кадр у нас може бути зміна у положенні вводу і це може ламатись при змінній кадровій частоті.

Оскільки вікно вводу могло бути варіативним і змінний аналоговий ввід міг зчитатись невірну кількість часу, бо оновлення вводу є залежним від кадрової частоти, а не від реального часу. Саме такої схеми дотримувався розробник Retro City Rampage[ссилка] для дискретних кнопок. Для більш комплексних аналогових систем вводу цей варіант не є підходящим, і треба покадрово записувати і відтворювати вводи.

Як мінус методу можна виділити його велику залежність від кадрової частоти та неможливість пристосування до змін у навколишньому середовищі без зміни самого запису. Дуже добре підходить для комплексних сценаріїв, які відбуваються впродовж короткого періоду часу.

3.3. Опис методу відтворення через навігаційну сітку

Для використання методу навігаційної сітки нам потрібно мати на карті побудовану навігаційну сітку та її присутність на сегменті рівня, що тестується. Це робить цей метод більш підходящим для рівнів з рівною сполученою поверхнею.

На рисунку 3.2 ми можемо побачити приклад сполученої поверхні праворуч і не сполученої поверхні ліворуч. Права поверхня має зв’язок з основною навігаційною сіткою рівня, в той час як поверхня ліворуч не має такого зв’язку.

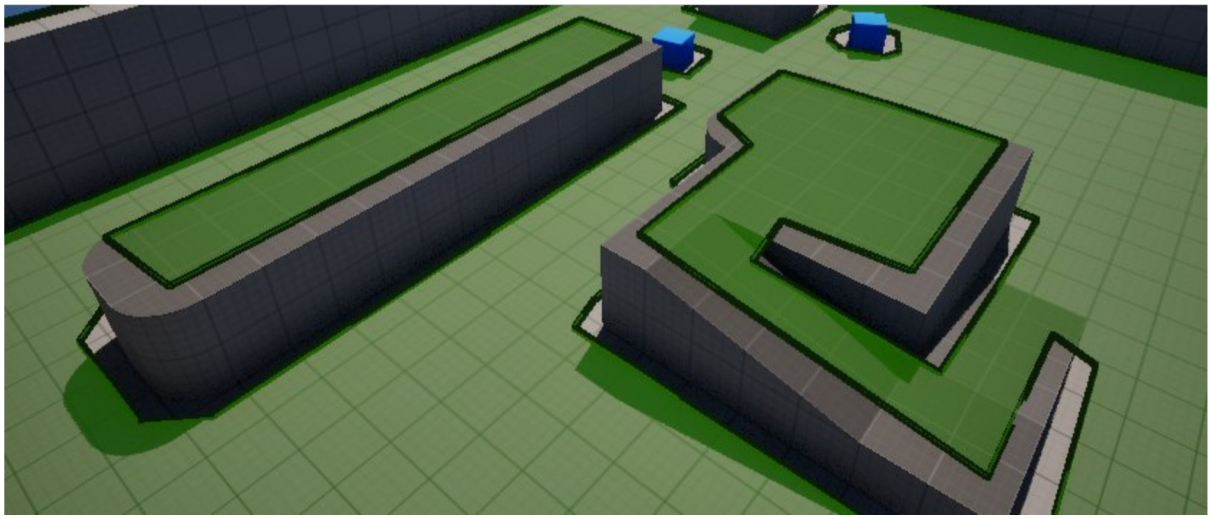

Рис. 3.2. Навігаційна сітка

Такі обмеження можна обійти, використовуючи навігаційні зв’язки, які можна побачити на рисунку 3.3. Для прототипу було прийнято рішення не інтегрувати навігаційні зв’язки, як песимістичний сценарій для реальної розробки через дефіцит ресурсів розробки чи апаратної частини цільової платформи.

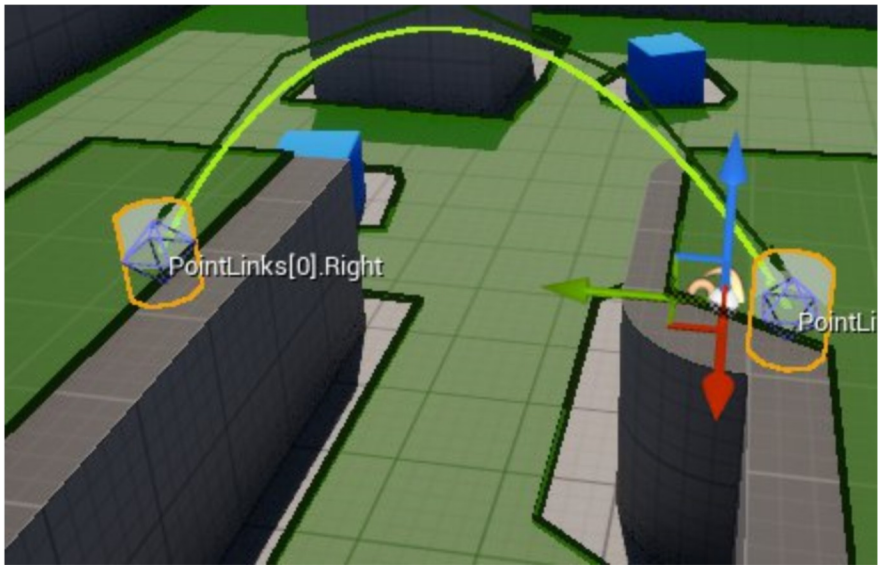

Рис. 3.3. Навігаційний зв’язок

Алгоритм використання методу навігаційної сітки складається з трьох частин:

- знаходження останньої точки на записаному маршруті, яка лежить на доступній навігаційній сітці;

- побудова найкоротшого шляху до цієї точки; • Слідування за цим шляхом.

Знаходження останньої точки на заданому маршруті рахується за допомогою алгоритму групового лінійного пошуку. Лінійний пошук з кінця, але робиться групами, щоб здешевити алгоритм проекції. Бінарний пошук тут не підходить, оскільки у нас немає гарантії, що після сходу з навігаційної сітки не буде повернення назад на неї, тож лінійний пошук з кінця це найкращий варіант для цього випадку. Робиться проекція кожної точки на навігаційну сітку, на якій зараз стоїть агент, щоб зрозуміти чи підходяща точка.

Побудова найкоротшого шляху робиться за допомогою алгоритму A*, який є стандартом для цього варіанту використання. Побудова нового, найкоротшого шляху дозволяє скоротити час тестування і зменшити непотрібні рухи, які використовуються для проходження маршруту. Також

особливістю є те, що треба збільшити радіус агента на навігаційній сітці, щоб випадкові колізії не збивали агента з маршруту, і шлях будувався з запасом.

Після складання маршруту слідування ним є достатньо простою задачею. Спочатку робиться переміщення у початкову точку, поворот камери і персонажа на наступну точку, зупинка усієї інерції персонажа, і симуляція вводу руху вперед, з постійною корекцією напряму руху гравця. Після досягнення наступної точки, ті ж самі кроки, що і для стартової точки. І так до кінця маршруту.

Цей алгоритм дозволяє детерміновано дійти до будь-якої точки на карті, якщо вона є в зоні досяжності навігаційної сітки, що робить його хорошим вибором у загальному випадку за доступності навігаційної сітки.

3.4. Опис методу відтворення через локацію

Практична реалізація цього методу виявилась не надійною та відкинутою за поганих результатів, які він показав. Головною причиною цього виявилось те, що цей метод, який мав покрити аспект переміщення ніяк не враховує контекст повороту камери під час руху. Якщо рухати камерою під час руху вперед, то траекторія може сильно змінитись, навіть якщо вводи переміщення ніяк не змінювались. Або ж локація змінилась та стара точка не є такою, якою вона була на записі. І якщо була пропущена точка зміни вводу, то відновлення стану видавалось дуже проблематичним. Через описані проблеми цього методу, було прийнято рішення не проводити подальших експериментів.

3.5. Загальний опис проведення експериментів

Для проведення експериментів було обрано найбільш усереднений варіант переміщення у тривимірному просторі з використанням камери від третього обличчя(за спиною). Даний формат візуалізації часто береться за основу у тривимірних відеоіграх зі складною системою навігації, тож можна

взяти його за базу, на якій можна буде оцінити використання запропонованих методів.

Візуалізацію цього можна побачити на рисунку 3.4.

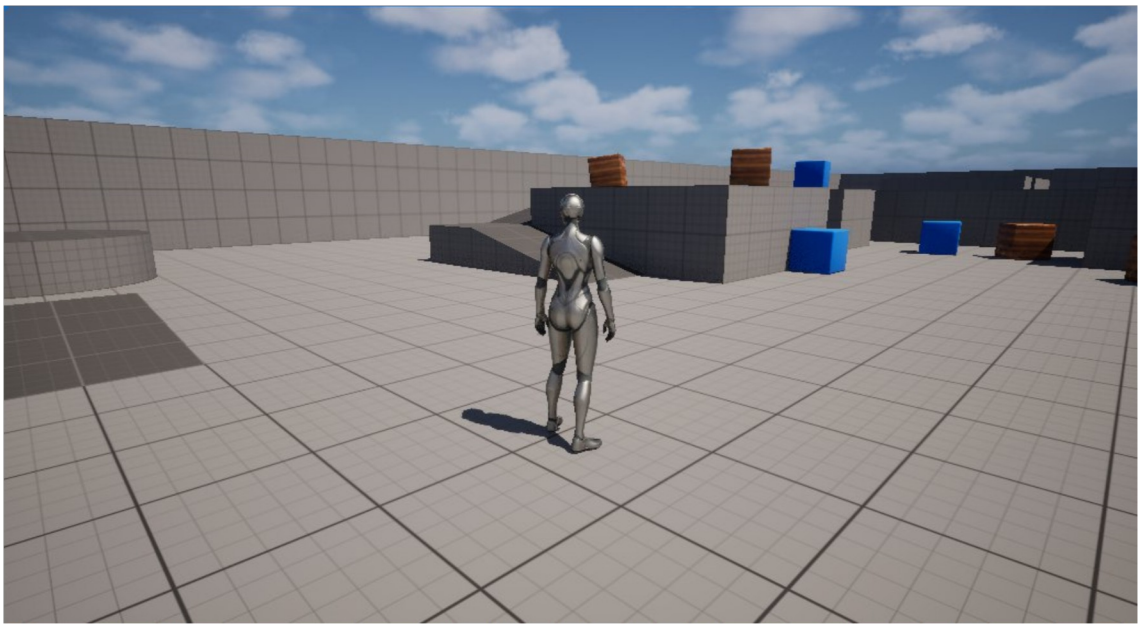

Рис. 3.4. Візуалізація середовища для експериментів

Як середовище для проведення експериментів було зібрано рівень, на якому зібрані різні перепони для проходження рівня:

- платформи, які не мають прямого навігаційного зв’язку з рештою рівня;
- перешкоди, які вирізають під собою навігацію (сині);
- перешкоди, які залишають під собою навігацію без змін (коричневі);
- підйоми, спуски.

Середовище для експериментів можна побачити на рисунку 3.5.

Рис. 3.5. Побудований рівень для проведення експериментів

Дане середовище має ставити виклик обом методам. Де розставлені перешкоди мають збивати потік відтворення через вводи гравця та слідування по навігації.

Кожен експеримент проводився по 5 спроб пройти з першого разу. У результатах проведення експериментів написано частоту успіху.

Були проведені декілька експериментів, щоб оцінити роботу різних методів відтворення та їх комбінування. Були протестовані такі сценарії:

a. проходження по повністю зв’язаному шляху без перешкод;

b. проходження по повністю зв’язному шляху з перешкодами, які невпливають на навігацію на шляху;

c. проходження по шляху з перешкодами, які впливають на навігацію на шляху, і розривають шлях;

d. падіння з висоти;

e. проходження по шляху з різних платформ, які не пов’язані між собою;

f. проходження по шляху з різних платформ, які не пов’язані між собою з перешкодами обох видів.

Результати проходження по повністю зв’язаному шляху без перешкод:

- метод відтворення вводів відпрацював без проблем, 5/5;
- метод навігаційної сітки відпрацював без проблем, 5/5.

Результати проходження по повністю зв’язному шляху з перешкодами, які не впливають на навігацію на шляху:

- метод відтворення вводів часто збивався зі шляху, і доводилось проходити цей самий відрізок заново, доки зсув не ставав прийнятним. 1/5;

- метод навігаційної сітки відпрацював без проблем. Після зіткнення з перешкодами відбувалось самокорегування і агент повертався на початковий шлях. 5/5.

Візуалізацію шляху можна побачити на рисунку 3.6.

Рис. 3.6. Перешкоди, що не впливають на навігацію

Результати проходження по шляху з перешкодами, які впливають на навігацію на шляху, і розривають шлях:

- метод відтворення вводів вів себе так само, як і з перешкодами, які не мають впливу на навігацію 1/5;

- метод навігаційної сітки просто не будував шлях через розірвану перешкодою навігацію 0/5;

- через погані результати на даному сценарії був використаний комбінований метод відтворення шляху. Для цього метод відтворення вводів переходив через перешкоди, де метод навігаційної сітки міг побудувати шлях

до потрібної точки і скорегувати траєкторію, щоб дійти у результуючу точку. Майже всі рази спрацювало 4/5.

Результати падінь з висоти:

- метод відтворення вводів на падіннях мав невелике відхилення під час падінь, що пов’язано з апромаксимованою природою фізичних обчислень, але в цілому показав непогані результати. 4/5;

- метод навігаційної сітки не може відтворити падіння без прямих навігаційних зв’язків верхнього та нижнього сегменту навігації. 0/5.

Результати проходження по шляху з різних платформ, які не пов’язані між собою**:**

- метод відтворення вводів відпрацював без проблем, 5/5;

- метод навігаційної сітки не може пройти на наступну платформу без зв’язку навігаційним зв’язком, 0/5.

Результати проходження по шляху з різних платформ, які не пов’язані між собою з перешкодами обох видів:

- метод відтворення вводів часто збивався зі шляху під час зіткнення з перешкодами. В комбінації з точністю, яку потрібно було досягти не вдалось пройти жодного разу без проблем, оскільки відхилення на перестрибуванні послідуючих перешкод ставало завеликим, 0/5;

- метод навігаційної сітки не може пройти на наступну платформу без зв’язку навігаційним зв’язком, 0/5;

- був використаний комбінований метод, де комбінувався метод відтворення та метод навігаційної сітки. Версія без синхронізації і поточкового відтворення стану теж виявилась невдалою, 0/5. Вона не могла пройти через перешкоду, яка представлена на рисунку n.

Ця перешкода не дає побудувати навігацію на платформі та також при цьому заважає проходженню за допомогою відтворення, бо зміщує агента і відтворення починає розходитись із записом. Для вирішення цієї проблеми було вдосконалено цей метод поточковою синхронізацією та відновленням та відкатом станів. Кожна успішна точка на землі фіксувалась, і коли

відбувалось розходження запису з відтворенням, то відбувався відкат на декілька точок і зміна метода.

Завдяки цьому методу вдалося вирішити дану задачу, та загалом підвищилась загальна надійність методу відтворення, оскільки навіть неточності можна скоригувати через фіксування точок та відтворення стану з будь-якої записаної точки. Якщо стан повністю не можна відновити, то він відкатується ще трохи далі і потім відновлює його під час відтворення більш старих точок. 5/5.

Демонстрацію процесу проходження можна побачити на рисунку 3.7. Результати проведення експериментів можна побачити у таблиці 3.1.

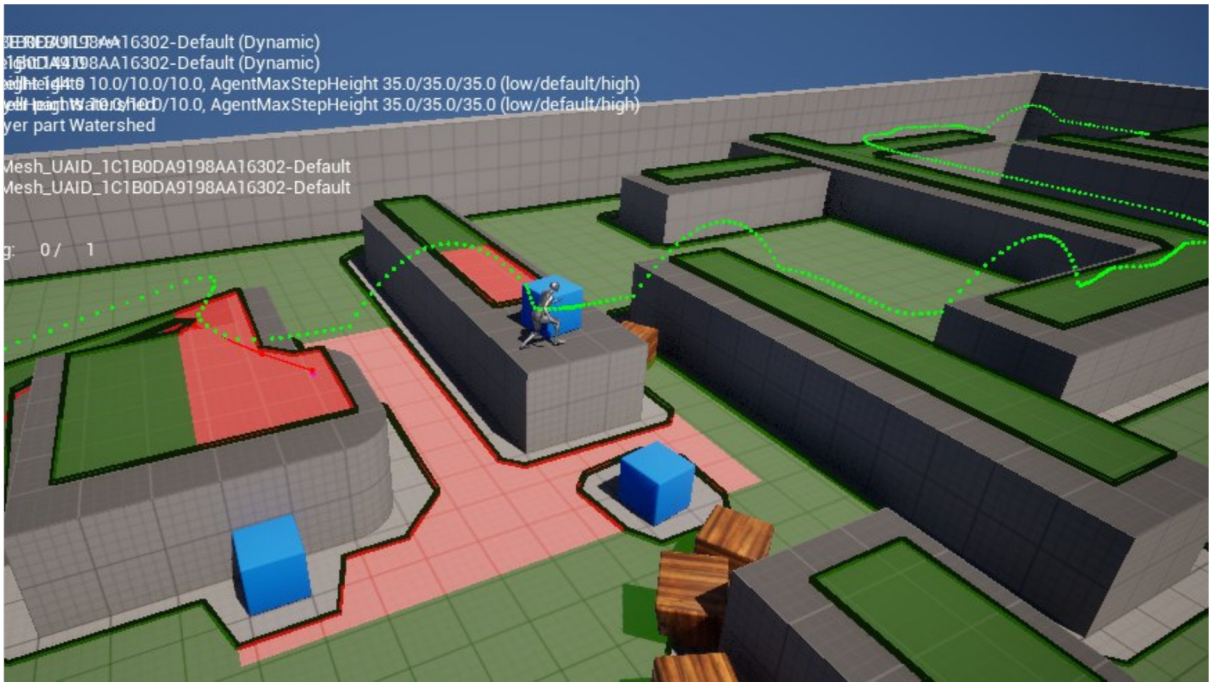

Рис. 3.7. Проходження експерименту з використанням комбінованого методу з відслідковуванням точок

Таблиця 3.1. Порівняння методів відтворення

Експеримент	Відтворення вводів	Навігаційна сітка	Комбінований метод	Комбінований з фіксуванням точок
a	5/5	5/5	5/5	5/5
b	1/5	5/5	5/5	5/5

c	1/5	0/5	4/5	5/5
d	4/5	0/5	5/5	5/5
e	5/5	0/5	5/5	5/5
f	0/5	0/5	0/5	5/5

Висновок до розділу 3

У третьому розділі були розглянуті реалізація та проведення експериментів з методами end-to-end тестування на різних сценаріях.

Був розглянутий опис технологічного стеку для реалізації. Для реалізації програми був обраний рушій Unreal Engine, оскільки він має розвинуту екосистему, яку будуть доповнювати реалізовані методи End-to-End тестування.

Описаний формат даних, щоб відтворювати дії користувача: які дані та для чого використовувались. Також описані особливості адитивних, неадитивних, цифрових та аналогових вводів за механізми їх відтворення.

Наступними були досліджені методи відтворення: описані особливості їх реалізації та алгоритмів, які лежать в їх основі. Метод відтворення по локації при реалізації виявився непрактичним, тож було вирішено від нього відмовитись і натомість зосередитись на методі відтворення вводів та методі навігаційної сітки та їх комбінації.

У кінці були описані експерименти для перевірки методів відтворення дій гравця. Найкращі результати показав комбінований метод відтворення з фіксуванням пройдених точок.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЗАПРОПОНОВАНОГО МЕТОДУ АСТОМАТИЗОВАНОГО ТЕСТУВАННЯ

4.1. Тестування генерацією тестових оракулів

У цьому експерименті досліджено, чи можна вважати тестування з багатьох реалізацій як один з методів тестування моделей.

У цьому експерименті запускають десять різних реалізацій алгоритмів автоенкодера з варіаційним засобом (VAE) з однаковими тестовими вхідними даними та конфігураціями.

Враховуючи всі вихідні дані моделі, визначено більшість голосів відповідей цих реалізацій як тестового оракула.

Використано набір даних MNIST [22], безкоштовний набір даних з понад 30 000 ігрових зображень.

Для тестового вводу створено два зображення.

Ці тестові вводи допомагають нам побачити, наскільки добре різні реалізації можуть генерувати зображення. Я використовую ці тестові вводи для вимірювання якості тестових вводів моделі.

Рисунки 3.3 та 3.4 – це два зразки тестових входів, використаних для відповіді на питання 1 у цьому експерименті.

Автоенкодери є методом навчання без учителя, який використовує нейронні мережі для завдання навчання представлення даних. Архітектура нейронної мережі спроектована таким чином, щоб в мережі був вузький місток для стиснення вихідного представлення знань.

Якщо властивості вхідних даних незалежні один від одного, це стискання та подальша реконструкція буде складним завданням. Однак, якщо дані мають деякі структури, наприклад, кореляції між вхідними ознаками, ця структура може бути вивчена та використана при примусовому проходженні даних через місток.

Як зображено на рисунку 4.1, немаркований набір даних можна розглядати як задачу навчання з вчителем, що має на меті виведення xˆ, відтворення оригінального вводу x.

Bottleneck – це важлива характеристика мережевого дизайну. Без інформаційного bottleneck [46] мережа може швидко вивчити запам'ятовувати значення вводу, передаючи їх через мережу. Bottleneck обмежує кількість інформації, яка може пройти повну мережу, змушуючи навчитися стисненню вхідних даних.

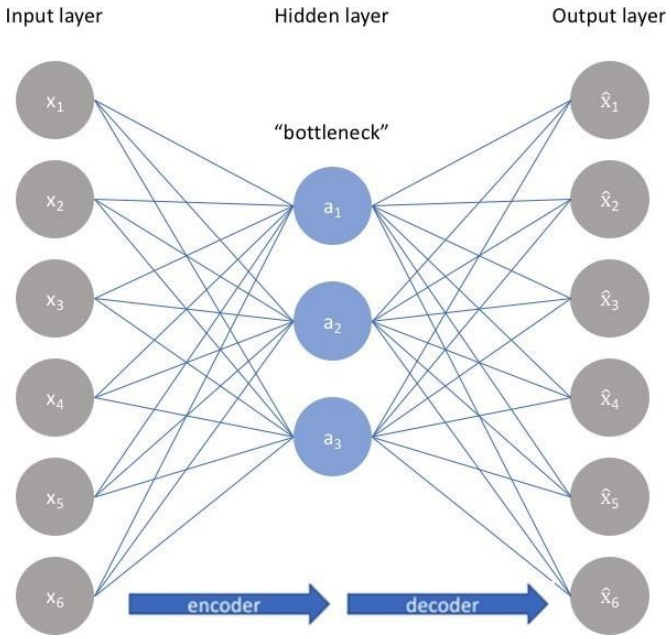

Рис. 4.1. Загальна структура моделі автокодера

Найпростіша архітектура для побудови автоенкодера полягає в обмеженні кількості вузлів у прихованому(их) шарі мережі, обмежуючи кількість інформації, що може пройти через нього. Шляхом покарання мережі за реконструкційну похибку, модель може навчитись основних атрибутів вхідних даних та найкращого способу відтворення початкового введення зі "стану кодування".

В ідеалі, це кодування навчається і описує латентні атрибути вхідних даних.

У автоенкодера в кодері є кілька повністю зв’язаних шарів, які беруть вхід і стискають його до меншого представлення з меншою кількістю

вимірів, ніж вхід. Це представлення називається затором або прихованим шаром. Потім з затору він намагається відновити вхід за допомогою генеративно-адверсарної мережі.

Використання варіаційного автоенкодера бере протилежний підхід. Розподіл, який слідується латентними векторами, не є питанням, і цільовий розподіл - те, до чого модель намагається дійти.

За лише три роки Варіаційні Автоенкодери (VAE) [29] стали одним з найпопулярніших підходів до навчання без допомоги учителя складних розподілів. VAE мають привабливість тому, що вони ґрунтуються на стандартних апроксимаціях функцій (нейронних мережах) і можуть навчатися за допомогою стохастичного градієнтного спуску [26].

У відміну від автоенкодерів, які перетворюють вхідні дані на фіксовані вектори, Варіаційні автоенкодери (VAE) перетворюють їх на розподіл. Цю ідею можна пояснити так: у стандартному автоенкодері зображення перетворюється в вектор, а в VAE вектори, що отримуються з зображення, перетворюються в розподіл. Це зображено на рисунку 4.2, де замість звичайного bottleneck у VAE використовують два окремі вектори, один з яких представляє середнє значення, а інший - стандартне відхилення розподілу. Таким чином, коли потрібен вектор для подачі в мережу декодеру, з розподілу береться випадковий зразок і підсилюється через декодер.

У всіх реалізаціях алгоритму VAE встановлено наступні параметри:

- Кількість епох = 50
- Кількість шарів латентних змінних = 20
- Розмір пакету = 128
- Кількість нейронів: 100
- Розмір вхідного зображення: 488 x 488 пікселей.

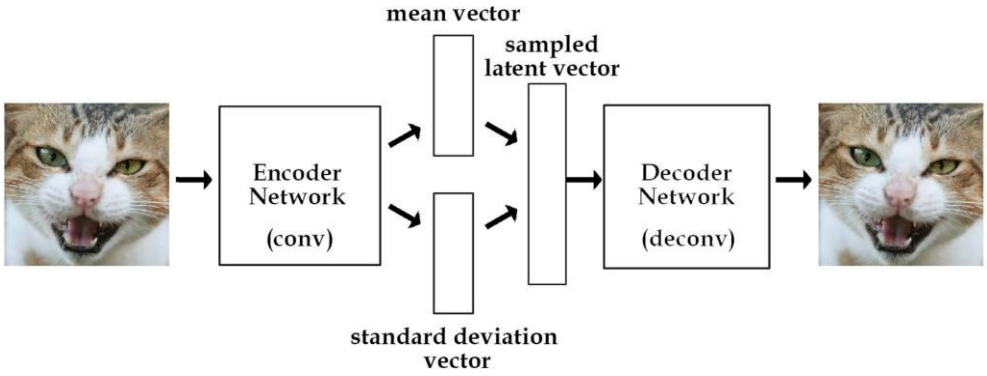

Рис. 4.2. Модель варіаційного автокодера з точки зору його функціональності

Де кількість епох вказує кількість ітерацій для кожного алгоритму, Кількість шарів латентних змінних вказує шари звуження, що представляють вектор, Розмір пакету – кількість прикладів в одній партії, для яких виконуються передбачення одночасно. Кількість нейронів вказує кількість вузлів у кожному шарі, крім звуження, а розмір вхідного зображення вказує розмір кожного зображення для навчання. Я використовував всі конфігурації, базовані на найчастіших числах для кожної функції в усіх реалізаціях.

Для оцінки згенерованих зображень використано метрику, яка називається Fre´chet Inception Distance [16]. Fre´chet Inception Distance (FID) – це одна з метрик для автоматичної оцінки якості генеративних моделей зображень [16]. FID – це вимірювання, яке порівнює два зображення піксель за пікселем. Розмір зображення та його розмитість можуть впливати на оцінку, яка є числом з плаваючою точкою та показує схожість зображень.

Зазначається, що порівняння одного зображення з самим собою буде мати оцінку FID 0,00. У цьому експерименті я використовую FID для порівняння вхідних зображень зі згенерованими зображеннями всіма реалізаціями. Нижче наведено повне визначення оцінки FID.

Inception Score (IS) [26] використовує якість згенерованих зображень та їх різноманітність як два критерії. Я хочу, щоб умовна ймовірність P(y/x) була дуже передбачуваною в згенерованих зображеннях. Тому я використовую мережу Inception для класифікації згенерованих зображень та передбачення P(y/x) - мітки x. Це відображає якість зображень.

Якщо згенеровані зображення є різноманітними, розподіл даних для y повинен бути рівномірним. Щоб поєднати ці два критерії, обчислюємо їх KL-дивергенцію та використовуємо рівняння нижче для обчислення IS.

$$IS(G) = exp(E_{x\sim p_a} D_{KL}(p(y|x)||p(y)))$$

FID використовує мережу Inception для вилучення ознак з проміжного шару. Потім, для цих ознак я моделюю розподіл даних за допомогою багатовимірного гаусівського розподілу з математичним очікуванням μ та коваріаційною матрицею Σ. FID між реальними зображеннями x та згенерованими зображеннями g обчислюється за допомогою формули, яку я наводжу нижче.

$$FID(x,g) = ||\mu_x = \mu_g||_2^2 + Tr(\Sigma_x + \Sigma_g - 2(\Sigma_x\Sigma_g)^{\frac{1}{2}})$$

Тут Tr позначає суму всіх діагональних елементів матриці. Менші значення FID означають кращу якість та різноманітність зображень. Відстань збільшується при симуляції пропущених мод. FID більш стійкий до шумів, ніж IS. Якщо модель генерує лише одне зображення для кожного класу, то відстань буде високою. Тому FID є кращим показником для різноманітності зображень. У FID є досить висока систематична похибка, але низька дисперсія. Якщо обчислити FID між набором даних для навчання та набором даних для тестування, то я маю очікувати, що FID буде рівним нулю, оскільки обидва набори містять реальні зображення. Однак при тестуванні з різними партіями вибірки для навчання FID показує ненульове значення.

Для оцінки якості протестовано всі 10 реалізацій з двома тестовими вхідними зображеннями. Кожна реалізація зайняла близько п'яти годин, щоб згенерувати результати у відношенні часу виконання. Для оцінки якості згенерованих зображень обчислено відстань між згенерованим зображенням та оригінальним зображенням з порогом FID 120.

Цей емпіричний поріг оцінювався вручну, оскільки не існує іншої попередньої роботи з тестуванням декількох реалізацій на VAE.

Щоб перевірити, чи залежить оцінка FID згенерованих зображень від конфігурації або моделі DNN, я запустив усі моделі з їх наборами конфігурацій та вхідними даними. Нарешті, маючи тестового оракула, я порівняв різні реалізації вручну, щоб побачити, чи вказує різниця між оцінками FID оригінального зображення та згенерованого зображення на наявність якихнебудь недоліків.

Далі я розгляну відповіді на дослідницьке запитання експерименту, обговоривши результати. Таблиця 4.1 показує результати кожної реалізації для алгоритму VAE, де I_i – це *i*-та реалізація.

Таблиця 4.2 також показує результати всіх реалізацій на основі другого входу.

Таблиця 4.3 показує значення FID для кожної реалізації з її параметрами за замовчуванням. Незважаючи на те, що інші дві таблиці використовують параметри, зазначені вище, у цій таблиці параметри встановлені так само, як і за замовчуванням для кожної реалізації при клонуванні. Чим більше число епох, тим більше часу потрібно для компіляції кожної реалізації.

Кінцевим кроком було поєднання FID оцінок наших реалізацій у таблиці 4.4, відсортувавши їх у порядку зростання FID, який можна знайти в таблицях 4.1, 4.2 і 4.3, що відповідають першому введенню, другому введенню та різним параметрам та введенням відповідно. Вони були відсортовані, починаючи з найменшого FID.

Нарешті, був встановлений поріг α для знаходження більшості згодних реалізацій. Оскільки не було попередніх робіт з встановленням α, вибрано α = 120 на основі спостережень та порівняння якості згенерованих зображень.

Це свідчить про те, що зміна конкретного параметру може стати причиною різниці в якості згенерованих зображень. Як показано в таблиці 3.3, за замовчуванням кількість прихованих шарів для цієї реалізації становить 50,

Таблиця 4.1. Розрахована оцінка FID для кожної реалізації для першої картинки

Ідентифікатор реалізації	FID Оцінка
*І*1	158.89
*І*2	229.28
*І*3	117.49
*І*4	67.5
*І*5	205.37
*І*6	62.43
*І*7	107.74
*І*8	95.05
*І*9	54.38
*І*10	135.01

Таблиця 4.2. Розрахована оцінка FID для кожної реалізації для другої картинки

Ідентифікатор реалізації	FID Оцінка
*І*1	166.23
*І*2	257.67
*І*3	125.11
*І*4	84.55
*І*5	193.49
*І*6	76.21
*І*7	115.89
*І*8	102.9
*І*9	68.21
*І*10	143.70

Таблиця 4.3. Результати реалізацій з параметрами та вхідними даними за замовчуванням

Ідентифікатор реалізації	FID Оцінка	Кількість епох	Кількість латентних шарів	Розмір партії	Кількість нейронів
*I*1	200.09	10	20	100	100
*I*2	233.42	20	20	128	128
*I*3	115.77	75	3	100	100
*I*4	79.82	10	20	100	100
*I*5	35.37	20	50	128	128
*I*6	89.07	50	32	64	64
*I*7	40.27	60,000	20	32	32
*I*8	84.62	20	20	128	128
*I*9	24.1	1000	32	64	64
*I*10	172.21	301	20	100	100

Таблиця 4.4. Сортування реалізацією FID спочатку враховує найнижчий FID, виходячи з наших трьох таблиць

Табл. 4.1	Табл 4.2	Табл 4.3
I9	I9	I9
I6	I6	I5
I4	I4	I7
I8	I8	I4
I7	I7	I8
I3	I3	I6
I10	I10	I3
I1	I1	I10
I5	I5	I1
I2	I2	I2

тоді як кількість прихованих шарів, встановлених для всіх моделей, дорівнює 20.

Рисунок 4.3 показує вхідне зображення, шумове вхідне зображення та згенероване зображення на першій та десятій епохах для реалізації I_1.

Розглядаючи згенероване зображення за допомогою I_1 на рисунку 4.3, можна побачити, що додавання шуму може призвести до генерації гірших зображень. Реалізація I_2 також використовує такий же підхід для генерації зображень.

Рисунок 4.4 показує процес генерації зображень з шумом для I_2. Варто зазначити, що можуть існувати різні типи додавання шуму до вхідного зображення, які не входять до області дослідження цього дослідження, але вони можуть впливати на якість згенерованих зображень.

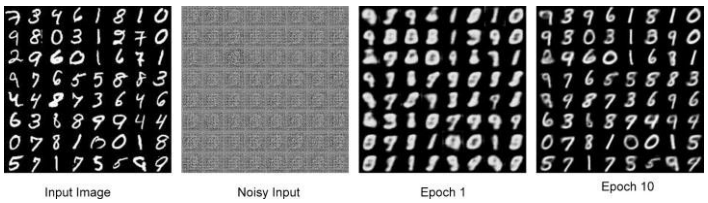

Рис. 4.3. Процес додавання шуму для реалізації I_1

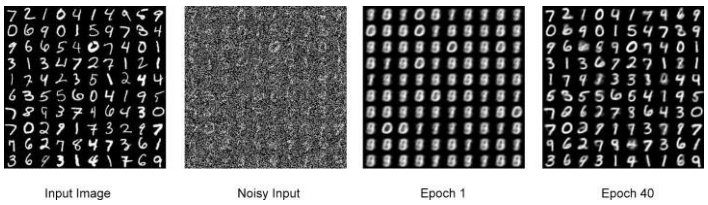

Рис. 4.4. Процес додавання шуму для реалізації I_2

Отже, під час зберігання параметрів в межах приблизно однакових значень немає зміни у відсортованих результатах FID. Це означає, що якщо при кожній реалізації параметри мають приблизно однакові значення, якість згенерованих зображень залишається незмінною, незалежно від вхідного зображення. З іншого боку, останній стовпець показує, що можна отримати різні результати змінюючи параметри та вхідне зображення.

Використання MIT може допомогти виявити будь-які помилки, що спричиняють різні виходи на різних імплементаціях для моделей. Отже, MIT може допомогти виявити будь-які помилки або відмінності в імплементації моделей автоматизованого навчання з допомогою нейронних мереж.

4.2. Оцінка адекватності тесту

У цьому експерименті критерії адекватності тестування для моделей оцінюються за допомогою порівняння технік тестування на основі покриття та мутацій.

Раніше не проводилось прямого порівняльного дослідження між двома основними групами метрик (тобто критеріїв покриття та мутаційної здатності), щоб виявити адверсарні приклади.

У цьому експерименті запропоновано підхід для порівняння метрик Surprise Coverage та Model Mutation здатності для визначення того, який з них може мати більший приріст за допомогою додавання адверсарних прикладів до вихідного набору тестів. У цьому експерименті чутливість визначена як приріст метрики при додаванні частини адверсарних прикладів. Тому, більший приріст в балів метрик показує вищу чутливість до адверсарних атак.

Використано набір даних CIFAR-10 [19], колекцію часто використовуваних зображень для тренування алгоритмів машинного навчання та комп'ютерного зору. Ці два набори даних також використовувалися в обох початкових статтях, які запропонували SA і DeepMutation. Тому наш вибір не буде спрямований на одну метрику.

Цільовими глибокими моделями навчання для наборів даних CIFAR-10 є мережі з п'ятьма шарами згортки та дванадцятьма шарами згортки відповідно, тоді як у [23] використовується LeNet та GoogleNet відповідно.

Налаштування:

• Конфігурація Surprise Coverage: Був встановлений типовий поріг активації для метрики LSC на рівні 105.

• Конфігурація Mutation Model: Для методу мутації моделей потрібно встановити дві конфігурації. По-перше, оператор мутації, який генерує змінені моделі, а по-друге, швидкість мутації. Для цього експерименту вибрано Gaussian Fuzzing (GF), оскільки він був ефективнішим за інші

методи. Генеруючи десять мутованих моделей для кожної моделі, враховуємо її внутрішню випадковість.

Мета також полягає в вимірюванні чутливості метрик щодо змін параметрів. Тому необхідно встановити кілька конфігурацій. Кожна конфігурація змінює один аспект експерименту:

- Обрання набору даних та моделі.
- Вибір шару для покриття в Surprise Coverage: одним з важливих гіперпараметрів LSC є вибір шару для вимірювання покриття.
- Вибір рівня мутації: набір даних MNIST та модель LeNet, та випадковий атакувальний метод JSMA.
- Вибір оператора мутації: вибрано один оператор (GF) і досліджено два інших оператори (NS і WS).

Для критерію Surprise Adequacy використано метрику покриття Likelihood-based Surprise Coverage (LSC), а не DSA. DSA використовується лише для класифікації атакованих та оригінальних прикладів, а не для розрахунку покриття нейронів.

Таблиця 4.5 показує кількість атакованих тестових вхідних даних для кожної атаки з використанням набору даних CIFAR-10.

Таблиця 4.5. Кількість прикладів спроб зловмисників на одну атаку, на один набір даних

Ворожі атаки	CIFAR-10
Black-Box	1000
Deepfool	1000
FGSM	2000
JSMA	2000
C&W	1000

На рисунку 4.5 зображено збільшення показника (LCR та LSC), коли до тестового набору додаються деякі випадкові атаки.

На кожному кроці додається 1% додаткових атак.

При детальному розгляді графіків можна побачити, що для всіх атак на обох моделях глибокого навчання метрика, заснована на мутації (LCR), має значно більші збільшення, ніж метрика, заснована на покритті (LSC).

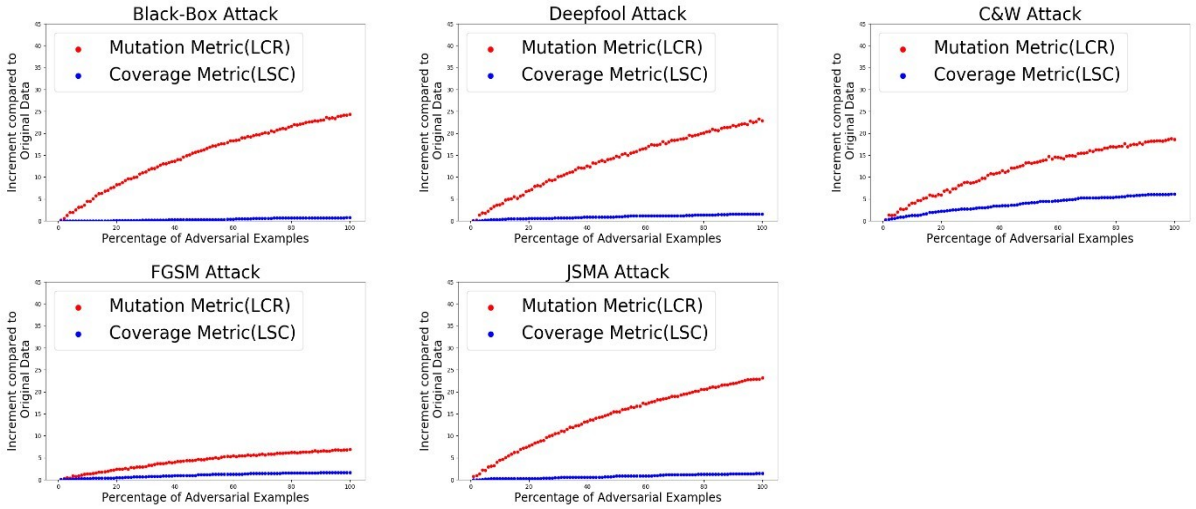

Рис. 4.5. Набір даних CIFAR-10 націлений на 12-шарову згорткову модель з п'ятьма різними ігровими прикладами

Як показано на рисунку 4.6, зміна шару призвела до більшого збільшення покриття несподіванки (LSC), а також зробила LSC краще, ніж LCR. Цей один приклад був достатнім, щоб продемонструвати, що вибір параметра гіперпараметра адекватності метрики (тут шар в LSC) важливий для виявлення атак. Іншими словами, з правильною настройкою можна використовувати LSC порівняно з LCR або навпаки.

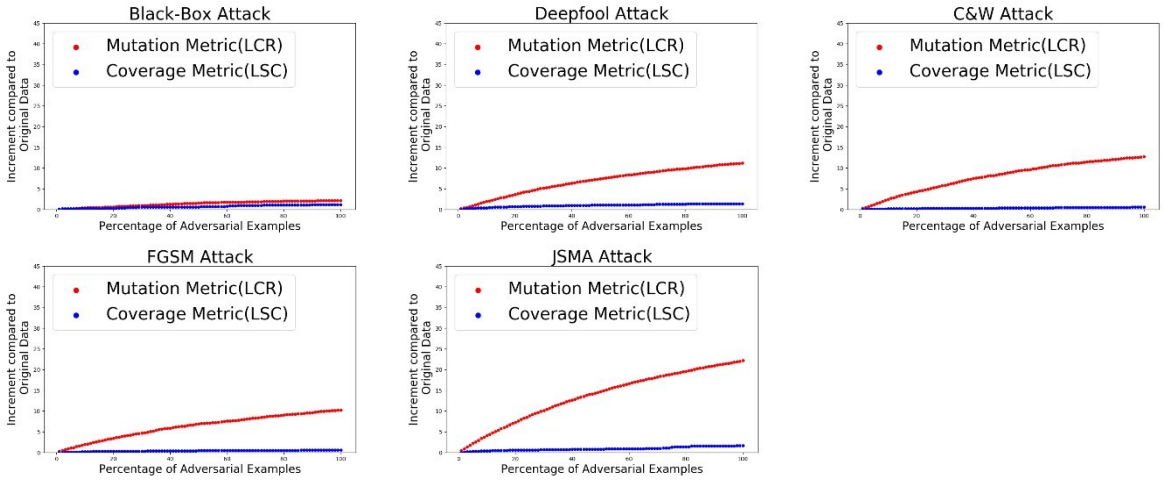

Рис. 4.6. Набір даних CIFAR-10 націлений на модель GoogleNet з п'ятьма різними змагальними іспитами.

Таблиця 4.6. Коефіцієнт кореляції Пірсона між "Відсотком атакованих прикладів" та "Збільшенням оцінки адекватності (LCR та LSC) порівняно з вихідними тестовими даними".

	Модель	Black-Box	Deepfool	FGSM	JSMA	C&W
SC	LeNet	0.987	0.929	0.99	0.978	0.947
	Conv 5	0.994	0.982	0.98	0.985	0.986
	GooglLeNet	0.988	0.956	0.963	0.969	0.972
	Conv 12	0.98	0.989	0.98	0.99	0.986
Mutation	LeNet	0.985	0.984	0.984	0.982	0.984
	Conv 5	0.984	0.984	0.985	0.984	0.984
	Googl LeNet	0.978	0.984	0.985	0.984	0.982
	Conv 12	0.983	0.989	0.98	0.9985	0.98

4.3. Генерація тестових вхідних даних

У третьому експерименті досліджується третій аспект автоматизованого тестування ігор за допомогою нейромереж – генерація вхідних тестів.

Мета полягає в тому, щоб створювати вхідні тести (адверсальні зразки) для моделей, які знаходять проблеми з моделлю (наприклад, помилкову класифікацію) і потім виправляють модель, перетренуючи її зі згенерованими вхідними тестами.

У цьому експерименті спочатку генеруються адверсарні приклади за допомогою трьох різних методологій, потім досліджується продуктивність моделі, і нарешті перетреновується модель з використанням згенерованих прикладів тестування, і перевіряється, чи є якісь покращення в F1-показнику.

У сфері програмної інженерії існує три відомих інструменти для вбудовування коду під назвами Code2vec [26], Code2seq [27] та CodeBERT [28]. Усі три моделі пропонують метод вбудовування фрагментів коду, оскільки широке коло застосувань у програмній інженерії, таких як підсумовування коду, документація та пошук, вимагають їх використання.

Code2vec пропонує нейронну модель для представлення фрагментів коду у вигляді неперервно розподілених векторів, тоді як Code2seq запропонувала трансформацію для генерації послідовностей природньої мови з фрагментів вихідного коду. Нарешті, обидві моделі оцінюють свій метод, передбачаючи назву методу на основі вихідного коду.

CodeBERT навчається універсальним представленням, що підтримує додаткові завдання у сфері програмної інженерії, такі як пошук коду за природною мовою і генерація документації для коду. Це двохмодельна переднавчена модель для природної мови (Natural Language, NL) та мов програмування (Programming Language, PL), таких як Python та Java.

У цьому дослідженні також використовуються оригінальні необроблені java-файли як вхідні дані та генеруються адверсарні приклади за допомогою оригінальних.

Оригінальні файли Java: Як Code2vec, так і Code2seq підтримують мови програмування Java і C# як вхідні кодові фрагменти. Code2seq також підтримує мову Python. CodeBERT підтримує згадані мови програмування, а також JavaScript, PHP, Ruby та Go.

Датасет, опублікований Code2vec, має передопрацьований формат, але в цьому дослідженні потрібні були сирі файли, щоб створити атакуючі приклади. Він також розділяє датасет на тренувальні, валідаційні та тестові набори за одним файлом Java, тоді як Code2seq розділяє їх за проектними папками. На щастя, датасети, які супроводжуються Code2seq, містять сирі файли Java. CodeBERT також забезпечив передопрацьовані набори даних для тренування та валідації і надав код для передопрацювання тестового датасету.

Отже, використовуємо датасет Java-Large на сторінці Github Code2seq для всіх трьох моделей, включаючи 9000 проектів Java для тренування, 200 для валідації та 300 для тестування. Цей датасет містить близько 16 мільйонів прикладів.

Таблиця 4.7. Оригінальна оцінка F1 моделі наведена у відповідній статті з конфігураціями за замовчуванням на наборі даних за замовчуванням

Інструмент	F1-score	Подальше завдання
Code2vec	58.4	Прогнозування назви методу
Code2seq	59.19	Прогнозування назви методу
CodeBERT	74.84	Пошук коду

Інструменти Code2vec, Code2seq та CodeBERT є доступними для повторення досліджень. У таблиці 4.7 показано значення F1-оцінки, які були звітовані з кожного відповідного інструменту.

Хоча навчена модель для Code2seq та CodeBERT відповідає продемонстрованій продуктивності, Code2vec не міг досягнути F1-оцінки, через відсутність публічно доступних наборів даних.

Крім того, через перенесення всіх java-файлів у єдину теку, значення F1оцінки для Code2vec також відрізняється від тої, що використовувала набір даних Java-Large.

У експерименті встановлено кількість епох 20 і частоту мутації 0,05, оскільки дослідження показують, що це хороша частота мутації для проблеми генетичного алгоритму [13].

Метрикою оцінки є F1-оцінка.

ROUGE, або Recall-Oriented Understudy for Gisting Evaluation [14], це набір метрик та програмного забезпечення, що використовуються для

оцінювання програмного забезпечення автоматичної резюмування та машинного перекладу в обробці природної мови. Метрики порівнюють автоматично створену резюме або переклад з людською резюме або перекладом.

Доступні наступні п'ять метрик оцінки:

- ROUGE-N: Належність N-грамів [6] між системою та довідковими резюме.
- ROUGE-1 відноситься до належності одиночних слів (уніграм) між системою та довідковими резюме.
- ROUGE-2 відноситься до належності біграм між системою та довідковими резюме.
- ROUGE-L: статистика на основі найдовшої спільної послідовності (LCS) [13]. Задача знайдення найдовшої спільної підпослідовності враховує подібність структури на рівні речень та автоматично ідентифікує найдовші спільно входящі в послідовність n-грами.

Варто зазначити, що деякі випадкові техніки рефакторингу можуть бути непридатні для методу Java.

1-кратна мутація: як показано на рисунку 4.7, у методі 1-кратної мутації рефакторяться всі початкові файли всього один раз. Точніше кажучи, спочатку модель виділяє кожен метод Java, а потім випадковим чином вибирає оператор рефакторингу з пулу операторів, який застосовується до методу.

Рис. 4.7. Процес 1-кратної мутації, який було використано у третьому експерименті

Наприклад, якщо певний метод не містить жодного циклу, то випадково обраний метод покращення циклу не може бути застосований тут.

Після того, як всі методи виділені та рефакторовані, генеруються відверсійні файли Java. На рисунку 4.8 наведено зразок 1-разового рефакторингу, за допомогою оператора додавання аргументу.

К-кратна мутація: Підхід к-кратної мутації схожий на підхід 1-разової мутації, з рефакторингом одного методу к-разів. Після видобування кожного методу Java застосовується випадково вибраний оператор рефакторингу, і цей процес повторюється к-разів для кожного методу.

Іншими словами, кожен метод Java рефакторіться к-разів з к-випадково вибраними операторами, як показано на рисунку 4.9.

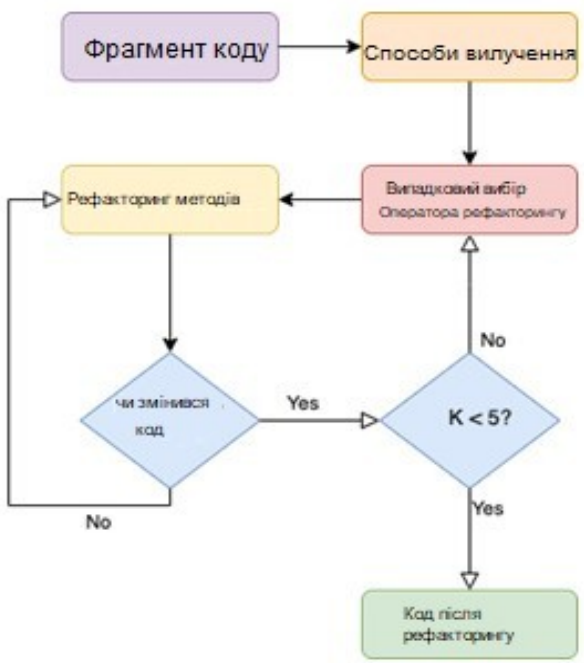

Рис. 4.8. процес 5-кратної мутації, що використовувався в третьому експерименті.

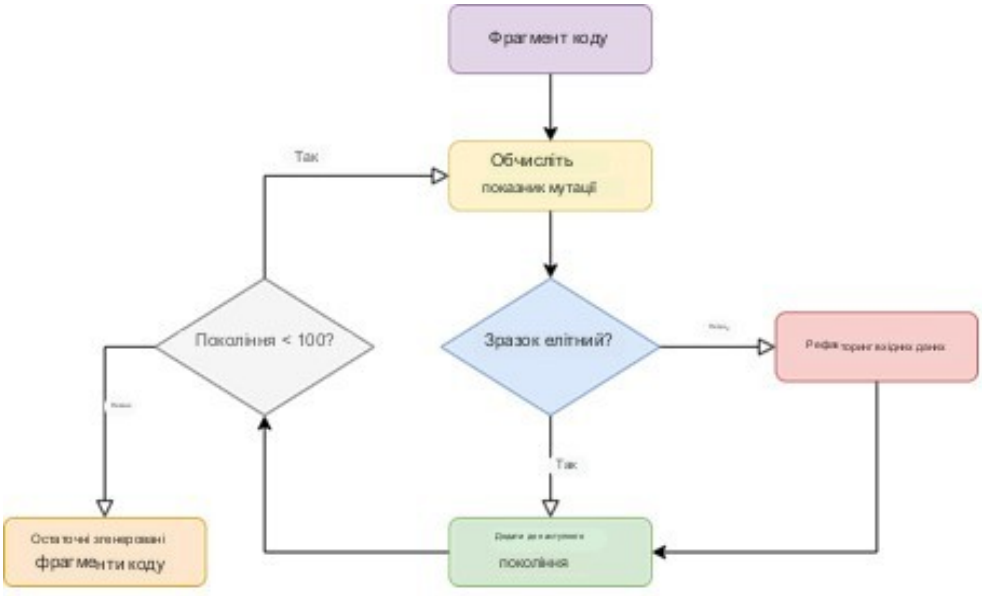

Рис. 4.9. Процес керованої мутації, використаний у третьому експерименті

Знову ж таки, деякі випадкові техніки рефакторингу можуть бути непридатними. Тому ітерують процес з різними операторами, щоб переконатися, що метод був рефакторингований. Розглядається виключно K = 5, оскільки в [25] оцінили різні значення K та запропонували, що K = 5 має найкращий показник F1-оцінки.

Цей експеримент запропонував методологію Guided Mutation (GM) для генерації адверсарних вхідних даних для вбудовування коду. Метод Guided mutation є еволюційною стратегією, натхненною генетичним алгоритмом (GA) за структурою, але без застосування кросовера на генерацію вхідних даних.

Після того, як визначено генетичну репрезентацію та функцію пристосованості, GA продовжує ініціалізувати популяцію рішень та поліпшувати її за допомогою повторних застосувань мутації та кросовера. Як обговорювалось, у методі GM застосовано лише мутацію, а не кросовер на вхідних даних. Причина полягає в тому, що зміна фрагментів коду за допомогою кросовера може призвести до багатьох помилкових (навіть некомпільованих) фрагментів коду, не кажучи вже про збереження функціональності коду.

Елітизм у генетичному алгоритмі передбачає копіювання невеликої частини найбільш пристосованих кандидатів без змін до наступного покоління, що може мати драматичний вплив на продуктивність, забезпечуючи, що еволюційний алгоритм не витрачає час на повторне відкриття раніше відкинутих часткових рішень. Кандидати, які залишаються незмінними завдяки елітизму, можуть бути вибрані батьками для розвитку наступного покоління.

Як показано на рисунку 4.9, для генерації адверсних зразків за допомогою методології GM потрібно виконати наступні кроки:

1. Обчислити бал мутації для поточного покоління.
2. Вибрати елітні кандидати на основі найвищого балу мутації та скопіювати їх у наступне покоління.
3. Мутувати решту кандидатів з вказаними швидкостями мутації.
4. Повторювати крок перший до досягнення критерію зупинки (100 ітерацій).

DeepMutation++ [24] – це фреймворк для тестування мутацій з метою аналізу якості тестових даних. Оператори мутацій представлені в таблиці 4.8.

Таким чином, у цій дисертації я використовую метрику балів мутації в DeepMutation++ для оцінки якості тестових даних, які я генерую за допомогою методології GM.

Таблиця 4.8. Оператори мутації

Рівень		Оператор	Пояснення
Статичний	**Вага**	Фаззінг ваги за допомогою гаусівського розподілу (WGF)	Фаззінг ваги
		Зменшення точності ваги (WPR)	Зменшення точності ваг
Динамічний	**Статус**	Очистка стану до 0 (SC)	Очистити стан до 0
		Скидання стану до попереднього стану (SR)	Скинути стан на попередній стан
		Фаззінг значення стану за допомогою гаусівського розподілу (SGF)	Фаззінг значення стану
		Зменшення точності значення стану (SPR)	Зменшення точності значення стану
	Гейт	Очистка значення гейту до 0 (GC)	Очистити значення гейту до 0
		Фаззінг значення гейту за допомогою гаусівського розподілу (GGF)	Фаззінг значення гейту
		Зменшення точності значення гейту (GPR)	Зменшення точності значення гейту

Спочатку DeepMutation [23] вперше запропонував вісім операторів на рівні моделі для глибинних нейронних мереж, щоб створювати модельні мутанти. Пізніше DeepMutation++ запропонував дев'ять нових операторів, спеціалізованих на рекурентних нейронних мережах. Він підтримує статичне генерування мутантів для аналізу тестових даних в цілому та динамічне генерування мутантів для виявлення вразливих сегментів тестового вводу. У таблиці 4.8 показано 9 операторів, визначених DeepMutation++.

DeepMutation++ спочатку використовує надані оператори мутації для генерації набору високоякісних мутантів з певним заданим порогом якості. Після створення певної кількості мутантів DeepMutation++ аналізує відмінності у поведінці початкової мережі та згенерованих мутантів на наданих вхідних даних. Нарешті, вона виводить якість тестових даних, надаючи мутаційний бал, враховуючи конкретну модель.

Перетренування моделі вбудування коду за допомогою адверсарних зразків покращує продуктивність моделі, незалежно від завдання оцінювання моделі.

```
private static JoinPoint currentJoinPoint() {
  MethodInvocation mi = ExposeInvocationInterceptor.currentInvocation();
  if (!(mi instanceof ProxyMethodInvocation)) {
    throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ mi);
  }
  ProxyMethodInvocation pmi = (ProxyMethodInvocation) mi;
  JoinPoint jp = (JoinPoint) pmi.getUserAttribute(JOIN_POINT_KEY);
  if (jp == null) {
    jp = new MethodInvocationProceedingJoinPoint(pmi);
    pmi.setUserAttribute(JOIN_POINT_KEY , jp);
  }
  return jp;
}
```

Рис. 4.10. Оригінальний фрагмент коду Java з набору даних JavaLarge

```
private static JoinPoint currentJoinPoint(String currentJoinPoint) {
  MethodInvocation mi = ExposeInvocationInterceptor.currentInvocation();
  if (!(mi instanceof ProxyMethodInvocation)) {
    throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ mi);
  }
  ProxyMethodInvocation pmi = (ProxyMethodInvocation) mi;
  JoinPoint jp = (JoinPoint) pmi.getUserAttribute(JOIN_POINT_KEY);
  if (jp == null) {
    jp = new MethodInvocationProceedingJoinPoint(pmi);
    pmi.setUserAttribute(JOIN_POINT_KEY , jp);
  }
  return jp;
}
```

Рис. 4.11. Згенерований код змагальний за допомогою методу 1-разового рефакторингу

```
private static JoinPoint currentjoinpoint(String new_JoinPoint) {
  MethodInvocation im = ExposeInvocationInterceptor.currentInvocation();
  if (!(im instanceof ProxyMethodInvocation)) {
    throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ im);
    System.out.print("MethodInvocation is not a Spring ProxyMethodInvocation: "+ im);
  }
  ProxyMethodInvocation pmi = (ProxyMethodInvocation) im;
  JoinPoint jp = (JoinPoint) pmi.getUserAttribute(JOIN_POINT_KEY);
  for (i = 0; i < 1; i++){
    if (jp == null) {
      jp = new MethodInvocationProceedingJoinPoint(pmi);
      pmi.setUserAttribute(JOIN_POINT_KEY , jp);
    }
  }
  return jp;
}
```

Рисунок 4.12. Згенерований код змагальний за допомогою методу 5-разового рефакторингу.

```
private static JoinPoint currentJoinPoint() {
  ProcessInvocation returningName = ExposeInvocationInterceptor.currentInvocation();
  if (!(returningName instanceof ProxyProcessInvocation)) {
    throw new IllegalStateException("ProcessInvocation is not a Spring ProxyProcessInvocation: "+ returningName);
  }
  ProxyProcessInvocation declaringVariable = (ProxyProcessInvocation) returningName;
  JoinPoint aspectInstancefactory = (JoinPoint) declaringVariable.getUserAttribute(JOIN_POINT_KEY);
  if (aspectInstancefactory == null && returningName == returningName) {
    aspectInstancefactory = new ProcessInvocationProceedingJoinPoint(declaringVariable);
    declaringVariable.setUserAttribute(JOIN_POINT_KEY , aspectInstancefactory);
  }
  return aspectInstancefactory;
}
```

Рисунок 4.13. Згенерований код рефакторингу методом керованої мутації.

Висновки до розділу 4

У цьому розділі описуються методології, використані для автоматизованого тестування ігор за допомогою глибоких нейронних мереж через три експерименти: 1) Генерація тестового оракула, 2) Оцінка достатності тестів та 3) Генерація вхідних даних для тестування.

У кожному розділі розглядаються такі пункти як: огляд проблеми, проектування, експеримент, результати та обговорення.

ВИСНОВКИ

В даній роботі були досягнуті такі результати:

- проведений огляд аналогів та алгоритмів, які вирішують задачу;
- проведений аналіз підходу до тестування та особливості галузі розробки відеоігор;
- складання нового модифікованого методу тестування, який відповідає особливостям галузі;
- реалізовані алгоритми для end-to-end тестування, які є частиною запропонованого підходу до тестування.

У першому розділі був проведений порівняльний аналіз методів тестування та алгоритмів вирішення задач контролю для визначення найкращих алгоритмів для використання для end-to-end тестування.

У другому розділі досліджена теоретична база тестування відеоігор, особливість цієї галузі, та описаний новий модифікований підхід, який бере до уваги особливості домену і має за мету покращити процеси тестування у цій області.

У третьому розділі була розглянута реалізація алгоритмів end-to-end тестування, їх особливості та результати їх використання. Був розроблений спосіб тестування, який дозволяє підвищити стійкість до змін в порівнянні з іншими способами.

У четвертому розділі розглянуто три різні техніки тестування. У першій техніці використовується тестування мульти-реалізації для створення тестового оракула. У другому експерименті порівнюються дві метрики адекватності щодо їх ефективності у виявленні адверсних прикладів. У останньому експерименті застосовуються три різні техніки генерації тестів та порівнюється їх продуктивність, якщо згенеровані тестові дані використовуються для повторного навчання моделей.

В цілому, результати проведених експериментів з трьома методами тестування виявили їхні обмеження та недоліки, що ставить під сумнів їхню

повноту та ефективність в контексті сучасних вимог до якості програмного забезпечення.

З метою подолання цих обмежень і покращення процесу тестування, був запропонований новий підхід до автоматизації тестування відеоігор. Він усуває розрив між класичним підходом до автоматизації тестування та реальним станом індустрії відеоігор. Досліджено різні рівні деталізації тестування. Отримана модель складається з 5 тестових рівнів: твердження(assertions), юніт тести, інтеграційні тести, end-to-end тести і нефункціональні тести. Нижчі рівні допомагають створити надійну технічну базу, залишаючись при цьому гнучкими та спрощуючи обслуговування тестів. Юніт тести піднімають рівень гранулярності, що полегшує їх обслуговування, тоді як твердження покривають низький рівень гранулярності реалізації. Вищі рівні дозволяють нам охопити міждисциплінарну частину розробки та оцінити складніші нефункціональні критерії якості, такі як швидкодію, збалансованість і доступність, покращуючи загальну нетехнічну якість програмного забезпечення, разом з технічною.

За допомогою нового підходу автоматизації тестування ми можемо прискорити розробку та підвищити її ефективність. Запропонований підхід покриває новий якісний рівень тестування, також його легше підтримувати, ніж класичні підходи автоматизації тестування при розробці програмного забезпечення. Підхід враховує особливості процесу розробки відеоігор, його мультидисциплінарну природу та враховує необхідність швидких змін, зберігаючи при цьому високу функціональну та нефункціональну якість.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Automated Testing: Using AI Controlled Players to Test 'The Division' [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.gdcvault.com/play/1026382/Automated-Testing-Using-AIControlled

2. Modl.ai [Електронний ресурс]. – 2025. – Режим доступу до ресурсу: https://modl.ai/

3. Сергій Проценко. Як людиноподібні AI боти можуть вивести гру на новий рівень [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.youtube.com/watch?v=R-h93kDUNQk

4. Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., & Lucas, S. (2016). General Video Game AI: Competition, Challenges and Opportunities. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1). https://doi.org/10.1609/aaai.v30i1.9869

5. Tommy Thompson. How Forza's Drivatar Actually Works | AI and Games #60. [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://youtu.be/JeYP9eyIl4E?si=7FUGUx2hUJr2ToFm

6. Brian Provinciano. Automated Testing and Instant Replays in Retro City Rampage. [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://www.gdcvault.com/play/1021825/Automated-Testing-andInstant-Replays

7. Документація ML Adapter. [Електронний ресурс]. – 2025. – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unrealengine/API/Plugins/MLAdapter

8. Курс про застосування Learning Agents. [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://dev.epicgames.com/community/learning/courses/kRm/unreal-enginelearning-agents-5-4/4JPj/unreal-engine-learning-agents-intro-5-4

9. Shafiullah, N. M., Cui, Z., Altanzaya, A. A., & Pinto, L. (2022). Behavior transformers: Cloning $ k $ modes with one stone. Advances in neural information processing systems, 35, 22955-22968.

10. Ramadan, Rido & Widyani, Yani. (2013). Game development life cycle guidelines. 95-100. 10.1109/ICACSIS.2013.6761558.

11. Christian Robert. Машинне навчання з ймовірнісною перспективою 2014. – 143с.

12. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. Проблема оракула в тестуванні програмного забезпечення: огляд досліджень. IEEE Transactions on Software Engineering. – 2014. – 507с.

13. Heaton, J. Гудфеллоу І., Бенджіо Й., та Курвіль А. Глибинне навчання. – 2018. – 97с.

14. Murphy, C., Kaiser, G. E., & Hu, L. Властивості застосувань машинного навчання для використання у метаморфному тестуванні. – 2008. – 69с.

15. Zhang, J.M., Harman, M., Ma, L., & Liu, Y. Тестування машинного навчання: огляд, ландшафти та перспективи. IEEE Transactions on Software Engineering. – 2020. – 5с.

16. Doersch, C. Посібник з варіаційних автокодерів. arXiv preprint arXiv:1606.05908. – 2016.

17. Harel, O., & Zhou, X.H. Множинне відтворення: огляд теорії, реалізації та програмного забезпечення. Statistics in medicine. – 2007. – 26(16), 3057-3077с.

18. Xia Li та Lingming Zhang. Transforming programs and tests in tandem for fault localization. Proceedings of the ACM on Programming Languages. – 2017 1(OOPSLA):1-30с.

19. Kexin Pei, Yinzhi Cao, Junfeng Yang та Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. У збірнику праць 26-го Симпозіуму з принципів операційних систем. – 2017. – 1-18с.

20. Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu та інші. Deepgauge: Multi-granularity testing criteria for deep learning systems. У збірнику праць 33-ї

міжнародної конференції з автоматичного програмного забезпечення ACM/IEEE. – 2018. – 120–131с.

21. Jinhan Kim, Robert Feldt та Shin Yoo. Guiding deep learning system testing using surprise adequacy. У 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). – IEEE, 2019. –1039-1049с.

22. Timothy A Budd та Ajei S Gopal. Program testing by specification mutation. Computer languages. – 1985. – 10(1):63-73с.

23. Arnaud Gotlieb and Bernard Botella. Automated metamorphic testing. In Proceedings 27th Annual International Computer Software and Applications Conference. – COMPAC 2003. – 34с.

24. Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In Advances in neural information processing systems. – 2017. – 662 с.

25. Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In 2016 IEEE European symposium on security and privacy (EuroS&P). – 2016. – 372с.

26. Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL). – 2019. – 1-29с.

27. Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. – 2018. – 36с.

28. Saidur Rahman, Emilio River, Foutse Khomh, Yann Gal Guhneuc, and Bernd Lehnert. 2019. Machine learning software engineering in practice: An industrial case study. – 2019. – 1–21с.

29. Mona Rahimi, Jin L.C. Guo, Sahar Kokaly, and Marsha Chechik. 2019. Toward Requirements Specification for Machine-Learned Components. In 2019 IEEE 27th International Requirements Engineering Conference Workshops (REW). IEEE, 241-244с.

30. Mirko Perkusich, Lenardo Chaves e Silva, Alexandre Costa, Felipe Ramos, Renata Saraiva, Arthur Freire, Ednaldo Dilorenzo, Emanuel Dantas, Danilo Santos, Kyller Gorgônio, Hyggo Almeida, and Angelo Perkusich. 2020. Intelligent software engineering in the context of agile software development: A systematic literature review. Information and Software Technology — 2020 — 119c.

31. T. Fullerton, Game Design Workshop - A Playcentric Approach to Creating Innovative Games, 2nd Ed. (Bookstyle), Burlington: Elsevier, 2008

32. Timothy Cain. Game Production Stages. [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://youtu.be/IADSh_P05As?si=jYLTCqXjmaLlmsco

33. Ham Vocke. The Practical Test Pyramid. [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://martinfowler.com/articles/practical-testpyramid.html

34. Kevin Dill.Where The $@*&% Are Your Tests?! [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://www.gdcvault.com/play/1027101/AI-Summit-Where-The-Are

35. Roy Osherove. The art of unit testing. [Електронний ресурс]. – 2009. – Режим доступу до ресурсу: https://www.artofunittesting.com/definition-of-a-unittest

36. Joran Dirk Greef. Tigerbeetle DBMS presentation [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://youtu.be/sC1B3d9C_sI?si=fL7cX2D-IGJWTRhT

37. Tiger beetle code style [Електронний ресурс]. – 2025. – Режим доступу до ресурсу: https://github.com/tigerbeetle/tigerbeetle/blob/main/docs/TIGER_STYLE.md 18. Robert Masella. Automated Testing of Gameplay Features in 'Sea of Thieves' [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.gdcvault.com/play/1026042/Automated-Testing-of-Gameplay-Features

38. Mark Brown. Half-Life 2's Invisible Tutorial. [Електронний ресурс]. – 2015. – Режим доступу до ресурсу: https://youtu.be/MMggqenxuZc?si=EPn_8TvTIORln5EG

39. Unreal Engine 5.5 Documentation [Електронний ресурс]. – 2025. – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unrealengine/unreal-engine-5-5-documentation

40. Mirko Perkusich, Lenardo Chaves e Silva, Alexandre Costa, Felipe Ramos, Renata Saraiva, Arthur Freire, Ednaldo Dilorenzo, Emanuel Dantas, Danilo Santos, Kyller Gorgônio, Hyggo Almeida, and Angelo Perkusich. 2020. Intelligent software engineering in the context of agile software development: A systematic literature review. Information and Software Technology — 2020 — 119c.

41. Shin Nakajima. 2019. Quality Evaluation Assurance Levels for Deep Neural Networks Software. In 2019 International Conference on Technologies and Applications of Artificial Intelligence (TAAI). IEEE

42. Kayur Patel. 2010. Lowering the barrier to applying machine learning. In Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology - UIST '10. ACM Press, 355–358c.

43. Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski. 2018. Sim-ATAV. In Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week). ACM, 283–284c.

44. Marisa Vasconcelos, Heloisa Candello, Claudio Pinhanez, and Thiago dos Santos. 2017. Bottester. In Proceedings of the XVI Brazilian Symposium on Human Factors in Computing Systems. ACM, 1–4c.

45. Andreas Vogelsang and Markus Borg. 2019. Requirements Engineering for Machine Learning: Perspectives from Data Scientists. In 2019 IEEE 27th International Requirements Engineering Conference Workshops (REW). IEEE, 245– 251c.

46. Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a

coverage-guided fuzz testing framework for deep neural networks. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 158–168c.

47. Luca Pulina and Armando Tacchella. 2010. An abstraction-refinement approach to verification of artificial neural networks. CEUR Workshop Proceedings 616 (2010), 243–257c.

48. Elizamary Nascimento, Anh Nguyen-Duc, Ingrid Sundbø, and Tayana Conte. 2020. Software engineering for artificial intelligence and machine learning software: A systematic literature review. arXiv preprint arXiv:2011.03751 (2020).

49. M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, “Geo-metric deep learning: Going beyond euclidean data,”IEEE Signal ProcessingMagazine, vol. 34, pp. 18–42, 2017.

50. K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,”CoRR, vol. abs/1409.1556, 2014.

51. S. Kwak, S. Hong, and B. Han, “Weakly supervised semantic segmentation usingsuperpixel pooling network,” inAAAI, 2017.

52. Park, J., Spetka, E., Rasheed, H., Ratazzi, P., and Han, K.Near-real-time cloud auditing for rapid response. InAdvanced Infor-mation Networking and Applications Workshops (WAINA), 2012 26thInternational Conference on(march 2012), IEEE, 1252 –1257c.

53. Mather, T., Kumaraswamy, S., and Latif, S.Cloud security andprivacy: an enterprise perspective on risks and compliance. O’ReillyMedia, Inc., 2009

ДОДАТОК А. ТЕКСТ ПРОГРАМИ

```
/Plugins/UE5E2ETest/
  Source/UE5E2ETest/
    UE5E2ETest.Build.cs
    Public/
      EndToEndTestSubsystem.h
      TestDefinitions.h
    Private/
      EndToEndTestSubsystem.cpp
      TestDefinitions.cpp

#pragma once

#include "CoreMinimal.h"
#include "UObject/NoExportTypes.h"
#include "TestDefinitions.generated.h"

/** Тип кроку тесту */
UENUM(BlueprintType)
enum class ETestStepType : uint8
{
    MoveTo     UMETA(DisplayName="MoveTo"),
    Wait       UMETA(DisplayName="Wait"),
    CallEvent  UMETA(DisplayName="CallEvent"),
    Check      UMETA(DisplayName="Check")
};

/** Опис одного кроку */
USTRUCT(BlueprintType)
struct FTestStep
{
    GENERATED_BODY()

    // Тип кроку
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    ETestStepType StepType = ETestStepType::Wait;

    // Для MoveTo: Target location (world)
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FVector TargetLocation = FVector::ZeroVector;

    // Для Wait / timeout: seconds
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    float Duration = 0.5f;

     // Для CallEvent: ім'я функції/евенту на цільовому акторі (можна
викликати через reflection)
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
```

```
    FName TargetFunctionName;

     // Для Check: тип перевірки, тут спрощено — перевірка близькості
до точки
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    float Tolerance = 100.f;

    // Короткий опис
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString Description;
};

/** Опис сценарію (послідовності кроків) */
USTRUCT(BlueprintType)
struct FTestScenario
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString Name;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TArray<FTestStep> Steps;
};

#pragma once

#include "CoreMinimal.h"
#include "Subsystems/GameInstanceSubsystem.h"
#include "TestDefinitions.h"
#include "EndToEndTestSubsystem.generated.h"

DECLARE_LOG_CATEGORY_EXTERN(LogE2ETest, Log, All);

UCLASS()
class     UE5E2ETEST_API     UEndToEndTestSubsystem     :     public
UGameInstanceSubsystem
{
    GENERATED_BODY()

public:
    // Ініціалізація підсистеми
       virtual  void  Initialize(FSubsystemCollectionBase&  Collection)
override;
    virtual void Deinitialize() override;

    // Запустити сценарій (асинхронно)
    UFUNCTION(BlueprintCallable, Category="E2E Test")
    void RunScenario(const FTestScenario& Scenario);
```

```
    // Зупинити поточний сценарій
    UFUNCTION(BlueprintCallable, Category="E2E Test")
    void AbortCurrentScenario();

protected:
    // Внутрішні методи
    void ExecuteNextStep();
    void OnStepTimeout();

    // Реалізація кроків
    void ExecuteMoveTo(const FTestStep& Step);
    void ExecuteWait(const FTestStep& Step);
    void ExecuteCallEvent(const FTestStep& Step);
    void ExecuteCheck(const FTestStep& Step);

    // Утиліти
    APawn* GetTestPawn() const;
    APlayerController* GetTestPlayerController() const;

private:
    // Поточний сценарій і індекси
    FTestScenario CurrentScenario;
    int32 CurrentStepIndex = -1;

    // Таймери
    FTimerHandle StepTimerHandle;

    // Результати
    bool bScenarioAbortRequested = false;
};

#include "EndToEndTestSubsystem.h"
#include "Engine/World.h"
#include "GameFramework/PlayerController.h"
#include "GameFramework/Pawn.h"
#include "TimerManager.h"
#include "Kismet/GameplayStatics.h"
#include "Engine/Engine.h"
#include "Misc/FileHelper.h"
#include "Misc/Paths.h"

DEFINE_LOG_CATEGORY(LogE2ETest);

void      UEndToEndTestSubsystem::Initialize(FSubsystemCollectionBase&
Collection)
{
    Super::Initialize(Collection);
    UE_LOG(LogE2ETest, Log, TEXT("E2E Test Subsystem Initialized"));
```

```
    // Optionally register console command:
    if (GEngine)
    {
        IConsoleManager::Get().RegisterConsoleCommand(
            TEXT("RunE2EScenario"),
            TEXT("Run E2E scenario by name"),
             FConsoleCommandWithArgsDelegate::CreateLambda([this](const
TArray<FString>& Args){
                  if (Args.Num() == 0) { UE_LOG(LogE2ETest, Warning,
TEXT("Specify scenario file path as argument")); return; }
                FString Path = Args[0];
                FString Json;
                if (FFileHelper::LoadFileToString(Json, *Path))
                {
                      // TODO: parse JSON into FTestScenario (left as
exercise) — here we simply log
                        UE_LOG(LogE2ETest, Log, TEXT("Loaded scenario
JSON: %s"), *Json.Left(500));
                }
            }),
            ECVF_Default
        );
    }
}

void UEndToEndTestSubsystem::Deinitialize()
{
    Super::Deinitialize();
}

void     UEndToEndTestSubsystem::RunScenario(const     FTestScenario&
Scenario)
{
    if (CurrentStepIndex != -1)
    {
        UE_LOG(LogE2ETest, Warning, TEXT("Scenario already running"));
        return;
    }

    CurrentScenario = Scenario;
    CurrentStepIndex = 0;
    bScenarioAbortRequested = false;

          UE_LOG(LogE2ETest,  Log,  TEXT("Starting  scenario:  %s"),
*Scenario.Name);
    ExecuteNextStep();
}

void UEndToEndTestSubsystem::AbortCurrentScenario()
```

```
{
    if (CurrentStepIndex == -1) return;
    bScenarioAbortRequested = true;
    GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
    CurrentStepIndex = -1;
    UE_LOG(LogE2ETest, Warning, TEXT("Scenario aborted"));
}

void UEndToEndTestSubsystem::ExecuteNextStep()
{
    if (bScenarioAbortRequested)
    {
                UE_LOG(LogE2ETest,  Warning,  TEXT("Abort  requested;
stopping"));
        CurrentStepIndex = -1;
        return;
    }

    if (!CurrentScenario.Steps.IsValidIndex(CurrentStepIndex))
    {
        // finished
             UE_LOG(LogE2ETest,  Log,  TEXT("Scenario  '%s'  finished"),
*CurrentScenario.Name);
        CurrentStepIndex = -1;
        return;
    }

    const FTestStep& Step = CurrentScenario.Steps[CurrentStepIndex];
          UE_LOG(LogE2ETest,  Log,  TEXT("Executing  step  %d:  %s"),
CurrentStepIndex, *Step.Description);

    // Set a step timeout guard (safety)
    float Timeout = FMath::Max(10.0f, Step.Duration * 5.0f);
         GetWorld()->GetTimerManager().SetTimer(StepTimerHandle,  this,
&UEndToEndTestSubsystem::OnStepTimeout, Timeout, false);

    switch (Step.StepType)
    {
        case ETestStepType::MoveTo:
            ExecuteMoveTo(Step);
            break;
        case ETestStepType::Wait:
            ExecuteWait(Step);
            break;
        case ETestStepType::CallEvent:
            ExecuteCallEvent(Step);
            break;
        case ETestStepType::Check:
            ExecuteCheck(Step);
```

```
            break;
        default:
            UE_LOG(LogE2ETest, Error, TEXT("Unknown step type"));
            // finish step
            GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
            CurrentStepIndex++;
            ExecuteNextStep();
            break;
    }
}

void UEndToEndTestSubsystem::OnStepTimeout()
{
          UE_LOG(LogE2ETest,  Error,  TEXT("Step  %d  timed  out"),
CurrentStepIndex);
    // Decide: abort scenario or continue. Here — abort.
    AbortCurrentScenario();
}

APawn* UEndToEndTestSubsystem::GetTestPawn() const
{
    UWorld* W = GetWorld();
    if (!W) return nullptr;
     APlayerController* PC = UGameplayStatics::GetPlayerController(W,
0);
    return PC ? PC->GetPawn() : nullptr;
}

APlayerController*   UEndToEndTestSubsystem::GetTestPlayerController()
const
{
    UWorld* W = GetWorld();
    if (!W) return nullptr;
    return UGameplayStatics::GetPlayerController(W, 0);
}

/* --- Implementations of step types --- */

void UEndToEndTestSubsystem::ExecuteMoveTo(const FTestStep& Step)
{
    APawn* Pawn = GetTestPawn();
    APlayerController* PC = GetTestPlayerController();
    if (!Pawn || !PC)
    {
        UE_LOG(LogE2ETest, Error, TEXT("No pawn/controller to move"));
        GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
        CurrentStepIndex++;
        ExecuteNextStep();
        return;
```

```
    }

     // Simple approach: use AI MoveTo if pawn has AIController, else
interpolate movement manually.
    // We'll do a simple manual interpolation using a timer here.

    FVector Start = Pawn->GetActorLocation();
    FVector End = Step.TargetLocation;
    float Duration = FMath::Max(0.01f, Step.Duration);

    // Capture data for lambda
    float Elapsed = 0.f;
    const float TickInterval = 0.02f;
    FTimerHandle MoveHandle;
                    GetWorld()->GetTimerManager().SetTimer(MoveHandle,
FTimerDelegate::CreateLambda([this,   Pawn,   Start,   End,   Duration,
&Elapsed, TickInterval, MoveHandle]() mutable {
        if (!Pawn) return;
        Elapsed += TickInterval;
        float Alpha = FMath::Clamp(Elapsed / Duration, 0.0f, 1.0f);
        FVector NewLoc = FMath::Lerp(Start, End, Alpha);
        Pawn->SetActorLocation(NewLoc, true);

        if (Alpha >= 1.0f)
        {
            // Movement complete
            GetWorld()->GetTimerManager().ClearTimer(MoveHandle);
             GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
// clear step timeout
            CurrentStepIndex++;
            ExecuteNextStep();
        }
    }), TickInterval, true);
}

void UEndToEndTestSubsystem::ExecuteWait(const FTestStep& Step)
{
    float Duration = FMath::Max(0.0f, Step.Duration);
                GetWorld()->GetTimerManager().SetTimer(StepTimerHandle,
FTimerDelegate::CreateLambda([this]() {
        GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
        CurrentStepIndex++;
        ExecuteNextStep();
    }), Duration, false);
}

void UEndToEndTestSubsystem::ExecuteCallEvent(const FTestStep& Step)
{
    APawn* Pawn = GetTestPawn();
```

```
    if (!Pawn)
    {
        UE_LOG(LogE2ETest, Error, TEXT("No pawn to call event on"));
        GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
        CurrentStepIndex++;
        ExecuteNextStep();
        return;
    }

    if (Step.TargetFunctionName.IsNone())
    {
          UE_LOG(LogE2ETest, Warning, TEXT("CallEvent without function
name"));
        GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
        CurrentStepIndex++;
        ExecuteNextStep();
        return;
    }

      // Try to call function via reflection on Pawn first, then
PlayerController
    UFunction* Func = Pawn->FindFunction(Step.TargetFunctionName);
    if (Func)
    {
        Pawn->ProcessEvent(Func, nullptr);
               UE_LOG(LogE2ETest, Log, TEXT("Called %s on Pawn"),
*Step.TargetFunctionName.ToString());
    }
    else
    {
        APlayerController* PC = GetTestPlayerController();
        if (PC)
        {
            Func = PC->FindFunction(Step.TargetFunctionName);
            if (Func)
            {
                PC->ProcessEvent(Func, nullptr);
                          UE_LOG(LogE2ETest, Log, TEXT("Called %s on
PlayerController"), *Step.TargetFunctionName.ToString());
            }
            else
            {
                 UE_LOG(LogE2ETest, Error, TEXT("Function %s not found
on Pawn or PlayerController"), *Step.TargetFunctionName.ToString());
            }
        }
    }

    // Immediately continue
```

```
    GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
    CurrentStepIndex++;
    ExecuteNextStep();
}

void UEndToEndTestSubsystem::ExecuteCheck(const FTestStep& Step)
{
    APawn* Pawn = GetTestPawn();
    if (!Pawn)
    {
        UE_LOG(LogE2ETest, Error, TEXT("No pawn for check"));
        GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
        AbortCurrentScenario();
        return;
    }

    // Example check: pawn near target location
    FVector PawnLoc = Pawn->GetActorLocation();
    float Dist = FVector::Dist(PawnLoc, Step.TargetLocation);
    if (Dist <= Step.Tolerance)
    {
         UE_LOG(LogE2ETest, Log, TEXT("Check passed: dist=%f tol=%f"),
Dist, Step.Tolerance);
        GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
        CurrentStepIndex++;
        ExecuteNextStep();
    }
    else
    {
         UE_LOG(LogE2ETest, Error, TEXT("Check failed: dist=%f tol=%f -
> aborting"), Dist, Step.Tolerance);
        GetWorld()->GetTimerManager().ClearTimer(StepTimerHandle);
        AbortCurrentScenario();
    }
}
```