ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

КВАЛІФІКАЦІЙНА РОБОТА

Тема: «Комп’ютерна гра «Endless» з використанням генеративних алгоритмів та інтелектуальною моделлю поведінки неігрових персонажів»

Ступінь вищої освіти – бакалавр
Спеціальність – 122 «Комп’ютерні науки»
Освітня програма «Комп’ютерні науки»

ПОЯСНЮВАЛЬНА ЗАПИСКА

Виконав (-ла): здобувач (ка) 4 курсу
групи КН-21
Руслан ШМАЛЬ

Керівник: к.е.н., с.н.с, доцент, зав
кафедри комп’ютерних
наук
Сергій МІЧКІВСЬКИЙ

Засвідчую, що кваліфікаційна робота оформлена відповідно до ДСТУ 3008:2015 та не містить запозичень з праць інших авторів без відповідних посилань.
Здобувач: ___________
(підпис)

м. Київ – 2025 рік

ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«УНІВЕРСИТЕТ ЕКОНОМІКИ ТА ПРАВА «КРОК»

ЗАТВЕРДЖУЮ:
завідувач кафедри
комп’ютерних наук
Сергій МІЧКІВСЬКИЙ
«_____»_____20___р

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
Шмаль Руслан Сергійович

Тема роботи	Комп’ютерна гра «Endless» з використанням генеративних алгоритмів та інтелектуальною моделлю поведінки неігрових персонажів
Номер та дата наказу про затвердження теми	№121-7 від 24 грудня 2024 року
Коротка постановка завдання	Розробити комп’ютерну гру в жанрі шутер з використанням алгоритмів генерації залів-рівнів та інтелектуальною моделлю поведінки неігрових персонажів.
Посилання на джерела інформації (не більше п’яти найменувань, які рекомендує науковий керівник)	1. Міллінгтон Іан Artificial Intelligence for Games // Taylor & Francis eBooks – URL: https://doi.org/10.1201/9781315375229 (дата звернення: 20.12.2024) 2. Шелл Джесс The Art of Game Design: A Book of Lenses // Taylor & Francis eBooks – URL: https://doi.org/10.1201/b22101 (дата звернення: 20.12.2024)
Вимоги до кваліфікаційної роботи	Кваліфікаційна робота має передбачити теоретичне, системотехнічне або експериментальне дослідження складного спеціалізованого завдання або практичної проблеми в галузі комп’ютерних наук, яке характеризується комплексністю та невизначеністю умов і потребує застосування теорій і методів інформаційних технологій.

Дата видачі завдання 27 грудня 2024 р.

Керівник ___________Сергій МІЧКІВСЬКИЙ

Здобувач освітнього ступеня бакалавра ___________Руслан ШМАЛЬ

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання	Примітка
	Підготовчий етап		
1	Вибір напряму дослідження	02.12.2024 р.	*виконано*
2	Формування теми та призначення керівника	16.12.2024 р.	*виконано*
3	Затвердження теми кваліфікаційної роботи	23.12.2024 р.	*виконано*
4	Затвердження завдання на кваліфікаційну роботу	27.12.2024 р.	*виконано*
	Основний етап		
5	Розробка концепції кваліфікаційної роботи	13.01.2025 р.	*виконано*
6	Підбір та вивчення джерел інформації з напряму дослідження. Огляд існуючих аналогів	20.01.2025 р.	*виконано*
7	Затвердження розширеної постановки завдання. Підготовка та подання керівникові розділу 1 кваліфікаційної роботи	10.03.2025 р.	*виконано*
8	Проєктування. Підготовка та подання керівникові розділу 2 кваліфікаційної роботи	24.03.2025 р.	*виконано*
9	Підготовка доповіді для експертизи стану виконання кваліфікаційної роботи (проміжний контроль)	31.03-04.04.2025 р.	*виконано*
10	Реалізація. Підготовка та подання керівникові розділу 3 кваліфікаційної роботи	07.04.2025 р.	*виконано*
11	Підготовка та подання керівнику першого варіанту всієї кваліфікаційної роботи	14.04.2025 р.	*виконано*
12	Доопрацювання кваліфікаційної роботи з урахуванням зауважень керівника та представлення керівникові доопрацьованого варіанту кваліфікаційної роботи	21.04.2025 р.	*виконано*
	Завершальний етап		
13	Представлення рукопису для перевірки на плагіат	28.04-04.05.2025 р.	*виконано*
14	Підготовка презентації та доповіді на передзахист	05.05-11.05.2025 р.	*виконано*
15	Передзахист кваліфікаційної роботи	12.05-16.05.2025 р.	*виконано*
16	Доопрацювання роботи за результатами передзахисту	19.05-06.06.2025 р.	*виконано*
17	Експертиза роботи керівником та зовнішнім експертом	09.06-15.06.2025 р.	*виконано*
18	Доопрацювання доповіді та презентації для захисту	09.06-15.06.2025 р.	*виконано*
19	Захист кваліфікаційної роботи	16.06-22.06.2025 р.	*виконано*

Керівник Сергій МІЧКІВСЬКИЙ

Здобувач освітнього ступеня бакалавра Руслан ШМАЛЬ

Шмаль Р. С. Комп'ютерна гра «Endless» з використанням алгоритмів генерації залів-рівнів та інтелектуальною моделлю поведінки неігрових персонажів (ботів).

Пояснювальна записка кваліфікаційної роботи за спеціальністю 122 – Комп’ютерні науки (освітня програма - Комп’ютерні науки) СО Бакалавр. – ВНЗ «Університет економіки та права «КРОК», навчально-науковий інститут інформаційних та комунікаційних технологій, кафедра комп’ютерних наук, Київ, 2025.

У даній роботі розглянуто проблеми індустрії комп’ютерних ігор на прикладі розробки гри «Endless». Розроблено комп'ютерну гру «Endless» з використанням алгоритмів генерації залів-рівнів та інтелектуальною моделлю поведінки неігрових персонажів (ботів).

Ключові слова: комп’ютерна гра, шутер, алгоритми генерації, поведінка неігрових персонажів.

Рис. 33. Бібліограф. 42.

Shmal R. S. Computer game «Endless» with the use of algorithms for generating hall-levels and an intelligent model of non-playable characters (bots) behaviour.

Explanatory note of the qualification work in the specialty 122 – Computer Science (educational program - Computer Science) Bachelor's degree. – «KROK» University, Educational and Scientific Institute of Information and Communication Technologies, Department of Computer Science, Kyiv, 2025.

This work considers the problems of the computer game industry on the example of the development of the game “Endless”. The computer game “Endless” was developed using algorithms for generating level halls and an intelligent model of the non-playable characters (bots) behaviour.

Keywords: Computer game, Shooter, generating algorithms, intelligent non-playable characters behaviour.

Fig. 33. Bibliography. 42

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ 6

ВСТУП 7

РОЗДІЛ 1 ПОСТАНОВКА ЗАВДАННЯ З РОЗРОБКИ КОМП'ЮТЕРНОЇ ГРИ "ENDLESS" 9

1.1 Опис та аналіз предметної області 9

1.2 Огляд аналогів 11

1.3 Постановка завдання на кваліфікаційну роботу 18

Висновки до розділу 20

РОЗДІЛ 2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ 21

2.1 Моделювання поведінки продукту 21

2.2 Моделювання структури продукту 26

2.3 Опис архітектури продукту 30

Висновки до розділу 44

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ 46

3.1 Реалізація та констуювання програмного продукту 46

3.2 Тестування програмного продукту 57

3.3 Використання програмного продукту 59

Висновки до розділу 61

ВИСНОВКИ 62

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 63

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

Unity (Unity3d) – ігровий рушій, що лежить в основі цього проекту [1].

C# (C-sharp) – основна мова програмування, що використовується в даному проекті [7].

Config [34] – конфігураційний файл або об'єкт.

SO (ScriptableObject) [35] – це об'єкт, що використовується для конфігурації статичних даних і використовується для створення конфігів.

UI (User interface) [32] – термін, що означає користувацький інтерфейс.

DI (Dependencies injection) [31] – методика, що використовується в архітектурі застосунку для контролю та впровадження залежностей до сутностей.

BT (Behavior Tree) [28] – патерн проектування, що означає древо поведінки, що є підходом до реалізації комплексної поведінки персонажів або інших сутностей, поведінку яких можна описати розгалуженою структурою.

FSM (Finite state machine) [27] – патерн проектування, що описує стан сутності кінцевими станами з можливістю переходу з одного стану в інший.

Object Pool [30] – патерн проєктування, що використовується з метою оптимізації застосунку шляхом зменшення частоти створення і знищення об'єктів, які часто зустрічаються.

Movement [37] – термін, що означає модель пересування персонажа.

NPC (Non-playable character) [39] – неігровий персонаж (бот), що контролюється поведінковими алгоритмами (BT).

GA (Generation algorithms) [33] – термін, що означає алгоритми процедурної генерації, що вказують на генерацію в процесі роботи програми.

Soundtrack (саундтрек) [38] – термін, що означає музичний супровід.

Gameplay (геймплей) [36] – термін, що означає ігровий процес.

ВСТУП

Актуальність теми. Останні десятиліття комп'ютерні ігри є невід'ємною частиною дозвілля для безлічі людей із різними вподобаннями та віковими категоріями, серед яких є гравці, що бажають випробувати свої здібності в іграх, які є вимогливими до навичок, але натомість пропонують цікаві виклики та різноманітний геймплей, що особливо актуально для проектів з унікальним набором відносно глибоко продуманих ігрових механік.

Проблемна ситуація. Незважаючи на великий вибір різноманітних існуючих продуктів, ігрова індустрія є перспективною та такою, що швидко розвивається і готова до нових інтерпретацій і комбінацій існуючих жанрів, що дає змогу створювати нові ніші та задовольняти попит на них.

Мета дослідження є розробка комп'ютерної гри «Endless» з використанням генеративних алгоритмів та інтелектуальною моделлю поведінки неігрових персонажів (ботів).

Завдання дослідження:

проаналізувати сучасні тенденції в розробці комп’ютерних ігор та наявні проекти зі схожим жанром, позначити їхні сильні та слабкі сторони;

визначені вимоги до програмного забезпечення системи та автоматизовані функції;

розглянуто алгоритми та підходи до генерації локацій гри та поведінки неігрових персонажів;

спроектовано архітектуру програмного забезпечення гри «Endless»;

розробити програмне забезпечення гри «Endless», реалізувавши всі аспекти гри згідно з планом.

Об'єктом дослідження є ігровий процес управління персонажем та поведінка неігрових персонажів, процеси генерації ігрових локацій.

Предметом дослідження є комп'ютерна гра з використанням генеративних алгоритмів та інтелектуальною моделлю поведінки неігрових персонажів(ботів).

Методи дослідження. Дослідження проводилося з використанням різних наукових та дослідницьких методів, як аналітичні методи дослідження, порівняльний аналіз. Аналітичні методи були використані для вивчення різних технологій та інструментів для розробки ігор.

Практичне значення. Розроблена комп'ютерна гра «Endless» з використанням генеративних алгоритмів та інтелектуальною моделлю поведінки неігрових персонажів (ботів), що дає змогу урізноманітнити відпочинок, пропонуючи гравцям виклик для перевірки власних навичок.

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та списку посилань (42 найменувань). Пояснювальна записка містить 33 рисунків. Загальний обсяг пояснювальної записки складає 66 сторінок, основний зміст викладено на 62 сторінках.

РОЗДІЛ 1
ПОСТАНОВКА ЗАВДАННЯ З РОЗРОБКИ КОМП'ЮТЕРНОЇ ГРИ «ENDLESS»

1.1 Опис та аналіз предметної області

Основною метою комп'ютерних ігор є розвага гравця, однак це жодним чином не перешкоджає можливості навчання в процесі гри або тренуванню певних навичок – усе залежить від конкретного жанру або гібриду з кількох, що може запропонувати незвичайний досвід, що містить у собі найкраще одразу з кількох напрямів, а для кращого розуміння жанрів. Розглянемо кілька основних жанрів.

Пригодницькі ігри (Adventures) – надають перевагу історії, дослідженню, розв'язанню головоломок та управлінню запасами, а не екшену. Гравці розкривають сюжет у власному темпі, досліджуючи ігрові світи. Головоломки перевіряють логіку гравців та їхні навички управління обмеженими ресурсами [40].

Платформери (Platformers) – це 2D-ігри з бічною прокруткою, які змушують гравців бігати, стрибати та лазити небезпечними трасами. Ворогів та пастки потрібно долати з точним розрахунком часу та спритністю. Платформери домінували в ранньому геймінгу і залишаються популярними донині [40].

Стратегічні ігри (Strategy) спонукають до ретельного планування, управління ресурсами та прийнятті важливих рішень, а не на спритності чи рефлексах. Вони заохочують аналіз, творче вирішення проблем і довгострокове стратегічне бачення [40].

Екшен-ігри (Action) перевіряють і винагороджують рефлекси гравців, моторну координацію та швидкість реакції. Ігровий процес розвивається в швидкому темпі, з акцентом на боях, вибухах та епічних моментах. До популярних піджанрів належать шутери, файтинги, бродилки тощо [40].

Рольові ігри (RPG) дозволяють гравцям створювати персонажів з унікальними навичками, предметами та здібностями. Насичені історії розгортаються через дослідження, квести та побічні дії. Битви поєднують стратегічне управління ходом поєдинку з екшеном й вимагають дотримуватись тактики для підвищення ефективності [40].

Шутерами (Shooter) називають будь-яку гру, в якій стрілецька зброя займає центральне місце. На відміну від точних FPS-ігор, шутер може застосовуватися до різного геймплею та стилів ігрового мистецтва, в яких присутня зброя [40].

Це декілька основних жанрів, кожен із котрих має немало піджанрів та може комбінуватись з іншими, що дозволяє створювати дійсно унікальний ігровий досвід. Даний проект можна описати гібридом із кількох жанрів, серед яких:

шутер – у грі є різні типи доступної для гравця зброї;

екшен – гра є динамічною та досить вимоглива до навичок гравця і винагороджує за ефективне проходження, вміння швидко реагувати на динамічну зміну ситуації, глибоке розуміння ігрових механік;

RPG (Рольова складова) – у грі є поліпшення для персонажа, унікальні предмети, деяка варіативність проходження.

Що стосується монетизації, існують різні підходи та їхні варіації, що дають змогу підібрати найбільш підходящі канали доходу від проекту, відштовхуючись від специфіки гри. Таким чином, можна виділити кілька основних підходів до монетизації.

Повна купівля гри – зазвичай передбачає одноразове придбання користувачем гри, після чого він назавжди має доступ до неї. Цей підхід, зазвичай, може поєднуватися з іншими і нерідко застосовується до масштабних ігор, що відразу містять велику кількість контенту [41].

Мікротранзакції – купівля предметів або валюти всередині гри, найчастіше зустрічаємо у free-to-play проектах [42].

Додатковий контент або розширення (DLC) – одноразова плата за певне розширення або додатковий контент, нерідко має на увазі якісні доповнення для гри, що пропонують розширення ігрового досвіду і чимало нового контенту [41].

Підписка – спосіб монетизації, у межах якого користувач купує тимчасовий доступ до розширених можливостей або додаткового контенту. Найчастіше цей підхід зустрічається у вигляді варіації «Бойовий пропуск», що дає можливість в обмежений період отримати унікальні предмети за виконання сезонних завдань [41].

Як основний тип монетизації, у даному проекті використовується повна купівля гри з безкоштовною підтримкою (вихід усіх загальних оновлень) та подальшою можливістю купівлі платних розширень (DLC).

Цільовими платформами для дистрибуції є Steam [5] і Epic Games [6], оскільки ці платформи найбільш популярні та пропонують механізми рекомендацій продуктів для користувачів і вирізняються корисними інструментами для розробників, за допомогою яких можна аналізувати відгуки, охоплення, продажі та інші важливі показники, що відносяться до проекту й відображають його поточний стан і конкурентоспроможність.

1.2 Огляд аналогів

Для створення більш унікального продукту проаналізуємо існуючі аналоги, щоб розуміти сильні та слабкі сторони, потенційні поліпшення та інші можливі коригування проекту.

ULTRAKILL (рис. 1.1) – це динамічний ретро-шутер із глибокою та якісною реалізацією ігрових механік, незвичайним саундтреком, що запам'ятовується, стилістично відсилає до першовідкривачів жанру, таких як Quake [3].

Що стосується геймплею, в ULTRAKILL він дуже різноманітний: гра, в основному, розбита на рівні, а після завершення рівня, гравцеві виводяться

його результати, де враховується час проходження, виконання додаткового завдання і збір секретних артефактів.

Рисунок 1.1 – Прев'ю гри ULTRAKILL

Джерело: [2]

У гравців є широкий вибір зброї, є можливість обирати додаткові предмети, що впливатимуть на проходження (наприклад, гак-кішка, за допомогою якого гравець зможе притягувати до себе легких ворогів або самому притягуватися до важких ворогів). Також є поняття босів, з якими доведеться битися наприкінці глави, кожен з яких унікально веде бій з гравцем, використовуючи різні прийоми і хитрощі. Також слід відзначити чудовий левел-дизайн, де кожна з глав не тільки може запропонувати унікального боса і геймплейні особливості, а й унікальний стиль локацій: від лабіринтів до великих відкритих просторів або багатошарових будівель.

Переваги:

- висока динаміка;
- зброя чуйна і передбачувана;

- широкий вибір стилю гри;
- якісний левел-дизайн;
- унікальний саундтрек;
- наявність секретних завдань.

Рисунок 1.2 – Приклад геймплею ULTRAKILL

Джерело: [2]

Недоліки:

- немає холодної зброї (melee);
- модель пересування персонажа досить легка для розуміння, але відносно поверхнева;
- іноді може здаватися занадто монотонним;
- може бути незрозумілим для не знайомих з подібним жанром;
- візуальний стиль дійсно унікальний, але підходить не всім.

Рисунок 1.3 – Приклад геймплею ULTRAKILL

Джерело: [2]

Загалом, ULTRAKILL є дуже якісним продуктом, однак ніша, наразі, не зайнята великою кількістю подібних ігор і, що важливо, «Endless», порівняно з іншими проектами, хоч і є найближчим до ULTRAKILL, але не є прямим аналогом, що додатково зменшує шанс бути в тіні більшого проекту, адже напрямок Movement-Shooter тільки починає знаходити свою аудиторію.

Doom Eternal (рис. 1.4) – культовий динамічний шутер від першої особи, що пропонує варіативний геймплей, є найбільш впізнаваним представником платформера і пригодницького жанрів, має вкрай якісний саундтректа якісно витриману похмуру стилістику (рис. 1.5).

Проходження полягає в подоланні великої кількості ворогів, битві з різноманітними босами (супротивники підвищеної складності) у міру просування сюжетом, відкритті озброєння. Локації є коридорними, але присутні альтернативні шляхи або скорочення, а арени для боїв із босами

завжди наділені унікальними об'єктами, які можуть використовуватися гравцем для спрощення або урізноманітнення бою.

Рисунок 1.4 – Логотип гри Doom Eternal

Джерело: [4]

Переваги:

- висока динаміка;
- якісний левел-дизайн;
- різноманітні ворожі неігрові персонажі;
- якісний саундтрек;
- цікавий сюжет;
- широкий вибір зброї.

Недоліки:

- ігровий процес може бути складним і заплутаним для нових гравців;
- загальна стилістика є вкрай похмурою, у грі присутня велика кількість жорстокості, що може не підійти деяким гравцям;
- прямолінійність локацій.

В цілому, Doom Eternal є дуже якісним динамічним шутером від першої особи, пропонує велику кількість цікавих локацій і різноманітних супротивників, однак не є прямим конкурентом розроблюваному продукту, оскільки попри схожість жанрів, підходи до реалізації та коренева основа геймплею дуже відрізняються.

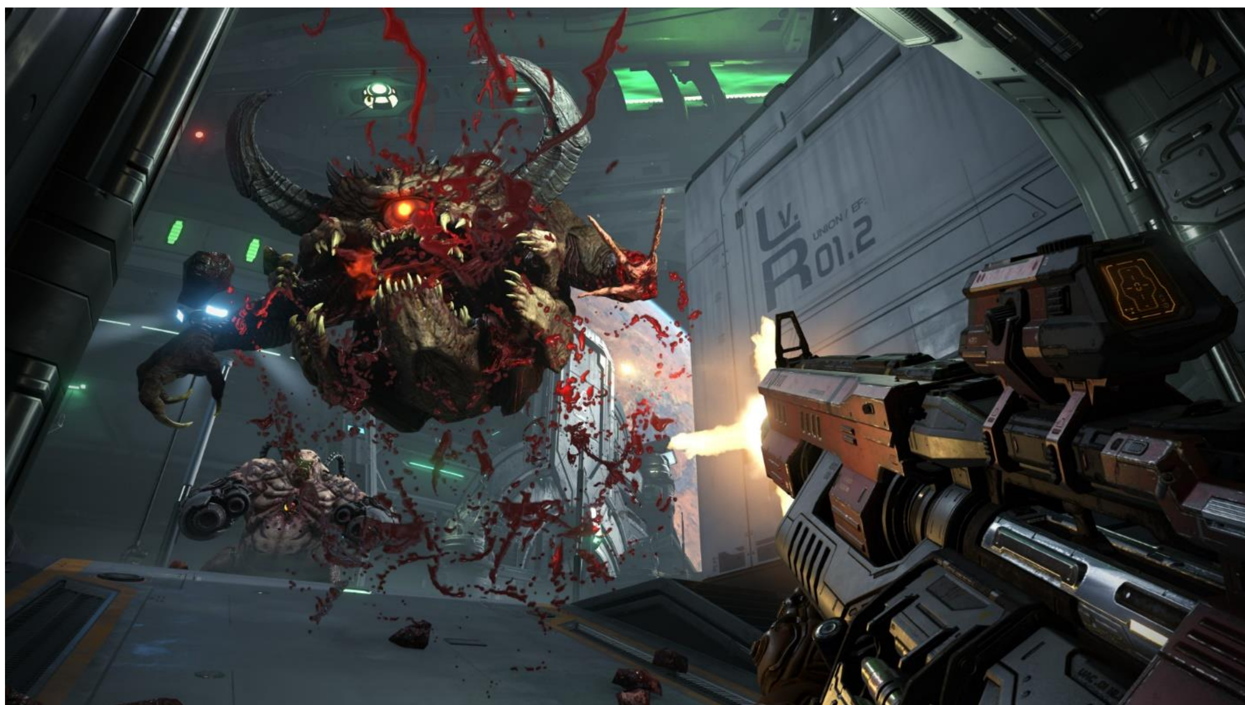

Рисунок 1.5 – Приклад геймплею Doom Eternal

Джерело: [4]

BLOW-UP: AVENGE HUMANITY (рис. 1.6) – шутер від першої особи, де гравцеві доведеться боротися з інопланетною расою, граючи за промислового робота. Гра складається з рівнів, локації представляють собою досить прості переважно коридорні ділянки, що наповнені різними ворогами (рис. 1.7). Гравець може використовувати оточення для ефективного пересування та ухилення від ворожих атак, а за швидке проходження з виконанням усіх побічних завдань (челенджів), передбачено отримання

додаткової винагороди та можливість змагатися з іншими гравцями за місце в таблиці лідерів.

Рисунок 1.6 – Прев'ю гри BLOW-UP: AVENGE HUMANITY

Джерело: [23]

Переваги:

- якісна модель пересування персонажа, наявність прийомів для швидшого проходження;
- унікальна зброя;
- велика кількість різних супротивників.

Недоліки:

- невизначений візуальний стиль;
- простий поведінковий інтелект неігрових персонажів;
- простота і прямолінійність локацій;
- монотонність саундтреку та рівнів.

Загалом, BLOW-UP: AVENGE HUMANITY містить динамічний геймплей (рис. 1.7) та чимало рідкісних і унікальних механік, але не позбавлений поверхнево реалізованих аспектів. Розглянутий продукт є вельми

близьким за загальною ідеєю до продукту, що розробляється але стилістично подібностей практично немає, через що потенційна аудиторія абсолютно різна.

Рисунок 1.7 – Приклад геймплею BLOW-UP: AVENGE HUMANITY

Джерело: [23]

1.3 Постановка завдання на кваліфікаційну роботу

Основним завданням кваліфікаційної роботи є розробка комп'ютерної гри «Endless», що належить до кількох жанрів, поєднуючи шутер, екшен і рольову складову (RPG), пропонує просунуту модель пересування персонажа, алгоритми генерації залів-рівнів та інтелектуальну модель поведінки неігрових персонажів (NPC). Також до обов'язкового функціоналу належить реалізація в грі зброї, системи взаємодії з об'єктами, система вибору рівнів.

Ігровий процес має містити можливість вибору між заздалегідь створеними і згенерованими рівнями, на кожному з яких присутні ворожі неігрові персонажі, зброя для бою, ресурси для збору, розхідні предмети

(аптечки для відновлення здоров'я тощо). У разі програшу, гравцеві пропонується спробувати знову або вийти до головного меню, його прогрес (зібрані ресурси та предмети тощо) при цьому зберігаються. У разі успішного завершення рівня, пропонується продовжити проходження, перейшовши до наступного або вийти до головного меню.

Функціональні вимоги:

- користувацький інтерфейс;
- генерація ігрових рівнів;
- інтелектуальна поведінка неігрових персонажів;
- система інвентарю, можливість взаємодіяти з об'єктами, збирати ресурси тощо;
- система збережень.

Нефункціональні вимоги:

- стабільна робота застосунку – якісна оптимізація та висока кількість кадрів в секунду;
- якісний левел-дизайн;
- зрозуміле та передбачуване управління;
- інтуїтивний інтерфейс.

Автоматизовані функції: збереження прогресу, спавн неігрових персонажів і предметів, обробка та інтерпретація введення, шкала здоров'я, анімації персонажів і предметів.

Вхідні дані. Налаштування користувачем гри, дії гравця (виконання додаткових завдань, збір артефактів), обраний користувачем рівень та його параметри.

Вихідні дані. Оцінка ефективності проходження, відображення даних через інтерфейс (меню, інтерфейс у процесі гри, що демонструє здоров'я, ресурси тощо), нарахування ігрової валюти гравцеві, прогрес персонажа.

Висновки до розділу 1

Розглянуто та проаналізовано сучасну ігрову індустрію загалом, основні жанри та їхній вплив або відношення до продукту, що розробляється.

Було проведено аналіз схожих наявних продуктів для ширшого уявлення про напрямок і розуміння сильних і слабких сторін інших проєктів, що скоригувало окремі складові «Endless».

Визначено основне завдання, функціональні та нефункціональні вимоги, автоматизовані функції, вхідні та вихідні дані та їхні складові.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Моделювання поведінки продукту

Для відображення поведінки продукту в загальному вигляді, слід розглянути діаграму варіантів використання (Use Case Diagram) на рис 2.1.

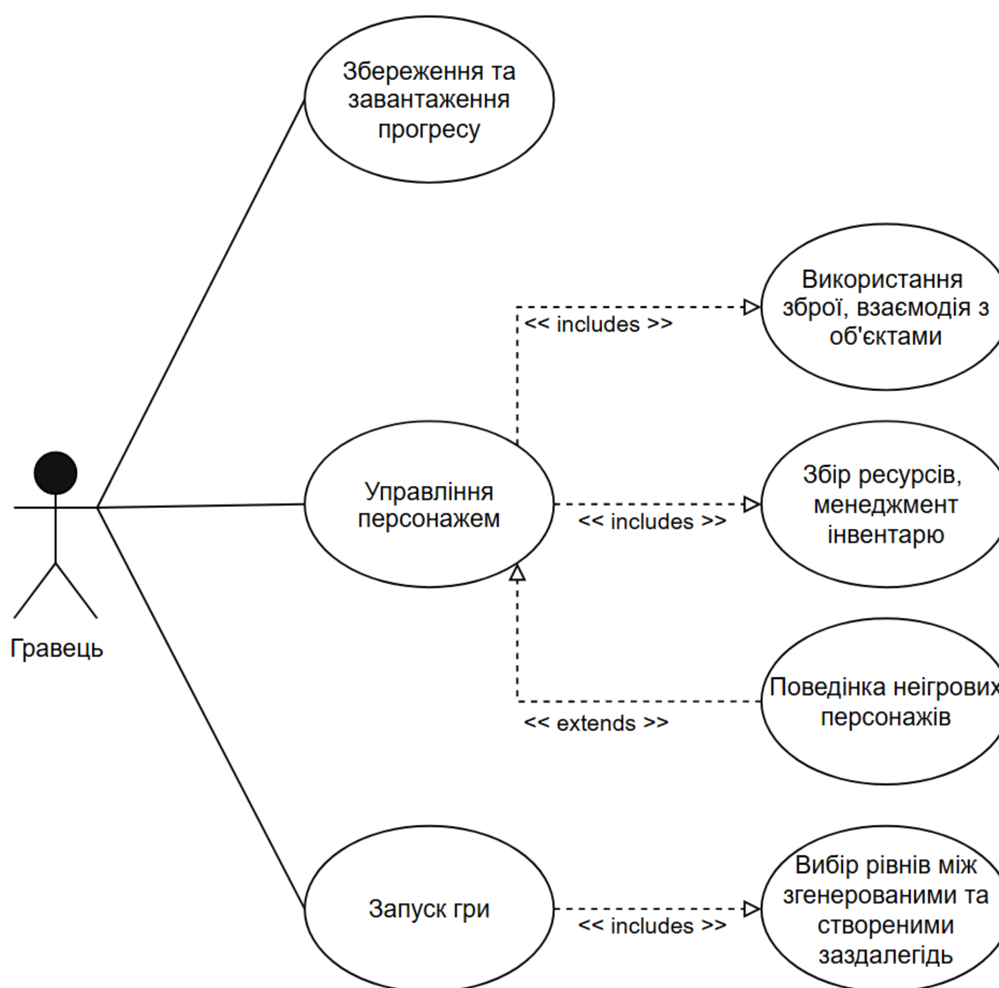

Рисунок 2.1 – Діаграма варіантів використання (Use Case Diagram)

Джерело: розроблено автором

Пояснення діаграми варіантів використання.

- актором (actor) є гравець, взаємодія якого з системою розглядатиметься;

- представлені функціональні частини (Use Case), що описують у загальному вигляді конкретну частину системи та її функціонал, що є видимим для користувача;
- “<< extends >>” зв'язок вказує на другорядну, опціональну дію, у той час як “<< includes >>” вказує на додаткову дію, що завжди відбувається з тією, від якої вона залежна.

Після запуску програми (гри), гравець, після завантаження, потрапляє до головного меню, звідки може виконати такі дії:

- відкрити вікно налаштувань – призведе до появи вікна з можливістю вибору мови, коригування чутливості миші тощо;
- вийти з гри – додаток збереже прогрес і закриється;
- почати вибір рівня для проходження (заздалегідь створений рівень або згенерований) – після вибору розпочнеться ігровий процес (геймплей).

При переході до геймплею, список можливих дій такий:

- проходження рівня – гравець досліджує локації, бореться з ворогами, збирає цінні предмети, взаємодіє з об'єктами;
- повернення в головне меню;
- відкриття налаштувань гри;
- програш – персонаж гравця вмирає або знищує необхідний для проходження предмет, після чого може повернутися до головного меню або спробувати пройти ще раз, перезапустивши рівень.
- виграш – гравець успішно проходить рівень, після чого може перейти в наступний або повернутися до головного меню.

Розглянута діаграма відображає поведінку застосунку та його реакцію на основні дії користувача (гравця), але для наочного відображення «потоку дій/управління», діаграму варіантів використання варто супроводити іншими, тому також представлено діаграму діяльності, що демонструє, куди ведуть певні рішення, прийняті безпосередньо гравцем (рис. 2.2).

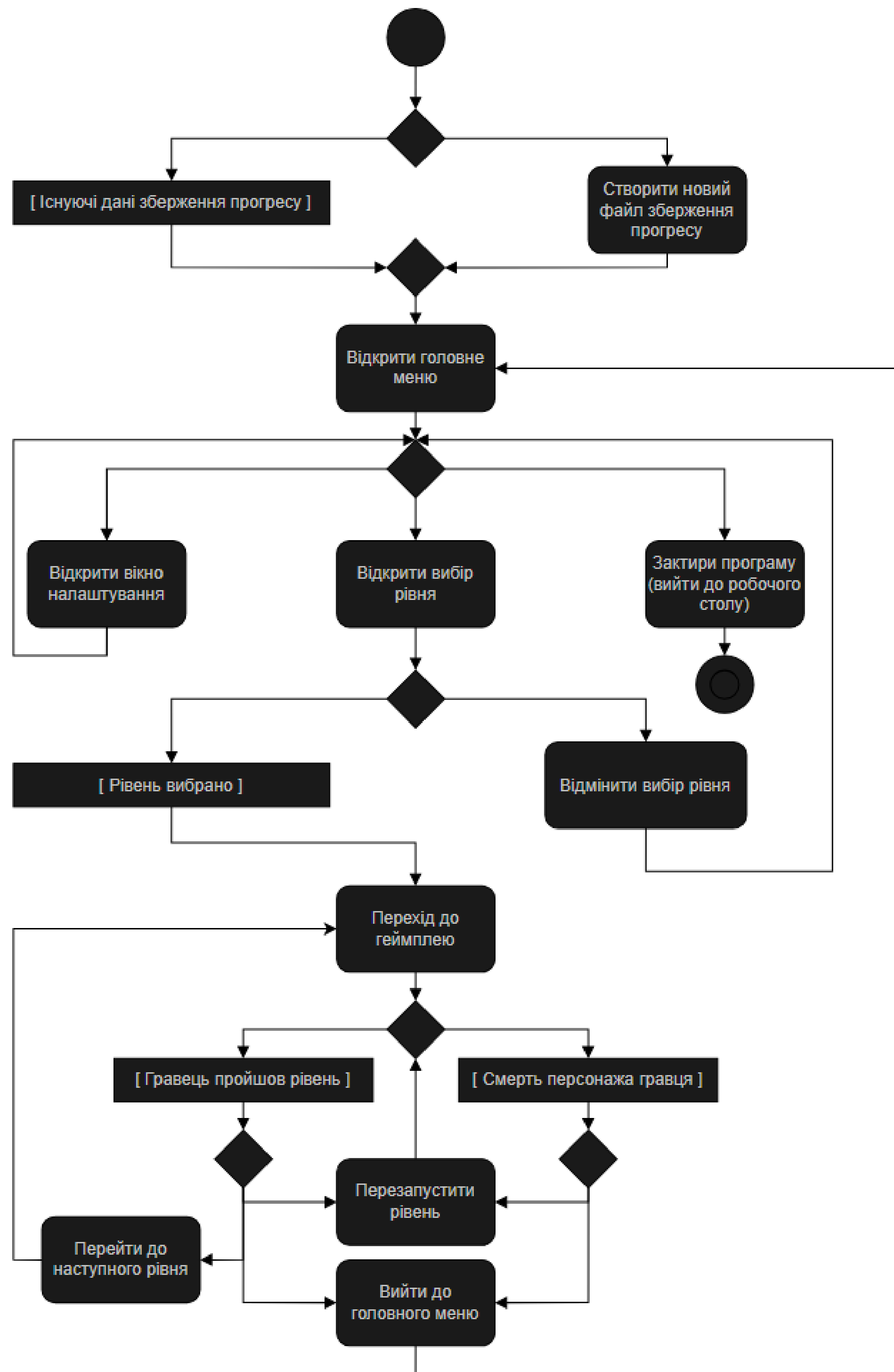

Рисунок 2.2 – Діаграма діяльності (Activity Diagram)

Джерело: розроблено автором

Пояснення діаграми діяльності. Діаграма доповнює попередню, демонструючи вплив рішень, прийнятих гравцем, і те, до чого набір певних рішень веде.

Представлено діаграму послідовності (Sequence Diagram) (рис. 2.2), що демонструє послідовну взаємодію між основними сутностями, показуючи це як повідомлення, яке послідовно проходить через усі сутності, які беруть участь у його обробці (послідовність виклику будується залежно від конкретного повідомлення).

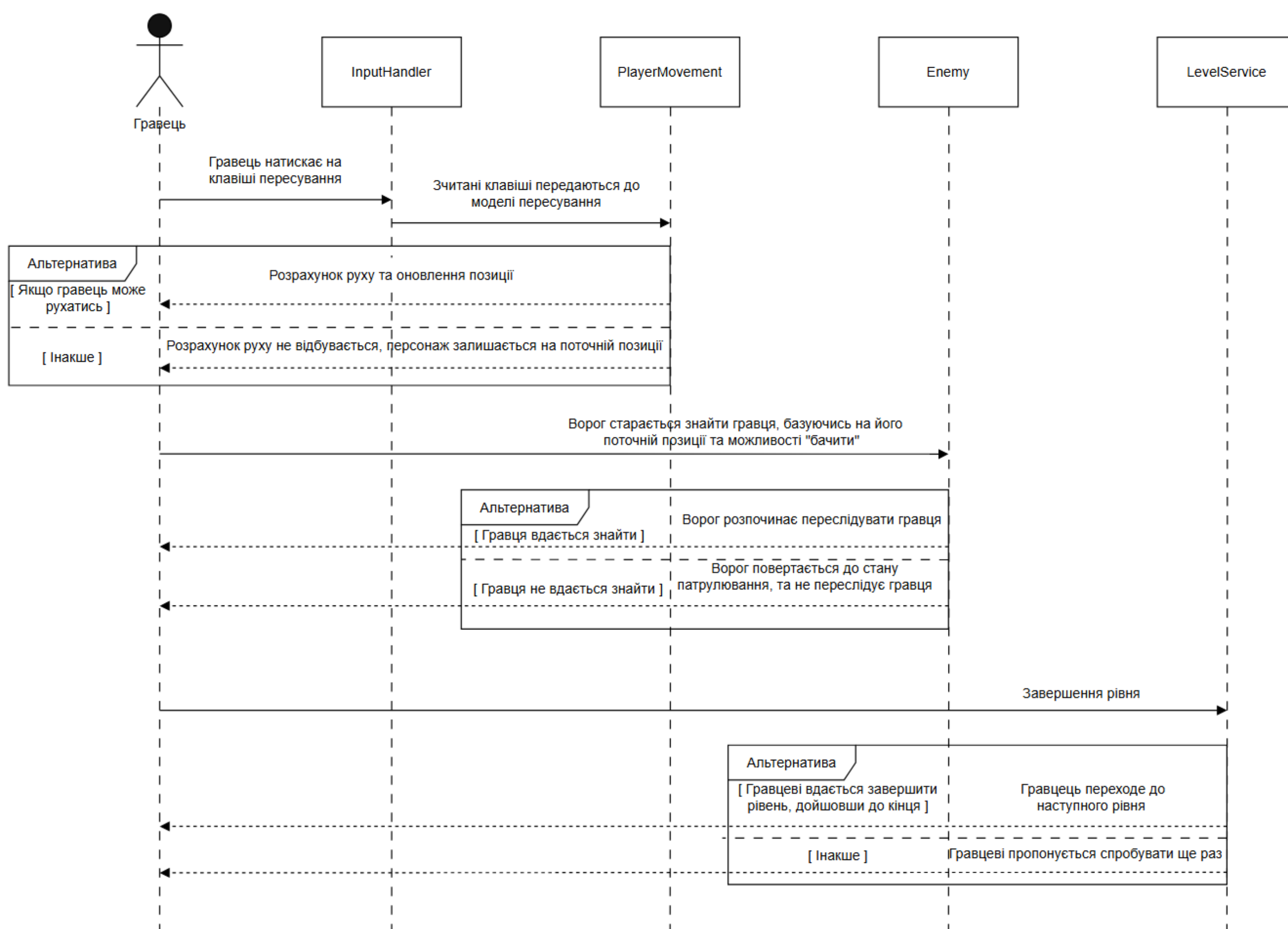

Рисунок 2.3 – Діаграма послідовності (Sequence Diagram)

Джерело: розроблено автором

Пояснення діаграми послідовності. Діаграма демонструє послідовності взаємодії між основними сутностями в Endless. Оскільки цей продукт містить безліч різних сутностей, що забезпечують роботу конкретної частини функціоналу, класи, що представлені на діаграмі, є узагальненням і насамперед

відображають структуру взаємодії, об'єднуючи в собі безліч інших, отже, кожен із них можна описати як:

Player (гравець) – цю сутність представлено в загальному вигляді, демонструючи вплив користувача на «повідомлення» (під повідомленням мається на увазі подія, що ініціює послідовність взаємодії між оброблювальними сутностями), що оброблятимуться сутностями, які є учасниками системи (гри);

InputHandler (обробник введення) – сутність, що репрезентує систему вводу, спроектовану в цьому додатку, не зачіпаючи деталі реалізації, на діаграмі показано процес зчитування клавіш і передавання інформації про введення зацікавленим сутностям;

PlayerMovement (система пересування персонажа) – також не є одним класом, а скоріше описує роботу кількох, що інтерпретують дані, отримані від оброблювача вводу і, на їхній основі, прораховують пересування персонажа, якщо це можливо;

Enemy (ворог) – узагальнює поняття супротивника, вказуючи на спробу ворожих неігрових персонажів завадити гравцеві на прикладі зміни стану, що описує поведінку бота у конкретний відрізок часу;

LevelService (сервіс рівнів) – сутність, що займається загальним менеджментом рівнів, опрацьовує логіку побудови залежно від режиму і типу рівня, може керувати сутностями, що безпосередньо створюють локації, реалізують спавн ворогів і предметів, перевіряють умови для проходження тощо.

Розглянуті діаграми описують поведінку програми (гри) з погляду користувача, що дає розуміння, як застосунок (гра) реагує на різні дії гравця, а також відображають ланцюжки взаємодії деяких із систем, що містяться в цьому продукті.

2.2 Моделювання структури продукту

На рис. 2.4 представлено діаграму класів, що показує відносини та залежності між класами, які стосуються гравця, ворогів і роботи системи рівнів. Більша частина з цих класів належать до геймплею (ігрового процесу), але вони можуть використовувати сервіси, які не стосуються безпосередньо до гемплею, наприклад, StaticDataService(через інтерфейс IStaticDataService) – надає конфігураційні файли для всіх сутностей, які цього потребують. Далі, під час опису архітектури, буде детально описано роботу всього застосунку, а наразі розглянуто ключові сутності.

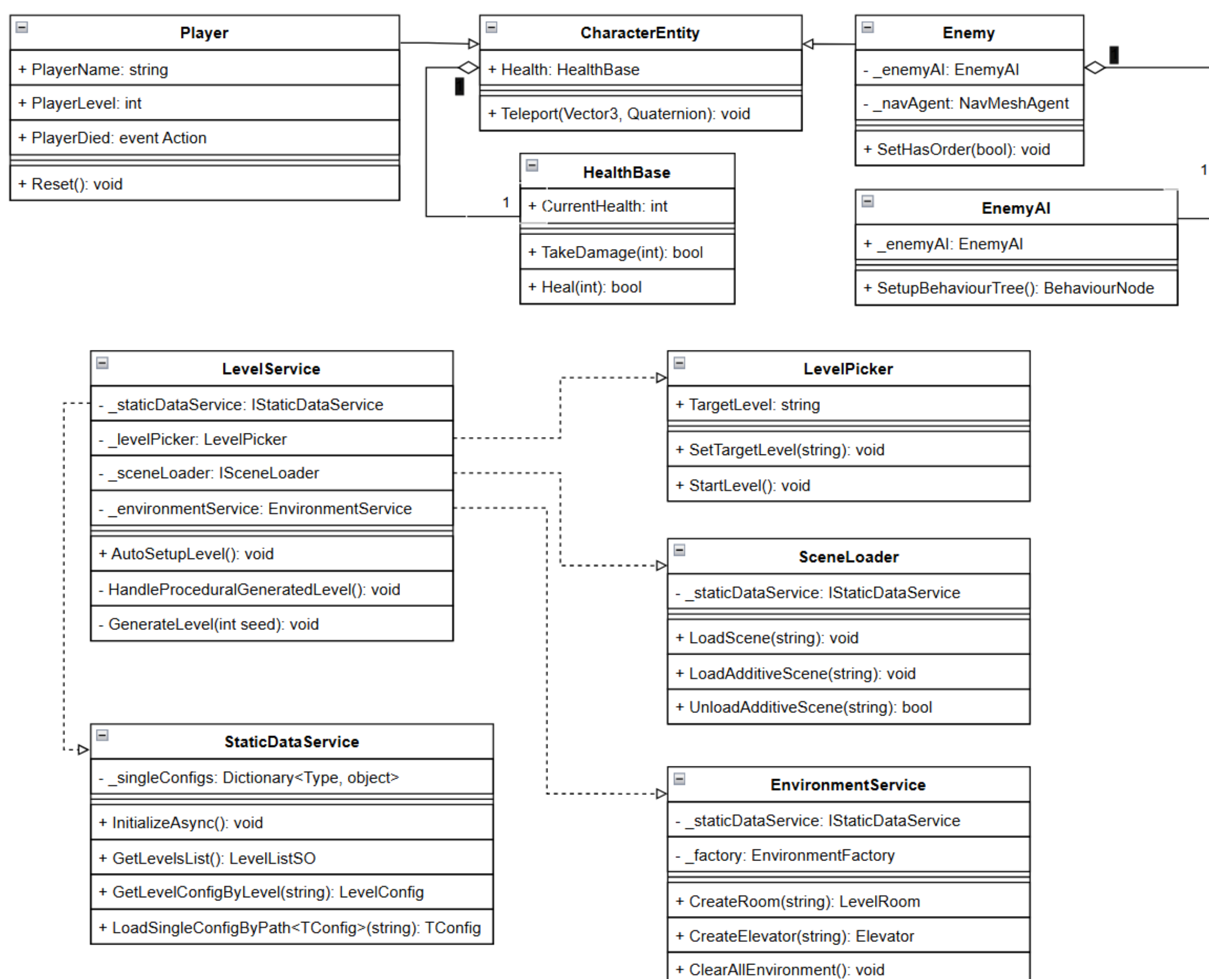

Рисунок 2.4 – Діаграма класів (Class Diagram)

Джерело: розроблено автором

Класи, розглянуті в даній діаграмі, не описано повністю, оскільки кожен із них делегує виконання логіки на кілька інших сутностей, повний розгляд яких на даному етапі буде складним для повноцінного розуміння – натомість, присутні їхні основні частини (члени класів), завдяки яким працює відповідна частина програми. Перш ніж розглядати кожен із класів, важливо уточнити, що Player і Enemy успадковуються від одного базового класу, який одразу надає метод для телепортації персонажа і поле, чий тип даних описує здоров'я персонажа і метод отримання шкоди або відновлення здоров'я, після чого розширюються і доповнюються необхідними їм членами класу (наприклад, клас Enemy отримує додатковий член класу у вигляді поля, що посилається на компонент, який описує інтелектуальну поведінку цього неігрового персонажа – EnemyAI, тощо). Більш детально про кожну сутність із діаграми:

клас гравця (Player) є максимально високорівневою обгорткою над класами, що реалізують основну частину функціоналу – у цьому класі немає безпосередньої логіки виконання чогось, але він, маючи посилання на інші класи/компоненти, може звертатися до інших класів, чия відповідальність полягає в наданні певного функціоналу;

клас ворога (Enemy) також вельми високорівневий і просто надає публічні члени класу для «перенаправлення» виконання логіки іншим класам, що реалізують певну частину роботи ворожих неігрових персонажів, наприклад, EnemyAI, Health тощо;

клас здоров'я (HealthBase) – сутність, що надає публічні методи і властивості для можливості нанесення шкоди гравцеві або неігровим персонажам, викликає події (events) під час зміни здоров'я або смерті персонажа;

клас постачальник статичних даних (StaticDataService) був описаний вище – його відповідальністю є «роздавати» конфігураційні файли всім, хто цього потребує;

завантажувач сцен (SceneLoader) відповідає за менеджмент сцен у проєкті, перехід між основними і завантаження/знищення дочірніх сцен, надає сцену завантаження, запуску програми тощо;

сервіс оточення (EnvironmentService) бере на себе управління всіма елементами оточення (всі об'єкти, що є будівлями або інші статичні об'єкти) – слідкує за правильним знищенням непотрібних об'єктів, завжди має посилання на кожен створений об'єкт для можливості повністю керувати його життєвим циклом. Створення конкретних нових об'єктів делегує на сутність EnvironmentFacotry, що безпосередньо займається процесом створення елементів оточення;

LevelPicker є сутністю, що приховує деталі реалізації старту ігрового процесу за публічним методом StartLevel(), також вказує сервісу рівнів (LevelService) на цільовий рівень під час переходу в геймплей, зберігає дані про поточний рівень, використовується для перезапуску рівня, надаючи необхідну інформацію для повторного завантаження;

сервіс рівнів (LevelService) відповідає за побудову рівня, є високорівневим сервісом, що використовує LevelPicker (для отримання даних про цільовий або поточний рівень для побудови), StaticDataService (відштовхуючись від конфігураційних файлів, у разі побудови рівня, що генерується в реальному часі, по-різному підходить до безпосередньої генерації), SceneLoader (для довантаження дочірніх сцен із локаціями(рівнями)), EnvironmentService (для делегування процесу створення локацій або кімнат), SpawnService (для спавну неігрових персонажів та предметів та контролю їх життєвого циклу), реалізуючи загальну логіку контролю життєвого циклу рівня, процес генерації або завантаження створеного заздалегідь рівня, не вдаючись до деталей реалізації кожного з вищезазначених сервісів.

На рис. 2.5, розглянуто діаграму розгортання (Deployment Diagram), що відображає вимоги для роботи застосунку на користувацькому пристрої.

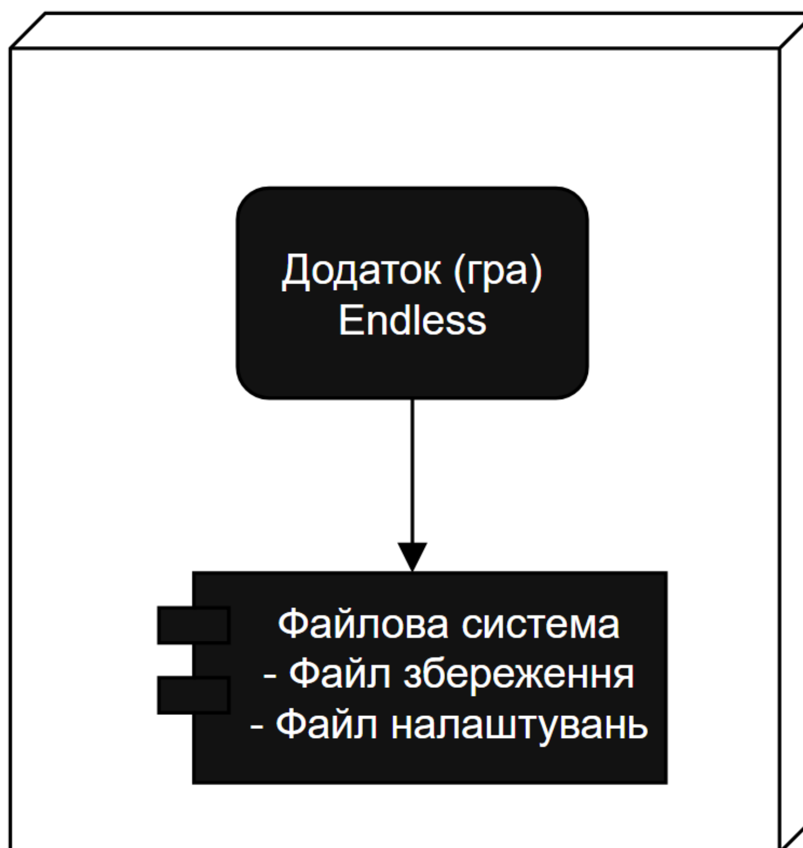

Рисунок 2.5 – Діаграма розгортання (Deployment Diagram)

Джерело: розроблено автором

Розгортання наразі доволі нескладне, оскільки застосунок (гра) працює локально, усі сервіси, що відповідають за функціональні частини, містяться в самому білді гри, а для зберігання файлів збереження ігрового прогресу та налаштувань потрібен лише доступ до файлової системи. Хоча наразі процес розгортання простий, надалі планується додавання опціональної можливості гри через мережу (мультиплеєр), для чого вже знадобиться підключення до інтернету та можливість комунікації із зовнішнім сервером, а також, до наявних сервісів буде розглянуто додавання зовнішніх, на прикладі античита (програма, що здатна розпізнати спробу отримання переваги користувачем через зміну коду або окремих компонентів гри).

2.3 Опис архітектури продукту

Через велику кількість складових, опис архітектури продукту поділено на підпункти, у межах яких детально розглянуто деталі роботи кожного шару архітектури, у порядку від пояснення загальних, об'єднуючих сутностей, до низькорівневих (тих, що здійснюють конкретну логіку).

Перед початком розгляду конкретних сутностей, необхідно описати та пояснити патерни, що використовувалися для їхньої реалізації.

Singleton – це патерн проєктування, що дає змогу обмежити кількість екземплярів певного класу до єдиного (зазвичай), у межах якого на рівні сутності відсутній публічний конструктор, але є метод, який, використовуючи приватний конструктор, створює екземпляр цього класу, якщо його немає, а в іншому випадку – створює новий замість вихідного або повертає посилання на вже створений. Доступ до екземпляра даної сутності здійснюється через публічні статичні поля або властивості. Цей підхід втрачає актуальність із появою Service Locator або Dependencies Injection патернів (розглянуті далі), оскільки вони не вимагають створення штучних і необов'язкових обмежень на рівні сутності.

Dependencies Injection (введення/ін'єкція залежностей) – це патерн проєктування, що використовується в усіх напрямках розроблення програмного забезпечення, забезпечує додаткову модульність, тестованість і підтримуваність різним компонентам системи шляхом надання залежностей (посилань на інші сутності) кожному з компонентів, які цього потребують. У цьому продукті використовується контейнер для залежностей Zenject [14], що в парі зі своїм аналогом VContainer [15] є промисловим стандартом для розроблення ігор із використанням рушія Unity [1]. Даний контейнер надає 3 контексти: Project Context (контейнер, життєвий цикл сутностей якого – увесь час роботи застосунку), Scene Context (контейнер, життєвий цикл сутностей якого збігається з часом існування його активної сцени) та GameObject Context (життєвий цикл його сутностей збігається з об'єктом, до якого відноситься сам

контейнер). Контейнер, чий рівень глобальності нижчий за інший, завжди може отримувати посилання на сутності, які перебувають у контейнері більшої глобальності, наприклад: контейнер із GameObjectContext завжди може отримати залежності зі SceneContext або ProjectContext, але ProjectContext контейнер не має доступу до залежностей, зареєстрованих у контейнері нижчого порядку (GameObjectContext, SceneContext). Також цей патерн дає змогу відмовитися від використання патерну Singleton, та реєструвати екземпляри важливих сутностей в одному з контекстів, що знімає необхідність на рівні сутності обмежувати створення екземплярів. Загалом, патерн впровадження залежностей є дуже просунутим сервіс-локатором (Service Locator), котрий, у свою чергу, є патерном, що пропонує зберігати посилання на важливі для системи компоненти в одній сутності (для можливості звернутися до неї для отримання того чи іншого сервісу), а екземпляри певних сервісів створювати при першому зверненні.

ObjectPool (патерн пулу об'єктів) – патерн проєктування, що часто використовується для оптимізації роботи застосунку шляхом зменшення частоти створення та знищення об'єктів, які часто зустрічаються, що передбачає замість знищення певних об'єктів, поміщати їх до списку неактивних, після чого під час наступного запиту на створення видавати один із неактивних об'єктів запитуваного типу, зменшуючи затримки та використання ресурсів пристрою для створення та знищення об'єктів.

Finite State Machine (патерн машина кінцевих станів) – патерн проєктування, який використовують для сутностей, у яких може бути тільки один стан у конкретний момент часу і, відштовхуючись від перевірок умов для зміни станів, можливість переходу в інші стани. Як правило, сутність, спроєктована за цим патерном, має публічні методи для реєстрування нових і зміни станів вручну, а самі стани є типом даних, що описує логіку під час входу, виходу і, опціонально, оновлення поточного стану, а виклик цих методів є відповідальністю машини станів. На рис. 2.6 представлено діаграму з прикладом використання патерну машини кінцевих станів. Хоча цей підхід дає

можливість повністю розділити логіку конкретних станів, його головним недоліком є змішування виконання конкретної логіки або дій та перевірки умов для переходу в інші стани.

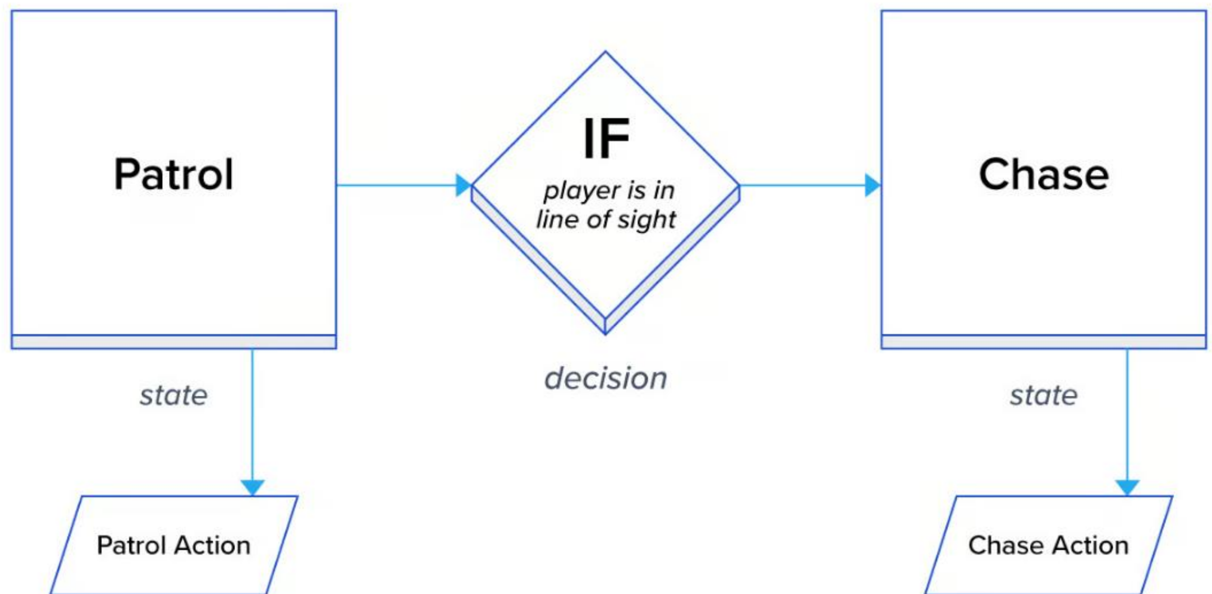

Рисунок 2.6 – Діаграма прикладу використання патерну машини кінцевих станів (Finite State Machine)

Джерело: [8]

Behaviour Tree (дерево поведінки) – патерн проектування, що використовується для опису складних поведінкових моделей, описуючи поведінку сутностей як дерево з набором гілок, що складаються з вузлів (Node). Кожен із вузлів може повертати три стани: успіх (Success), провал (Failure), вузол у процесі виконання (Running). Вузли можна поділити на перевірочні (CheckNode) і ті, що виконують певну дію (TaskNode), таким чином, перевірочні вузли не містять логіки виконання будь-чого, а контролюють потік, повертаючи Success або Failure, що зрештою призведе до певної дії або їхнього набору, а вузли, які виконують конкретну дію, не містять перевірок умов виконання, а просто відпрацьовують свою частину логіки,

завжди повертаючи тільки Running – такий підхід дає змогу відокремити перевірку умов від безпосереднього виконання дій, дозволяючи повторно використовувати вузли в абсолютно різних контекстах і поєднаннях, що і є основною відмінністю від машини кінцевих станів з її станами. Приклад дерева поведінки для неігрового персонажа відображено на рис 2.7.

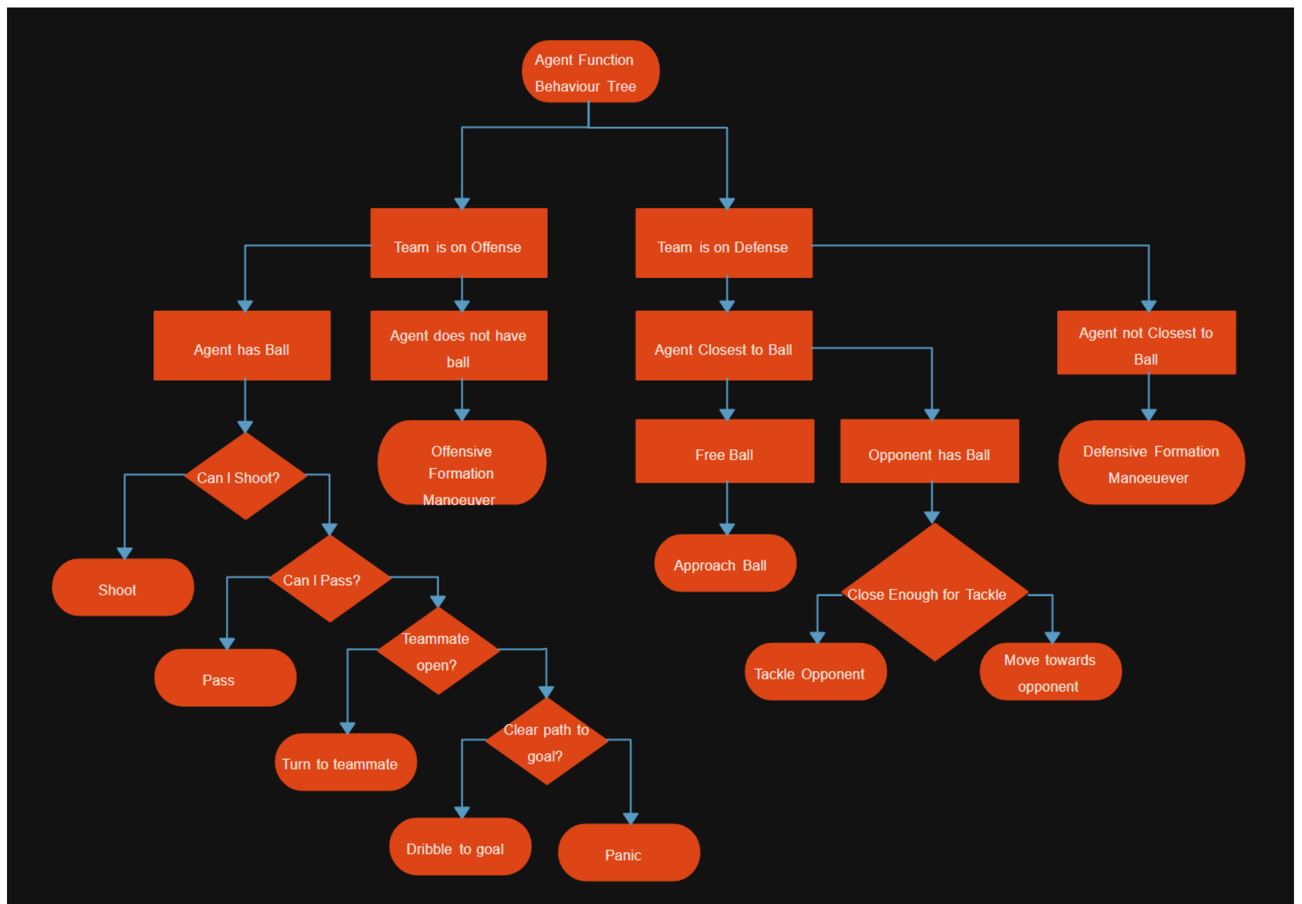

Рисунок 2.7 – Приклад дерева поведінки *(Behaviour Tree)*

Джерело: [9, 28]

Оскільки інтерфейс є дуже важливою частиною, а його гнучкість і модульність є дуже важливими в контексті тестування і можливості міняти або редагувати відображення, не зачіпаючи логіку роботи тих чи інших сутностей, використовувався MVP патерн – патерн MVX сімейства, що стосується проєктування інтерфейсу, який передбачає поділ конкретного елемента

інтерфейсу на три ключові сутності, що допомагає повністю розділити бізнес-логіку та логіку відображення: модель (model), видима частина (view), представник (presenter). На рис. 2.8 представлено діаграму, що демонструє зв'язок компонентів MVP.

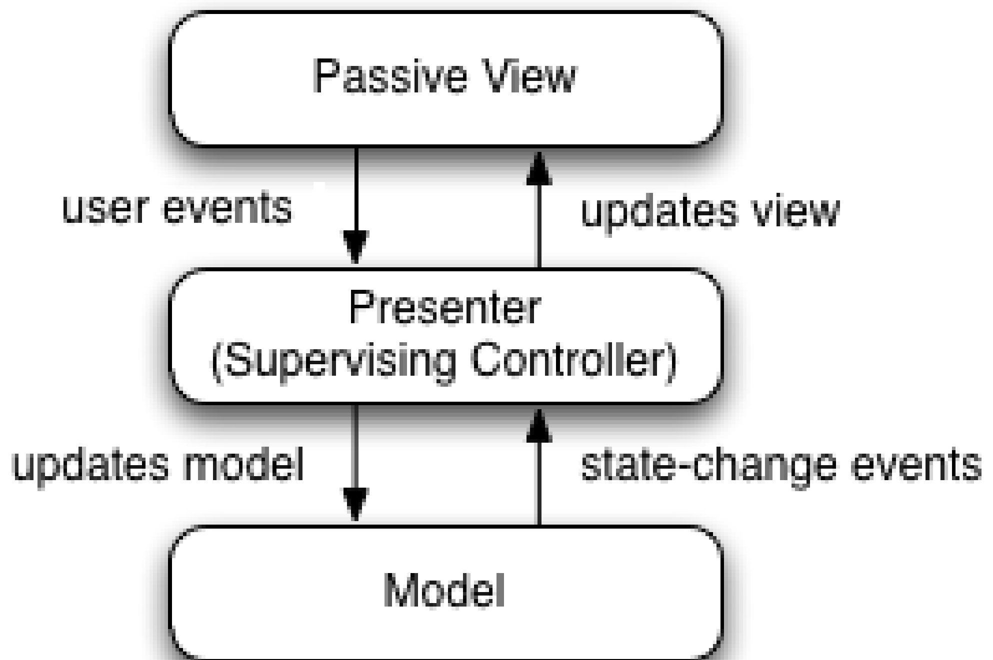

Рисунок 2.8 – Діаграма MVP патерну

Джерело: [10]

Докладніше про кожну з сутностей MVP:

модель – мається на увазі сутність, що містить певну бізнес-логіку (наприклад, клас або набір класів, що описують інвентар) з публічними членами класу, наприклад, подіями (event), на активацію яких реагуватиме представник (presenter) або публічними методами, що можуть бути викликані представником;

видима частина – сам об'єкт інтерфейсу (наприклад, вікно інвентарю), що викликає свої публічні події (event) після натискання на кнопки або інші елементи, які відносяться до нього – якщо конкретизувати, то патерн, який

використовується для організації інтерфейсу, є MVP Passive-View різновидом, оскільки видима частина ніколи не “знає” про те, що відбувається після взаємодії з її елементами – вона тільки активує події, що означають певну взаємодію (натискання на кнопку тощо), а будь-якою логікою обробки дій після натискання займається представник (presenter);

представник – сполучна ланка між моделлю і видимою частиною, яка повністю контролює видиму частину (крім того, будь-яка взаємодія з видимою частиною завжди відбувається через представника). Якщо попередні сутності, у вигляді моделі та видимої частини, є пасивними і не повідомлені про існування представника, то представник – це той, хто певною мірою знає про деталі реалізації моделі та видимої частини, може сам підписуватися на події моделі з метою поновлення даних видимої частини або, наприклад, на події видимої частини, щоб реагувати на натискання кнопок, викликаючи певні методи моделі тощо.

Фабрика (Factory) – патерн проєктування, що використовується для сутностей, чия роль, за допомогою публічних методів, що приховують деталі створення, делегувати процес безпосереднього створення об'єктів, надаючи можливість іншим сутностям створювати об'єкти без необхідності вдаватися до їхніх подробиць реалізації, натомість просто повертаючи створений об'єкт. Контекст використання цього патерну полягає в централізації створення об'єктів, що дає змогу вносити зміни в цей процес лише в одному місці.

Також важливо пояснити поняття контекстів проєкту в розумінні їхньої глобальності – у подальших поясненнях, під глобальністю маються на увазі рамки життєвого циклу тієї чи іншої сутності, отже, сутність, що перебуває в контейнері ProjectContext, тобто, в глобальному контейнері для всього проєкту, є єдиним екземпляром цього класу в усьому проєкті, створюється разом із цим контейнером (не обов'язково одразу, зазвичай за першим запитом на отримання залежності на сутність певного типу даних) та існує доти, доки її батьківський контейнер не буде знищено (принцип залежності рамок життєвого циклу від

існування батьківського контейнера однаковий для всіх контейнерів та сутностей, які до них відносяться, різниця тільки у рівні глобальності контейнеру, а отже, його сутностей), а якщо розглянути SceneContext контейнер – він ідентичний першому, тільки існує доти, доки не знищено сцену, де він перебуває, відповідно, життєвий цикл сутностей, що до нього належать, обмежений існуванням сцени, що й містить сам SceneContext контейнер. Для GameObejctContext контейнера все те ж саме, тільки його рівень глобальності – об'єкт на сцені (мінімальний з розглянутих).

Розглянувши основні підходи та патерни проєктування, далі присутній опис кожного з шарів архітектури та його сутностей.

Спочатку розглянуто основні шари, після чого більш детально описано кожен із них, однак зважаючи на чималу кількість різних сутностей, що відносяться до роботи додатка, здебільшого, пояснення стосується основної суті роботи та найбільш важливих учасників кожного шару.

Оскільки в проєкті використовується підхід ін'єкції залежностей (Dependencies Injection), спочатку розглянуто глобальний контекст (ProjectContext) і його контейнер, де створюються такі сутності (список не повний):

ApplicationStateMachine – глобальна машина кінцевих станів рівня програми, що диктує глобальний стан програми, деякі зі станів:

- ApplicationGameplayState – глобальний стан геймплею, що відкриває кореневу сцену геймплею, де ініціюється локальний контейнер рівня сцени, що створює локальну машину станів SceneStateMachine (з її станами, що не впливають на глобальний стан застосунку, а натомість реалізують логіку кожного стану в межах геймплею) – механізм, де в межах кожного глобального стану створюється локальна машина кінцевих станів рівня поточної кореневої сцени, є однаковим для всіх глобальних станів застосунку;

- ApplicationMainMenuState – глобальний стан головного меню;
- ApplicationBootstrapState – перший глобальний стан, у який переходить глобальна машина кінцевих станів, завантажуючи налаштування додатка та ініціалізуючи деякі з основних сервісів, наприклад, InputService, StaticDataService тощо;
- ApplicationLoadingState – глобальний стан, що описує перехід до сцени, локальна машина кінцевих станів якої займається завантаженням даних, які не стосуються налаштувань додатка, після чого передає дані всім зацікавленим сервісам.

GameProgressProvider – сутність, що надає збереження гри та налаштування застосунку;

SceneLoader – завантажувач, менеджер основних і дочірніх сцен;

StaticDataService – сервіс, що надає конфігураційні файли;

ApplicationBootstrapper – відповідає за реєстрацію станів глобальної машини станів під час запуску програми, після чого запускає стан ApplicationBootstrapState, де відбувається ініціалізація основних сервісів;

ApplicationRunner – сутність, що, використовуючи SceneLoader, завантажує основну сцену ініціалізації в момент запуску застосунку та перевіряє наявність ApplicationBootstrapper;

InputService – сервіс, що надає введення, на яке можуть реагувати інші учасники системи, містить різні схеми вводу (меню, геймплей) та можливість ними керувати (перемикати активну схему тощо);

LevelPicker – високорівневий сервіс, що містить інформацію про цільовий для переходу або поточний рівень, приховує запит зміни глобального стану застосунку на геймплей тощо;

UIRootView – кореневий інтерфейс рівня застосунку з різними шарами: шар для екранів або спливаючих вікон рівня застосунку, шар кореневого інтерфейсу рівня певного контексту (геймплей, основне меню, де є свої

контейнери для екранів або спливаючих вікон), шар екранів завантаження. На рис. 2.9 відображена ієрархічна структура інтерфейсу даного продукту.

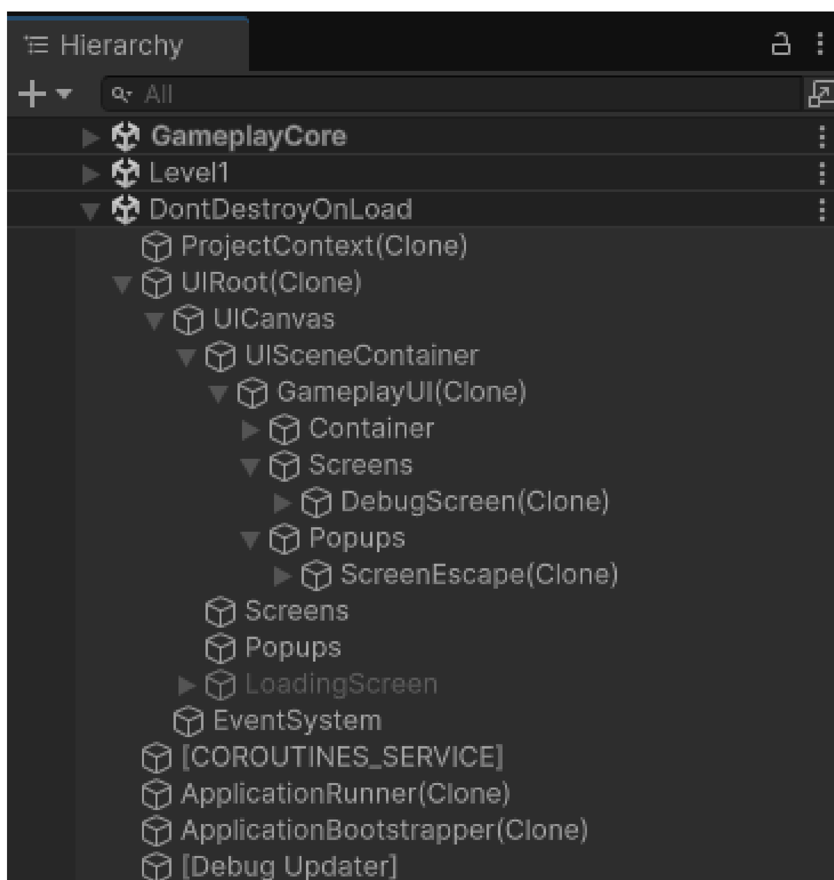

Рисунок 2.9 – Ієрархічна структура інтерфейсу (приклад з геймплею)

Джерело: розроблено автором

На зображенні продемонстровано ієрархічну структуру інтерфейсу, коренем якої є UIRootView (назва самого об'єкта – UIRoot), а GameplayUI є кореневим контейнером для поточного контексту (геймплей) зі своїми екранами (Screens) і спливаючими вікнами (Popups), у розділі 3.1 більш конкретно розглянуто процес створення та додавання елементів інтерфейсу, їхній менеджмент і деякі деталі реалізацій самих елементів інтерфейсу.

Після розгляду більшої частини основного шару архітектури, далі, на прикладі ігрового процесу, описано роботу в межах конкретного глобального стану (у цьому випадку, пояснення того, що відбувається після переходу

глобальної машини станів у ApplicationGameplayState). Крім специфічних систем для ігрового процесу (генерація рівнів, модель поведінки неігрових персонажів), тут також описуються загальні процеси (наприклад, робота інтерфейсу, обробки та інтерпретації вводу тощо).

У подальших поясненнях фігуруватиме поняття кореневої сцени, що вказує на підхід, згідно з яким, реалізація ігрового процесу повністю розділяє сцени з локаціями (рівень, тощо – їх може бути багато) та сцену, яка є фундаментом ігрового процесу (GameplayCore – дана сцена одна), до якої відноситься контейнер, що містить всі інфраструктурні сутності (SceneContext контейнер, що описаний далі). Тепер сфокусуємося на тому, що відбувається в межах одного глобального стану (у даному випадку, на рівні кореневої сцени ігрового процесу), так само як і для глобального контексту всього застосунку, тут першим розглянемо SceneContext контейнер і сутності, що в ньому створюються (список також не повний, але розглянуто усі ключові сутності):

SceneStateMachine – машина кінцевих станів рівня кореневої сцени, є практично ідентичною ApplicationStateMachine, але нижчою за рівнем глобальності впливу, нижче представлено деякі зі станів:

- GameplayInitializationState – стан ініціалізації ігрового процесу, у межах якого відбувається створення та «підв'язування» до UIRoot кореневого інтерфейсу цього контексту (GameplayUI контейнер на рис. 2.9), створюється ігровий персонаж, ініціалізуються необхідні постійні (детальніше пояснення роботи в розборі загальної роботи інтерфейсу) елементи інтерфейсу (шкала здоров'я, приціл тощо);
- GameplayPreparationState – стан, що описує етап підготовки до початку безпосереднього ігрового процесу, де LevelService видаляє попередній рівень (якщо такий є) і будує новий (опис роботи LevelService і сервісів, які він використовує, у розділі 2.2), наново спавнить ботів (віддає наказ сервісу для створення і розміщення неігрових персонажів);

- GameplayState – стан самого ігрового процесу, що активує обробку ігрової схеми введення (GameplayInputHandler через сервіс InputService), робить можливим виклик інтерфейсу, який безпосередньо стосується геймплею (інвентар, статистика і т.д.), віддає наказ діяти неігровим персонажам.

GameplayCoreBootstrapper – сутність, що на кшталт ApplicationBootstrapper, заповнює машину станів, але вже не глобального рівня, а в межах кореневої сцени ігрового процесу і запускає перший зі станів – GameplayInitializationState;

LevelService, EnvironmentFactory, EnvironmentService були розглянуті в розділі 2.2;

SpawnService – сервіс, що відповідає за спавн неігових персонажів і предметів, відстежує їхнє коректне знищення або потрапляння в ObjectPool з метою оптимізації, контролюється сервісом рівнів (LevelService);

UIViewInstantiator, UIFactory, UIManager<TSceneRootUI>, SceneRootUI сутності, що забезпечують роботу користувацького інтерфейсу, контролюють життєвий цикл кожного зі створених елементів інтерфейсу;

InventoriesService – сервіс інвентарів, що обробляє передачу предметів з одного інвентарю в інший, надає швидкий пошук конкретного інвентарю тощо (інвентарем може бути, наприклад, торговець, сховище, сам інвентар гравця).

Після опису роботи кореневої сцени геймплею (GameplayCore), можна перейти до пояснення роботи системи рівнів: щоб розпочати перехід у геймплей, необхідно, використовуючи LevelPicker, запустити рівень через його публічний метод StartLevel(string levelName), передавши назву рівня, після чого цей сервіс, звернувшись до ApplicationStateMachine, змінить глобальний стан на ApplicationGameplayState, а далі відбудеться перехід на кореневу сцену ігрового процесу (потім, як описувалося вище, розпочнуть свою роботу контейнер SceneContext разом з усіма супутніми сутностями, що забезпечують роботу геймплею).

Рисунок 2.10 – Меню Endless з наведеним курсором на кнопку вибору рівня

Джерело: розроблено автором

Назви рівнів (назва для відображення може бути якою завгодно, головний пункт – назва відповідної сцени) зберігаються в конфігураційних файлах кожного з рівнів, а список рівнів, що відображаються як опції для проходження (рисунок 2.10) після натискання на вибір рівня в головному меню (рисунок 2.11) формується, відштовхуючись від іншого конфігураційного файлу, що містить перелік рівнів, які мають бути доступними для проходження. Після вибору рівня, довантажується дочірня (additive) сцена, що існує до моменту знищення основної (GameplayCore), а всі присутні на ній сутності мають доступ до всіх інфраструктурних об'єктів з кореневої сцени (наприклад, сервіс для спавну персонажів, сервіс для створення будівель тощо), але дочірня сцена, окрім унікальної логіки для конкретного рівня, не містить практично ніяких сутностей, а натомість, тим самим процесом спавну, як правило, займаються сутності з контейнера кореневої сцени у відповідних станах (GameplayPreparationState), оскільки ідея поділу цих сцен полягає в

тому, щоб логіка та вся інфраструктура частина містилася в рамках завжди єдиної кореневої сцени (якщо конкретизувати, то в сутностях, чий життєвий цикл з нею збігається), а дочірня сцена з рівнем завжди була тільки самим рівнем, на якому, крім локацій, можуть бути точки спавну (Spawnpoint (приклад на рис. 2.12) – точка, що вказує на місце, де потрібно створити, наприклад, неігрового персонажа, крім цього, вказує на тип створюваного об'єкта, але не конкретний тип, а той, що пізніше інтерпретується сервісом спавна, і він уже обирає, який конкретно об'єкт створювати) або інші вказівники для сервісів, що так чи інакше реагують на їхню наявність.

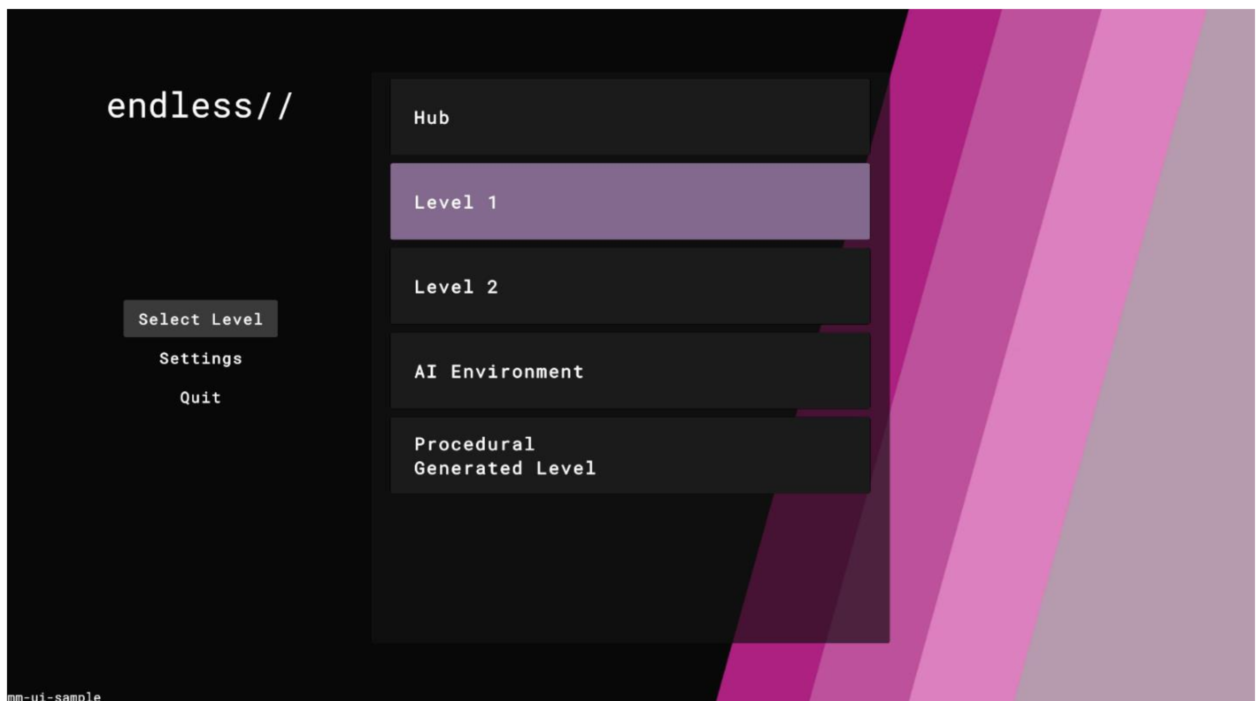

Рисунок 2.11 – Вікно вибору рівнів

Джерело: розроблено автором

Якщо обрано заздалегідь створений рівень, довантажується відповідна йому дочірня сцена, а в разі вибору рівня, що випадково генерується із заготовлених ділянок, завантажується дочірня сцена ProceduralGeneratedLevel, а LevelService не лише наказує SceleLoader завантажити рівень, а й генерує сам

рівень (процес генерації та принципу роботи моделі поведінки неігрових персонажів описано в розділі 3.1), після чого відбуваються ті ж самі процеси, що й для звичайних рівнів: спавн ботів (завдяки SpawnService), увімкнення опрацювання вводу, тощо.

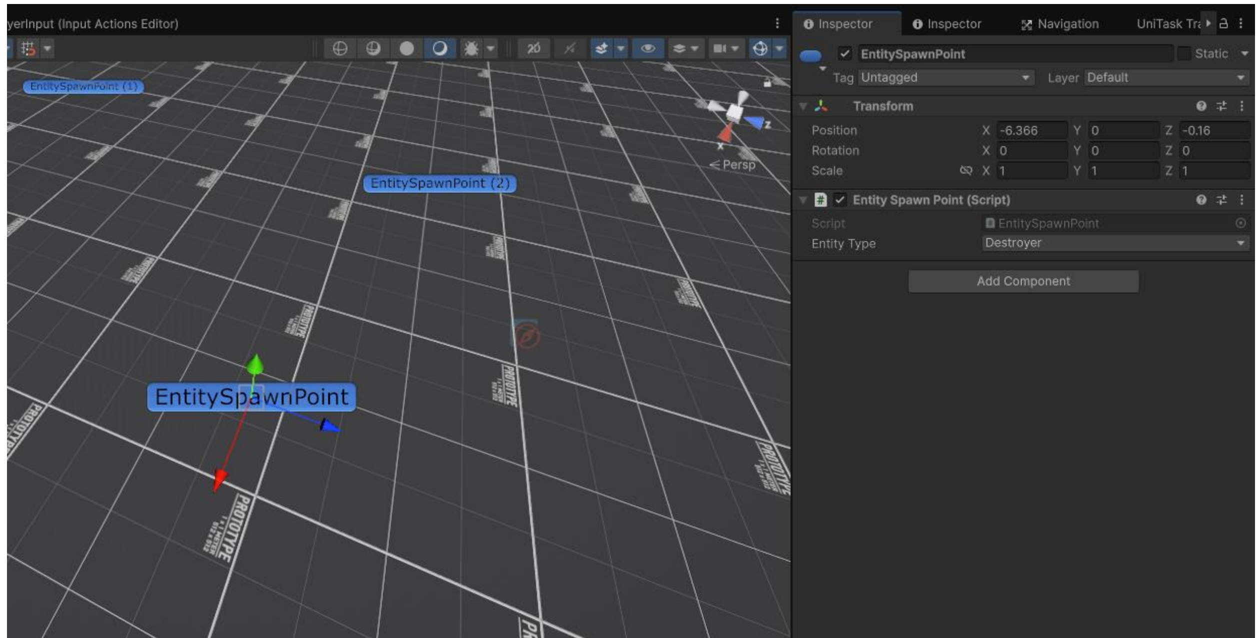

Рисунок 2.12 – Приклад точки спавну всередині редактору сцен рушія UnityEngine [1] (Spawnpoint)

Джерело: розроблено автором

На цьому зображенні показано приклад точки спавна та її компонент з можливістю вибрати тип об'єкта (зокрема ворога), що буде створено. Також існують точки спавна для предметів, що підбираються, та окремих генерованих будівель.

На рис. 2.13 представлено вікна ієрархії на різних рівнях, що демонструють, що для кожного з рівнів коренева сцена завжди є одною (GameplayCore), а відрізняється тільки дочірня (в даному прикладі, це Level1, Level2, ProceduralGeneratedLevel та EndlessHub), що безпосередньо містить рівень (локації, точки спавну тощо).

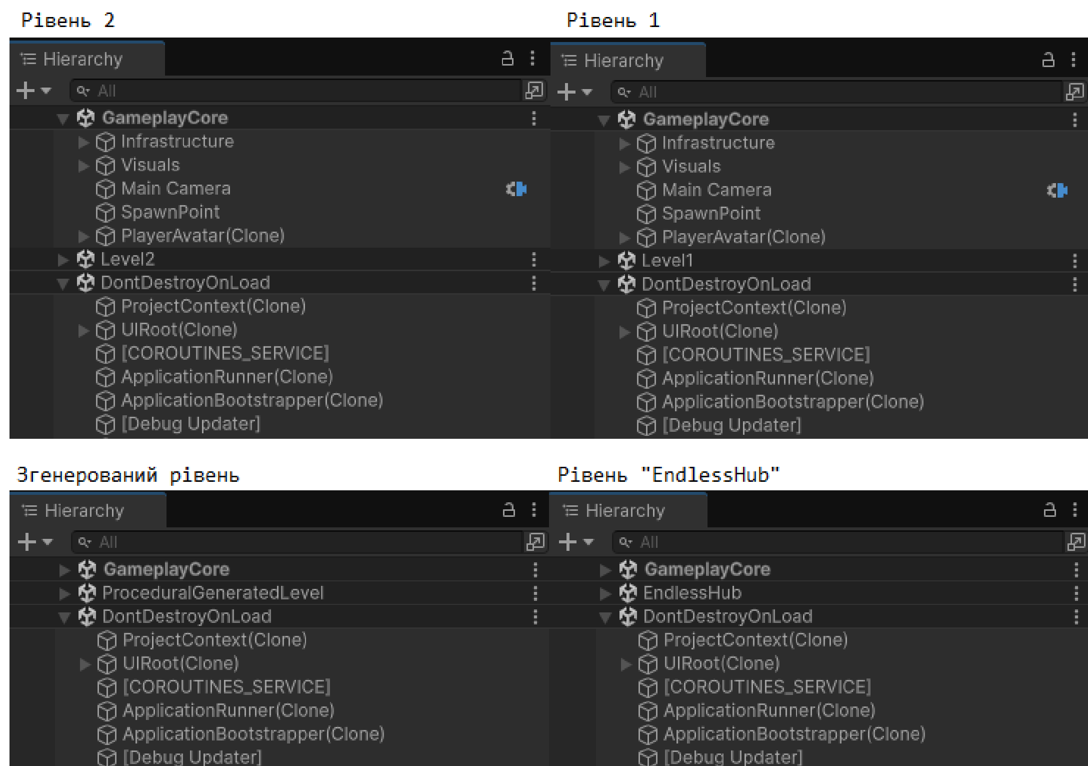

Рисунок 2.13 – Демонстрація ієрархії об'єктів, що, зокрема, відображає структуру рівнів

Джерело: розроблено автором

Було розглянуто основні шари та ієрархію архітектури, описано основні підходи до проєктування та структуризації проєкту. У наступному розділі, секція 3.1 у тому числі наповнена поясненням реалізації користувацького інтерфейсу в даному продукті, де розкрито підхід до створення окремих елементів інтерфейсу, їхнього менеджменту та використання в різних контекстах.

Висновки до розділу 2

У цьому розділі було розглянуто та описано підхід до проєктування і створення структури продукту комп'ютерної гри Endless із поясненням загальної поведінки застосунку, різних шарів і принципів архітектури,

патернів, що використовуються. У рамках моделювання продукту описано діаграми, кожна з яких пояснює певну частину загальної поведінки гри.

Під час опису процесу моделювання структури гри розглянуто та продемонстровано за допомогою діаграм взаємодію між деякими з основних сутностей програми, їхній зв'язок між собою, ієрархію.

Відносно детально розглянуто архітектуру застосунку, її шари та їхню ієрархію, пояснено деякі з основних патернів проєктування та причини їхнього вибору, наводилися конкретні приклади функціональних модулів з описом їхньої роботи.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Реалізація та конструювання програмного продукту

Опис технологій, що використовуються для реалізації розробленого продукту.

Вибір ігрового рушія: на даний момент є кілька основних ігрових рушіїв, що підходять для реалізації даного продукту.

Unreal Engine [25] (логотип на рис. 3.1) – ігровий рушій, розроблений компанією Epic Games [6], є однією з найбільш якісних пропозицій на ринку, містить велику кількість якісних вбудованих інструментів, підтримує останні технології в галузі графіки. В якості нативної мови програмування використовується змінена версія C++, що використовує механізм збирання сміття (контроль використання пам'яті, шляхом відстеження наявності посилань на об'єкти та їх знищення в разі, якщо посилань не знайдено) і рефлексію.

Рисунок 3.1 – Логотип рушію Unreal Engine

Джерело: [25]

Godot Engine [26] (логотип на рис. 3.2) – ігровий рушій з відкритим вихідним кодом, містить непоганий набір вбудованих інструментів, проте в контексті 3D ігор, поступається можливостями Unity Engine і Unreal Engine, тому частіше використовується для більш простих 3D проектів або 2D ігор.

Рисунок 3.2 – Логотип рушію Godot Engine

Джерело: [26]

Як ігровий рушій, що лежить в основі цього продукту, використовується Unity Engine [1] (логотип на рис. 3.3) – один з найбільш конкурентоспроможних рушіїв на даний момент, вибір якого заснований на постійній підтримці та нововведеннях великій кількості якісних вбудованих інструментів, вкрай стабільній роботі (цей рушій вже не один десяток років зарекомендував себе як чудовий вибір для побудови на ньому проєкту будь-якого масштабу), актуальності технологій, що містяться у ньому, підтримці різних API рендерінгу та операційних систем, завдяки чому він забезпечує бездоганну роботу додатка на найрізноманітніших системах (особливо на цільових для цього проекту: Windows [11], Linux [12], MacOS [13]), що і робить його чудовим кандидатом лежати в основі проекту.

Рисунок 3.3 – Логотип рушію Unity Engine

Джерело: [1]

Мовою програмування, що нативно підтримується даним рушієм, є розроблена компанією Microsoft мова, що належить до сімейства С-подібних, C# [7] (логотип показано на рис. 3.4) – вперше презентована 2001 року, сучасна об'єктно-орієнтована високопродуктивна мова програмування загального призначення, яка постійно розвивається й підтримується та використовується практично в усіх сферах розроблення (від вебсайтів до комп'ютерних ігор). Уся кодова база продукту Endless створена цією мовою.

Рисунок 3.4 – Логотип Microsoft .NET (C#)
Джерело: [7]

Середовищем розробки було обрано Microsoft Visual Studio 2022 [20] (рис. 3.5), що надає безліч інструментів і зручного функціоналу для ефективнішого процесу написання коду, постійно підтримується, дає змогу використовувати плагіни та є одним із найбільш перевірених часом інструментів розробника та особливо добре підходить у зв'язці з C#, що і стало основною причиною вибору цього середовища розробки.

Рисунок 3.5 – Логотип Microsoft Visual Studio
Джерело: [20]

Для створення та редагування 3D моделей використовувався Blender [19] (рис. 3.6) – насамперед, вкрай потужний 3D-редактор, який використовується в усіх напрямках, але крім цього, володіє багатим набором можливостей, зокрема скульптуруванням, текстуруванням, редагуванням відео тощо. Постійно покращується і загалом дуже зручний у використанні, а в цьому продукті використовувався для складніших моделей, тоді як вбудований редактор ProBuilder у рушій Unity [1] часто добре підходить для найрізноманітнішого оточення, надаючи також дуже великий набір різних зручних інструментів для створення моделей тощо.

Рисунок 3.6 – Логотип Blender

Джерело: [19]

При створенні двовимірних зображень (спрайти тощо) використовувався Aseprite [21] (рис. 3.7) – редактор для піксель-арту, один із найпоширеніших, зважаючи на кількість і якість інструментів, що надаються, а також простоту у використанні.

Рисунок 3.7 – Логотип Aseprite

Джерело: [21]

Пояснення алгоритму генерації рівнів: під час запуску рівня, що генерується з певних ділянок (кімнат), сервіс рівнів LevelService отримує рандомне (отримуване псевдовипадковим чином) значення (seed), відштовхуючись від якого генерує набір кімнат різних типів:

- перша й останні кімнати відрізняються від решти, оскільки вони є кінцевими (їх неможливо пройти, оскільки вони «запечатані») – кімнат такого типу може бути багато, але на одному рівні існує лише одна початкова й остання кімнати, які обираються випадковим чином із доступних;
- проміжні кімнати можуть відрізнятися за складністю проходження, вимагати від гравця різного набору вмінь (тактику, якщо сегмент рівня переважно полягає в битві з великою кількістю ворогів; маневреність, якщо є складні ділянки, які потребують особливої акуратності під час проходження тощо.) – кімнати такого типу завжди мають вхід та вихід, кожна наступна кімната обирається випадковим чином (або за складністю) із набору сумісних з попередньою та з'являється на спеціальній мітці (ConnectorOut), що розміщена на виході з попередньої кімнати, приклад на рис. 3.8;
- з дуже низьким шансом, на рівні може з'явиться бонусна кімната з секретним артефактом або іншими ресурсами.

Після генерації локацій рівня розпочинається спавн ворогів, зброї і предметів, з якими можна взаємодіяти, а залежно від налаштувань, їхні початкові позиції можуть бути заданими заздалегідь, випадковими або в межах заданої відстані від заздалегідь розставлених точок.

Саме таким чином працює генерація і, до того ж, у майбутньому планується додавання випадкової генерації предметів та ворогів на всіх ділянках, щоб окрім завжди унікального порядку і кількості кімнат рівня, щоразу був різний набір ворогів, статичних предметів і тих, з якими можна взаємодіяти.

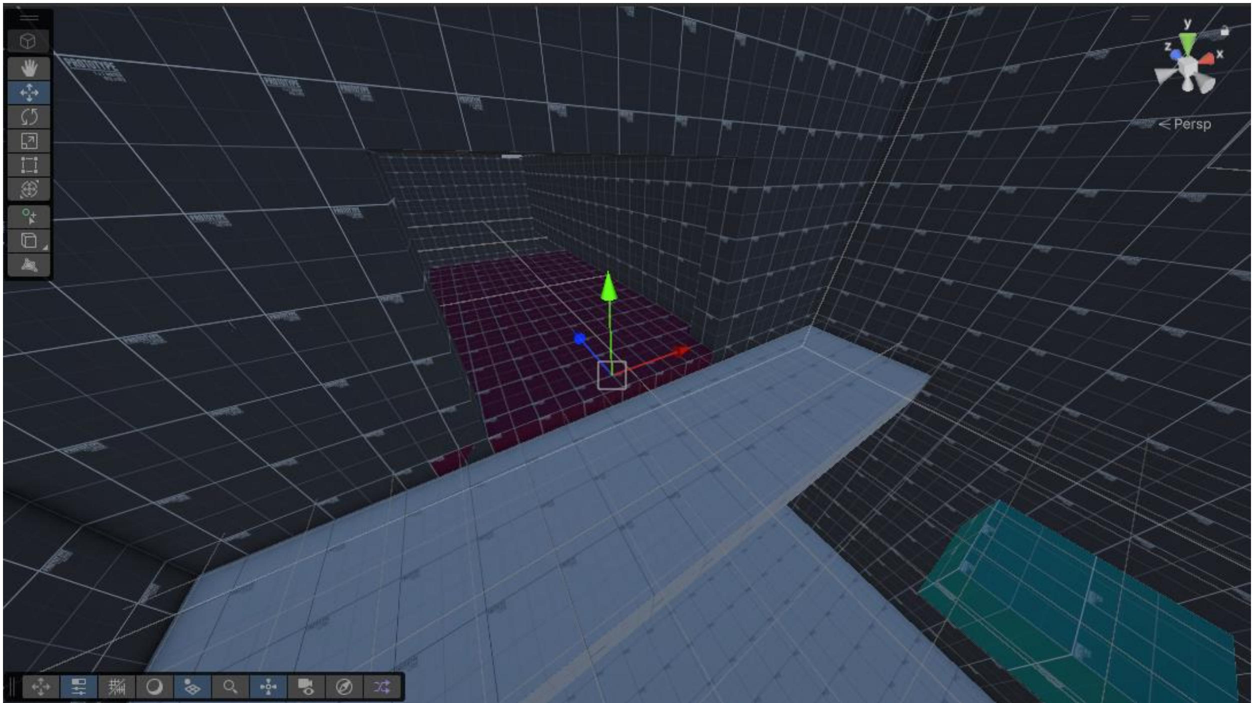

Рисунок 3.8 – Приклад точки виходу із кімнати

Джерело: розроблено автором

Пояснення підходу до реалізації моделі поведінки неігрових персонажів: з метою створення комплексної, модульної та легко розширюваної моделі поведінки, чудово підійде підхід із використанням патерну поведінокового дерева (Behaviour Tree). Головна його перевага в можливості розділяти вузли, що відповідають за перевірку певних умов, вузли, що контролюють загальний потік, і вузли, які реалізують логіку виконання конкретної дії, а також вибудовувати, як свідчить сама назва, дерево поведінки, де може бути багато гілок (кожна з яких може розгалужуватись), які описують складну поведінку неігрових персонажів (але може застосовуватися не тільки до персонажів, а до будь-яких сутностей, нерідко є заміною патерну Finite State Machine, описаного раніше).

Для пояснення моделі інтелектуальної поведінки неігрових персонажів (що описана у 2.3), був створений бот і оточення для нього (локація, вороги), щоб у деталях розібрати принцип його роботи. На рис. 3.9 показано скріншот

його дерева поведінки, на прикладі якого пояснюється кожен елемент (вузол), його відповідальність і вплив на підсумкову поведінку.

```
        2 references | yukineqqe, 62 days ago | 1 author, 1 change
34      protected override BehaviourNode SetupTree()
35      {
36          BehaviourNode rootNode = new Selector(new List<BehaviourNode>()
37          {
38              new Sequence(new List<BehaviourNode>()
39              {
40                  new CheckHasOrder(_input),
41                  new Selector(new List<BehaviourNode>()
42                  {
43                      new Sequence(new List<BehaviourNode>()
44                      {
45                          new LowHealthCheck(_health),
46                          new Selector(new List<BehaviourNode>()
47                          {
48                              new Sequence(new List<BehaviourNode>()
49                              {
50                                  new CheckSafeDistance(transform, _animator, _agent, _safeDistance),
51                                  new TaskUseHealthItem(_animator, _agent, _health)
52                              }),
53                              new TaskEscape(transform, _animator, _agent, _speed, _safePlace)
54                          })
55                      }),
56                      new Sequence(new List<BehaviourNode>()
57                      {
58                          new CheckEnemyInAttackRange(transform, _animator, _attackingRange),
59                          new TaskAttack(transform, _animator, _agent, _damage)
60                      }),
61                      new Sequence(new List<BehaviourNode>()
62                      {
63                          new FindEnemyInRange(transform, _animator, _agent),
64                          new FollowTargetTask(transform, _animator, _agent, _speed)
65                      }),
66                      new TaskPatrol(_animator, _agent, transform, _waypoints, _speed)
67                  })
68              }),
69              new TaskGoToBase(new Vector3(-25f, 0f, 15), _agent)
70          });
71
72          return rootNode;
73      }
74  }
```

Рисунок 3.9 – Дерево, що описує поведінку неігрового персонажа

Джерело: розроблено автором

Метод SetupTree() повертає саме дерево поведінки в поле для кореневого вузла (у ньому ж задається ієрархія вузлів), є членом базового класу(перевизначає (override) логіку цього методу з початкового класу) для компонента, що описує поведінку об'єкта BehaviourTree.

```
using UnityEngine;
using Zenject;

namespace BehaviourTree
{
    public abstract class BehaviourTree : MonoBehaviour
    {
        private BehaviourNode _root;

        private void Start()
        {
            _root = SetupTree();
        }

        private void Update()
        {
            if (_root != null)
            {
                _root.Evaluate();
            }
        }

        protected abstract BehaviourNode SetupTree();
    }
}
```

Рисунок 3.10 – Базовий клас BehaviourTree

Джерело: розроблено автором

На рис 3.9 продемонстровано, що компонент SampleBotAI (що є спадкоємцем BehaviourTree, що представлено на рис. 3.10) наповнюється набором вузлів, що описують його поведінку, а перш ніж розглянути кожен із них, варто зачепити клас, що лежить в їхній основі – BehaviourNode, а крім нього, описати контролюючі Selector і Sequence вузли:

Клас BehaviourNode (рис. 3.11) є базовим для всіх вузлів, містить такі члени класу:

- поле, що зберігає його поточний стан;
- список дочірніх вузлів;

- словник, куди за допомогою методів SetData(string key, object value), GetData(string key) та ClearData(string key) можна записувати, отримувати й видаляти дані, що є важливими для цього вузла, а також отримувати доступ до даних з батьківських вузлів (але не навпаки);
- метод Attach(BehaviourNode node) для додавання інших вузлів до списку його дочірніх;
- та ключовий метод, Evaluate(), який повертає стан вузла (що може бути Success, Failure, Running – описувалося в розділі 2.2), де відбувається сама логіка конкретного вузла, який і викликається BahaviourTree, до якого він належить.

```
using System.Collections.Generic;

namespace BehaviourTree
{
    public abstract class BehaviourNode
    {
        protected NodeState _state;
        protected List<BehaviourNode> _childNodes = new List<BehaviourNode>();
        private Dictionary<string, object> _dataContext = new Dictionary<string, object>();
        public BehaviourNode Parent { get; set; }

        public BehaviourNode()
        {
            Parent = null;
        }

        public BehaviourNode(List<BehaviourNode> childNodes)
        {
            foreach (var childNode in childNodes)
            {
                Attach(childNode);
            }
        }

        private void Attach(BehaviourNode node)
        {
            node.Parent = this;
            _childNodes.Add(node);
        }

        public virtual NodeState Evaluate()
        {
            return NodeState.FAILURE;
        }

        public void SetData(string key, object value)
        {
            _dataContext[key] = value;
        }

        public object GetData(string key)
        {
            object value = null;

            if (_dataContext.TryGetValue(key, out value))
            {
                return value;
            }

            BehaviourNode node = Parent;
            while (node != null)
            {
                value = node.GetData(key);
                if (value != null)
                {
                    return value;
                }

                node = node.Parent;
            }

            return null;
        }

        public bool ClearData(string key)
        {
            object value = null;

            if (_dataContext.ContainsKey(key))
            {
                _dataContext.Remove(key);
                return true;
            }

            BehaviourNode node = Parent;
            while (node != null)
            {
                bool cleared = node.ClearData(key);
                if (cleared)
                {
                    return true;
                }

                node = node.Parent;
            }

            return false;
        }
    }
}
```

Рисунок 3.11 – Базовий клас BehaviourNode

Джерело: розроблено автором

Поняття контролюючих вузлів – вузли цього типу не є перевірочними та не описують конкретну дію, а контролюють виконання дочірніх вузлів. До їх числа на даний момент входять Sequence і Selector вузли:

Sequence – цей вузол контролює виконання дочірніх за принципом ланцюжка: якщо один із дочірніх вузлів поверне стан провалу (Failure), то цей вузол, будучи батьківським, також негайно припинить виконання логіки в дочірніх вузлах, що залишилися, і поверне стан провалу. Інакше кажучи, принцип його роботи можна описати як «до першого провалу» одного з його дочірніх вузлів;

Selector – здійснює виконання кожного з дочірніх вузлів доти, доки хоча б одним із них не буде повернуто стану успіху або «в процесі» (Success / Running), після чого контроль потоку поведінки буде передано цьому вузлу, а якщо таких немає – Selector поверне стан провалу. Інакше кажучи, відштовхуючись від нього відбувається вибір наступної гілки поведінкового дерева.

Після опису основних складових сутностей, що є основою для інтелектуальної моделі, далі розглянуто кожен із вузлів поведінкового дерева демонстраційного бота (рис. 3.9):

- CheckHasOrder – вузол, що перевіряє, чи є наказ діяти в неігрового персонажа;
- TaskGoToBase – вузол, що описує завдання повернення на базу;
- LowHealthCheck – вузол, що перевіряє, чи низький поточний запас здоров'я;
- CheckSafeDistance – вузол, що перевіряє, чи достатньо далеко ворог (ціль) від бота;
- TaskUseHealthItem – вузол, що описує відновлення здоров'я шляхом використання предмета для регенерації;
- TaskEscape – вузол, що описує процес втечі від цілі;

- CheckEnemyInAttackRange – вузол, який перевіряє, чи достатньо близький ворог, щоб почати його атакувати;
- TaskAttack – вузол, що описує атаку по цілі;
- FindEnemyInRange – вузол, що перевіряє, чи бачить неігровий персонаж ціль;
- FollowTargetTask – вузол, що описує процес переслідування цілі;
- TaskPatrol – вузол, що описує патрулювання за заздалегідь заданими або отриманими випадково точками.

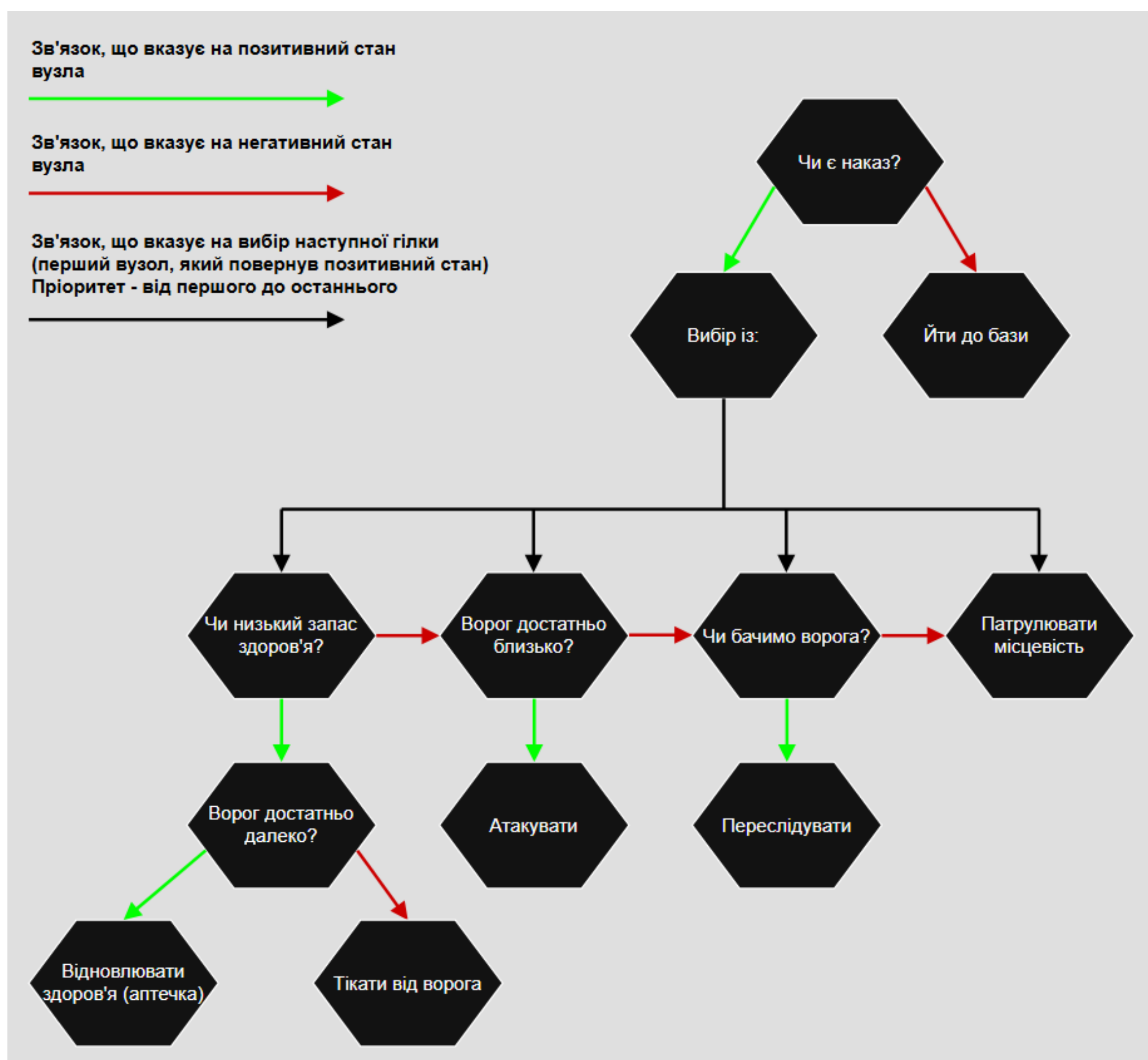

Рисунок 3.12 – Діаграма, що описує структуру інтелектуальної моделі бота

Джерело: розроблено автором

Відштовхуючись від опису вузлів, можна зробити висновок, що кожен із них є максимально самостійним і може використовуватися неодноразово у різних контекстах, оскільки між ними немає прямого зв'язку, але складаючи з них поведінкове дерево, можна отримати дуже розширювану і модульну інтелектуальну модель.

Опис поведінки бота, чию структуру інтелектуальної моделі з вищеописаних вузлів відображено діаграмою на рис 3.12:

Якщо в даного неігрового персонажа немає наказу діяти, він просто піде на базу, а інакше буде вибір однієї з гілок поведінки (обирається перша гілка, чий базовий вузол поверне успіх (Success), в іншому разі, шукається наступна гілка):

- Спочатку перевіряється, чи низький запас здоров'я і, в разі якщо це так, відбудеться перевірка, чи достатньо велика відстань від цілі (для безпеки бота) для використання предмета, що відновлює здоров'я, після чого персонаж почне використовувати аптечкою (предмет для регенерації), а інакше – тікати доти, доки не буде достатньо далеко від ворога;
- Потім буде перевірка, чи достатньо близько ворог для того, щоб його атакувати та якщо це так – виконуватиметься завдання (вузол, що описує конкретну дію) з безпосередньою атакою;
- Далі йде перевірка, чи бачить неігровий персонаж ціль і якщо це так, він починає її переслідувати;
- Якщо жодна з попередніх гілок не виконалася, то бот патрулюватиме місцевість.

На прикладі цього неігрового персонажа пояснено підхід до створення інтелектуальної моделі поведінки в Endless.

3.2 Тестування програмного продукту

Функціональне тестування є вкрай важливим етапом, перевірка якості роботи дає змогу виявити наявність дефектів або неоднозначних рішень під

час конструювання та усунути їх до випуску на ринок, що критично важливо, оскільки дає змогу запобігти випуску продукту в неналежному стані. Програмний продукт Endless було протестовано як вручну так і за допомогою автоматизації на кожному з етапів розробки:

- для тестування кожного з модулів застосунку, було відтворено спеціальні умови, що автоматизовано перевірили його на предмет некоректної роботи безліччю повторень і різних ввідних даних;
- ручне тестування проводилося як для окремих функціональних частин, так і для декількох, що пов'язані між собою, щоб не допустити дефектів або непередбачуваної поведінки додатка;
- модель інтелектуальної поведінки неігрових персонажів перевірено на предмет якості прийнятих ботом рішень, його конкретних дій у різних ситуаціях;
- загальна логіка і механізми роботи гри протестовані в різних умовах – наразі забезпечують стабільну і якісну роботу програми: гра не закривається несподівано, в ігровому процесі підтримується висока частота кадрів;
- генерація рівнів також пройшла низку тестувань – автоматизовані, в рамках яких сотні тисяч разів створювалися різні рівні та оцінювалися алгоритмом і, крім того, було перевірено крайні випадки, що підтвердило повноцінну працездатність цього модуля.

Також, для тестування продуктивності та можливості наочно побачити конкретну проблемну частину, використовувався вбудований інструмент у рушій Unity Engine [1] – Profiler, за допомогою якого можна відстежувати дані споживання системних ресурсів застосунком і виявити моменти підвищеного споживання або витоку пам'яті, різке падіння кількості кадрів за секунду, низьку чуйність тощо.

У процесі тестування було знайдено та усунуто низку помилок, дефектів та інших прикладів некоректної роботи та наразі кожен із модулів застосунку

працює коректно і без нерозривних залежностей, завдяки чому продукт гнучкий і легко розширюваний.

3.3 Використання програмного продукту

Опис можливостей розробленого програмного продукту(Інструкція користувачеві). Розроблений продукт Endless може запропонувати унікальний ігровий досвід динамічного шутера, що кидає виклик вправним гравцям, вимагаючи швидкості прийняття рішень і знання механік гри для ефективного проходження заздалегідь створених або згенерованих локацій, наповнених противниками з інтелектуальною моделлю поведінки.

Даний продукт поширюється через платформи електронної дистрибуції Steam [5] і Epic Games [6], разово купується цифрова копія базового видання, що включає безкоштовні оновлення загального призначення та платні розширення, які додають нові геймплейні механіки, ворогів та локації.

Після встановлення гри, під час першого запуску, для користувача створиться файл збереження, куди записуватиметься весь подальший прогрес гравця.

Під час запуску, після завантаження, користувач потрапляє в головне меню, звідки може:

- відкрити налаштування – відкриється повний список налаштувань застосунку, включно з можливістю зміни мови;
- вийти з гри – прогрес і поточні налаштування програми будуть збережені, після чого відбудеться закриття гри;
- почати ігровий процес, натиснувши на відповідну кнопку, після чого відкриється меню вибору рівнів, доступних для проходження, де потрібно буде вибрати між створеними заздалегідь рівнями або згенерованими в реальному часі.

Переходячи в геймплей, перед гравцем стоїть головна мета завершити рівень, проте є й другорядні: збір предметів зі схованок, швидке проходження. Увесь прогрес (уражені вороги, зібрані предмети) зберігається за умови виходу

з рівня, але найбільша нагорода чекає на гравця в разі проходження рівня до кінця. Якщо персонаж гравця помирає, йому пропонується спробувати знову або вийти в головне меню, а в разі успішного проходження зараховується винагорода, пропорційна ефективності проходження (враховується кількість нейтралізованих ворогів, швидкість проходження тощо) і з'являється можливість перейти до наступного рівня. При проходженні двох і більше рівнів без поразок, нагорода за завершення наступних рівнів підвищується.

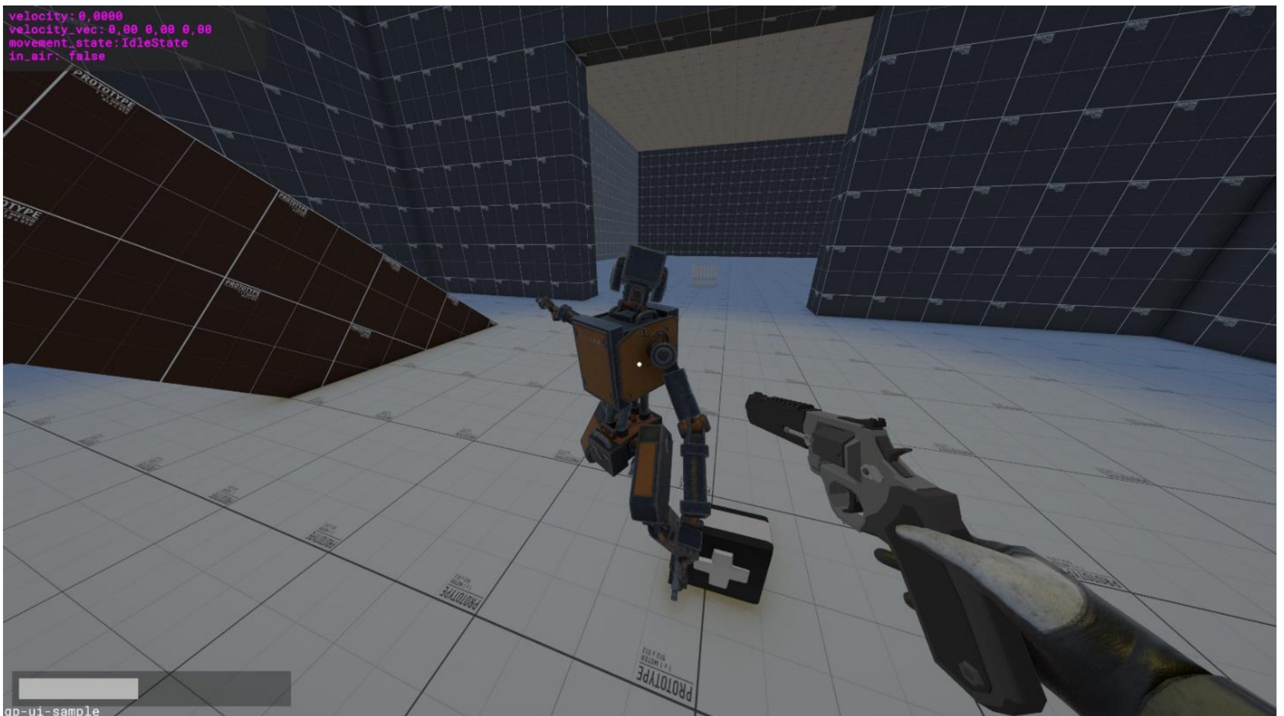

Рисунок 3.13 – Приклад геймплею Endless
Джерело: розроблено автором

Для підвищення швидкості проходження, на деяких ділянках рівнів передбачені альтернативні короткі шляхи, а крім того, в даному продукті модель пересування персонажа сильно нагадує Apex Legends [17] / Valorant [18], пропонуючи різні техніки для прискорення персонажа (своєчасне використання слайдінгу у зв'язці зі стрибками) та вкрай чуйний контроль у момент перебування у повітрі (Air Strafing [22]), також є дуже маленьке

тимчасове «вікно», у рамках якого, якщо гравець встигне натиснути на стрибок під час приземлення, він не отримає уповільнення від тертя об поверхню і зможе підтримувати високу швидкість пересування (ця техніка часто іменується як BunnyHope), завдяки чому, вивчаючи тонкощі гри, є можливість швидше проходити рівні, розуміючи комбінації наявних ігрових механік та їхнє застосування в різних ситуаціях.

Висновки до розділу 3

У цьому розділі міститься опис інструментів для створення продукту Endless, пояснення їхнього вибору:

- розглянуто більш детальний опис підходів до безпосередньої реалізації моделі інтелектуальної поведінки неігрових персонажів із розкриттям ключових складових, наведено приклад поведінкового дерева бота, що явно демонструє структуру прийняття рішень і пріоритет їхнього вибору.
- описано алгоритми генерації рівнів, розглянуто процес побудови рівня, його етапи та основні сутності, що беруть участь;
- пояснюється процес тестування гри на прикладі ручного та автоматизованого підходів, згадується висока результативність проведення тестування та виявлення дефектів у роботі програми;
- розглядається використання програмного продукту, що описує можливості програми та контексти використання;
- інструкція користувача – розглядає основні дії, які може вчинити гравець, починаючи від встановлення застосунку і закінчуючи шляхами поліпшення якості проходження гри.

ВИСНОВКИ

В ході розроблення програмного продукту комп'ютерної гри Endless було проаналізовано аналоги та їхні сильні або слабкі сторони, актуальність і перспективи цього продукту, завдяки чому вдалося підібрати та скоригувати процес реалізації.

Розглянуто моделювання загальної поведінки продукту, що забезпечило розуміння очікувань і вимог. Описано структуру продукту, пояснено основні патерни проєктування та їхні аналоги або варіації, показано практики, що застосовуються, відштовхуючись від чого спроєктовано багатошарову архітектуру застосунку, що забезпечує модульність, гнучкість і розширюваність розробленої гри.

Реалізовано та описано гнучкий процес генерації рівнів, логіку переходу між ними, наведено приклади.

Реалізовано інтелектуальну модель поведінки неігрових персонажів, в основі якої лежить підхід поведінкових дерев [28].

Описано інструменти, що використовуються для розробки даного продукту, до яких належить ігровий рушій Unity Engine [1], мова програмування C# [7], середовище розробки Microsoft Visual Studio [20], редактор 3D моделей Blender [19] та редактор для піксель-арт (2D) зображень Aseprite [21].

Проведено автоматизоване та ручне тестування всіх модулів програми, знайдено та усунуто дефекти в роботі функціональної частини програми, проконтрольовано рівень продуктивності та стабільності роботи гри. Створено інструкцію для користувача.

За результатами виконання кваліфікаційної роботи опубліковано тези доповідей в матеріалах V Міжнародної наукової конференції «Сучасний менеджмент організації: витоки, реалії та перспективи розвитку 2025», (м. Київ, 2-3 травня 2025 р.) [24].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Ігровий рушій Unity Engine/ URL: https://unity.com/ (дата звернення 12.03.2025).
2. Комп'ютерна гра ULTRAKILL/ URL: https://store.steampowered.com/app/1229490/ULTRAKILL/ (дата звернення 12.03.2025).
3. Комп'ютерна гра Quake/ URL: https://store.steampowered.com/app/2310/Quake/ (дата звернення 12.03.2025).
4. Комп'ютерна гра Doom Eternal/ URL: https://store.steampowered.com/app/782330/DOOM_Eternal/ (дата звернення 12.03.2025).
5. Платформа цифрової дистрибуції Steam/ URL: https://store.steampowered.com/about/ (дата звернення 12.03.2025).
6. Платформа цифрової дистрибуції Epic Games/ URL: https://www.epicgames.com/site/en-US/about/ (дата звернення 12.03.2025).
7. Платформа Microsoft .NET та мова програмування C#/ URL: https://dotnet.microsoft.com/en-us/languages/csharp/ (дата звернення 12.03.2025).
8. Діаграма патерну машини кінцевих станів/ URL: https://www.toptal.com/unity/unity-ai-development-finite-state-machine-tutorial/ (дата звернення 12.03.2025).
9. Behaviour Tree Design Pattern Example/ URL: https://creately.com/diagram/example/i87iorxo1/behaviour-tree-classic/ (дата звернення 12.03.2025).
10. MVP Design Pattern (Model–view–presenter)/ URL: https://en.wikipedia.org/wiki/Model–view–presenter/ (дата звернення 12.03.2025).

11. Microsoft Windows Operating System/ URL: https://www.microsoft.com/en-us/windows?r=1/ (дата звернення 12.03.2025).
12. Linux Ubuntu (один з найпопулярніших дистрибутивів Linux)/ URL: https://ubuntu.com/ (дата звернення 12.03.2025).
13. MacOS Operating System/ URL: https://www.apple.com/macos/macos-sequoia/ (дата звернення 12.03.2025).
14. Extenject (Zenject) Dependencies Injection Container/ URL: https://assetstore.unity.com/packages/tools/utilities/extenject-dependency-injection-ioc-157735/ (дата звернення 12.03.2025).
15. VContainer Dependencies Injection Container/ URL: https://vcontainer.hadashikick.jp/ (дата звернення 12.03.2025).
16. MVP Design Pattern (Model–view–presenter)/ URL: https://en.wikipedia.org/wiki/Model–view–presenter/ (дата звернення 12.03.2025).
17. Комп'ютерна гра Apex Legends (Respawn Entertainment)/ URL: https://store.steampowered.com/app/1172470/Apex_Legends/ (дата звернення 12.03.2025).
18. Комп'ютерна гра Valorant (Riot)/ URL: https://playvalorant.com/en-us/ (дата звернення 12.03.2025).
19. Blender/ URL: https://store.steampowered.com/app/365670/Blender/ (дата звернення 12.03.2025).
20. Microsoft Visual Studio/ URL: https://visualstudio.microsoft.com/vs/professional/ (дата звернення 12.03.2025).
21. Aseprite/ URL: https://store.steampowered.com/app/431730/Aseprite/ (дата звернення 12.03.2025).
22. Air Stafing explanation/ URL: https://steamcommunity.com/sharedfiles/filedetails/?id=184184420/ (дата звернення 12.03.2025).

23.Комп’ютерна гра BLOW-UP: AVENGE HUMANITY/ URL: https://store.steampowered.com/app/2301940/BLOWUP_AVENGE_HUMANITY/ (дата звернення 12.03.2025).

24.Шмаль Р.С, Поліщук А.О. ОСОБЛИВОСТІ СУЧАСНИХ ПІДХОДІВ ДО РОЗРОБКИ КОМП'ЮТЕРНИХ ІГОР // Збірник матеріалів V Міжнародної наукової конференції «Сучасний менеджмент організації: витоки, реалії та перспективи розвитку 2025», (м. Київ, 2-3 травня 2025 р.) / ВНЗ «Університет економіки та права «КРОК» – Київ, 2025. URL: https://conf.krok.edu.ua/MMO/MMO-2025/paper/view/2902

25.Unreal Engine (Epic Games)/ URL: https://www.unrealengine.com/en-US/ (дата звернення 12.03.2025).

26.Godot Engine/ URL: https://godotengine.org/ (дата звернення 12.03.2025).

27.Патерн Finite State Machine/ URL: https://ocw.mit.edu/courses/6-004-computation-structures-spring-2017/pages/c6/ (дата звернення 12.03.2025).

28.Патерн Behaviour Tree/ URL: https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work/ (дата звернення 12.03.2025).

29.Патерн Singleton/ URL: https://medium.com/@ravipatel.it/understanding-the-singleton-design-pattern-in-c-fdb9ce04d795/ (дата звернення 12.03.2025).

30.Патерн Object Pool/ URL: https://learn.unity.com/tutorial/use-object-pooling-to-boost-performance-of-c-scripts-in-unity?uv=6&projectId=67bc8deaedbc2a23a7389cab#67a0d722edbc2a14cfcb7401/ (дата звернення 12.03.2025).

31.Патерн Dependencies injection/ URL: https://discussions.unity.com/t/design-patterns-singleton-and-dependency-injection/712774/ (дата звернення 12.03.2025).

32. Поняття користувацького інтерфейсу User interface/ URL: https://ocw.mit.edu/courses/6-831-user-interface-design-and-implementation-spring-2011/ (дата звернення 12.03.2025).
33. Алгоритми генерації (Generation algorithms)/ URL: https://cssesw.easyscience.education/cssesw2024/CSSESW2024/paper21.pdf/ (дата звернення 12.03.2025).
34. Configs/ URL: https://www.sciencedirect.com/topics/computer-science/software-configuration/ (дата звернення 12.03.2025).
35. Scriptable object (SO)/ URL: https://docs.unity3d.com/Manual/class-ScriptableObject.html/ (дата звернення 12.03.2025).
36. Значення геймплею (Gameplay)/ URL: https://www.diva-portal.org/smash/get/diva2:835337/FULLTEXT01.pdf/ (дата звернення 12.03.2025).
37. Значення моделі пересування для гри Movement/ URL: https://hopefulhomies.com/2017/02/18/movement-mechanics/ (дата звернення 12.03.2025).
38. Саундтрек в іграх ULTRAKILL/ URL: https://illustratemagazine.com/the-role-of-soundtracks-in-video-games-how-music-enhances-immersive-gaming// (дата звернення 12.03.2025).
39. Неігрові персонажі (NPC)/ URL: https://www.larksuite.com/en_us/topics/gaming-glossary/non-playable-character-npc/ (дата звернення 12.03.2025).
40. Жанри комп’ютерних ігор/ URL: https://pixune.com/blog/video-game-genres/ (дата звернення 12.03.2025).
41. Монетизація комп’ютерних ігор/ URL: https://developer.mozilla.org/en-US/docs/Games/Publishing_games/Game_monetization/ (дата звернення 12.03.2025).
42. Монетизація комп’ютерних ігор/ URL: https://www.gameanalytics.com/blog/traditional-monetization-strategies/ (дата звернення 12.03.2025).