МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Розробка комп'ютерної гри у жанрі Stealth**

Виконав: студент 4 курсу, групи 6.1211-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
Б. А. Горлач
(ініціали та прізвище)
Керівник доцент, к.т.н., доцент В. І. Заяц
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дискус»
Р. О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Т.В. Критська
“ 01 ” березня 2025 року

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

Горлачу Богдану Андрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка комп’ютерної гри у жанрі Stealth

керівник роботи Заяц Валерій Іванович, доцент, к.т.н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 27.12.2024 № 2029-с

2. Строк подання студентом кваліфікаційної роботи 01.06.2025

3. Вихідні дані бакалаврської роботи

- комплект нормативних документів ;
- технічне завдання до роботи.

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

- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми збору та аналізу даних з веб-сторінок;
- створення програмного продукту та його опис;
- дослідження поставленої проблеми та розробка висновків.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
17 слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.03.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів бакалаврської роботи	Примітка
1	Аналіз предметної області	5.03 – 12.03.2025	виконано
2	Формування основної задачі дипломної роботи. Узгодження теми з викладачем.	13.03 – 19.03.2025	виконано
3	Аналіз існуючих ігрових рішень.	20.03 – 26.03.2025	виконано
4	Дослідження можливостей Unity та розширень до нього для розробки гри.	27.03 – 5.04.2025	виконано
5	Визначення базових систем та механік для реалізації в проекті.	6.04 – 9.04.2025	виконано
6	Узгодження дій та подальшої розробки з науковим керівником.	10.04 – 15.04.2025	виконано
7	Розробка гри з системами та механіками, що були визначені та узгодженні.	16.04 – 7.05.2025	виконано
8	Представлення поточних результатів науковому керівнику. Узгодження подальшого плану розробки.	8.05 – 11.05.2025	виконано
9	Виправлення наявних проблем, «багів», створення демонстраційного рівня.	12.05 – 30.05.2025	виконано
10	Оформлення звіту	31.05 – 07.06.2025	виконано

Студент _____ Горлач Б.А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Заяц В.І.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено
Нормоконтролер _____ Скрипник І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок — 67

Рисунків — 12

Джерел — 17

Горлач Б.А. Розробка комп’ютерної гри у жанрі Stealth : кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник В. І. Заяц. Запоріжжя : ЗНУ, 2025. 67 с.

Мета полягає у створенні комп’ютерної гри у жанрі *Stealth*, на основі якої дослідити процес створення ігор, різних механік, способів оптимізації, та штучного інтелекту супротивника.

У процесі розробки була розглянута проблема створення ігор, використовуючи сучасні інструменти. У результаті була створення гра у жанрі *стелс* «Snack Hunt», яка створена з використанням *рушія Unity*. Гра складається з 7 рівнів, де гравець у ролі головного героя гри, єнота, має пробиратися у будинки та викрадати їжу, уникаючи зустрічей з мешканцями будинку. Гра надає можливість взаємодіяти з середовищем та використовувати різні предмети для досягнення кінцевої цілі. Візуально застосунок стилізовано під стиль «*low-poly*» та мальованих коміксів. А завдяки *Unity* та його крос-платформеності, результуючий білд, можна публікувати на різні ігрові крамниці.

Ключові слова*: Stealth(стелс)-гра, Unity, штучний інтелект, оптимізація, ігрові механіки, low-poly.*

ABSTRACT

Pages — 67

Drawings — 12

Sources — 17

Horlach B.A. Development of a Computer Game in the Stealth Genre: bachelor's thesis in specialty 121 "Software Engineering" / Science. manager V. I. Zayats. Zaporizhzhia: ZNU, 2025. 67 p.

The goal is to develop a computer game in the Stealth genre to explore the process of game development, various mechanics, optimization methods, and artificial intelligence for opponents.

During development, the problem of creating games using modern tools was examined. As a result, a stealth game titled "Snack Hunt" was created using the Unity engine. The game consists of seven levels, where the player, taking on the role of a raccoon as the main character, must sneak into houses and steal food while avoiding encounters with the residents. The game allows interaction with the environment and the use of various objects to achieve the ultimate goal.

Visually, the application is designed in a low-poly and hand-drawn comic book style. Thanks to Unity's cross-platform capabilities, the final build can be published on various game stores.

Keywords: *Stealth game, Unity, artificial intelligence, optimization, game mechanics, low-poly*.

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 12
1.1 Огляд літературних джерел 12
1.2 Огляд існуючих ігрових рішень 13
1.2.1 Alien: Isolation 13
1.2.2 Dishonored 15
1.2.3 Thief 16
1.3 Постановка задачі 18
1.4 Висновок до розділу 1 18
2 ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ РОЗРОБКИ 20
2.1 Ігрові рушії 20
2.1.1 Unity 20
2.1.2 Unreal Engine 22
2.1.3 Godot 23
2.2 Вибір технологій 24
2.3 Висновок 24
3 РОЗРОБКА ГРИ З ВИКОРИСТАННЯМ РУШІЯ UNITY 25
3.1 Опис предметної області та мета роботи 25
3.2 Визначення систем для реалізації 25
3.2.1 Пересування гравця 25
3.2.2 Взаємодія гравця з об’єктами 26
3.2.3 Програвання анімацій 26
3.2.4 NPC 26
3.2.5 Вороги 27
3.2.6 Відслідковування стану гри 27
3.2.7 Пошук гравця неігровими персонажами 27
3.2.8 UI 28
3.3 Вимоги до апаратного забезпечення системи 28
3.4 Вимоги до програмного забезпечення системи 28
3.5 Реалізація 29
3.5.1 Пересування гравця 29

3.5.2 Вскарабкування на предмети 33
3.5.3 Взаємодія гравця з об’єктами 33
3.5.3.1 Захоплення предмета 34
3.5.3.2 Кидання предметів 36
3.5.3.3 Взаємодія з об’єктом оточення 37
3.5.4.1 Базовий аніматор 40
3.5.4.2 Аніматор взаємодії з об’єктами оточення 41
3.5.5 NPC 45
3.5.5.1 Автомат станів 45
3.5.5.2 Поведінкові компоненти 49
3.5.6 Вороги 54
3.5.7 Відслідковування стану гри 56
3.5.7.1 Зарахування знайденої їжі 56
3.5.7.2 Кінець гри 57
3.5.7.3 Пауза 58
3.5.8 Пошук гравця неігровими персонажами 58
3.5.9 UI 61
3.5.9.1 Відображення лічильника досвіду та кількості здоров’я 61
3.5.9.2 Візуальні підсказки 62
3.5.9.3 Меню паузи 63
3.6 Висновки до розділу 3 64
ВИСНОВКИ 65
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 66

ВСТУП

Актуальність теми

Ігрова індустрія, на даний момент, є дуже розвиненою та популярною в сфері розважального контенту. Все більше людей знаходять щось своє, та те що припадає їм до душі. З'являються різні нагороди, рейтинги, змагання та різні події присвячені або в цілому цьому напрямку або якійсь конкретній грі. Це відбувається з різних причин, такі як:

Доступність. Багато людей сьогодні мають мобільні пристрої та персональні комп'ютери, з яких можна запустити гру. І це навіть не кажучи за ігрові приставки чи консолі, які спрямовані саме на це галузь.

Прогрес в розробці. З часом, в іграх, все більше почали ставати чимось більше ніж розвага на 5-10 хвилин. Став покращуватись візуал, історії, механіки. Для деяких творів це стало шляхом оповідання своїх творів чи фантазій.

Ігри як культурне явище. Відеоігри стали частиною масової культури. Їх згадування можна зустріти у фільмах, рекламі, музиці і тд..

Різноманіття. Можна зустріти різні проекти: від надважких та гіпер-реалістичних шутерів (жанр комп'ютерних ігор, де гравець використовує стрілецьку зброю) до стилізованих візуальних новел, що порушують питання буття.

Також вагомим, з боку бізнес складової, є прибутковість. Згідно з даними від платформи Statista [1], за 2024 рік заробіток світового ігрового ринку оцінюється майже в 455 мільярдів доларів США. Основними джерелами доходу є продаж ігор, підпискові сервіси, внутрішньо ігрові покупки, реклама та кібер-спорт. Зростаючий попит на ігровий контент, робить сферу розробки ігор є досить перспективною з точки зору інвестування.

Якщо ж розглядати *геймдев* розробку з боку професійного розвитку, то в цілому спеціалісти цієї сфери можуть працюють не тільки над розробкою ігор. Здобуті навички в цій галузі можна використати в різних напрямках,

наприклад, у кіноіндустрії для створення візуальних ефектів, графіки, процедурних анімацій; в освіті для розробки інтерактивних навчальних програм і симуляторів, з використанням віртуальної та доповненої реальності.

Можливості для незалежних розробників також значно розширилися. Сьогодні завдяки платформам на кшталт Steam, itch.io, Google Play та App Store навіть невеликі команди або окремі ентузіасти можуть випускати власні ігри та знаходити свою аудиторію. Це робить ігрову розробку доступнішою та сприяє появі нових унікальних ідей у цій сфері.

Загалом, сама по собі сфера розробки ігор, розвивається та є досить актуальною галуззю в розробці програмного забезпечення.

Досліджуючи жанр *stealth* (в перекладі українською – *скритність*), можна дізнатись що його елементи використовується у різних іграх в поєднані з різними механіками та іншими жанрами. Це пов’язано з особливостями взаємодії гравця з ігровим простором та створення варіативності проходження, що дарує свій досвід кожному хто проходить той чи інший рівень локації гри.

Unity в свою чергу є досить популярним серед безкоштовних ігрових рушіїв. В ньому вже реалізований різний функціонал, що спростить розробку механік, роботи з графікою, анімаціями, візуальними ефектами тощо. Окрім цього має активну спільноту розробників, постійно оновлюється та вдосконалюється.

Підсумовуючи все, можна сказати, що розробка ігор є актуальною темою в сфері розваг. Вона не лише забезпечує користувачів якісним контентом, але й відкриває широкі можливості для творчості. Індустрія продовжує еволюціонувати, пропонуючи ще більше інноваційних підходів та технологій до створення контенту. Це створює попит, а відповідно захоплює все більше уваги людей та різних компаній, які побажають інвестувати в цю сферу.

Мета дослідження

Метою дослідження є аналіз та створення комп’ютерної гри, що поєднує елементи прихованого проходження, штучного інтелекту супротивників та інтерактивного ігрового середовища, способи оптимізації, і вивчення методів та підходів до розробки.

Завдання дослідження

Аналіз існуючих рішень та шляхів створення гри. Вивчення принципів *гейм-дизайну*, з метою розуміння побудови цікавого та затагуючого *геймплею*; ігрових механік; рушія Unity та методів оптимізації в ньому. Розробка гри, яка зможе забезпечити захопливий ігровий досвід.

Об’єкт дослідження

Об’єктом дослідження є процес розробки гри жанру *стелс*.

Предмет дослідження

Предметом дослідження є вивчення принципів створення комп’ютерних ігор з використанням рушія Unity. Аналіз сучасних популярних ігор, для врахування вподобань гравців.

Методи дослідження

Аналіз літератури та порівняння вже існуючих рішень, що стосуються методів та розбору створення ігор; особливостей Unity; *гейм-дизайну*; огляд популярних ігрових механік, з метою розуміння які можуть стати корисними для кінцевої гри; апробація кінцевих результатів.

Практичне значення одержаних результатів

Результати дослідження є комп’ютерна гра, яку в свою чергу можна використати як приклад для розробки проектів, з елементами жанру *стелс* із

застосуванням рушія Unity. Також отриманні напрацювання можуть стати корисним початківцям у сфері *геймдеву*, які прагнуть удосконалити свої навички.

Зібрані дані та шляхи вирішення різних проблем можуть бути адаптовані для інших проектів, що дозволить ефективніше використовувати Unity для створення власних проектів.

Окрім цього, дослідження сприяє популяризації *інді-розробки* та демонструє можливості використання сучасних технологій у створенні ігрового контенту.

Глосарій

Геймдев (*GameDev*) – скорочення від "game development" (розробка ігор), що охоплює всі аспекти створення відеоігор, від ідеї до фінального продукту.

Гейм-дизайн (*Game Design*) – процес створення правил, механік, рівнів і загальної концепції гри для забезпечення цікавого ігрового досвіду.

Геймплей (*Gameplay*) – ігровий процес, тобто взаємодія гравця з грою, її механіками та середовищем.

Інді-розробка (*Indie Development*) – створення ігор незалежними розробниками або малими студіями без підтримки великих видавців.

Стелс (*Stealth*) – жанр або механіка гри, що передбачає приховане проходження рівнів, уникнення ворогів і використання хитрощів замість прямого бою.

NPC (Non-Player Character) – персонаж у грі, поведінка якого визначається програмно і не може контролюватися гравцем на пряму.

Місії – внутрішньо-ігрові завдання, які також можуть бути і не обов’язковими, для проходження гри.

Ассети – різні ресурси гри: моделі, медіа, код, текстури та інше.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд літературних джерел

Розробка комп’ютерних ігор є складним і багатоаспектним процесом, що охоплює різні дисципліни, включаючи програмування, дизайн та штучний інтелект. Для створення якісного продукту необхідно спиратися на наукові праці, технічну документацію та досвід професійних розробників.

Враховуючи стрімкий розвиток ігрової-індустрії, під час пошуку літератури, значна увага приділялась до актуальності методик та підходів до розробки. Оскільки технології та способи до створення ігор постійно оновлюються, особливо важливо враховувати сучасні дослідження, що охоплюють інноваційні рішення у сфері гейм-дизайну, оптимізації продуктивності та покращення ігрового досвіду.

Літературні джерела з геймдеву містять інформацію про основи створення механік. У дослідженнях з ігрової теорії та дизайну аналізуються принципи занурення гравця у віртуальний світ та методами балансування складності.

Зокрема, була досліджена, література до створення ігор з використанням рушія Unity. Серед розглянутих тем, основними є: підходи до створення механік, роботі з графікою та візуальними ефектами, шляхами оптимізації гри.

Основні теми, аналізу літературних джерел:

1. Опис рушія Unity, його функціоналу та можливостей у розробці ігор [2, 3].
2. Аналіз популярних ігрових механік та патернів проектування. [4, 5, 8].
3. Штучний інтелект в іграх [6].
4. Геймдизайн, особливості ігрового процесу та балансу [5, 7, 8].

В свою чергу жанр *стелс* не є досить складним в реалізації, з боку програмування, але стає проблемним у створенні умов для нього в самій грі.

Більшість механік використовуються і в інших жанрах, і не є унікальними, тому їх можна легко створити чи навіть знайти вже готові рішення. Аналіз існуючих рішень дасть зрозуміти, які саме є основні особливості жанру, для створення списку його базових складових, за яким в подальшому можна буде вести роботу.

1.2 Огляд існуючих ігрових рішень

Вся суть обраного жанру *стелс* полягає у прихованому проходженні гри, уникаючи прямого контакту та конфронтації з ворогами чи *NPC*.

Здебільшого в сучасних іграх можна зустріти саме елементи цього жанру, чи окремі *місії*, які орієнтуються на ньому. Хоча також є і проекти що засновуються саме на такому типі.

1.2.1 Alien: Isolation

Події гри *Alien Isolation* [12], від розробників **Creative Assembly**, розгортаються в всесвіті серії фільмів "*Чужий*", де головна героїня Аманда Ріплі, потрапивши на космічну станцію, стає свідком жахливих подій. Корабель захоплений прибульцем, який несе загрозу для людей, стає пасткою для всіх кому не вдалось його покинути, а ті що вижили зійшли з розуму, та також несуть небезпеку.

Гра базується на жанрах хоррор-екшен та *стелсом*, і вимагає від гравця обдумувати кожен свій крок, щоб не спровокувати ворогів та пройти повз них для досягнення кінцевих цілей. Для цього гравцю надається можливість користуватись різними пристроями та середовищем, щоб відслідковувати та ховатись від потенційної небезпеки, приклад зображено на рисунку 1.1.



Рисунок *1.1 – Геймплей Alien Isolation*

Основні елементи гри, які відповідають за механіки прихованого проходження:

1. Можливість ховатись від ворогів: в темних кутках, під столами, в шафах чи вентиляції.
2. Використання різних пристроїв для відволікання: фаєри, димові шашки, звукові генератори та інші.

Механіки що полегшують чи допомагають у проходженні:

1. Озброєння – дає змогу позбутись від деяких ворогів.
2. Відслідковування – завдяки пристроям чи звукам гравець може дізнатись де знаходиться ворог, та побудувати стратегію до проходження локації.

Умови що вимагають від гравця грати за правилами:

1. Дефіцит спорядження – на локаціях більше ворогів чим набоїв до зброї чи якихось предметів для відволікання.
2. Неможливість позбутись головного ворога – прибулець, «чужий», не отримує пошкоджень та є головною проблемою гравця. Він здатен вільно пересуватись по всій локації, на якій знаходиться гравець, та чути його дії.

1.2.2 Dishonored

Події гри *Dishonored* [13], розробленої студією Arkane Studios, відбуваються у вигаданому місті Дануолл, натхненному вікторіанською Англією з елементами стімпанку. Головний герой, Корво Аттано, несправедливо звинувачений у вбивстві імператриці, і стає втікачем, прагнучі помсти та очищення свого імені.

Гра базується на поєднанні жанрів *стелс*, пригодницького *екшену* та *RPG*, де рішення гравця напряму впливають на сюжет, світ гри та її фінал. Завдяки широкому набору здібностей і інструментів, гравець має повну свободу у виборі стилю проходження. *Геймплей* наведено на рисунку 1.2.



Рисунок 1.2 – Геймплей Dishonored.

Основні елементи гри, які відповідають за механіки прихованого проходження:

1. Використання архітектури локації: пересування по дахах, карнизах, вентиляційних шахтах, в затемнених місцях і за перешкодами для укриття.
2. Приховані атаки ззаду, що дозволяють нейтралізувати ворога без шуму.

Механіки, що полегшують чи допомагають у проходженні:

1. Надприродні здібності, які можна покращувати чи заміняти.
2. Використання різних видів озброєння.
3. Можливість вивчення розкладів ворогів і обхід маршрутів патрулювання.

Умови, що вимагають від гравця грати за правилами:

1. Система хаосу – чим більше ворогів вбито, тим важче стає світ гри (з’являються більше щурів, хворих NPC, змінюється фінал).
2. Обмежені ресурси – боєприпаси, еліксири мани та здоров'я є обмеженими.
3. Розумні вороги, що викликають підкріплення при виявленні гравця.
4. Рейтингове завершення місій – чим не помітніше, тим краще оцінюється проходження.

1.2.3 Thief

Події гри *Thief* [14], випущеної у 2014 році від студії Eidos-Montréal, відбуваються у вигаданому місті, у якому гравець бере на себе роль Гаррета — досвідченого крадія, який повертається в місто, охоплене хворобами, безладом і диктатурою. Його мета — розкрити таємниці влади й збагачуватись, виконуючи різні завдання, пов’язані з крадіжками, проникненням у заборонені зони та пошуком цінних предметів.

Гра належить до жанрів стелс та екшен-адвенчур, з акцентом на обережне переміщення, уникання ворогів і збирання ресурсів. Гравець має широкий вибір інструментів і можливостей, щоб проходити рівні без бою, повністю покладаючись на прихованість. Головна особливість – свобода дій та багато способів досягнення цілей. Один із прикладів *геймплею* показано на рисунку 1.3.



Рисунок 1.3 – Геймплей Thief

Основні елементи гри, що реалізують механіки прихованого проходження:

1. Система освітлення — гравець повинен уникати яскравих ділянок і залишатись у тіні, щоб залишатись невидимим.
2. Система звуку — гравець повинен уникати шуму при ходьбі або дії, особливо на скрипучих поверхнях.

Механіки, що полегшують або допомагають у проходженні:

1. Інстинкт крадія (focus mode) — тимчасовий режим, що дозволяє краще бачити інтерактивні об’єкти та ворогів.
2. Система апгрейдів — можливість покращувати спорядження й здібності для кращого стелс-проходження.

Умови, що змушують гравця дотримуватись правил прихованості:

1. Розумні вороги — відкритий бій зазвичай швидко призводить до смерті, вороги мають добру реакцію та якщо ворог підозрює щось, він кличе інших, і вся локація стає агресивною.

2. Дефіцит ресурсів — стріли, інструменти та ліки трапляються нечасто, що стимулює економити.

1.3 Постановка задачі

Метою кваліфікаційної роботи, є створення гри з елементами жанру *стелс*, на основі рушія Unity. Постановка задачі включає наступні етапи:

1. Аналіз потреб і вимог. Визначення основних потреб та вимог до гри, які повинні відповідати опису обраному жанру. Це включає підтримку різних механік, розробки штучного інтелекту, та оптимізації кінцевого проекту.
2. Розробка архітектури, що буде об’єднувати компоненти гри між собою. Вона має бути гнучкою та підтримуваною, щоб в майбутньому була можливість до розширення та удосконалення.
3. Розробка механік гри. Реалізація основних ігрових механік які визначаються жанром *стелс*: відслідковування шуму, гравця, штучний інтелект та інші; та базові, які є в будь-якій грі: керування, збереження, виконання квоти і тд..
4. Створення рівнів, що об’єднують в собі та використовують створенні механіки, демонструючи їх функціонал. Також забезпечити зміну складності гри в залежності від рівню.
5. Апробація. Виконання перевірок коректності роботи кінцевих результатів: механік, рівнів, тощо.

1.4 Висновок до розділу 1

1. Було проведено огляд проблем створення ігор, а також базових механік і штучного інтелекту до них.

2. Проаналізовано існуючи ігри обраного жанру та визначено особливості та механіки, що відіграють побудові стелс складової у них.
3. Сформульовано постановку задачі для кваліфікаційної роботи.

2 ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ РОЗРОБКИ

2.1 Ігрові рушії

Комп’ютерна гра, з точки зору програмування, може бути сумісністю різних компонентів, але абстрактно їх можна поділити на такі частини:

1. Візуалізація – графіка, анімації, візуальні ефекти, UI і тд.
2. Ігрова логіка – механіки, скрипти чи будь-який інший код, що описує поведінку гри.
3. Взаємодія з пристроєм користувача – зчитування введення, збереження даних, тощо.

І для всіх них є різні методи реалізація, в залежності від вимог до розробки.

Існують багато фрейворків, які дозволяють працювати з описаними вище окремо, додаючи зручний інтерфейс та кросплатформеність системи. Але найчастіше обирають ігрові рушії, що об’єднують в собі всі переваги, та пришвидшують розробку проектів та нерідко мають в собі розповсюдженні механіки.

Серед популярних є: Unity, Unreal Engine, Godot; і щоб обрати один з них, треба проаналізувати, який базовий функціонал вони надають, що зможе знадобитись у розробці.

2.1.1 Unity

Unity [15] – кросплатформений ігровий рушій, створений компанією Unity Technologies, що призначений для розробки 2D, 3D, AR, VR проектів, для десктопних, консольних, мобільних та Web застосунків, в основі якого лежить компонента архітектура.

Здобув значну популярність в мобільній розробці та серед *інді-розробників*, завдяки простоті розробки та можливостям до кастомізації.

Рушій реалізує системами фізики, анімації, аудіо та UI, що допомагає у створенні проектів. Також серед додаткових є також і такі види механік та розширень:

- **Unity AI –** серед головних варто визначити **Unity NavMesh** та **Unity Begavior**. Перше розширення надає можливість побудови навігаційної сітки, з можливістю її динамічної зміни. Друге в свою чергу надає графічний інтерфейс для реалізації NPC, створюючи граф його поведінки.
- **Netcode –** базовий фреймворк рушія для створення багатокористувацьких ігор, що підтримує синхронізацію станів, RPC (Remote Procedure Calls), авторитету об'єктів та реплікації.
- **Timeline –** розширення для створення кат-сцен, що дозволяє візуально створювати послідовності подій у реальному часі: анімації, зміни станів, виклик подій, тощо.
- **VFX Graph –** графічний редактор візуальних ефектів, що засновано на системі частинок з широкими можливостями: колізії, турбулентність, залежність від фізики та інші.
- **Shader Graph –** графічний редактор шейдерів, що заснований на ShaderLab, фреймворку Unity для підтримки кросплатформеності з поєднанням HLSL та GLSL.
- **Visual Scripting –** візуальна система, для створення скриптів. Полегшує роботу новачкам чи для створення прототипів.

Також однією з основних переваг Unity є велика та активна спільнота розробників, які створюють власний навчальний контент, що знижує поріг входження в ігрову індустрію. Окрім того, Unity має велику кількість власних навчальних матеріалів, що робить рушій доступним як для новачків, так і для досвідчених розробників.

2.1.2 Unreal Engine

Unreal Engine [16] — кросплатформений ігровий рушій, створений компанією Epic Games, що використовується для розробки 2D, 3D, AR, VR проектів під мобільні, десктопні, консольні та Web-платформи. В основі рушія лежить концепція акторів і компонентів, які керуються ієрархією класів у C++.

Unreal Engine здобув популярність завдяки фото-реалістичному рендерингу, розширеній підтримці C++ та великій кількості готових інструментів. Найчастіше використовується у високо-бюджетній AAA-розробці, але також підтримує інді-проєкти.

Рушій містить вбудовані системи фізики, анімації, звуку та UI. Крім того, є низка додаткових механік і розширень:

- **AI** — рушій реалізує **Behavior Tree**, систему поведінки для NPC з використанням blackboard-пам’яті та дерева завдань. Також підтримується **Navigation Mesh**, динамічна побудова маршрутів та зон.
- **Multiplayer/Networking** — Unreal має реплікацію на рівні ядра, підтримку клієнт-серверної моделі, RPC та авторитет сервера, що дозволяє будувати масштабні мережеві ігри з мінімальними додатковими залежностями.
- **Sequencer** — візуальний редактор для створення кат-сцен, анімацій, кінематографічного контролю камери, аудіо, скриптів. Підтримує keyframe-анімацію і треки подій.
- **Niagara VFX** — система візуальних ефектів нового покоління, яка дає змогу створювати складні ефекти з урахуванням фізики, колізій, симуляцій та GPU-обчислень.
- **Material Editor** — візуальний редактор матеріалів, що дозволяє створювати складні PBR-матеріали без написання HLSL вручну.
- **Blueprints** — візуальна система програмування, що дозволяє створювати логіку гри без кодування. Має повну інтеграцію з C++, дає змогу швидко створювати прототипи або повноцінні системи.

Серед ключових переваг Unreal Engine — відкритий код ядра рушія на C++, що дає змогу кастомізувати будь-яку підсистему під власні потреби. Також рушій підтримує найновіші технології: Lumen (глобальне освітлення у реальному часі), Nanite (віртуалізована геометрія).

2.1.3 Godot

Godot Engine [17] — це повністю відкритий, безкоштовний і кросплатформений ігровий рушій, який підтримує розробку 2D, 3D, мобільних, десктопних, консольних та Web-проєктів.

Рушій набуває дедалі більшої популярності завдяки своїй легкій вазі, гнучкості, вбудованому редактору коду, кросплатформеності та повністю відкритому коду під ліцензією MIT, що дозволяє модифікувати рушій під власні потреби без будь-яких обмежень.

Рушій має вбудовану підтримку 2D та 3D графіки, фізики, UI, анімації, аудіо, системи сцен і сигналів. Також існує низка додаткових або частково реалізованих механік:

- **AI та навігація** — рушій містить підтримку **NavigationServer**, що дозволяє генерувати навігаційні сітки, знаходити шляхи, і динамічно змінювати навігаційні області. Є також **NavigationAgent** для базового NPC переміщення. Розширення **Godot AI Toolkit** додає поведінкові дерева та FSM.
- **Мережева система** — рушій реалізує multiplayer API з підтримкою RPC) авторитету, синхронізації станів, peer-to-peer або client-server архітектури, з підтримкою WebSocket, WebRTC та ENet.
- **Visual Shader Editor** — візуальний редактор шейдерів, який дозволяє створювати GLSL-шейдери у вигляді графа. Працює як для 2D, так і для 3D, з підтримкою додаткового коду.
- **Visual Scripting** — вбудований візуальний редактор логіки на основі вузлів, який орієнтований на новачків або швидке створення прототипів.

Godot підходить як для *інді-розробки,* прототипування, так і для повноцінних комерційних проектів. Версія *Godot 4.x* принесла значні покращення в 3D графіці, продуктивності, та рендерингом (зокрема *Vulkan*).

2.2 Вибір технологій

Згідно, з наведеним в попередньому пункті, аналізом існуючих ігрових рушіїв – сучасні рішення мають схожі можливості, і здебільшого, особливо для *інді-розробки*, мають вирішальне значення саме зручність для розробника. В моєму випадку, для мене пріоритетним є Unity, завдяки досвіду роботі з ним у минулому.

Тому, для виконання кваліфікаційної роботи, я обрав наступні технології: ігровий рушій Unity та мова програмування C#.

Серед додаткових технологій є розширення **DOTween** [11], яке призначене для створення динамічних анімацій з коду, та **PlayableGraph Visualizer** [9] для візуалізації змішування динамічно доданих анімацій.

2.3 Висновок

1. Досліджено основні компоненти та технології, з яких складається гра.
2. Досліджено сучасні ігрові рушії, та їх функціонал.
3. Обрано стек, на якому буде реалізовано кваліфікаційну роботу.

3 РОЗРОБКА ГРИ З ВИКОРИСТАННЯМ РУШІЯ UNITY

3.1 Опис предметної області та мета роботи

Предметна область, цієї кваліфікаційної роботи, включає в себе розробку ігрових механік та подальшого їх використання, для побудови гри на базі Unity. Дана предметна область в собі поєднує використання навичок програмування, розробки з урахуванням особливостей рушія, та роботі з графікою.

Дипломна робота зосереджується на розробці гри, а саме на реалізації та застосуванні механік, що притаманні для жанру *стелс*, де особлива увага приділяється реалізації механік прихованого пересування, взаємодії з оточенням, виявлення гравця штучним інтелектом, реакцій NPC, а також надання гравцю свободи дій у межах рівня. В свою чергу обраний рушій, Unity, забезпечує широкий функціонал для створення різних проектів. До того ж має низький поріг входження, якісну графіку, різні оптимізації, та можливість створювати ігри під різні платформи.

Також, під час виконання роботи, будуть розглянуті шляхи розробки, включно з архітектурними та функціональними рішеннями.

Основною метою цієї кваліфікаційної роботи – це покращення навичок програмування та знань у галузі розробки ігор, а заразом створення комп’ютерної гри з використанням Unity.

3.2 Визначення систем для реалізації

3.2.1 Пересування гравця

Гра має надавати можливість рухатись по ігровому простору, зчитуючи введення з пристрою користувача.

В свою чергу система має передбачувати такі характеристики:

- Забезпечення всебічного руху (комбінація руху по сторонах та стрибок).
- Варіація швидкості пересування: біг, сповільнений, нормальний.
- Можливість піднятися на об’єкти, що дозволить діставатись високих місць, на які звичайного стрибка може не вистачати.

Створення цієї механіки, дасть змогу гравцю пересуватись по рівню, досліджувати локацію, та вирішувати поточні місії гри.

3.2.2 Взаємодія гравця з об’єктами

Взаємодія з ігровим простором, не тільки зможу оживити гру та зробити її більш реалістичною.

Серед можливих варіацій, система може підтримувати такі функціональні можливості:

- Підбір і використання предметів (ресурси, чи різні об’єкти).
- Активація механізмів або тригерів (кнопки, двері чи механізми).

Реалізація цієї механіки дозволяє гравцю більш глибоко зануритися у світ гри та впливати на її стан, задаючи умови вирішення рівню.

3.2.3 Програвання анімацій

В грі необхідно відслідковувати поточний стан об’єктів та персонажів, щоб програвати їх анімації. Тоді персонажі будуть виглядати більш живими.

Також треба забезпечити динамічне програвання анімація, для візуалізації взаємодії з різними предметами.

3.2.4 NPC

Щоб в грі було від кого ховатись, необхідно додати персонажів, які в свою чергу будуть знаходитись на локації, і шукати гравця, якщо на це буде необхідність. До них в свою чергу треба розробити автомат-станів, який буде описувати їх поведінку в грі.

3.2.5 Вороги

На основі базових NPC, також треба реалізувати ворогів, які стануть головною проблемою гравця, і від якого буде важче сховатись. Вони будуть мати змогу атакувати та, на відміну від звичайних неігрових персонажів, активно обшукувати локацію та швидше реагувати на дії гравця.

Така варіація спонукатиме гравця до стратегічного мислення та продумувати стратегію проходження, в залежності від рівня.

3.2.6 Відслідковування стану гри

Ця система повинна буде здебільшого очікувати активацій тригерів гри, які будуть означати якусь подію гри, наприклад:

- Програш – якщо гравця часто помічали NPC або ловили вороги.
- Пауза – коли гравець викликає меню натискає певну клавішу.
- Перемога – коли всі місії рівня були виконанні і гравець вийшов з локації.

3.2.7 Пошук гравця неігровими персонажами

Щоб змусити NPC чи ворога шукати гравця, треба якось повідомити їх про його присутність. Це в свою чергу реалізується за допомогою звичайного патрулювання чи списку дій персонажа по локації, у певних частинах локації де був зареєстрований шум, який буде створюватись різними подіями, наприклад:

- Шум від гравця – біг чи стрибок.
- Взаємодія з предметами – звук від відкривання дверей чи повороту камери для хатніх тварин.
- Заскриптовані події.

Також слід реалізувати чи попадає гравець в поле зору персонажа, та чи нема перешкод перед ним, які будуть заважати.

3.2.8 UI

Система має відповідно до поточного стану гри та гравця відображати необхідну інформацію через елементи інтерфейсу.

Очікуваний функціонал:

- Перемикання в та з меню паузи.
- Відображення поточної кількості здоров’я.
- Візуальні підказки взаємодії.

3.3 Вимоги до апаратного забезпечення системи

Під час розробки та апробації, було визначено, що мінімальними системні вимогами, при яких гра може видавати комфортний fps, та на яких відбувалась перевірка – це:

1. Процесор - Intel Pentium 6405U 2.40GHz.
2. Оперативна пам’ять - 1 ГБ на 2400 МГц.
3. Графічний процесор - NVIDIA MX 130 2 ГБ.
4. Кількість вільної пам’яті на HDD або SSD – 300 МБ.

3.4 Вимоги до програмного забезпечення системи

Вимоги до програмного забезпечення є наступними, і складались на основі тої системи та того кінцевого білда, на яких проводилась апробація результатів:

1. Операційна система Windows 10 x64 і вище.
2. Для графічних процесорів Nvidia драйвера версії 555.99 і вище.

3.5 Реалізація

3.5.1 Пересування гравця

Для реалізації керування гравцем в Unity застосовувались наступний функціонал рушія:

1. Input System – вбудоване розширення, яке надає інтерфейс для визначення системи зчитування введення та подій у разі активації.
2. CharacterController – компонент, що визначає інтерфейс для пересування, без застосування компонента Rigidbody, що відповідає за фізику об’єкта.

Перш за все варто зазначити, що сам компонент CharacterController, був обраний через те, як саме відбувається пересування об’єкта з ним, порівняно з Rigidbody, а саме:

1. CharacterController зміщує об’єкт в просторі, натомість Rigidbody штовхає в заданому напрямку. Це може вплинути на зручність керування, бо в обраному варіанті, рух буде завжди передбачуваний, без урахунку накопиченої сили на персонажа.
2. В CharacterController можна задати характеристику персонажа, такі як: висота, ширина, максимальний кут нахилу руху, та ширину зміщення кроку.

А враховуючи що їх комбінація може викликати конфлікти в оброці всередині рушія, то в коді буде також реалізована симуляція притяжіння.

Також варто зазначити, що підписки на події Input System будуть задані в інспекторі Unity, де зазначається об’єкт, компонент та метод що будуть викликатись, і виглядатимуть наступним чином, як на рисунку 3.1.

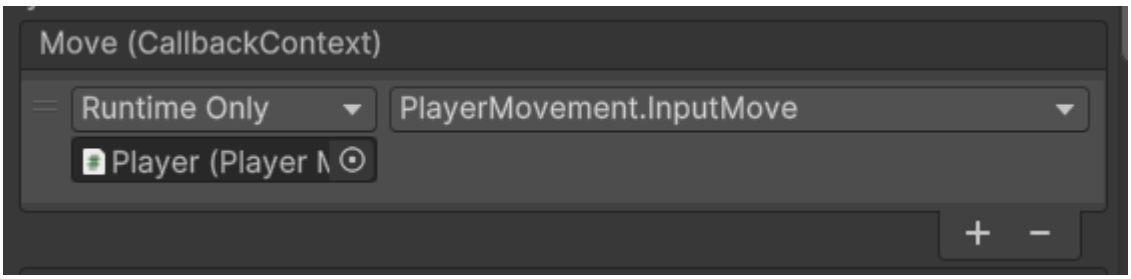


Рисунок 3.1 – Приклад підписки на подію введення в інспекторі

Сам клас компонента PlayerMovement має наступні методи зчитування введення, які можна побачити в Лістинг 1.

Лістинг 1 *Обробка подій введення*:

```
    public void InputMove(InputAction.CallbackContext
context)
    {
        if (_isMoveToTarget)
            return;

        _isInput = context.performed;

        _inputDirection = context.ReadValue<Vector2>();
    }

    public void Jump(InputAction.CallbackContext context)
    {
        if (IsCanJump == false || context.started == false)
            return;

        PrepareOnJump();
        Invoke("UnlockCamera", Time.deltaTime);
    }

    public void
SetSilentWalkMode(InputAction.CallbackContext context)
    {
        _isSilentWalk = context.performed;
    }

    public void SetSprintMode(InputAction.CallbackContext
context)
    {
        _isSprint = context.performed;
    }
```

```
    public void
InputUnlockCamera(InputAction.CallbackContext context)
    {
        if (_playerState.IsThrowing)
            return;

        _isLockedCameraDirection = context.canceled;

        _cameraBufferDirection = new
CameraDirectionBuffer(_camera.forward, _camera.right);
    }
```

Всього методів зчитування 5, а саме для: руху, стрибка, повільної ходьби, бігу та блокування камери.

Сам рух обробляється за допомогою методів Update, FixedUpdate, що успадковані від класу MonoBehaviour, який базовим класом, від якого наслідують всі компоненти рушія, які потребують постійного оновлення разом з грою. Також допоміжним є метод OnControllerColliderHit, що зчитує колізію з іншими колайдерами, та з якого беруться дані про площину.

Реалізація обробки руху представлена в Лістинг 2.

Лістинг 2 *Обробка руху гравця*:

```
    private void
OnControllerColliderHit(ControllerColliderHit hit)
    {
        float angle = Vector3.Angle(hit.normal, Vector3.up);

        SetSurfaceNormal(angle, hit);
        ResetVerticalVelocityOnHit(angle);
        AdjustJumpMoveSpeedOnHit(angle);
    }

    private void FixedUpdate()
    {
        if (_playerState.IsPaused)
            return;

        ApplyGravity();
    }

    private void Update()
    {
```

```
        if (_playerState.IsClimbing ||
_playerState.IsPaused)
            return;

        if (_isJump || IsCanMove)
            ProcessingMovement();
        else
            Move(Vector3.zero, CurrentSpeed);
    }
```

Алгоритм руху виглядає так:

1. Зчитування даних про нормаль поверхні та обробка зіштовхненя.
2. Застосування гравітації.
3. Обробка руху.

Сам метод Update також має умову, що якщо рух не може відбутись то відбувається рух за вектором Vector3.zero, або ж простіше кажучи – нульовим вектором. Такий підхід потрібен на випадок якщо гравець падає і не відбувається ввід, щоб персонаж продовжував падати, а не зупинявся в повітрі.

Обробка руху, має наступний алгоритм, реалізація якого наведено в Лістинг 3:

1. Розрахунок вектору руху, в залежності від повороту камери чи цілі.
2. Рух з заданому напрямку.
3. Поворот в бік напрямку.

Лістинг 3 *Головний метод руху:*

```
    private void ProcessingMovement()
    {
        bool hasTarget = _targetPosition.HasValue;

        Vector3 moveDirection = hasTarget == false ?
                        CalculateMoveDirection() :

CalculateMoveDirection(_targetPosition.Value -
MainTransform.position);

        if (hasTarget)
            MoveToTarget(moveDirection);
```

```
        else
            Move(moveDirection, CurrentSpeed);

        RotateToDirection(moveDirection);
    }
```

Повна реалізація в додатку, пункт PlayerMovement.cs.

3.5.2 Вскарабкування на предмети

Також серед умов була реалізація функціоналу підняття на високі об’єкти. За це буде відповідати компонент Climber, який в свою чергу, при виконанні умов (наявності перешкод, вільного для гравця простору), буде викликати подію для виклику анімації, що при завершенні буде пересувати гравця в кінцеву точку.

Методи що виконують цей функціонал наведені в Лістинг 4.

Лістинг 4 *Методи підняття*:

```
private void MakeClimb()
{
    if (IsCanClimb == false)
        return;

    OnClimb?.Invoke();
    IsClimbing = true;
    SetPosition();
}

public void ClimbFinished()
{
    _mainTransform.position = _finalPosition;
    _root.localPosition = Vector3.zero;
    IsClimbing = false;

    OnClimbFinished?.Invoke();
}
```

3.5.3 Взаємодія гравця з об’єктами

Загалом в грі кодом визначено 4 види взаємодії гравця, які зображенні на діаграмі рисунку 3.2.



Рисунок 3.2 – UML діаграма взаємодії гравця в грі.

За ці дії відповідають наступні класи:

1. ObjectGrabber (підняти або кинути на землю предмет).
2. ObjectThrower (кидати предмет в напрямку камери).
3. PlayerInteraction (взаємодія з об’єктом оточення).

3.5.3.1 Захоплення предмета

Для реалізації захоплення знадобилось визначити додатковий компонент Item, який в собі зберігає інформацію про позиціювання об’єкта в просторі. А для знаходження був створений компонент ObjectFinderByMask, який в свою чергу зчитує входження в простір тригерної зони та викликає події, на які відбувається підписка для подальшої взаємодії. Детальніше в додатку за пунктами Item.cs та ObjectFinderByMask.cs.

В самому коді під час захоплення отримується дані про предмет, якому встановлюється батьківський об’єкт, за позицією якого він буде слідкувати. А для того щоб виконати зворотне, треба повернути налаштування до попереднього стану.

Також, під час захоплення, встановлюється дані для інверсної кінематики, а саме про позиціювання рук на об’єкті, які під час програвання анімацій будуть наче тримати об’єкт.

Методи що реалізують описаний функціонал наведені в Лістинг 5.

Лістинг 5 *Захоплення об’єкта*:

```
    private void OnEnable()
    {
        _itemFinder.OnObjectDetect += FindObject;
        _itemFinder.OnObjectLost += LostObject;
    }

    private void OnDisable()
    {
        _itemFinder.OnObjectDetect -= FindObject;
        _itemFinder.OnObjectLost -= LostObject;
    }

    public bool TryTakeObject()
    {
        if (_foundItem == null || Item != null)
            return false;

        Item = _foundItem.GetComponent<Item>();
        Grab(Item);
        DisableRigidbody(Item.Rigidbody);

        return true;
    }

    public void SetNewPlace(Transform newPlace)
    {
        DropFromHands();
        Item.transform.SetParent(newPlace);
    }

    public void DropObject()
    {
        if (Item == null || _playerState.IsOnGround ==
false)
            return;

        SetNewPlace(null);
        EnableRigidbody(Item.Rigidbody);
        Item = null;
```

```
        OnDrop?.Invoke();
    }
```

3.5.3.2 Кидання предметів

Алгоритм такої взаємодії полягає у таких кроках:

1. Перехід в режим прицілювання, якщо це можливо.
2. Кинути піднятий предмет в напрямку камери.

Для реалізації написано два метода, що підписуються на події InputSystem, та поділяються на підготовку до кидка, та сам кидок. Їх код наведено в Лістинг 6.

Лістинг 6 *Зчитування введення для кидка об'єкта*:

```
public void GetReadyToThrow(InputAction.CallbackContext
context)
{
    if (_playerState.IsOnGround == false)
        return;
    if (Application.isFocused == false ||
_objectGrabber.IsHoldingObject == false)
    {
        Cancele();
    }
    else if(context.performed)
    {
        IsReadyToThrow = true;

        OnPrepareToThrow?.Invoke();
    }
    else if(context.canceled)
    {
        Drop();
    }
}

public void ThrowObject(InputAction.CallbackContext
context)
{
    if (context.canceled && IsReadyToThrow)
    {
        Cancele();
```

```
            PrepareToThrow();
            OnThrow?.Invoke();
            IsThrowing = true;
        }
}
```

В свою чергу після виклику події кидка, програється анімація, в якій викликається метод, який наведено в Лістинг 7, що і буде застосовувати силу до компонента Rigidbody, для кидка об’єкту.

Лістинг 7 *Кидання об’єкту, в напрямку камери*:

```
    public void ThrowToCameraDirection()
    {
        if (_objectGrabber.Item == null)
            return;

        Rigidbody itemBody = _objectGrabber.Item.Rigidbody;
        _objectGrabber.DropObject();

        itemBody.freezeRotation = false;
        itemBody.linearVelocity = Vector3.zero;
        itemBody.angularVelocity = Vector3.zero;
        itemBody.AddForce(_throwDirection * _throwStrength,
ForceMode.Impulse);
    }
```

3.5.3.3 Взаємодія з об’єктом оточення

Сама реалізації цієї механіки передбачає повторне використання і для NPC, тому в собі буде містити один базовий клас, від якого будуть успадковуватись інші компоненти для розширення функціоналу. Те, що буде містити головний клас, та хто буде успадковуватись від нього, зображено на рисунку 3.3.

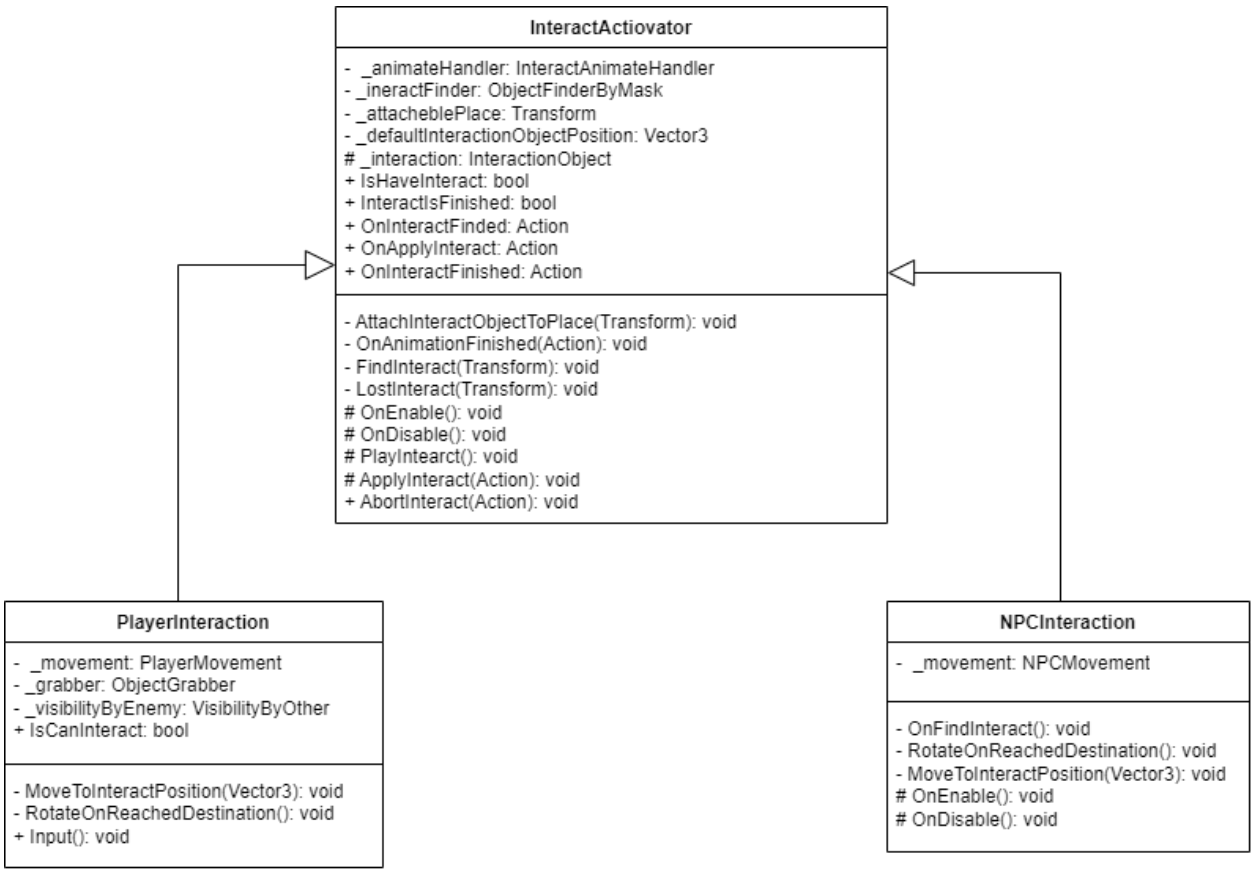


Рисунок 3.3 – Діаграма класів взаємодії з об’єктами оточення

Реалізація цієї механіки полягає у находженні об’єктів з компонентами InteractObject, які в свою чергу зберігають дані про позиціювання та список анімацій, що будуть програватись.

У випадку з PlayerInteraction, після знаходження та введенням, перед програваманням анімації герой буде пересуватись до точки взаємодії, та після цього буде активуватись сама дія. Код цієї обробки наведено у Лістинг 8, а деталі реалізації батьківського компонента у додатку, в пункті InteractActivator.cs.

А у випадку з NPC, потреба у введенні відсутня, програвання дії спрацьовує при знаходженні автоматично, детальніше у додатку, в пункті NPCInteraction.cs.

Лістинг 8 *Обробка взаємодії з об’єктами оточення*:

```
private void MoveToInteractPosition(Vector3 position)
{
```

```
        _movement.MoveTo(position,
RotateOnReachedDestination);
    }

    private void RotateOnReachedDestination()
    {
        _movement.MainTransform
            .DORotate(_interaction.InteractRotate, 1f)
            .OnComplete(base.PlayIntearct);
    }


    public void Input()
    {
        if (IsCanInteract == false)
            return;

        if (_interaction.IsActive)
            base.AbortInteract(null);
        else
            base.ApplyInteract(() =>
MoveToInteractPosition(_interaction.InteractPosition));
    }
```

3.5.4 Програвання анімацій

Серед основних систем анімацій є наступні:

1. Аніматор гравця.
2. Аніматор NPC.
3. Аніматор ворогів.
4. Аніматор UI.

Вони в свою чергу працюють через управління параметрів автомату станів анімацій, що реалізується компонентом Animator, приклад візуалізації якого наведено на рисунку 3.4.

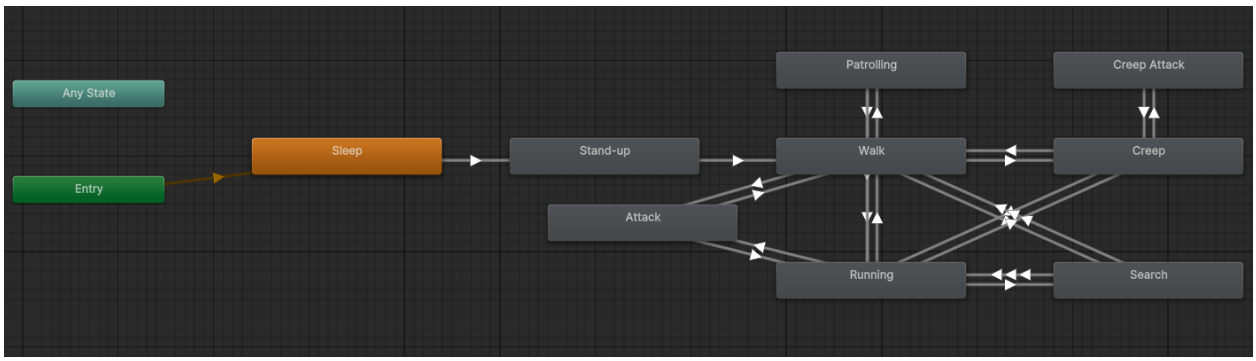


Рисунок 3.4 – Автомат станів аніматора ворога.

А враховуючи схожість написання компонентів, буде розглянуто лише NPCAnimateHandler та InteractAnimateHandler, а всі інші, разом з зазначеними, будуть додані в додаток, в пунктах UI, Гравець, NPC/Вороги.

3.5.4.1 Базовий аніматор

Для зміни переходу від однієї анімації до іншої, а Animator викликаються методи встановлення вказаних в ньому полів. Так в компоненті NPCAnimateHandler, в методі Update, код якого наведено в Лістинг 9.

Лістинг 9 *Код аніматора NPC*:

```
    private void OnEnable()
    {
        _playerDetecter.OnPlayerDetect +=
SetPlayerPosition;
        _playerDetecter.OnPlayerLost +=
ResetPlayerPosition;
    }


    private void OnDisable()
    {
        _playerDetecter.OnPlayerDetect -=
SetPlayerPosition;
        _playerDetecter.OnPlayerLost -=
ResetPlayerPosition;
    }


    private void Update()
    {
        float magnitude = _movement.Velocity.magnitude;
```

```
        float normalizedVelocity = Mathf.Clamp01(magnitude
/ _movement.CurrentSpeed);

        _animator.SetFloat("Velocity", normalizedVelocity,
0.1f, Time.deltaTime);

        if (_playerPosition.HasValue)

_headRotator.RotateHeadTowards(_playerPosition.Value);
    }


    private void SetPlayerPosition(Vector3 position)
    {
        _playerPosition = position;
    }


    private void ResetPlayerPosition(Vector3 position)
    {
        _playerPosition = null;
        _headRotator.LostTarget();
    }
```

Також, як видно з вище наведеного коду, під час активації події знаходження гравця зберігається його позиція, в бік якої в подальшому, в методі Update, повертається голова персонажа за допомогою компонента HeadRotator, код якого в додатку в пункті NPC/Вороги, Анімування.

3.5.4.2 Аніматор взаємодії з об’єктами оточення

За динамічне програвання анімацій відповідає компонент InteractAnimateHandler, що створено з використанням Playable API [10], що дозволяє створювати граф змішування анімацій, який в свою чергу може виступати як надбудова над базовим компонентом Animator.

Для ініціалізації, код якої наведено в Лістинг 10, необхідно створити сам PlayableGraph, який стане вихідною точкою всіх анімацій, та міксери, за допомогою яких і будуть змішуватись станами. Після чого створюються головний міксер, `_topLevelMixer`, до якого будуть під’єднанні аніматор та змішувачі для динамічних анімацій.

Лістинг 10 *Ініціалізація InteractAnimateHandler*:

```
    private void Start()
    {
        _playableGraph =
PlayableGraph.Create(transform.gameObject.name);

        AnimationPlayableOutput playableOutput =
AnimationPlayableOutput.Create(_playableGraph, "Animation",
_animator);
        _topLevelMixer =
AnimationMixerPlayable.Create(_playableGraph, 3);
        playableOutput.SetSourcePlayable(_topLevelMixer);

        _animatorPlayable =
AnimatorControllerPlayable.Create(_playableGraph,
_controller);
        _topLevelMixer.ConnectInput(0, _animatorPlayable,
0);

        _playableGraph.GetRootPlayable(0).SetInputWeight(0,
1f);
        _playableGraph.Play();
        GraphVisualizerClient.Show(_playableGraph);
    }
```

В результаті, за допомогою розширення PlayableGraph Visualizer [9], можна буде відкрити вікно візуалізації графа, яке зображене на рисунок 3.5. А у відкритому вікні буде зображено, який стан графу, що до нього підключено, та який поточний шлях анімації йде на вихід.

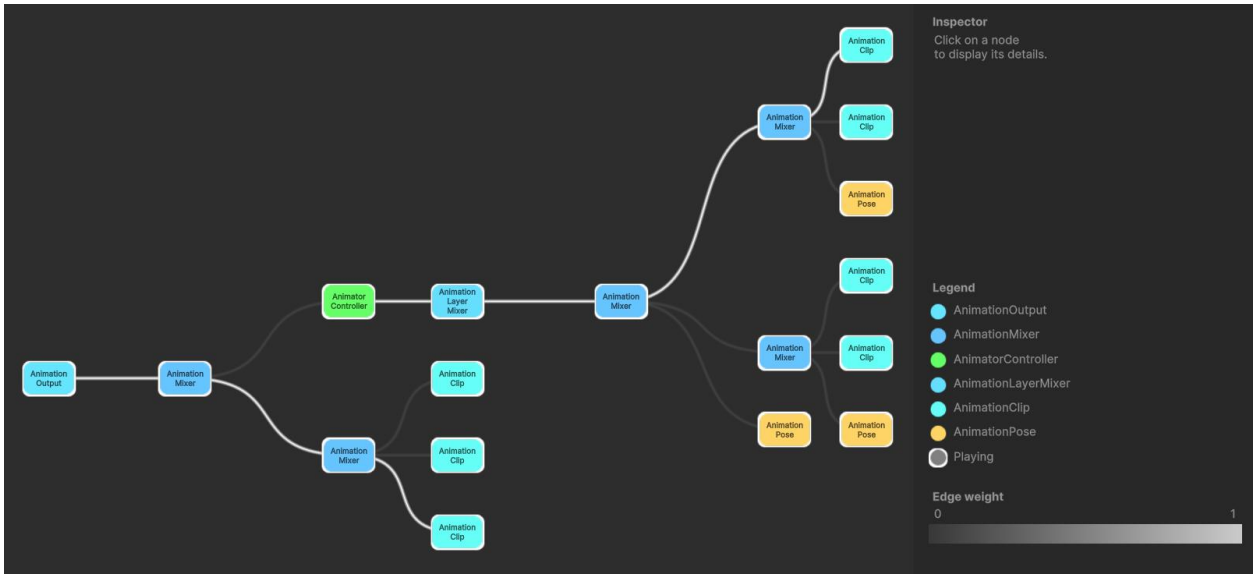


Рисунок 3.5 – Візуалізація графа анімацій

Якщо говорити про саме змішування анімацій, то воно відбувається зміною ваги підключення до міксера, і виконується двома методами MakeBlend та Blend в компоненті , що наведені у Лістинг 11.

Лістинг 11 *Методи змішування*:

```
    private IEnumerator Blend(float duration, Action<float>
blendCallback, float delay = 0f, Action callback = null)
    {
        if (delay > 0f)
            yield return new WaitForSeconds(delay);

        float blendTime = 0f;
        while (blendTime < 1f)
        {
            yield return new WaitUntil(() => !_isPaused);

            blendTime += Time.deltaTime / duration;
            blendCallback(blendTime);
            yield return blendTime;
        }

        blendCallback(1f);

        callback?.Invoke();
    }


    private void MakeBlend(AnimationMixerPlayable mixer,
int indexFrom, int indexTo,float weightEnd, float duration,
float delay = 0, Action callback = null)
    {
        float weightStart = weightEnd > 0 ? 0 : 1f;

        StartCoroutine(Blend(duration, (blendTime) => {
            float weight = Mathf.Lerp(weightStart,
weightEnd, blendTime);
            mixer.SetInputWeight(indexFrom, weight);
            mixer.SetInputWeight(indexTo, 1f - weight);
        }, delay, callback));
    }
```

Варто зазначити, що метод Blend, є корутиною, та запускається методом MonoBehaviour.StartCoroutine. Вона буде працювати паралельно з оновленням

кадрів рушія, та реалізують через IEnumerator, для повернення часу очікування.

А для програвання списку анімацій використовується інша корутина, PlaySequance, що буде додавати до міксера нові анімації, яку наведено в Лістинг 12.

Лістинг 12 Метод *PlaySequance*:

```
    private IEnumerator PlaySequance(AnimationMixerPlayable
mixer, int inputIndexFrom, int inputIndexTo, InteractClip[]
clips, bool automaticEndOfAnimation = true, Action callback
= null)
    {
        _animationInProcessing = true;
        float blendDuration =
CallculateBlendDuration(clips[0]);
        float outDelay = clips[0].Clip.length -
blendDuration * 2;

        AddClipOnMixer(clips[0], mixer);
        mixer.SetInputWeight(0, 1f);
        MakeBlend(_topLevelMixer, inputIndexFrom,
inputIndexTo, 0f, blendDuration);

        for (int i = 1; i < clips.Length; i++)
        {
            blendDuration =
CallculateBlendDuration(clips[i]);

            float t = 0f;
            while (t < outDelay)
            {
                yield return new WaitUntil(() =>
!_isPaused);
                t += Time.deltaTime;
                yield return null;
            }

            AddClipOnMixer(clips[i], mixer);
            MakeBlend(mixer, i - 1, i, 0f, blendDuration);

            outDelay = clips[i].Clip.length - blendDuration
* 2;
        }
```

```
        if (automaticEndOfAnimation)
        {
            MakeBlend(_topLevelMixer, 0, inputIndexTo, 1f,
blendDuration, outDelay, callback);
        }
    }
```

Завдяки компоненту InteractAnimateHandler,

Детальна реалізація обговорюваного компонента наведено у додаток, за пунктом Інше, Взаємодія, InteractAnimateHandler.cs.

3.5.5 NPC

3.5.5.1 Автомат станів

Створення неігрових персонажів полягає у написані автомату станів, що буде описувати їх поведінку у грі.

В своєму варіанті реалізації я маю перерахунок можливих станів та клас, з методами, для отримання та збереження інформації про переходи. Вони наведені у Лістинг 13.

Лістинг 13 *Перерахунок та функціоналі методи станів*:

```
enum NPCStateType
{
    Idle,
    Patrol,
    Search,
    Attack
}
```

```
        public void AddTransition(NPCState state, Func<bool>
condition, Action onTransition = null)
        {
            _transitionDictionary.Add(state, condition);
            _addedTransitionStates.Add(state);

            if (onTransition != null)
                _transitionEvents.Add(state, onTransition);
        }
```

```
        public bool CheckTransition(NPCState nextState)
        {
            if(_transitionDictionary.ContainsKey(nextState))
                return
_transitionDictionary[nextState].Invoke();

            return false;
        }


        public void InvokeTransitionEvent(NPCState
nextState)
        {
            _transitionEvents[nextState]?.Invoke();
        }
```

Ці методи класу стану, відповідають за наступне:

1. Додавання переходів – використовується як метод для ініціалізації, з зазначенням умови за колбеку, який зберігається в словнику.
2. Перевірка – має метод що повертає bool значення, чи виконанні умови переходу.
3. Переключення – метод що викликає збережений колбек за переданим станом.

Як видно цей клас не зберігає в собі чітких даних про те, який поточний стан, але це і не потребується. Він буде використовуватись як споміжний об’єкт, для зберігання даних про переходи.

Вся інша логіка лежить вже на автоматі станів, для прикладу буде оглядатись компонент NPCStateMachine, який буде перевіряти умови і при спрацьовуванні переключати необхідні компоненти, код ініціалізації якого наведено у Лістинг 14, а схема переходів на рисунку 3.6.

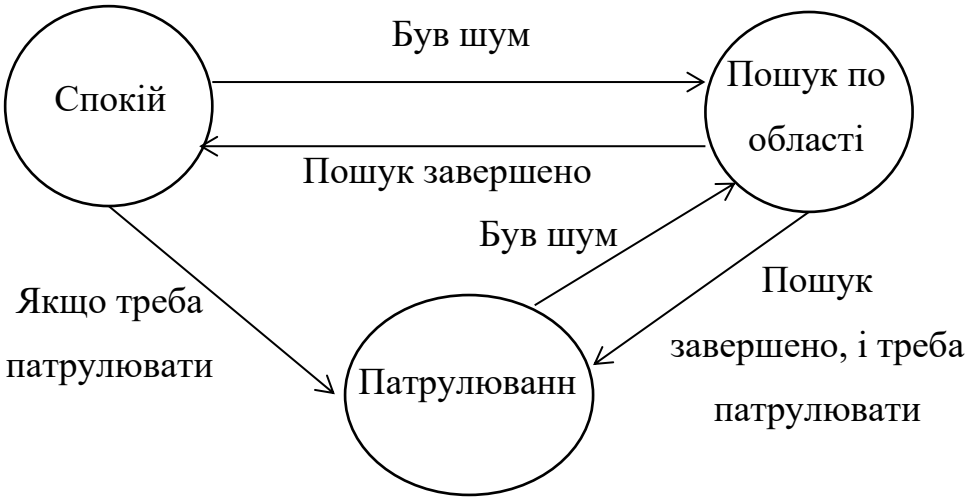


Рисунок 3.6 – Схема автомату станів NPC

Лістинг 14 *Ініціалізація автомату станів*:

```
    private void Awake()
    {
        InitializeStatesArray();
        InitializeStates();

        _currentState =
_enemyStates[(int)NPCStateType.Idle];
    }

    private void InitializeStatesArray()
    {
        _enemyStates = new NPCState[COUNT_STATES];

        for (int i = 0; i < COUNT_STATES; i++)
            _enemyStates[i] = new NPCState();
    }

    private void InitializeStates()
    {
        AddTransition(NPCStateType.Idle,
NPCStateType.Search, () => _isDetectedNoise, IdleToSearch);
        AddTransition(NPCStateType.Idle,
NPCStateType.Patrol, () => _needInteraction, ToPatroll);
        AddTransition(NPCStateType.Search,
NPCStateType.Idle, () => _isSearchingEnd &&
!_needInteraction, SearhToIdle);
```

```
        AddTransition(NPCStateType.Search,
NPCStateType.Patrol, () => _isSearchingEnd &&
_needInteraction, ToPatroll);
        AddTransition(NPCStateType.Patrol,
NPCStateType.Search, () => _isDetectedNoise,
PatrolToSearch);
    }

    private void AddTransition(NPCStateType from,
NPCStateType to, Func<bool> condition, Action onTransition)
    {
        NPCState currentState = _enemyStates[(int)from];
        NPCState nextState = _enemyStates[(int)to];

        currentState.AddTransition(nextState, condition,
onTransition);
    }
```

Коли сцена тільки запускається – викликається метод Awake, саме він, як видно з вище наведеного коду, і відповідає за створення станів. Враховуючи що перерахунок можна інтерпретувати як ціле число, я його використовував для індексації станів у масиві. Завдяки чому код став інтуїтивно зрозумілим, з якого стану в який піде далі обробка.

Також в Лістинг 14 є підписка на події різних компонентів, саме в переданих методах, змінюються поля що перевіряються у лямбда функціях, які і є перевірочними для переходу при додаванні станів.

Перевірка стану та умов переходу відбувається в методі Update, що наведено у Лістинг 15. В ньому викликається метод SwitchState, який і перевіряє умови та викликає колбек, для перемикання компонентів нового стану.

Лістинг 15 *Перевірка переходу у наступний стан*:

```
    private void Update()
    {
        if (_isFindPlayer)
            return;
        SwitchState();
    }
```

```
    private void SwitchState()
    {
        for (int i = 0; i < COUNT_STATES; i++)
        {
            NPCState enemyState = _enemyStates[i];

            if (_currentState.CheckTransition(enemyState))
            {

_currentState.InvokeTransitionEvent(enemyState);
                _currentState = enemyState;
                break;
            }
        }
    }
```

Також в методі Update є одна умова, що буде зупиняти перехід у разі спрацювання, чи знайдено гравця. Вона була додана з внутрішньо ігрових домовленостей, що у разі знаходження гравця нейтральним NPC, відбудеться відмотка часу, та перемикатись між станами не треба.

Повна код класів, згаданих у цьому пункті, наведено у додаток, в частині NPC/Вороги, Поведінка.

3.5.5.2 Поведінкові компоненти

Рух

Завдяки функціоналу Unity, реалізація руху по площині не є такою складною задачею. Його надає пакет UnityEngine.AI, який в свою чергу містить компоненти для створення навігаційної сітки та агентів (NavMeshSurface та NavAgent), що зможуть пересуватись по визначеній області, приклад наведено на рисунку 3.7.

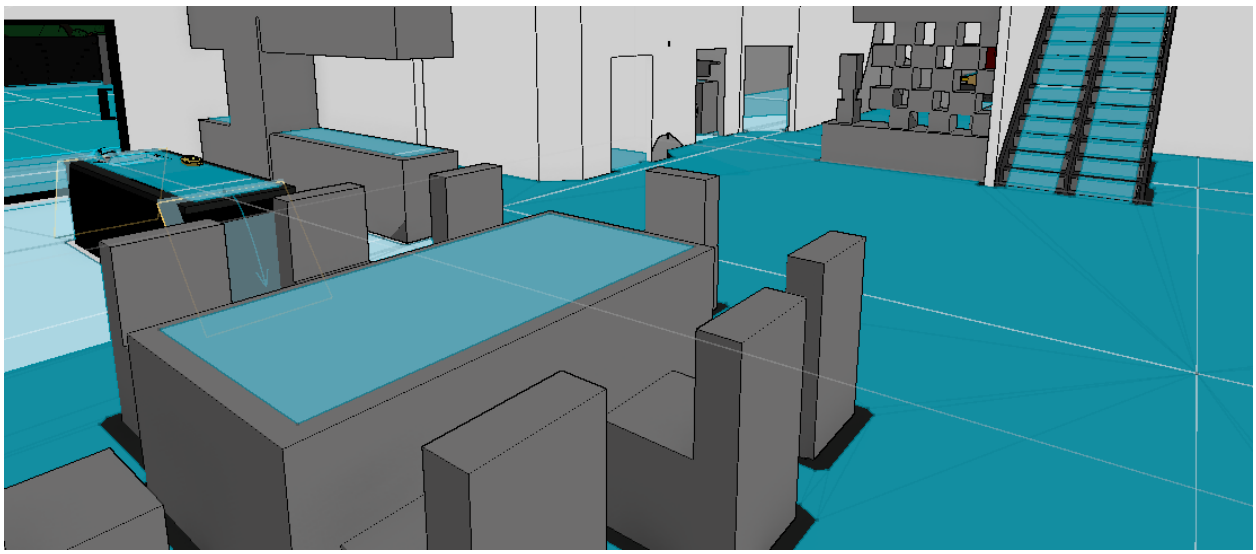


Рисунок 3.7 – Приклад навігаційної сітки (зображено блакитним кольором).

В моєму варіанті, компонент NPCMovement, має наступний функціонал, код якого наведено в Лістинг 16:

1. Встановлення пункту призначення.
2. Зробити ривок.
3. Зміна типу руху (нормальний, біг, тихий).

Лістинг 16 *Базовий функціонал компонента NPCMovement*:

```
    public float CurrentSpeed =>
                IsSprinting ? _sprintSpeed :
                IsCreepWalking ? _creepWalkSpeed :
                _moveSpeed;

    private void Update()
    {
        if (IsReachedDestination)
            return;

        Vector3 direction = _navAgent.steeringTarget -
CurrentPosition;

        if(CanMove == true && _isDashing == false)
            MoveToTarget(_lastTargetPosition,
CurrentSpeed);
        RotateToDirection(direction);

        IsReachedDestination = IsPathComplet();
    }
```

```
    private void MoveToTarget(Vector3 target, float speed)
    {
        _navAgent.speed = speed;
        _navAgent.SetDestination(target);
    }

    private IEnumerator DashCoroutine(Vector3
targetPosition)
    {
        _isDashing = true;
        MoveToTarget(targetPosition, _dashSpeed);

        yield return new WaitForSeconds(_dashCooldown);
        _isDashing = false;
    }

    public void SetMovementMode(NPCMovementMode
movementMode)
    {
        IsCreepWalking = movementMode ==
NPCMovementMode.CreepWalk;
        IsSprinting = movementMode ==
NPCMovementMode.Sprint;
    }
```

Працює це все наступним чином:

- Коли встановлюється пункт призначення, в методі Update, в компоненті NavAgent встановлюється поточна швидкість та позиція, куди треба рухатись.
- Метод ривка зупиняє звичайний рух, та встановлює свої дані, скидаючи блокування через певний час.
- Зміна типу руху визначається через перерахунок та застосовується через властивість.

Повна реалізація додано в додаток, пункт NPC/Вороги, Поведінка.

Патрулювання та пошук в областях.

В цілому, задача пошуку по місцевості, зводиться до переміщення персонажа по точках, що і реалізує компонент Patroller, а пошук по області вже

реагує на певний тригер, і при початку пошуку отримує список точок, які треба пройти. Тому враховуючи це, компонент SearcherInSpace, спадкується від Patroller, код якого наведено у Лістинг 17, для розширення функціоналу.

Лістинг 17 *Компонент патрулювання*:

```
private void Update()
 {
     if (_isSearching == false || _isSwitchingPoint)
         return;

     MoveToNextPoint();
 }

 private Vector3 GetNextPosition()
 {
     Vector3 nextPosition =
_patrolWaypoints[_currentIndex].position;
     _currentIndex = (_currentIndex + 1) %
_patrolWaypoints.Length;

     return nextPosition;
 }

 protected virtual void SwitchToNextTarget()
 {
     Vector3 nextPoint = GetNextPosition();
     _isSwitchingPoint = false;

     if (_isSearching)
         _enemyMovement.SetTarget(nextPoint);
 }

 private void MoveToNextPoint()
 {
     Vector3 position = _enemyMovement.transform.position;

     if (_enemyMovement.IsReachedDestination == false)
         return;

     OnReachingWaypoint?.Invoke();
     _isSwitchingPoint = true;
     Invoke("SwitchToNextTarget", _switchDelay);
 }
```

В свою чергу точки, які має пройти персонаж, встановлюються з інспектора Unity.

А от вже компонент SearcherInSpace сам встановлює необхідні точки пошуку, і відбувається це наступним чином:

1. Спрацьовує детектор шуму, та передає через подію позицію куди треба рухатись.
2. Автомат станів NPC переключається на потрібний стан та вмикає обробку у SearcherInSpace.
3. Компонент шукає область, серед можливих збережених, сортуючи по дистанції до точки шуму.
4. Після завершення обшуку компонент зупиняє обробку та повідомляє про це через подію.

Код наведеного алгоритму обробки SearcherInSpace наведено у Лістинг 18, а повний код цього та Patroller компонентів додано до додатку, в пунктах NPC/Вороги, Поведінка.

Лістинг 18 *Обробка пошуку по області*:

```
    private void AddSearchPosition(Vector3 position)
    {
        if (_isSearching)
            return;

        _noisePosition = position;
    }


    private SearchingSpace FindSearchingSpace(Vector3
noisePosition)
    {
        SearchingSpace closestSpace = _searchingSpaces
            .OrderBy(space => space.GetPoints()
                .Min(point =>
Vector3.Distance(point.position, noisePosition)))
            .FirstOrDefault();

        return closestSpace;
```

```
    }

    private void SortPointsByDistance(Transform[] points)
    {
        Array.Sort(points, (a, b) =>
        {
            float distA = Vector3.Distance(a.position,
_noisePosition);
            float distB = Vector3.Distance(b.position,
_noisePosition);

            return distA.CompareTo(distB);
        });
    }

    public override void StartSearch()
    {
        SearchingSpace searchingSpace =
FindSearchingSpace(_noisePosition);

        _patrolWaypoints = searchingSpace.GetPoints();
        SortPointsByDistance(_patrolWaypoints);

        _currentPointIndex = 0;

        base.StartSearch();
    }
```

3.5.6 Вороги

Для ворогів додається ще один стан атаки. Схема зображена на рисунку 3.8.

Оновлений автомат станів, має додатковий компонент – EnemyAttack. Він відслідковую поточну позицію гравця та шум, який спровокований його діями, та рухається до нього.

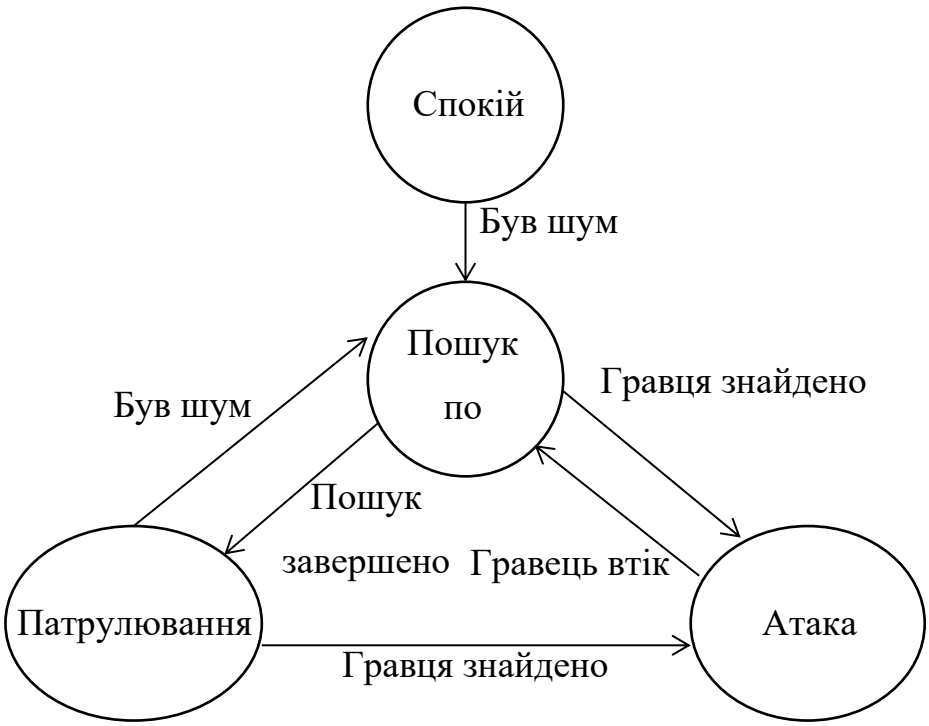


Рисунок *3.8 – Схема автомату станів ворога*

Оновлену ініціалізацію станів, для компонента EnemyStateMachine, наведено у Лістинг 19.

Лістинг 19 *Ініціалізація станів ворога*:

```
    private void InitializeStates()
    {
        AddTransition(NPCStateType.Idle, NPCStateType.Search, () =>
_isDetectedNoise, IdleToSearch);
        AddTransition(NPCStateType.Search, NPCStateType.Patrol, () =>
_isSearchingEnd && !_isPlayerDetected, SearhToPatrol);
        AddTransition(NPCStateType.Search, NPCStateType.Attack, () =>
!_isSearchingEnd && _isPlayerDetected, SearchToAttack);
        AddTransition(NPCStateType.Patrol, NPCStateType.Attack, () =>
_isPlayerDetected, PatrolToAttack);
        AddTransition(NPCStateType.Patrol, NPCStateType.Search, () =>
_isDetectedNoise, PatrolToSearch);
        AddTransition(NPCStateType.Attack, NPCStateType.Search, () =>
!_isPlayerDetected, AttackToSearch);
    }
```

Сам компонент EnemyAttack отримує позицію гравця завдяки компонентам PlayerDetector та NoiseDetector, які будуть розглянуті в наступних пунктах. Також має події, що сповіщають інші компоненти про

події атаки чи втрати гравця, що дає змогу переключитись до наступного стану.

Більш детальна реалізація додана в додаток, пункт NPC/Вороги, Поведінка.

3.5.7 Відслідковування стану гри

В цьому проекті за відслідковування стану гри слідкують наступні компоненти:

- FoodCollector – нараховує бали досвіду, за знайдену їжу по локації.
- GameOverTrigger – тригер, що відслідковує завершення гри, чи то гравцем самостійно чи то через програш.
- EndGameHandler – обробник завершення гри.
- PauseHandler – обробник паузи.

3.5.7.1 Зарахування знайденої їжі

Компонент FoodCollector реалізується через зчитування об’єктів, що потрапили в певне поле дії компонента ObjectFinderByMask, який знаходить усі предмети, що були позначені маскою їжі.

Під час спрацьовування події знаходження, отримуються дані про умовну вартість об’єкта, та нараховується до лічильника, код наведено у Лістинг 20.

Лістинг 20 *Метод нарахування досвіду за знайдену їжу*:

```
private void DetectFood(Transform food)
{
    Item foodInfo = food.GetComponent<Item>();

    if (_collector.Contains(foodInfo))
        return;

    _counter += foodInfo.ExpPoint;

    _collector.Add(foodInfo);
    OnCounterChange?.Invoke(_counter);
```

```
}
```

3.5.7.2 Кінець гри

Компонент EndGameHandler залежить від GameOverTrigger, і по суті єдиним хто сповіщає про завершення гри, після чого програється анімації виграшу чи програшу.

Сам тригер відслідковує наступне:

- Чи не закінчилось життя в гравця.
- Чи не намагається гравець покинути локацію.
- Чи не вийшов випадково гравець за можливу йому територію.

А враховуючи побудову головного рівня, будь яка спроба буде вимагатиме від гравця перелазити паркан чи знаходити як перестрибнути його, цей тригер спрацює в будь-якому з випадків.

Але залишається лише питання з програшом гравця. Щоб виконати умови програшу, треба щоб гравець залишився без здоров'я, а воно віднімається у наступних випадках:

1. Попастися на очі нейтральному персонажу.
2. Дати схопити себе ворожому персонажу.

Також, щоб гравець не втрачав швидко одиниці здоров'я, після кожної невдачі відбувається «відмотка часу». В грі це реалізовано в TimeRewinder, що знаходить усі об'єкти на яких був доданий компонент RecordingObject. А запис відбувається через об'єкт Recorder, який в свою чергу зберігає в собі всі останні позиції, які були в проміжку визначеного часу, реалізація наведена в Лістинг 21.

Лістинг 21 *Методи запису та видалення останніх позицій*:

```
    private void RecordPositions()
    {
        List<RecordingInfo> recordingPosition = new
List<RecordingInfo>();
```

```
        foreach (var obj in _recordingObjects)
            recordingPosition.Add(new RecordingInfo(obj,
obj.RecordPosition()));

        _recordingInfos.Enqueue(recordingPosition);
    }


    private void RemoveLastPositions()
    {
        if (_recordingTime >= _recordingDuration)
        {
            _recordingInfos.Dequeue();
            _recordingTime -= Time.deltaTime;
        }
    }
```

Якоїсь унікальної реалізації в EndGameHandler нема, він лише викликатиме певну анімацію та відправляє гравця до головного екрану гри.

3.5.7.3 Пауза

Ця системи паузи складається з одного компоненту та інтерфейсу: PauseHandler та IPause, який містить в собі два методи: Pause та Resume.

Кожен компонент, що реалізує IPause, сам визначає як він буде ставити себе на паузу та повертатись до обробки. Але сам перехід до зупинки гри може відбуватись по різних причинах, а саме:

1. Гравець сам натиснув на кнопку паузи.
2. Для програвання анімацій поразки.

3.5.8 Пошук гравця неігровими персонажами

До компонентів що входять до цієї категорії відносяться два наступні: PlayerDetecter (візуальний пошук) та NoiseDetecter (пошук шумів).

Якщо розбирати те як працює PlayerDetecter, то полягає він у перевірці вірності двох умов:

1. Чи входить гравець в поле зору.
2. Чи нема перешкод, за якими знаходиться гравець.

Таким чином, для перевірки потрібно два методи, що наведені в Лістинг 22.

Лістинг 22 *Методи пошуку*:

```
public bool SearchByRays(Transform target)
    {
        Vector3 rayPosition =
_searchSettings.rayPosition.position;
        Vector3 direction = (target.position -
rayPosition).normalized;
        float maxDistance = _searchSettings.rayDistance;
        float halfAngle =
_searchSettings.searchForwardRaysAngle / 2f;
        float angleStep =
_searchSettings.searchForwardRaysAngle /
(_searchSettings.rayCount - 1);

        for (int i = 0; i < _searchSettings.rayCount; i++)
        {
            float angle = -halfAngle + i * angleStep;
            Vector3 rayDirection = Quaternion.Euler(0,
angle, 0) * direction;

            float targetDistance =
GetDistanceToObjectFromRay(rayPosition, rayDirection,
maxDistance, _searchSettings.targetMask);
            float obstacleDistance =
GetDistanceToObjectFromRay(rayPosition, rayDirection,
maxDistance, _searchSettings.obstacleMask);

            if (targetDistance < obstacleDistance)
                return true;
        }
        return false;
    }

    public bool SearchByDotProduct(Transform transform,
Transform target)
    {
        Vector3 forwardByRays = _searchSettings.rayPosition
? _searchSettings.rayPosition.forward : transform.forward;

        Vector3 toTarget = target.position -
transform.position;
        float dotPruduct = Vector3.Dot(forwardByRays,
toTarget.normalized);
```

```
        float verticalAngle =
Vector3.Angle(toTarget.normalized, forwardByRays);

        return dotPruduct >= _searchSettings.dotThreshold
&& verticalAngle <= _searchSettings.searchVerticalAngle;
    }
```

Реалізація пошуку за полем зору, метод SearchByDotProduct, полягає у розрахунку скалярного добутку векторів: напрям до цілі та вперед персонажа; яке буде повертати значення від -1 до 1. І якщо поріг цього значення пройдено, то гравця буде видно.

А от в випадку з перевіркою на перешкоди застосовується трасування об’єктів. Але, враховуючи що одним промінь не здатен захопити всю ширину пошуку, застосовується декілька напрямків в бік цілі, як зображено на рисунку 3.9, для того щоб гравця знаходили, навість якщо його тільки частково видно.

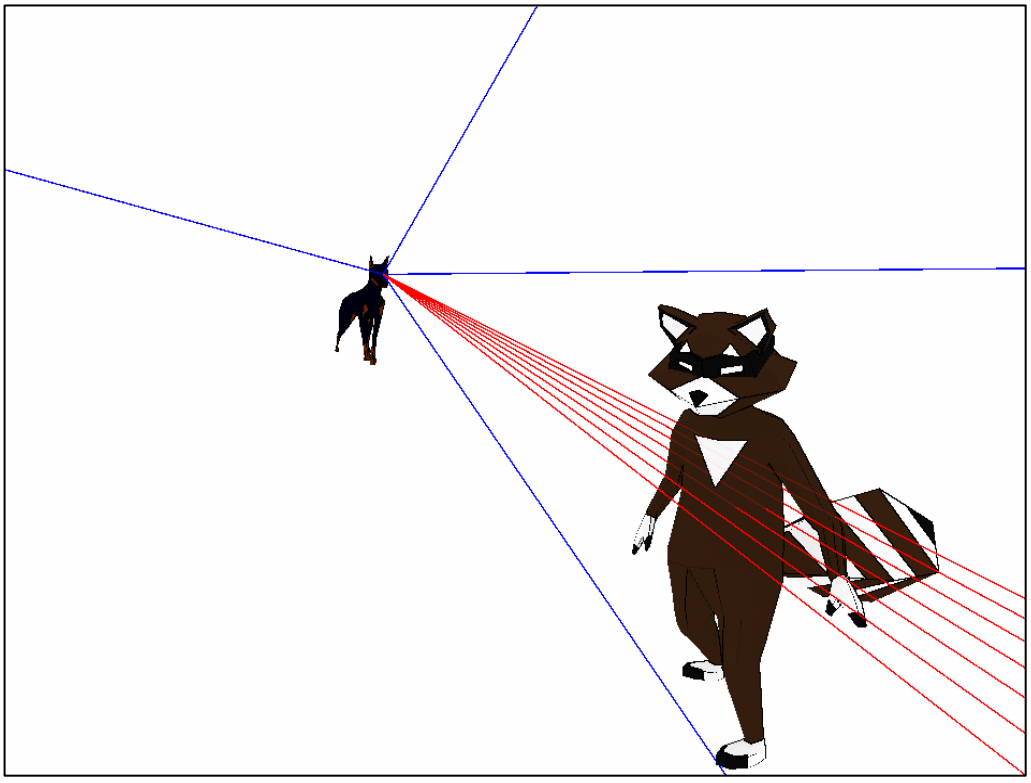


Рисунок 3.9 – Демонстрація роботи методів пошуків (синє – поле зору, червоне – трасування).

В результаті, після знаходження, залишається лише передати через подію позицію, де був помічений гравець.

3.5.9 UI

Основна задача UI полягає в відображенні потрібної інформації гравцю. У випадку з Unity можна всі потрібні переходи зробити за допомогою анімацій, які будуть керуватись через компонент Animator, та вказуванням потрібних параметрів чи тригерів. Але є також і ті елементи, якими можна, чи потрібно, керувати з коду.

3.5.9.1 Відображення лічильника досвіду та кількості здоров'я

Якщо порівнювати логіку відображення одиниць здоров'я та досвіду, та внутрішньо-ігрову логіку зміни обох значень, було вирішено що для лічильника набраних очок треба реалізувати динамічне нарахування.

Створений компонент HealthBar в реалізації не містить особливих підходів, і лише змінює поточне значення слайдера, з UnityEngine.UI.Slider. Тому його код буде наведений лише у додаток, в пункті Гравець, Здоров'є.

А от для ExperienceCounter використовує згаданий раніше FoodCollector, котрий після зарахування досвіду викликає подію OnCounterChange, передаючи актуальну кількість набраний очок.

Для того щоб зробити зміну числа плавним, застосовується інтерполяція значення з визначеною затримкою у корутині, яка наведена у Лістингу 23, що запускається після спрацювання події зміни лічильника.

Лістинг 23 *Метод оновлення лічильника*:

```
private IEnumerator UpdateCounter(int newExp)
{
    float elapsed = 0.0f;
    int startExp = Convert.ToInt32(_output.text);

    while (elapsed < _speedTransition)
    {
        float duration = elapsed / _speedTransition;
```

```
            elapsed += Time.deltaTime;

_output.SetText(Convert.ToInt32(Mathf.Lerp(startExp,
newExp, duration)).ToString());
            yield return null;
        }

        _output.SetText(newExp.ToString());
    }
```

3.5.9.2 Візуальні підсказки

Під візуальними підказками, я маю на увазі відображення того, що з об’єктом можна взаємодіяти. І це, в моєму проекті, реалізовано через появу літери клавіши, на яку зазначена дія.

Для спрощення модифікування параметрів візуалізації, а саме те як швидко з’являється підказка та на яку висоту підіймається текст, вся логіка появи була реалізовано з коду за допомогою розширення DOTween.

Сам метод, що наведений у Лістинг 24, змінює 2 параметри об’єкта-тексту в просторі: висота та масштаб. Він і викликає появу підсказки.

Лістинг 24 *Метод появи візуальної підказки*:

```
    private void ShowHint(Transform transform)
    {
        if (_interact.IsCanInteract == false || _lastDetect
== transform)
            return;

        _lastDetect = transform;

        DisableTweens();

        _tweens[0] = _hint.transform.DOScale(_baseScale,
_showDuration);
        _tweens[1] =
_hint.transform.DOLocalMoveY(_basePosition.y + _yOffset,
_showDuration);
    }
```

В результаті, новостворений компонент VisualHint, дає можливість гравцю знаходити об’єкти та бачити те, що з ними можна взаємодіяти.

3.5.9.3 Меню паузи

Виклик паузи гравцю дозволяє зупинити на час гру, але щоб підготувати гравця до переходу назад, бажаним може стати лічильник до запуску.

Якщо розглядати реалізацію цього з коду, то появу та зникнення меню з кнопками можна виконати з звичайної анімації, а от вже сам відлік стане корисним саме в програмному стилі, бо тоді буде можливість до швидкої модифікації.

Для цього було застосовано функціонал розширення DOTween.Sequence, який дозволяє збирати почергово список динамічних анімацій та запускати їх. Те як це виглядає в коді, наведено в Лістинг 25.

Лістинг 25 *Створення черги анімацій DOTween*:

```
for (int i = 3; i >= 0; i--)
{
    int currentNumber = i;

    countdownSequence.AppendCallback(() => {
        _counterOfBackToPlay.text = 
currentNumber.ToString();
        _counterOfBackToPlay.transform.localScale = 
Vector3.zero;
    });

countdownSequence.Append(_counterOfBackToPlay.transform.DOS
cale(startScale, _pauseDuration / 2));
    countdownSequence.AppendInterval(_pauseInterval);

countdownSequence.Append(_counterOfBackToPlay.transform.DOS
cale(Vector3.zero, _pauseDuration / 2));
}

countdownSequence.OnComplete(() => {
    PauseHandler.Play();
    _counterOfBackToPlay.enabled = false;
    _counterOfBackToPlay.transform.localScale = 
startScale;
    IsOnTransition = false;
});
```

Логіка наведеного коду вище наступна:

1. Встановлення значення лічильника.
2. Додавання анімації зміни масштабу від нульового до базового.
3. Додавання інтервалу очікування.
4. Додавання анімації зменшення розміру лічильника.
5. Додати колбек при завершенні, що запускає гру після паузи.

В результаті, після того як гравець вирішить повернутись до гри, та натисне потрібну кнопку чи клавішу, гра дасть час до того, як виконати сам вихід зі стану паузи.

3.6 Висновки до розділу 3

В результаті виконання цього розділу було:

1. Визначено механіки та системи які необхідно було реалізувати.
2. Описано вимоги до апаратного та програмного забезпечення, яке дозволить запустити створену гру.
3. Описано, що саме вдалось розробити та що було використано для реалізації.

ВИСНОВКИ

1. Проведено аналіз існуючих рішень та літературних джерел. На основі отриманих результатів дослідження, можна зробити висновок, що створення гри у жанрі стелс є досить комплексною задачею, бо необхідно реалізувати технології, які мають працювати разом.
2. Досліджено існуючи ігрові рішення, а саме такі проекти як: Alien: Isolation, Dishonored, Thief. Також було визначено основні елементи, що характеризують їх як ігри жанру стелс.
3. Досліджено та обрано технології розробки, а саме: ігровий рушій Unity, мова програмування C#, розширення DOTween та PlayableGrapth Visualizer.
4. Розроблено ключові ігрові механіки, характерні для жанру стелс, а також визначено мінімальні апаратні та програмні вимоги для кінцевого користувача.
5. Кінцевим результатом кваліфікаційної роботи є створення прототипу гри з елементами жанру стелс, з різними механіками та системами, такими як: базове пересування гравця, взаємодія з об’єктами та предметами оточення, NPC та вороги, візуальні підказки та оновлення UI.
6. Отриманий досвід може бути корисним у розробці проектів інших жанрів, за рахунок схожості та поширеності використовуваних механік.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Video game industry - Statistics & Facts. URL: https://www.statista.com/topics/868/video-games/#topicOverview (дата звернення: 05.06.2025).
2. Хокінг Дж., Хокінг Д.. Unity in Action: Multiplatform Game Development in C# with Unity 5. 1st Edition. Manning, 7 липня 2015 р. С. 352.
3. Learn game development Unity. Courses & tutorials in game design, VR, AR, & Real-time 3D. Unity Learn. URL: https://learn.unity.com/ (дата звернення: 05.06.2025).
4. Game Programming Patterns. URL: https://gameprogrammingpatterns.com/ (дата звернення: 05.06.2025).
5. Роджерс С. , Level Up! The Guide to Great Video Game Design, 2nd Edition, Wiley, 28 квітня 2014 р. С. 560.
6. Мілінгтон І., Artificial intelligence for games, 2nd Edition, Morgan Kaufmann, 6 серпня 2009 р. С. 896.
7. Шрайбер І., Ромеро Б., Game Balance, 1st Edition, CRC Press, 17 серпня 2021 р. С. 806.
8. Свінк С., Game Feel: A Game Designer's Guide to Virtual Sensation, 1st Edition, CRC Press, 27 липня 2017 р. С. 376.
9. About PlayableGraph Visualizer. URL: https://github.com/Unity-Technologies/graph-visualizer/blob/master/Documentation~/playablegraph-visualizer.md (дата звернення: 05.06.2025).
10. Unity - Manual: Playables API. URL: https://docs.unity3d.com/6000.1/Documentation/Manual/Playables.html (дата звернення: 05.06.2025).
11. DOTween (HOTween v2). URL: https://dotween.demigiant.com/ (дата звернення: 05.06.2025).

12. Alien: Isolation Wiki. URL: https://alienisolation.fandom.com/wiki/Alien:_Isolation_Wiki (дата звернення: 05.06.2025).
13. Dishonored Wiki | Fandom. URL: https://dishonored.fandom.com/wiki/Dishonored_Wiki (дата звернення: 05.06.2025).
14. Thief Wiki | Fandom. URL: https://thief.fandom.com/wiki/Thief:_The_Dark_Wiki (дата звернення: 05.06.2025).
15. Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. URL: https://unity.com/ (дата звернення: 05.06.2025).
16. The most powerful real-time 3D creation tool - Unreal Engine. URL: https://www.unrealengine.com/en-US (дата звернення: 05.06.2025).
17. Godot Engine - Free and open source 2D and 3D game engine. URL: https://godotengine.org/ (дата звернення: 05.06.2025).

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Горлач Богдан Андрійович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення освітньої програми “Програмне забезпечення систем”, адреса електронної пошти se121-02@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Розробка комп’ютерної гри у жанрі Stealth**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 02.06.2025 _____ Горлач Богдан Андрійович
(підпис) (прізвище та ініціали) (студент)
Дата 02.06.2025 _____ Заяц Валерій Іванович
(підпис) (прізвище та ініціали) (керівник)