ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
МІСЬКОГО ГОСПОДАРСТВА ІМЕНІ О. М. БЕКЕТОВА

Пояснювальна записка
до кваліфікаційної роботи бакалавра

на тему:
«Створення аудіосупроводу та інтеграція звукових ефектів у 2D-гру в середовищі Unity»

Виконав: здобувач вищої освіти,
групи КН 2021-1
спеціальності 122 – Комп’ютерні науки
Нікіта ДУХНИЦЬКИЙ
Керівник: Олександр КОСТЕНКО
Рецензент: Наталія БРАТЕРСЬКА

м. Харків – 2025

Харківський національний університет міського господарства імені О. М. Бекетова
(повне найменування закладу вищої освіти)
Навчально-науковий Інститут енергетичної, інформаційної та транспортної інфраструктури
Кафедра комп'ютерних наук та інформаційних технологій
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 Комп'ютерні науки
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КНтаІТ

 Марина НОВОЖИЛОВА
«26» 06 2025 року

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

Духницького Нікіти Олександровича
(прізвище, ім'я, по батькові)

1. Тема роботи «Створення аудіосупроводу та інтеграція звукових ефектів у 2D-гру в середовищі Unity»

керівник роботи Костенко Олександр Борисович к. ф.-м. н., доцент кафедри комп'ютерних наук та інформаційних технологій
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від «09» травня 2025 р. №341-03
2. Термін подання студентом роботи 21.06.2025
3. Вихідні дані до роботи: Проектування та реалізація Створення аудіосупроводу та інтеграція звукових ефектів у 2D-гру в середовищі Unity
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Аналіз предметного середовища, дослідження підходів розробки аудіосистем для ігор, аналіз існуючих аналогів, реалізація аудіосистеми - налаштування звуку, прив'язка до анімацій звуку, розробка унікальних взаємодій в ігрових механіках зі звуком, а також створення міні-ритмічної гри.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):

презентація 15аркушів

6. Консультанти розділів роботи

Розділ	Ім’я та Прізвище, посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Олександр КОСТЕНКО, к. ф.-м. н., доцент кафедри КН та ІТ	12.05.2025	24.05.2025
2	Олександр КОСТЕНКО, к. ф.-м. н., доцент кафедри КН та ІТ	24.05.2025	30.05.2025
3	Олександр КОСТЕНКО, к. ф.-м. н., доцент кафедри КН та ІТ	30.05.2025	09.06.2025
4	Вікторія МАЛИШЕВА, к. т. н., доцент кафедри ОП та БЖ	10.06.2025	14.06.2025

7. Дата видачі завдання 12 травня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вибір теми дипломної роботи	01.05.2025	Виконав
2	Затвердження тем, наукових керівників, завдань та календарного плану підготовки кваліфікаційної роботи	09.05.2025	Виконав
3	Написання І розділу	24.05.2025	Виконав
4	Написання ІІ розділу	30.05.2025	Виконав
5	Написання ІІІ розділу	09.06.2025	Виконав
6	Написання IV розділу	14.06.2025	Виконав
7	Подання дипломної роботи керівнику	15.06.2025	Виконав
8	Робота по усуненню зауважень керівника, уточнення і доповнення практичного матеріалу, оформлення додатків до роботи	18.06.2025	Виконав
9	Подання доопрацьованого варіанту роботи керівнику	20.06.2025	Виконав
10	Офіційний захист матеріалів дипломної роботи на засіданні екзаменаційної комісії	27.06.2025	Виконав

Студент _____ (підпис) Нікіта ДУХНИЦЬКИЙ (ім’я та прізвище)

Керівник роботи _____ (підпис) Олександр КОСТЕНКО (ім’я та прізвище)

АНОТАЦІЯ

Структура та обсяг роботи.

Пояснювальна записка кваліфікаційної роботи бакалавра студента групи КН 2021-1 спеціальності 122 Комп’ютерні науки Духницького Нікіти Олекснадровича на тему «Створення аудіосупроводу та інтеграція звукових ефектів у 2D-гру в середовищі Unity» має 4 розділи, включає 10 рисунків, 3 таблиць, 22 джерел, 3 додатки.

Предмет дослідження: аудіосистема до 2D-пригодницькой гри в жанрі квесту.

Мета роботи: розробити аудіосистему до гри, реалізувати налаштування звуку, створення ритмічної міні-гри, інтеграція звуків в ігрові механіки, прив'язка звуку до анімацій.

Методи дослідження: методи аналізу предметної області, проєктування підсистем, тестування програмного забезпечення.

Програмне забезпечення: Unity 2022.3.18f1, C#, Audacity (версія 3.7.3) BoscaCeoil, Visual Studio 2022.

Результати: Прив'язка звуку до анімацій, аудіосупровід для стелс механіки, аудіосупровід взаємодії з об'єктами, налаштування звуку в меню, а також інтеграція аудіосупроводу в UI, міні-ритм гра, проведено тестування на пристрої з мінімальними характеристиками.

Рекомендації щодо використання результатів: створений проєкт може застосовуватися у навчальних цілях для вивчення розробки ігор.

Галузь застосування: ігрова індустрія, навчальні проєкти.

Значущість роботи та висновки: розроблений проєкт сприяє розвитку навичок створення ігор у команді, розуміння аудіосистем та ігор, проєкт дозволяє побачити наративну і технічну складову в музиці, і одночасно з цим покращує сприйняття джерел створення ігор.

Ключові слова: 2D-ГРА, UNITY, АУДІО, C#, ІНТЕГРАЦІЯ, ОПТИМІЗАЦІЯ, РИТМ-ІГРИ.

ANNOTATION

Structure and Scope of the Work.

The explanatory note of the bachelor's qualification thesis by the student of group KN 2021-1, specialty 122 Computer Science, Dukhnytskyi Nikita Oleksandrovych, on the topic «Creation of audio accompaniment and integration of sound effects into a 2D-game in the Unity environment» has 4 sections, includes 10 figures, 3 tables, 22 sources, and 3 appendices.

Subject of research: The audio system for a 2D adventure game in the quest genre.

Objective of the work: To develop an audio system for the game, implement sound settings, create a rhythmic mini-game, integrate sounds into game mechanics, and link sound to animations.

Research methods: Methods of domain analysis, subsystem design, and software testing.

Software: Unity 2022.3.18f1, C#, Audacity (version 3.7.3), Bosca Ceoil, Visual Studio 2022.

Results: Linking sound to animations, audio accompaniment for stealth mechanics, audio accompaniment for object interaction, sound settings in the menu, as well as the integration of audio accompaniment into the UI, a mini-rhythm game, and testing conducted on a device with minimum specifications.

Recommendations for the use of the results: The created project can be used for educational purposes for studying game development.

Field of application: The game industry, educational projects.

Significance of the work and conclusions: The developed project contributes to the development of skills in team-based game creation, understanding of audio systems and games; the project allows one to see the narrative and technical components in music, and at the same time, it improves the perception of game creation sources.

Keywords: 2D-GAME, UNITY, AUDIO, C#, INTEGRATION, OPTIMIZATION, RHYTHM-GAMES.

ЗМІСТ

ВСТУП

Відеоігри – одна з найрізнобарвніших форм цифрового мистецтва. На відміну від кіно, літератури чи музики, вони поєднують у собі майже всі творчі й технічні галузі: від візуального оформлення та анімації до програмування, сценарного майстерності й, звісно, звуку. У кожній грі ці елементи переплітаються по-різному. В одних проєктах акцент робиться на технологічність і візуальну реалістичність, заради чого користувачі оновлюють своє «залізо», намагаючись побачити складніші ефекти та графіку, наближену до кінематографу. В інших – головне історія: світу, персонажів, ідей, закладених розробниками. І тут найважливішу роль починають грати звук і музика, адже саме вони можуть глибоко впливати на сприйняття.

Якщо в класичних іграх, таких як Doom, геймплей був на першому місці, а аудіосупровід лише підкреслював темп і напруження, то сьогодні ситуація змінилася. Технічні можливості зросли, і тепер навіть невеликі інді-команди можуть створювати унікальні за атмосферою світи, де звук і музика відіграють не менш важливу роль, ніж візуальна частина. Сюжет, звукова режисура, стилістика виконання – все це може занурити гравця в світ не гірше, ніж візуальний ряд. Прикладом є ігри, такі як Undertale та NieR: Automata, де музика та звукові ефекти активно взаємодіють з гравцем, передаючи емоційну складову історії та впливаючи на геймплей.

Сучасна розробка пропонує величезний вибір спеціалізованих програм: для малювання – під піксель-арт, шейдери, векторну анімацію; для аудіо – віртуальні інструменти, мікшери, секвенсори. Завдяки цьому звук більше не є фоновим елементом, а стає самостійним засобом вираження. Наприклад, в інді-іграх часто обмежені ресурси на складну графіку, але завдяки сильному музичному оформленню вдається створити глибоку емоційну атмосферу, передати настрій сцени, підкріпити характер персонажа чи навіть посилити геймплейні елементи.

Можливості обробки звуку сьогодні дозволяють не просто «вставити музику», а формувати власне звучання світу, використовувати унікальні тембри, тональності, шуми та ефекти, які стають частиною наративу. В перспективі можна уявити розвиток систем, здатних адаптувати музику залежно від поведінки гравця, що зробить звукове супроводження ще більш живим і інтерактивним.

Таким чином, аудіо складова – це не просто підтримка візуалу, а рівноцінний інструмент оповіді. Звук, як і картинка, може розповідати, реагувати, направляти. І саме в цьому полягає особливість відеоігор: вони дають відчуття участі й взаємозв’язку.

Ця дипломна робота виконується в межах командного проєкту, над яким працює четверо студентів. Кожен учасник відповідає за окрему сферу: геймдизайн, аудіо, графічне оформлення та візуальні ефекти. Над графікою працювали двоє учасників: один створював та наповнював гру графічним медіаконтентом (арти, фони, інтерфейс), інший – розробляв шейдери й накладав візуальні ефекти на вже підготовлений контент. Обоє також відповідали за анімацію і забезпечували візуальну динаміку об'єктів. Кожен не тільки розробляв контент, а й самостійно реалізовував його програмну інтеграцію до гри за допомогою засобів Unity і мови програмування C#.

Моєю зоною відповідальності, а також об'єктом дослідження є створення аудіосупроводу та програмна інтеграція звукових ефектів у 2D-гру. Особливість підходу полягає в тому, що аудіо не просто додається до сцен – воно інтегрується в ігрову логіку, реагує на події, змінюється залежно від дій гравця та ігрового контексту. Усе це реалізується за допомогою скриптів, написаних на C#.

Метою даної роботи є розробка модуля інтерактивного аудіосупроводу для 2D-гри, створеної в середовищі Unity, а також демонстрація його повноцінної роботи в рамках командного ігрового проєкту.

Відповідно до поставленої мети, завдання дипломної роботи були поділені на такі етапи:

– проведення аналізу предметної області та ігрових аудіосистем;

– огляд існуючих ігрових аналогів із якісною інтеграцією звуку;

– проєктування архітектури звукової системи в межах гри;

– створення аудіофайлів, їх обробка та підготовка до інтеграції;

– розробка та реалізація скриптів для інтерактивного звукового супроводу;

– тестування, налагодження та демонстрація роботи модуля аудіо в ігровому середовищі.

РОЗДІЛ 1
ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Опис предметного середовища

У своєму первинному розумінні відеоігри – це інтерактивні системи, що ставлять перед гравцем певні завдання для розв'язання. Ці завдання можуть бути суто механічними, вимагати логіки, стратегічного мислення або інших когнітивних навичок. Залежно від жанру гри, її цільової аудиторії та платформи, на якій вона розповсюджується, вимоги до гравця можуть суттєво відрізнятися. Але у будь-якому разі саме взаємодія – між гравцем і грою – є центральним елементом, навколо якого вибудовується вся структура продукту.

Прикладом механічної гри може слугувати «Asteroids» (1979) – класична аркада, у якій гравець керує космічним кораблем і знищує астероїди в космосі [1]. Геймплей цієї гри надзвичайно простий і, по суті, являє собою варіацію на тему тиру з базовим пересуванням. Основна мета – вижити якомога довше, розстрілюючи загрози, що з'являються на екрані. Через апаратні обмеження ігрових автоматів того часу складніші механіки були недоступні. Графіка складалася з простих геометричних форм, а звук – із коротких синтезованих сигналів. Аудіосупровід виконував переважно сигнальну функцію – надавав зворотний зв’язок на дії гравця: постріл, вибух, зіткнення. Звуки були короткими, циклічними, схожими на електронні біпи. Проте навіть у такому вигляді аудіо відігравало важливу роль: воно структурувало ігровий досвід, надавало йому ритму та динаміки.

З розвитком технологій аудіо стало не лише доповненням, а й невіддільною частиною геймплею. Звуковий супровід перетворився із другорядного компонента на повноцінний інструмент створення атмосфери, геймдизайну та навіть механіки гри. Яскравим прикладом цього є ритм-ігри, такі як «osu!» або «Guitar Hero» [2][3]. У «osu!» гравець виконує точні дії –

натискання, переміщення або утримання кнопок – у ритмі музики, орієнтуючись на візуальні і звукові підказки. Візуальний ряд синхронізований із музикою, що створює сильне відчуття ритму й залучення. У змагальному середовищі ця гра вважається надзвичайно складною дисципліною, де для досягнення високих результатів критично важливі швидкість реакції, точність та почуття ритму. Аудіо тут не просто супроводжує дії – воно керує ними, визначаючи темп і хореографію гравця.

Подібно до цього, Guitar Hero симулює гру на гітарі: натискання кнопок на спеціальному ігровому контролері синхронізуються з музикою та візуальними «нотам», що спадають на екран. Гравець не просто «слухає» музику – він її «виконує», отримуючи зворотний зв’язок у вигляді глядацької реакції (віртуальної) та аудіовізуального ефекту успішного або помилкового натискання. У таких іграх геймплей буквально «зшитий» з аудіо: без музичного супроводу гра втрачає свій сенс. Це приклад максимальної інтеграції звуку з ігровим процесом.

Проте роль звуку в ігровому середовищі не обмежується лише ритмічними чи механічними аспектами. Аудіо також може бути носієм наративу, атмосфери та емоцій. Сучасні ігри активно використовують музику і звукові ефекти для формування настрою, передачі психологічного стану персонажів, створення напруги чи навпаки – умиротворення. Яскравий приклад – гра Valiant Hearts: The Great War [4]. У цьому проєкті основний акцент зроблено на історії звичайних людей, які потрапили у вир подій Першої світової війни. Геймплей зосереджений не на бойових діях, а на вирішенні головоломок, дослідженні навколишнього середовища та емоційній взаємодії з іншими персонажами.

У Valiant Hearts персонажі не розмовляють у звичному сенсі: комунікація відбувається через візуальні хмари з зображеннями думок або намірів, короткі звуки, інтонації. Саме в такому форматі музика й звукові ефекти стають головними інструментами передачі емоцій, створення напруги чи підкреслення драматичних моментів. Наприклад, під час сцени розлуки між

персонажами сумна мелодія посилює візуальний образ, створюючи глибокий емоційний відгук. В іншій сцені, де головний герой рятує поранених, музика прискорюється, супроводжуючи наростаючу тривогу і відповідальність. Це доводить, що звук здатен не лише супроводжувати візуальну складову, а й конкурувати з нею у значущості.

Окрім того, у деяких проєктах звукова складова стає засобом інтерактивної розповіді. У низці інді-ігор, як-от Inside, Limbo або Journey [5], звук не лише доповнює атмосферу, але й служить навігаційним інструментом. У Journey гравець взаємодіє з іншими гравцями лише за допомогою музичних сигналів. Це створює унікальну форму комунікації, де звук – єдиний міст між двома учасниками гри. Такі приклади демонструють потенціал аудіо як повноцінного способу вираження сенсу, емоцій, а іноді – і сюжетних поворотів.

У підсумку, предметне середовище сучасної відеогри є комплексною взаємодією між візуальним, геймплейним та звуковим рівнями. Якщо раніше звук був лише допоміжним елементом, то нині – це інструмент формування ігрового досвіду, механіки і наративу. Усе частіше розробники звертають увагу не тільки на якість візуального оформлення, але й на звуковий ландшафт: фонова музика, звуки довкілля, реактивні аудіоефекти – усе це формує глибше занурення в гру. Саме тому при створенні гри звуковий супровід необхідно проєктувати не як «додаток», а як невіддільну складову всієї концепції.

Оскільки об’єктом дослідження є гра, створена в межах дипломного групового проєкту, яка належить до жанру квест і реалізована у середовищі Unity [6], доцільно розглянути особливості використання аудіосупроводу саме в цьому жанровому контексті, а також можливості рушія Unity для інтеграції звуку.

У контексті окремих жанрів, зокрема квестів, роль звукового оформлення набуває ще більшої ваги. Квест – це жанр, у якому наратив, дослідження світу та взаємодія з предметами і персонажами займають

центральне місце. Гравець зазвичай просувається по сюжету через розгадування головоломок, збирання предметів, діалоги та дослідження локацій. Через це темп таких ігор повільніший, а увага до деталей – вища. У такому середовищі аудіо стає ключовим елементом атмосфери: воно здатне направляти увагу, викликати емоції та створювати ефект занурення.

У квестах часто застосовуються динамічні фонові композиції, які змінюються залежно від подій на екрані: наприклад, напружена музика в сцені погоні або спокійна мелодія під час дослідження локацій. Важливу роль відіграють звуки взаємодії з об'єктами – відкривання шухляд, скрип дверей, клацання замків тощо. Вони не тільки підсилюють реалістичність середовища, але й можуть служити підказками для гравця. Наприклад, приглушений звук відкривання шафи може сигналізувати про її важливість у контексті задачі.

Окрім того, у квестах часто реалізуються голосові репліки персонажів, які супроводжують діалоги або внутрішні монологи героя. Якщо гра не має повної озвучки, важливу роль починає відігравати музичне оформлення сцен: воно задає настрій, може змінювати темп геймплею і навіть виконувати функцію емоційного коментаря до подій. Наприклад, меланхолійна тема у моменті втрати союзника або тривожна мелодія під час важливого вибору.

Усі ці можливості реалізації аудіо чудово підтримуються середовищем Unity. Unity є одним із найпоширеніших рушіїв для створення інді-ігор, і квести – не виняток. Його перевагою є гнучка система керування аудіо, яка дозволяє інтегрувати як прості звукові ефекти, так і складні музичні композиції з можливістю реакції на ігрові події.

Unity надає інструменти для роботи з аудіокліпами, дозволяючи відтворювати звуки, змінювати їхню гучність, просторове положення (3D-позиціювання), а також керувати ефектами накладання (наприклад, реверберація у закритих приміщеннях). Через систему скриптів на мові C# розробник має змогу програмно контролювати відтворення звуку, реалізовуючи складні сценарії – наприклад, поступову зміну музичної теми

при зміні локації або створення звукового ефекту, що запускається лише за певних умов.

Unity також підтримує аудіомікшування в реальному часі через компонент Audio Mixer, що дозволяє регулювати гучність різних груп звуків (музика, ефекти, голоси) і створювати ефекти заглушення, ехо, фільтрації тощо. Наприклад, у сцені, де персонаж перебуває під водою, можна змінити частотні характеристики усіх звуків, щоб вони звучали приглушено – це створює враження фізичної присутності у зміненому середовищі.

Ще одна сильна сторона Unity – це можливість інтеграції сторонніх аудіосистем, таких як FMOD або Wwise, які дають ще більше контролю над звуковим ландшафтом гри. Це особливо актуально для квестів, де важливі динамічні зміни звуку та точне позиціювання ефектів. За допомогою таких систем можна реалізувати, наприклад, зміну музичної теми в залежності від емоційного стану персонажа або накопичення тривожності у складних ситуаціях, що змінює музичну палітру гри.

Ще однією важливою перевагою Unity є зручність оптимізації аудіо для збереження продуктивності гри. У проєктах, де використовується велика кількість звукових файлів, критично важливо зменшити навантаження на пам’ять і процесор. Unity дозволяє оптимізувати аудіофайли за допомогою:

– форматів стиснення, таких як Compressed in Memory або Streaming – це дозволяє економити ресурси, особливо для довгих треків;

– зміни частоти дискретизації – для вторинних або фонових звуків можна використовувати нижчу якість (22 050 Гц замість 44 100 Гц) без значної втрати якості звучання;

– переведення в моно, якщо стереоефект не критичний (наприклад, для коротких звуків дій), що дозволяє зменшити обсяг у два рази;

– автоматичного препроцесингу – обрізання тиші, нормалізації, налаштування циклів відтворення без використання сторонніх редакторів;

– пулінгу та кешування аудіооб’єктів, щоб уникати повторного створення джерел звуку в реальному часі, що особливо ефективно для часто повторюваних ефектів (наприклад, кроків або клацань).

Крім того, тестування звукових подій можна здійснювати прямо в редакторі, що дозволяє значно пришвидшити цикл розробки. Можливість попереднього прослуховування звуків у відповідь на ігрові тригери дає змогу точно налаштувати реакцію на події.

Отже, використання Unity у розробці квестів дає розробнику широкий спектр інструментів для роботи зі звуком. Від простих ефектів до складної музичної драматургії – усе це можливо реалізувати без потреби у зовнішніх засобах. А якщо проєкт вимагає ще більшої гнучкості, середовище легко інтегрується з професійними аудіомодулями. Таким чином, Unity виступає не лише технічним каркасом для створення гри, а й повноцінним середовищем для творчого аудіодизайну, що дозволяє передавати через звук емоції, настрій і глибину ігрового світу.

1.2 Огляд наявних аналогів

У процесі розробки групового дипломного проєкту, спрямованого на створення 2D-гри в середовищі Unity [6], було проаналізовано низку вже реалізованих ігрових продуктів. Основна мета такого аналізу полягала у вивченні підходів до побудови аудіосистеми в 2D-квестах, визначенні технічних рішень, які забезпечують адаптивність звуку, занурення гравця в атмосферу гри, а також узагальненні творчих принципів створення фонової музики, звукових ефектів і зв’язку між аудіо та наративною частиною. Особливу увагу було приділено іграм, які поєднують якісне звукове оформлення з художнім стилем та добре структурованим геймплеєм.

З огляду на те, що наш проєкт реалізовувався в Unity, логічним було звернутися до вже реалізованих ігор, створених на цій же платформі. Найбільшу цінність у цьому плані становили My Brother Rabbit [7] та Fran Bow

[8] – дві відомі 2D-квест-гри, що активно використовують звук не просто як фоновий елемент, а як інструмент наративного розкриття. Обидва проєкти розроблені незалежними студіями, але при цьому демонструють високий рівень технічної реалізації, що є досяжним навіть в умовах обмежених ресурсів.

My Brother Rabbit [7] виявилася вкрай показовою з точки зору побудови звукового середовища, яке повністю заміщує собою традиційні діалоги. У грі повністю відсутні текстові репліки, тож уся емоційна комунікація покладена на музику, інтерфейсні звуки та зміну тональностей у залежності від подій. Із технічної точки зору гра використовує набір аудіо-компонентів Unity, включаючи стандартні AudioSource для відтворення локалізованих звуків, AudioMixer для контролю загального балансу гучності, а також тригери, які активують певні звукові сцени залежно від локації гравця або конкретних подій у грі. Саме такий підхід було адаптовано у нашому дипломному проєкті. Замість класичної подачі діалогів ми також частково поклалися на звукову атмосферу, що змінюється в залежності від місця перебування головного героя або взаємодії з певними об’єктами (рис. 1.1).

Рисунок 1.1 – Приклад гемплею гри «My Brother Rabbit»

Fran Bow [8], попри наявність діалогів і текстових вставок, також активно використовує звук як засіб створення психологічної напруги та відчуття сюрреалізму. У цій грі було помітно, наскільки правильно синхронізовані звукові ефекти зі зміною стану головної героїні: при переході між реальністю та галюцинаціями змінюється не лише візуальний стиль, а й аудіосупровід – посилюється фоновий шум, з'являються тривожні частоти, застосовуються сповільнені та реверсовані ефекти. Для реалізації таких механік в Unity використовуються аудіофільтри, динамічні шари та події, прив'язані до анімацій і перемикачів стану. Саме цей підхід надихнув на створення у нашій грі кількох рівнів фонових звуків, які активуються залежно від ситуації, що дозволяє ефективніше контролювати настрій сцени (рис. 1.2).

Рисунок 1.2 – Приклад гемплею гри «Fran Bow»

Крім технічного аналізу ігор на Unity, на творчий підхід до створення нашої гри суттєво вплинули й інші, не менш знакові проєкти. Зокрема, GTA: San Andreas – точніше, одна з її внутрішніх мініігор під назвою Lowrider Challenge [9] – стала натхненням для побудови окремого елемента нашого геймплею: ритмічної мінігри. В оригіналі гравець мав натискати стрілки на

клавіатурі в точному ритмі з музикою, що дозволяло виконувати танцювальні рухи автомобіля в унісон з треком. Ідея такого поєднання звуку, таймінгу та інтерактиву була адаптована під формат 2D-гри. У нашому проєкті реалізована мінігра, де гравець повинен натискати клавіші в ритм із музикою. Технічно це реалізовано через обробку таймерів, синхронізованих із темпом музики, та систему подій, які реагують на точність натискання (рис. 1.3).

Рисунок 1.3 – Приклад міні гри «Lowrider Challenge»

Ще одним важливим джерелом натхнення стала інді-гра Undertale [10], яка, незважаючи на візуальну простоту, пропонує дуже сильну музичну складову. Музика в грі створювалася з використанням простих віртуальних інструментів, але завдяки продуманим темам і повторюваним мотивам вона запам’ятовується й викликає емоційну реакцію. Важливо, що практично кожна мелодія в грі має наративну функцію: вона асоціюється з персонажем або важливим моментом, завдяки чому формується емоційний зв’язок із гравцем. Саме такий підхід до написання музики був узятий за основу у нашому проєкті – прості мелодії, які можуть варіюватися, але мають чітку музичну

ідентичність, створену спеціально для конкретної сцени чи персонажа (рис. 1.4).

Рисунок 1.4 – Приклад гемплею гри «Undertale»

Таким чином, вивчення наявних аналогів дало змогу не лише перейняти окремі технічні рішення, але й сформувати загальну концепцію інтеграції аудіо в ігровий процес. Головним висновком стало розуміння того, що звукове оформлення має бути інтерактивним, динамічним та контекстно залежним – лише в такому випадку воно сприяє зануренню гравця в світ гри. Як показали досліджені приклади, навіть у 2D-іграх з обмеженою візуальною складністю можна досягти глибокої емоційності саме завдяки правильно організованій роботі зі звуком. І це завдання повністю лягає на плечі не лише композиторів, але й розробників, які інтегрують аудіо в систему логіки гри.

Підсумовуючи, слід зазначити, що технічні приклади, як-от My Brother Rabbit і Fran Bow, показали ефективні способи роботи з аудіо в Unity, зокрема використання тригерів, шарів, мікшерів та динамічного управління гучністю. У свою чергу, творчі приклади з Undertale та GTA: San Andreas дозволили краще зрозуміти, як музика й інтерактив можуть гармонійно співіснувати,

збагачуючи ігровий досвід. Усі ці напрацювання були адаптовані в нашому проєкті відповідно до обраного жанру, художнього стилю й технічних можливостей платформи.

1.3 Постановка задачі

У межах дипломного проєкту передбачається розробка 2D-гри у жанрі квесту, дія якої розгортається у вигаданому світі напередодні воєнного конфлікту. Основною метою даної роботи є створення комплексного аудіосупроводу та інтеграція звукових ефектів у геймплей з використанням можливостей рушія Unity. Аудіо має не лише супроводжувати події, а й активно впливати на ігрове сприйняття, формувати атмосферу, емоційний фон і підсилювати взаємодію гравця з ігровим світом.

З урахуванням загального концепту проєкту та поставленої мети, можна сформулювати основні завдання, що потребують вирішення:

1. аналіз особливостей аудіодизайну в іграх жанру квест – визначити типові звукові рішення, які використовуються в подібних проєктах, та адаптувати їх для власної гри;
2. розробка структури аудіосупроводу – створити сценарій використання фонової музики, динамічних звукових ефектів та озвучення взаємодій з оточенням, з урахуванням особливостей геймплею та логіки ігрового світу;
3. інтеграція аудіо у середовище Unity – реалізувати систему звукового супроводу за допомогою стандартних засобів Unity (Audio Source, Audio Clip, Audio Mixer), використовуючи C#-скрипти для керування відтворенням, перемиканням та реакцією звуків на події в грі;
4. оптимізація аудіофайлів – налаштувати компресію, частоту дискретизації, формат збереження звуків для зменшення навантаження на систему без втрати якості звучання, використовуючи інструменти Unity;

5. реалізація звукових тригерів у сценах гри – додати до локацій звукові події, що активуються під час пересування гравця, взаємодії з предметами, NPC або зміни ігрового стану;

6. тестування та налагодження – перевірити роботу аудіосистеми в ігровому середовищі, усунути технічні помилки, забезпечити стабільність і коректність звукового супроводу в усіх сценаріях;

7. підготовка до масштабування – закласти гнучку систему, що дозволить легко додавати нові звуки, музичні треки або підключати сторонні аудіосистеми (наприклад, FMOD) у разі розширення гри.

Таким чином, реалізація зазначених завдань дозволить створити якісний, атмосферний і функціональний аудіосупровід для гри, що не лише супроводжуватиме геймплей, але й збагачуватиме ігровий досвід за рахунок глибокого емоційного занурення та інтерактивності.

Висновки до розділу

У першому розділі було розглянуто основні аспекти предметного середовища, в якому функціонує розроблена гра, а також особливості аудіодизайну в сучасних 2D-проєктах. Було встановлено, що звук у відеоіграх уже давно вийшов за межі фонової функції: він став повноцінним інструментом побудови атмосфери, впливу на геймплей і вираження наративу. Особливо це важливо для ігор жанру квест, де темп зазвичай уповільнений, а занурення в атмосферу – критично важливе для сприйняття історії.

У межах розділу проаналізовано приклади успішної інтеграції аудіо в ігри, зокрема My Brother Rabbit, Fran Bow, Undertale та інші. Вони показали, що навіть у технічно нескладних 2D-проєктах можна досягти високого рівня емоційного впливу через грамотно побудовану звукову систему. Цей аналіз дозволив виокремити низку ефективних технічних та творчих підходів, які стали основою для реалізації аудіосупроводу у власному проєкті.

Також було обґрунтовано вибір рушія Unity як технічної бази, що надає розробнику широкий спектр інструментів для роботи зі звуком – від простих аудіоефектів до складної динамічної музики, адаптованої до ігрового контексту. Таким чином, отримані знання сформували теоретичну та практичну основу для подальшого етапу – розробки архітектури аудіосистеми та її програмної реалізації.

РОЗДІЛ 2
ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Аналіз предметної області

У рамках даної кваліфікаційної роботи предметною областю є створення функціональної аудіосистеми для відеогри у жанрі 2D-квесту. Звукове оформлення виступає не просто супровідним елементом, а виконує інтегративну функцію: воно допомагає гравцеві зануритися в наратив, отримати зворотний зв'язок від дій та орієнтуватися у віртуальному просторі. Саме тому реалізація аудіосистеми потребує ретельного аналізу механік гри, особливостей локацій і поведінки гравця.

Ігровий процес розгортається в умовно-мирному середовищі, яке поступово змінюється під впливом сюжету. Основні механіки гри включають пересування головної героїні по кімнатах гуртожитку, взаємодію з предметами, NPC, виконання квестів та участь у мініграх. Звукове наповнення відіграє критичну роль на всіх етапах.

В рамках проєкту було визначено три основні рівні звукової системи, які відповідають за різні аспекти геймплею:

1) Фоновий рівень – музика та атмосферні звуки, що супроводжують гравця на локаціях. Наприклад, на верхньому поверсі гуртожитку звучить спокійна тема, яка поступово змінюється або затихає при наближенні до сюжетно важливої зони. Це дозволяє створити настрій та передати зміну ігрового стану. Перехід на нову локацію ініціює завантаження нової теми через систему AudioManager, або зміну параметрів існуючої (гучність, фільтрація, fade-in/fade-out).

2) Подієвий рівень – звуки, що виникають у відповідь на дії гравця або події в середовищі. Наприклад:

- натискання кнопок у меню;
- відкриття дверей;

– звук кроків (із врахуванням типу поверхні);

– використання предметів з інвентарю;

– старт та завершення діалогу з NPC.

Для реалізації цих подій передбачено систему тригерів та AudioSource-компонентів, які прикріплюються до об'єктів або керуються централізовано через EventAudioSystem.

3) Наративний рівень – звукові ефекти, які підкреслюють сюжетну динаміку. У певних зонах (так звані «емоційні тригери») активуються спеціальні звукові події: наприклад, зміна швидкості фонової музики, поява додаткового шуму (наприклад, тремтячий реверб), або приглушення решти звуків для зосередження на наративі. Такі ефекти реалізовуються за допомогою сценаріїв, в яких звукові події прив'язано до позиції гравця через OnTriggerEnter2D.

Окрім основного аудіосупроводу, важливою частиною проєкту є реалізація ритмічної мінігри. Вона виступає окремою геймплейною одиницею та потребує точного таймінгу відтворення звуків і синхронізації з візуальними елементами. У цій підсистемі центральною задачею є точна прив’язка подій до ритму, обробка інпуту користувача та надання миттєвого зворотного зв’язку через звуки влучання, промаху, зміни здоров’я тощо. Ритм і розклад нот зберігається в об'єкті StageData, а кожна нота через NoteSpawner генерується в заданий момент часу.

Також у проєкті реалізовано систему налаштувань гучності, яка дозволяє гравцеві керувати:

– загальною гучністю;

– гучністю музики;

– гучністю подій;

– гучністю фонових ефектів.

Кожна зміна відтворює тестовий звук для зручного налаштування – це дає змогу миттєво оцінити вплив змін. Усі значення зберігаються через PlayerPrefs і застосовуються через AudioMixer.

Таким чином, аналіз предметної області дозволяє сформулювати ключові вимоги до аудіосистеми гри: модульність, реактивність, чутливість до позиції гравця, можливість кастомізації та підтримка синхронізації із зовнішніми подіями.

2.1.1 Вхідні дані

Вхідними даними для побудови аудіосистеми гри виступають як користувацькі дії, так і внутрішньоігрові події, що ініціюють або змінюють звукове відтворення.

До основних вхідних даних належать:

1) Події взаємодії гравця:

– натискання клавіш (наприклад, клавіші взаємодії, пересування, участі в мінігрі);

– натискання UI-елементів (меню, повзунки гучності, кнопки “Назад”, “Грати” тощо);

– вибір опцій у меню налаштувань звуку.

2) Позиційні події:

– вхід або вихід гравця з тригерних зон (OnTriggerEnter2D);

– наближення до об’єктів, що мають звуковий супровід (NPC, двері, сюжетні точки).

3) Часові параметри:

– значення поточного ігрового часу (Time.time, Time.deltaTime);

– відлік до події у ритмічній мінігрі (на основі Note.spawnTime та StageData.startTime);

– затримка введення (input delay), яка може враховуватися для синхронізації нот.

4) Ігрові стани:

– зміна локації або сцени (SceneManager події);

– зміна стану персонажа (наприклад, початок діалогу, взаємодія з предметом);

– втрата або поповнення здоров’я в мінігрі.

5) Налаштування гравця:

– значення гучності збережені через PlayerPrefs;

– обрані параметри балансу, режимів (наприклад, “мовчазний режим”, “режим наративу”).

6) Сценарії та таблиці ритму:

– масиви NoteData[] зі StageData;

– типи нот (NoteType) та заплановані часові відмітки їх появи;

– стан здоров’я в мінігрі (Health value) – для визначення подальших ефектів.

Ці вхідні дані сприймаються через відповідні компоненти (наприклад, NoteInput, AudioMenuUI, NPCTriggerSound, SceneAudioController) та передаються до аудіосистеми через виклики подій або напряму через API Unity (AudioSource.Play(), AudioMixer.SetFloat() тощо).

2.1.2 Вихідні дані

Вихідними даними аудіосистеми є звукові сигнали, які формують акустичний зворотний зв’язок, атмосферу, а також забезпечують функціональну підтримку геймплею.

Основні типи вихідних даних:

1) Відтворення звукових ефектів (SFX):

– ефекти взаємодії: клацання кнопок, підбір предметів, відкриття дверей;

– контекстні ефекти: звук наближення NPC, запуск діалогу, відкриття нової зони;

– реакції в ритмічній мінігрі: влучання/промах (наприклад, HitSound, MissSound);

– вібраційні/ефекти напруги в спеціальних сценах (наприклад, lowpass фільтр при напрузі).

2) Відтворення фонової музики (BGM):

– музичні треки для кожної сцени або підсюжетної зони (через AudioManager);

– змінні параметри музики: гучність, затухання, темп;

– переходи між треками (crossfade, затримка або паралельне накладення).

3) Динамічні звукові реакції:

– відтворення реактивних звуків на дії користувача з урахуванням затримки (наприклад, відкладене влучання в ритмічній грі з компенсацією input lag);

– оновлення UI через звукові підказки (наприклад, зміна гучності – тестовий звук).

4) Регуляція гучності:

– застосування нових значень гучності до окремих аудіо-груп (Master, Music, SFX, Ambience) через AudioMixer.SetFloat() на основі зміни Slider.value;

– збереження змін у PlayerPrefs.

5) Системні сигнали для діагностики:

– повідомлення в консоль при певних подіях (наприклад, помилка відтворення звуку, пропущена нота, відсутній компонент);

– можливість розширення системи логування подій (у майбутньому може бути додано через кастомну систему SoundEventLogger).

Таким чином, вихідні дані реалізуються як аудіо-виходи для користувача та структуровані реакції системи на основі обробки подій. Звукова система відіграє як естетичну, так і функціональну роль, тому її виходи повинні бути точними, синхронізованими та адаптивними до ігрових сценаріїв.

2.2 Опис програмних модулів та конструкцій

Проєктування аудіосистеми для 2D-гри є важливою складовою загального архітектурного рішення, що визначає якість взаємодії користувача

з ігровим середовищем. Аудіосупровід у грі виконує не лише декоративну функцію, але й є невіддільною частиною ігрової механіки, яка підсилює атмосферу, формує настрій сцен, інформує про дії гравця та NPC, а також підкреслює ключові моменти наративу.

У межах розробки системи було поставлено завдання створити гнучкий, масштабований і зручний у підтримці програмний модуль, здатний забезпечити комплексну роботу з фоновою музикою, звуковими ефектами, голосовими вставками та звуками інтерфейсу. Особливу увагу було приділено можливості керування звуками з різних частин проєкту: меню, ігрових сцен, зон діалогу, а також ритмічної мінігри, яка реалізує окрему логіку синхронізації звуку з гравцем.

Система побудована з використанням об'єктно-орієнтованого підходу, що дозволяє легко розширювати функціональність, додавати нові типи аудіоподій, а також централізовано управляти усіма звуковими потоками. Для зберігання параметрів аудіо налаштувань реалізовано локальне збереження, що забезпечує зручність користування для гравця.

Загалом архітектура охоплює кілька ключових модулів: менеджер аудіо (який відповідає за відтворення та контроль гучності звуків), тригери для активації звукових ефектів у сценах, систему взаємодії з користувачем (включаючи елементи меню), а також спеціалізовану частину, що відповідає за роботу музичної мініігри.

У наступних підрозділах буде розглянуто проєктування структури зберігання аудіоданих, об'єктну модель системи та взаємозв’язки між основними компонентами програмного забезпечення.

Для реалізації повноцінної та функціонально багатої аудіосистеми в межах 2D-гри було розроблено низку програмних модулів, кожен із яких відповідає за конкретний аспект обробки та відтворення звукової інформації. Архітектура системи побудована за принципами об’єктно-орієнтованого програмування, що забезпечує модульність, гнучкість у використанні

компонентів та їх масштабованість у разі подальшого розвитку гри. Такий підхід дозволяє ізолювати логіку звукових ефектів та музичного супроводу від основної ігрової логіки, що покращує підтримку коду та спрощує тестування.

2.2.1. Загальна структура аудіосистеми

Основною «віссю» аудіосистеми є центральний менеджер – AudioManager. Він виконує роль єдиного посередника, який опрацьовує запити на відтворення звукових файлів, керує гучністю, переключенням треків і зберігає поточний стан аудіо. Завдяки цьому забезпечується централізований контроль за всіма аудіоджерелами, що підвищує узгодженість роботи системи та дає змогу реалізувати такі функції, як плавне перемикання музики, динамічне регулювання гучності та паузу відтворення (рис. 2.1).

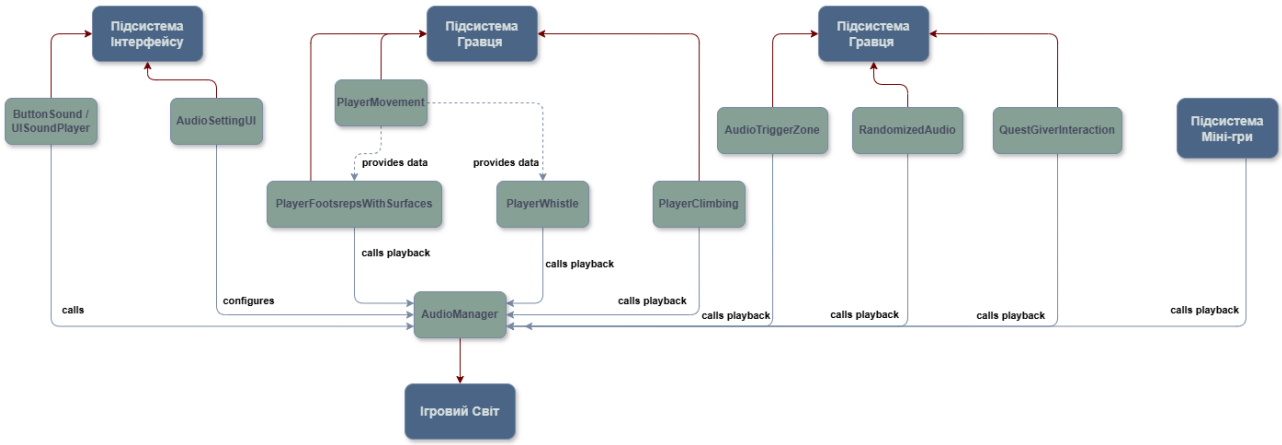

Рисунок 2.1 – Загальна архітектура аудіосистеми гри

Основні функції AudioManager:

1. завантаження та кешування аудіофайлів для швидкого доступу;
2. відтворення коротких звукових ефектів (SFX) та фонового музичного супроводу;
3. регулювання гучності для різних типів звуків (музика, ефекти, голос);
4. підтримка одночасного відтворення кількох аудіоджерел із пріоритетизацією;
5. реалізація паузи та відновлення звуку при переходах між сценами.

Цей модуль реалізований як сінглтон (Singleton), що гарантує існування єдиного екземпляра менеджера протягом життєвого циклу гри. Виклик методів AudioManager здійснюється через статичний інтерфейс, що спрощує інтеграцію з іншими модулями.

Щоб наочно представити архітектуру взаємодії аудіомодулів в інтерфейсі та основній грі, нижче подано UML-діаграму класів, яка відображає ключові компоненти та зв’язки між ними (рис. 2.2).

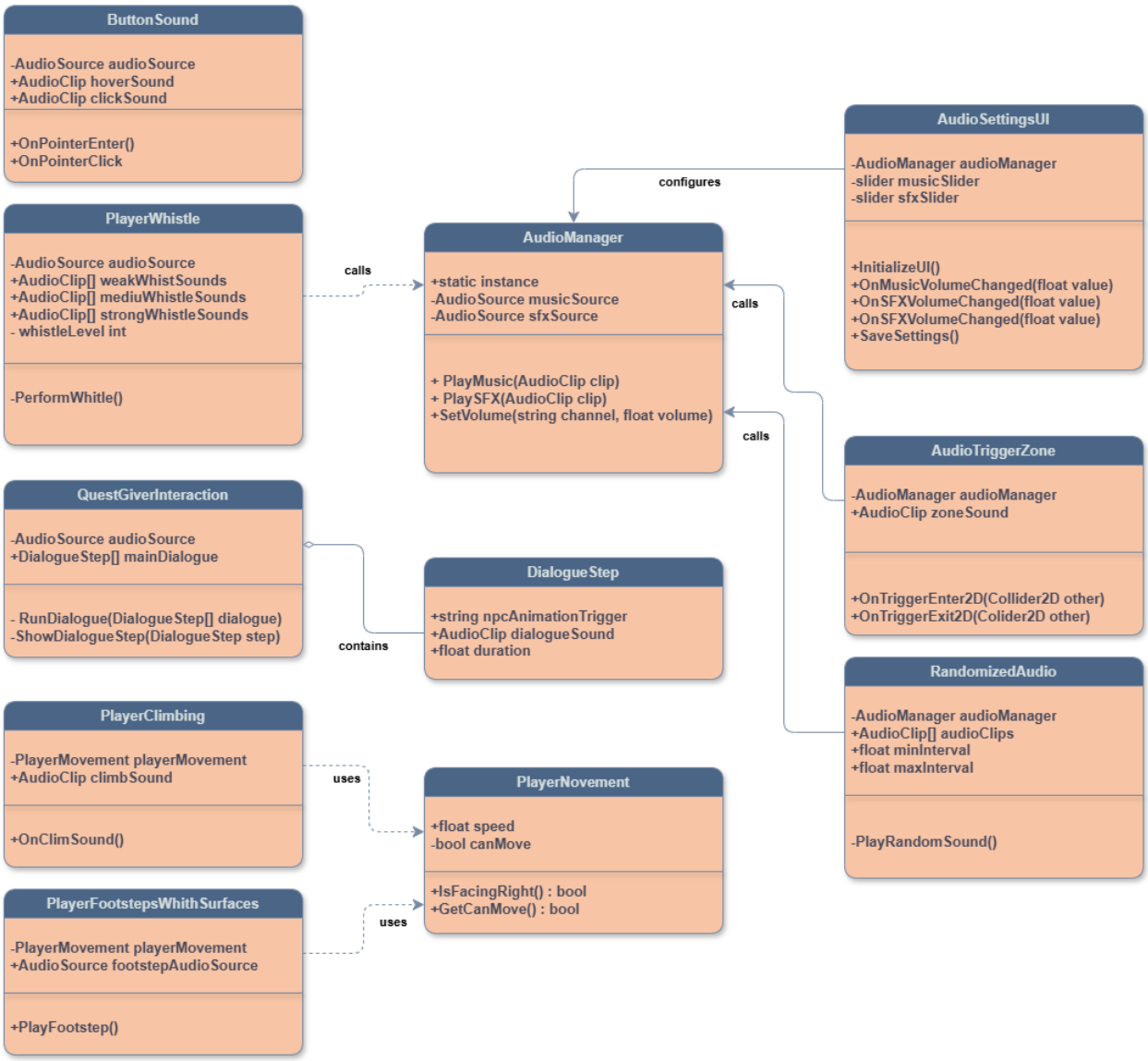

Рисунок 2.2 – UML-діаграма класів аудіосистеми для інтерфейсу та основної гри.

Дана діаграма ілюструє структуру основних класів, що беруть участь в організації аудіо в грі. Зокрема, показано центральний клас AudioManager та пов’язані з ним модулі: UISoundPlayer, AudioSettingsUI, DialogueSoundPlayer, AudioTriggerZone тощо. Відображено основні методи, поля та зв’язки між класами, включаючи агрегацію, наслідування та виклики.

2.2.2 Модулі для інтеграції аудіо в інтерфейс користувача

Інтерфейс користувача (UI) має значний вплив на загальне враження від гри, тому звукове оформлення кнопок, меню, сповіщень є важливою складовою. Для цього розроблено модуль UISoundPlayer, який відповідає за озвучення дій користувача в меню: натискань кнопок, наведення курсору, відкриття/закриття панелей.

Основні методи UISoundPlayer включають:

- PlayClickSound() – відтворення звуку натискання;
- PlayHoverSound() – відтворення звуку наведення;
- PlayTransitionSound() – звук переходу між меню або вкладками.

Взаємодія з AudioManager відбувається через передачу відповідних ключів аудіокліпів. Таким чином, UISoundPlayer не містить власних аудіоданих, а керує виключно логікою виклику відтворення.

Для надання користувачу можливості налаштування гучності музики та ефектів створено окремий модуль AudioSettingsUI. Цей компонент реалізує інтерфейс із повзунками (sliders), які дозволяють в реальному часі регулювати рівень звуку. Значення гучності зберігаються у налаштуваннях користувача (PlayerPrefs) та передаються до AudioManager, який коригує гучність відповідних аудіоканалів.

Функції AudioSettingsUI включають:

- ініціалізація UI-елементів відповідно до збережених налаштувань;
- обробка подій зміни значень повзунків;
- збереження оновлених параметрів гучності;
- повідомлення AudioManager про зміну параметрів.

2.2.3 Модулі для створення звукового оточення

Для створення живого та атмосферного звучання ігрового світу було розроблено два ключові підходи, що працюють у тандемі: статичні звукові зони та динамічні рандомізовані ефекти.

AudioTriggerZone – статичні звукові зони.

Цей модуль реалізований як компонент Unity, що додається до об’єктів сцени, які визначають певні звукові області (наприклад, ліс, кімната гуртожитку, коридор). Коли гравець заходить у таку зону, тригер активує відтворення постійного фонового звуку (ambience), що задає загальний настрій локації.

Логіка роботи: Використовує колайдери та тригери Unity для виявлення входу та виходу гравця. При вході він викликає AudioManager для запуску відповідного аудіо, а при виході забезпечує плавне його загасання.

RandomizedAudio – динамічні атмосферні ефекти.

Для поглиблення атмосфери та уникнення монотонності, що виникає від постійно повторюваного фонового звуку, був розроблений модуль RandomizedAudio. Його мета – відтворювати випадкові, короткі звукові ефекти (скрип дверей, крик птаха) через нерегулярні проміжки часу.

Архітектура модуля: Скрипт працює з масивом аудіокліпів та заданими часовими інтервалами. Після відтворення одного випадкового звуку він вичікує випадковий проміжок часу перед тим, як відтворити наступний. Для додання природності він також незначно змінює висоту тону та гучність кожного звуку, підтримуючи ефекти Fade In/Out для плавних переходів.

Комбінація AudioTriggerZone та RandomizedAudio дозволяє створювати багаторівневе звукове середовище: перший задає загальний тон локації, а другий наповнює її унікальними, непередбачуваними деталями, роблячи світ більш живим та реалістичним.

Якщо ці модулі відповідають за пасивний фоновий звук, що залежить від місцезнаходження гравця, то наступний комплекс модулів реалізує активний звуковий супровід, що генерується безпосередньо діями самого

персонажа. Проєктування цієї підсистеми є прикладом командної взаємодії: основні модулі руху, PlayerMovement та PlayerClimbing, були розроблені іншим учасником команди, а моїм завданням було інтегрувати в ці готові компоненти звукову логіку.

2.2.4 Модулі взаємодії гравця зі звуковим оточенням

Якщо AudioTriggerZone відповідає за пасивний фоновий звук, що залежить від місцезнаходження гравця, то наступний комплекс модулів реалізує активний звуковий супровід, який генерується безпосередньо діями самого персонажа. Проєктування цієї підсистеми є яскравим прикладом командної взаємодії: основні модулі руху, PlayerMovement та PlayerClimbing, були розроблені іншим учасником команди, який реалізував у них фізику переміщення та логіку взаємодії з оточенням. Моїм завданням було інтегрувати в ці готові компоненти звукову логіку, не порушуючи їх основної функціональності.

PlayerMovement – основа для звукової логіки руху.

Цей модуль, що керує фізикою переміщення персонажа, був використаний мною як джерело даних про поточний стан гравця: чи рухається він, в якому напрямку, чи дозволено йому рухатись взагалі. Було інтегровано систему, яка дозволяє іншим звуковим модулям, таким як PlayerFootstepsWithSurfaces, отримувати ці дані для ухвалення рішення про відтворення звуку.

PlayerFootstepsWithSurfaces – інтеграція динамічної системи звуків кроків.

Цей модуль, розроблений мною, відповідає за реалістичне озвучення ходьби персонажа, адаптуючись до поверхні. Його архітектура базується на динамічному аналізі оточення та тісній інтеграції з існуючими компонентами:

– Механізм визначення поверхні використовує скрипт використовує Raycast, спрямований вниз від позиції гравця. При зіткненні променя з поверхнею він зчитує її тег (наприклад, «Wood», «Carpet», «Tile»).

– Структура даних для звуків, в якій для кожного типу поверхні передбачено окремий набір аудіокліпів. Це реалізовано через масив об'єктів, де кожен об'єкт пов'язує тег поверхні з відповідними звуками для звичайної ходьби та ходьби у присяді.

– Інтеграція з анімацією є ключовим етапом моєї роботи стала прив'язка відтворення звуку до вже існуючих анімацій через додавання Animation Events безпосередньо в анімаційні кліпи ходьби. Це гарантує ідеальну синхронізацію звуку з моментом, коли нога персонажа візуально торкається землі.

PlayerClimbing – озвучення вертикальної взаємодії.

Аналогічно до попереднього модуля, було інтегровано звуковий супровід до вже готового скрипту PlayerClimbing. Логіка виявлення перешкод через Raycast вже була реалізована, тому в якості виконавчого завдання було додати відтворення звуку карабкання. Для цього також було використано Animation Event в анімації Climb, без втручання в основну логіку переміщення.

Такий підхід, де звукова система інтегрується в готові модулі руху, дозволив чітко розділити відповідальність між розробниками. Хоча ці модулі не використовують TriggerZone у класичному розумінні, їх механізм можна розглядати як систему мікротригерів, де Raycast та анімаційні події виступають точковими тригерами, що активують звукову логіку, створюючи глибоко інтерактивне звукове середовище.

2.2.5 Інтеграція звуку в інтерактивні ігрові механіки

Окрім базової взаємодії з оточенням, гра містить унікальні механіки, які потребували специфічного звукового супроводу. Модулі QuestGiverInteraction та PlayerWhistle, що реалізують логіку квестів та привернення уваги ворогів, були розроблені іншим учасником команди. Моїм завданням було інтегрувати в них звуковий шар, що підсилює емоційний відгук та інформативність цих систем.

QuestGiverInteraction – озвучення діалогових реплік.

Цей модуль керує складною логікою взаємодії з NPC, перевіркою умов виконання квесту та видачею нагород. Для пожвавлення діалогів було додано до них систему озвучення окремих реплік.

– Структура DialogueStep, в якій було розширено існуючу структуру даних, яка описує кожен крок діалогу, додавши до неї поле типу AudioClip. Це дозволило призначати унікальний звук для кожної репліки безпосередньо в інспекторі Unity.

– Логіка відтворення у методі, що відповідає за відображення діалогового вікна, яка перевіряє наявність аудіокліпу для поточного кроку і, якщо він є, відтворює його через AudioSource, що належить NPC. Це створює ефект «розмовної мови», де кожна репліка супроводжується коротким звуковим ефектом, що робить діалоги більш живими та динамічними.

PlayerWhistle – звуковий супровід для системи привернення уваги.

Модуль свисту дозволяє гравцеві створювати звук різної сили, щоб привернути увагу ворогів. Основна механіка (вибір сили свисту та визначення радіусу дії) вже була реалізована. Моя інтеграція полягала в додаванні звукового оформлення, яке б відповідало обраній силі.

– Диференціація звуків за силою: додає три окремі масиви AudioClip для слабкого, середнього та сильного свисту. Це дозволило призначити різні набори звуків для кожного рівня сили.

– Випадкове відтворення: При підтвердженні свисту моя логіка обирає відповідний масив звуків залежно від обраного гравцем рівня (whistleLevel) і відтворює з нього випадковий кліп. Це додає варіативності та запобігає монотонності.

– Інтеграція з AudioSource: забезпечує, щоб скрипт коректно використовував AudioSource, прикріплений до гравця, для відтворення звуку свисту, що створює правильне просторове позиціювання звуку.

2.2.6 Підсистема ритмічної мінігри

Окремий і найбільш складний комплекс становить підсистема ритмічної мінігри, що є частиною ігрового процесу. Вона реалізує інтерактивний аудіо-візуальний досвід, де гравець взаємодіє з нотами у ритмі музики.

Підсистема включає такі ключові модулі:

1. StageData – зберігає параметри поточного рівня, таймінги нот, тривалість композиції;
2. NoteSpawner – відповідає за динамічне створення нот на сцені згідно із заданим ритмом;
3. Note – клас, що представляє окрему ноту, включає логіку руху та перевірки попадання;
4. NoteInput – обробляє натискання гравця та порівнює час натискання із часом появи ноти;
5. HitZoneFeedback – візуальна та звукова реакція на попадання або промах, забезпечує зворотний зв’язок;
6. HealthDisplay – відображає стан здоров’я гравця у мінігрі, який зменшується при промахах.

Ця система працює за принципом синхронізації аудіо та вхідних подій користувача з високою точністю, використовуючи внутрішній таймер Unity (Time.deltaTime) для коректного відліку часу. Кожен модуль реалізовано як окремий компонент із чітким API, що дозволяє легко модифікувати або розширювати функціонал.

Для наочності на рисунку нижче подано UML-діаграму класів, яка відображає архітектуру підсистеми ритмічної мінігри, включаючи основні класи, їх поля, методи та зв’язки між ними (рис. 2.3).

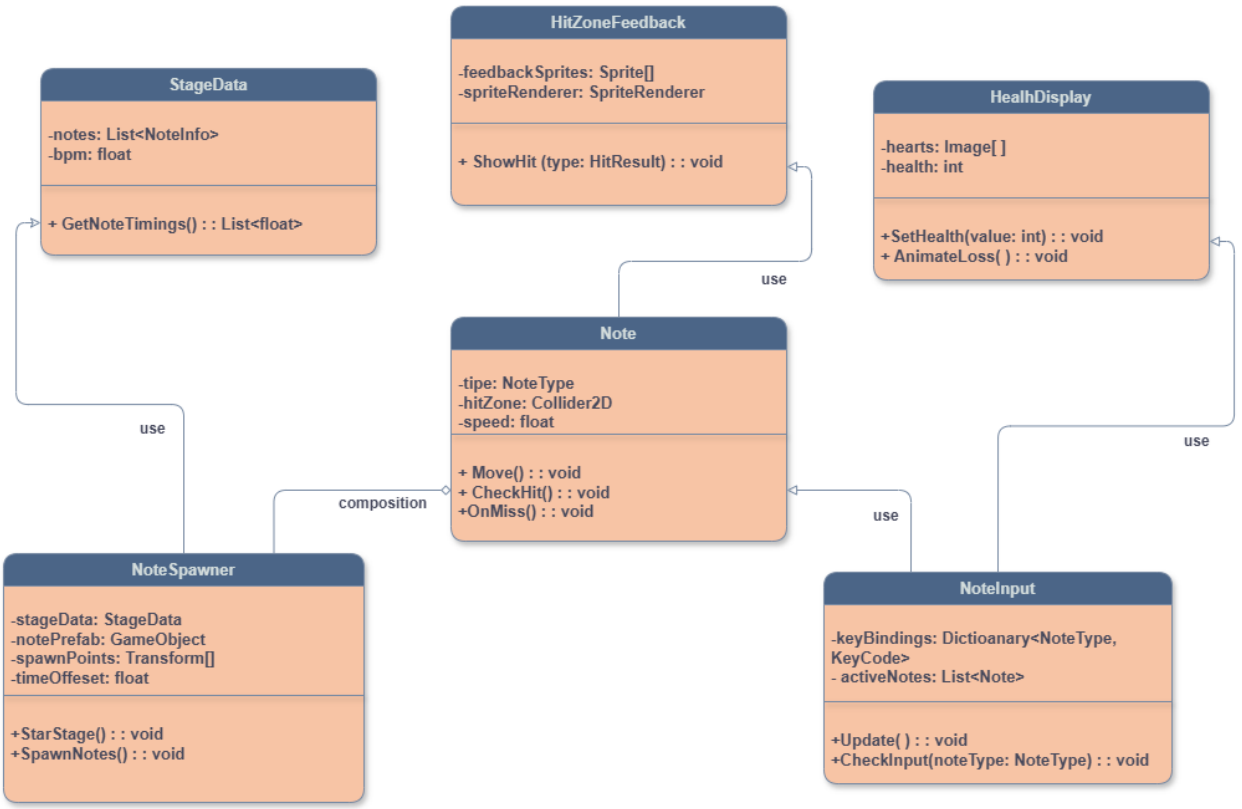

Рисунок 2.3 – UML-діаграма класів підсистеми ритмічної мінігри

2.2.7 Взаємодія та залежності модулів

Усі вищезазначені модулі працюють у єдиній архітектурній структурі з визначеними залежностями:

– AudioManager є центральним компонентом, через який відбувається взаємодія з усіма аудіоелементами;

– UI-модулі (UISoundPlayer, AudioSettingsUI) звертаються до AudioManager для керування звуками інтерфейсу;

– Звукові тригери (AudioTriggerZone) та DialogueSoundPlayer ініціюють відтворення аудіо через AudioManager відповідно до ігрових подій;

Підсистема ритмічної мінігри має власні спеціалізовані модулі, які координуються між собою, але при цьому використовують AudioManager для відтворення аудіо та ефектів.

Використання чітких інтерфейсів та абстракцій у коді дозволяє підтримувати слабке зв’язування між компонентами, що значно підвищує надійність системи та полегшує подальшу підтримку. Наприклад, зміна

аудіоформату або заміна конкретного звукового файлу потребує мінімальних змін у коді (табл. 2.1).

Таблиця 2.1 – Опис основних програмних модулів аудіосистеми

Назва модуля	Призначення	Ключові функції
1	2	3
AudioManager	Центральний менеджер аудіо, контроль гучності та відтворення	Завантаження аудіо, відтворення музики та ефектів, керування гучністю, сінглтон
UISoundPlayer	Озвучення дій користувача в UI	Відтворення звуків натискань, наведень, переходів
AudioSettingsUI	Інтерфейс налаштувань гучності	Регулювання та збереження рівня звуку музики та ефектів
AudioTriggerZone	Відтворення звуків у певних ігрових зонах	Активує звуки при вході/виході гравця, плавне загасання
PlayerMovement	Керування рухом гравця (основа для інтеграції звуку)	Реалізація фізики руху, надання даних про стан гравця для звукових модулів
PlayerFootsteps-WithSurfaces	Динамічне озвучення кроків гравця залежно від поверхні	Визначення поверхні через Raycast, інтеграція звуків в анімацію через Animation Events
PlayerClimbing	Інтеграція звуку для дії карабкання по стінах	Використання існуючої логіки Raycast для перешкод, синхронізація звуку з анімацією
QuestGiver-Interaction	Керування квестами та інтеграція звуку в діалоги	Прив'язка AudioClip до кожної репліки, відтворення звуку синхронно з текстом

Продовження таблиці 2.1

1	2	3
QuestGiver-Interaction	Керування квестами та інтеграція звуку в діалоги	Прив'язка AudioClip до кожної репліки, відтворення звуку синхронно з текстом
PlayerWhistle	Механіка привернення уваги та її звуковий супровід	Відтворення різних звуків залежно від обраної сили свисту (слабкий, середній, сильний)
StageData	Зберігання параметрів ритмічної мінігри	Таймінги нот, параметри рівня
NoteSpawner	Створення нот у ритмічній мінігрі	Динамічний спавн нот за таймінгом
Note	Логіка руху та обробки нот	Рух ноти, перевірка попадання
NoteInput	Обробка введення гравця	Порівняння часу натискання з таймінгом ноти
HitZoneFeedback	Візуальна та звукова реакція на попадання/промах	Відображення ефектів зворотного зв’язку
HealthDisplay	Відображення стану здоров’я гравця у мінігрі	Візуалізація здоров’я, зменшення при промахах

2.3 Математичне та алгоритмічне забезпечення

У сучасних інтерактивних іграх аудіосистема відіграє важливу роль у формуванні атмосфери, покращенні ігрового досвіду та підвищенні залученості гравця. В основі якісного аудіосупроводу лежить низка складних алгоритмів та математичних моделей, які забезпечують точну синхронізацію

звукових подій із діями користувача та ігровим процесом. Саме завдяки ретельно опрацьованим алгоритмічним рішенням аудіосистема може ефективно реагувати на введення гравця, динамічно змінювати гучність, мікшувати різні аудіоканали і забезпечувати плавне відтворення музики та звукових ефектів.

Математика в аудіосистемі використовується не лише для розрахунку часу появи звукових елементів, а й для корекції звуку з урахуванням особливостей людського слуху, забезпечення адекватної обробки сигналів та оптимізації ресурсів. Інтерактивна обробка звуку в іграх вимагає високої точності відліку часу та швидкості реагування на події, що виникають під час гри. Це досягається завдяки використанню внутрішнього таймера ігрового рушія (зокрема, Time.deltaTime в Unity), а також математичним моделям, що визначають правила взаємодії між компонентами аудіосистеми.

У цьому розділі детально розглянуто ключові алгоритмічні та математичні аспекти, що лежать в основі реалізації аудіосистеми нашої гри.

Зокрема, буде описано методи синхронізації нот із музичним ритмом, обробку введення користувача в ритмічній мінігрі, алгоритми зміни позиції нот, розрахунок здоров’я гравця, а також математичні принципи регулювання гучності звуку за допомогою AudioMixer. Ці елементи формують фундамент, на якому базується якісний аудіосупровід та інтерактивність гри.

2.3.1 Математичні аспекти налаштування звуку в Unity

В сучасних іграх якість звукового супроводу є надзвичайно важливою складовою загального враження. Для цього потрібно не лише правильно підібрати аудіоконтент, а й забезпечити коректне математичне опрацювання звукових параметрів, таких як гучність, баланс, просторове позиціювання тощо. Unity надає потужні інструменти для роботи зі звуком, серед яких ключову роль відіграє AudioMixer – компонент для мікшування кількох аудіоджерел із можливістю гнучкого керування гучністю та ефектами.

2.3.1.1 Основи роботи з гучністю: лінійна та логарифмічна шкала

В іграх і аудіоінженерії гучність звуку часто задається у децибелах (dB), що є логарифмічною шкалою [11]. Це пояснюється особливостями сприйняття людським слухом: наша здатність розрізняти зміни гучності не є лінійною, а наближається до логарифмічної залежності.

Гучність, виміряна в децибелах, обчислюється за формулою:

$$L = 20 \cdot \log_{10}(\frac{p}{p_0}), \quad (2.1)$$

де L – рівень звуку в децибелах, p – виміряна амплітуда звукової хвилі, p_0 – опорна амплітуда (поріг чутності).

Для цифрової обробки в Unity гучність подається у вигляді значення в діапазоні від 0 до 1 (лінійна шкала), але у внутрішніх налаштуваннях AudioMixer вона конвертується у децибели для коректного регулювання звуку.

2.3.1.2 Обробка гучності через AudioMixer і формули перетворення

Unity AudioMixer дозволяє задати гучність аудіоканалу у децибелах, де 0 dB – це вихідна гучність без ослаблення, а негативні значення означають зменшення гучності. Для конвертації з лінійного значення (наприклад, повзунка гучності від 0 до 1) у децибели використовується формула [12]:

$$\text{Volumed}_{\text{dB}} = 20 \cdot \log_{10}(\text{Volume}_{\text{linear}}) \quad (2.2)$$

Однак, оскільки $\log_{10}(0)$ не визначений, для значення 0 зазвичай задають мінімальний поріг, наприклад – 80dB, що еквівалентно майже повній тиші. В Unity це реалізовано так, щоб не виникало артефактів при повному вимиканні звуку.

Зворотна конвертація з децибелів у лінійне значення використовується, коли необхідно встановити гучність повзунка, базуючись на внутрішньому значенні AudioMixer:

$$\text{Volume}_{\text{linear}} = 10^{\frac{\text{Volume}_{\text{dB}}}{20}} \quad (2.3)$$

Ці перетворення дають змогу плавно регулювати гучність з урахуванням фізіологічних особливостей сприйняття звуку.

2.3.1.3 Принципи нормалізації звуку, адаптація до людського слуху

Крім конвертації гучності, важливо враховувати, що людський слух чутливий до різних частот по-різному. Наприклад, середні частоти (приблизно 2–5 кГц) сприймаються як більш гучні при тій самій амплітуді порівняно з дуже низькими або дуже високими частотами.

Для корекції цієї особливості в аудіосистемах використовуються різні фільтри, а також стандарти гучності, такі як LUFS (Loudness Units relative to Full Scale), що дозволяють нормалізувати загальний рівень гучності звуку в іграх і уникнути раптових перепадів.

У Unity AudioMixer можна застосовувати різноманітні ефекти і фільтри, наприклад, еквалайзери, компресори, які базуються на складних математичних алгоритмах цифрової обробки сигналів (DSP). Ці алгоритми працюють у реальному часі, коригуючи аудіопотоки залежно від заданих параметрів, що забезпечує більш природнє звучання.

2.3.2 Алгоритми динамічного регулювання гучності і мікшування

В процесі створення якісного аудіосупроводу в грі дуже важливим є динамічне регулювання гучності для створення природних переходів між звуками та уникнення різких змін, які можуть порушувати атмосферу або заважати користувачу. Також ефективне мікшування аудіопотоків дозволяє одночасно керувати декількома звуковими джерелами, наприклад музикою, ефектами та голосами NPC.

2.3.2.1 Алгоритми плавного переходу гучності (Fade In / Fade Out)

Плавний перехід гучності (fade) є базовою технікою для плавного увімкнення чи вимкнення звуку. Вона дозволяє поступово збільшувати (fade in) або зменшувати (fade out) гучність аудіокліпу протягом заданого проміжку часу.

Основна ідея реалізації – це лінійне (або нелінійне) інтерполювання значення гучності від початкового до кінцевого:

$$\mathrm{V(t)} = \mathrm{V_0} + (\mathrm{V_1} - \mathrm{V_0}) \times \frac{\mathrm{t}}{\mathrm{T}}, \tag{2.4}$$

де V(t) – поточне значення гучності у момент часу t, V_0 – початкова гучність (наприклад, 0 при fade in або 1 при fade out), V_1 – кінцева гучність (1 для fade in або 0 для fade out), T – загальна тривалість переходу, t – час, що минув від початку переходу, $0 \leq t \leq T$.

У Unity це часто реалізується через коригування параметру volume у AudioSource або AudioMixer за допомогою корутин або оновлення у методі Update(), де Time.deltaTime гарантує плавність змін незалежно від частоти кадрів.

Крім лінійного інтерполювання, для більш природного звучання використовують криві easing (наприклад, квадратичні або кубічні), що змінюють темп зміни гучності, імітуючи фізичні властивості затухання.

2.3.2.2 Керування параметрами через скрипти

Для динамічного керування звуковими параметрами в Unity використовуються скрипти на C#, які можуть регулювати гучність різних аудіоканалів у реальному часі.

Наприклад, для плавного переходу гучності виконується поступове змінення властивості volume AudioSource або параметрів AudioMixer за заданою формулою.

Скрипти можуть також враховувати ігрові події, такі як:

– зміна сцени (плавна зміна музики);

– активація спецефектів;

– пошкодження персонажа (зменшення гучності музики під час бою);

– налаштування користувача (регулювання гучності через UI).

Таке програмне управління забезпечує гнучкість і дозволяє створювати складні звукові композиції, які адаптуються під контекст гри.

3.3.2.3 Розподіл аудіопотоків (музика, ефекти, голос)

Для одночасного відтворення різних типів звуків у грі аудіосистема реалізує мікшування – поєднання декількох аудіоджерел в один звуковий потік.

У Unity для цього зазвичай застосовується AudioMixer з окремими каналами:

– Music Channel – для фонового музичного супроводу;

– SFX Channel – для звукових ефектів (кроки, натискання кнопок, вибухи тощо);

– Dialogue Channel – для озвучення персонажів і діалогів.

Кожен канал має власні налаштування гучності, які можна регулювати окремо як програмно, так і через UI.

Для правильного балансування звуку важливо враховувати пріоритетність потоків: наприклад, діалоги зазвичай мають вищий пріоритет і можуть приглушувати музику або ефекти, щоб гравець чітко чув репліки персонажів.

Крім того, AudioMixer дозволяє застосовувати різні ефекти (реверберацію, компресію, еквалайзер) до окремих каналів, що підвищує якість звучання.

З точки зору алгоритмів, мікшування реалізується шляхом сумування аудіосигналів із врахуванням їх гучності та панорамування.

2.3.3 Алгоритми ритмічної мінігр

Ритмічна мінігра є інтерактивним аудіо-візуальним компонентом, що потребує точного математичного та алгоритмічного забезпечення для синхронізації звукових подій і реакції на дії користувача.

2.3.3.1 Алгоритм синхронізації нот

Основою синхронізації нот є параметр BPM (beats per minute) – кількість ударів у хвилину, що задає темп музики. Для точного розміщення нот у часі використовується формула:

$$\text{noteTime} = \frac{60}{\text{BPM}} \times \text{beatNumber} + \text{timeOffset}, \quad (2.5)$$

де noteTime – час появи конкретної ноти в секундах, BPM – темп музики, beatNumber – порядковий номер біта (удару), timeOffset – додаткова корекція, що компенсує затримки відтворення або апаратні затримки.

Значення timeOffset може бути налаштоване експериментально для забезпечення максимальної точності синхронізації.

2.3.3.2 Обробка введення користувача

Для визначення точності попадання по ноті використовується клас NoteInput, який реагує на натискання клавіш або кнопок.

Коли користувач натискає клавішу, система порівнює час натискання з часом появи відповідної ноти noteTime. Для цього використовується перевірка попадання у визначеному часовому вікні (tolerance window), наприклад ±0.15 секунди.

Якщо різниця між часом натискання і noteTimenoteTimenoteTime не перевищує це вікно, вважається успішним попаданням (hit), інакше – промахом (miss).

Таймінг обробляється через внутрішній таймер, що базується на Unity API – Time.deltaTime, який забезпечує точний відлік часу між кадрами і дозволяє коректно співставити реальний час гри з музичним ритмом.

2.3.3.3 Зворотній зв’язок та відображення здоров’я

Після визначення результату попадання або промаху система виконує зворотній зв’язок – HitZoneFeedback, що включає:

– Візуальні ефекти (спалахи, анімації);

– Відтворення спеціальних звукових ефектів;
– Зміни стану інтерфейсу (пульсація, зміна кольору).

Водночас відбувається оновлення показника здоров’я гравця через компонент HealthDisplay, який відображає стан у вигляді шкали або індикатора. При промахах здоров’я зменшується, а при успішних попаданнях може відновлюватися або залишатися стабільним.

2.3.3.4 Таймінги і внутрішній таймер

Використання Time.deltaTime є ключовим для реалізації коректної роботи мінігри, оскільки дозволяє враховувати час, що минув між кадрами, та підтримувати синхронність ритму незалежно від частоти кадрів (FPS).

Інтервали оновлення позицій нот, перевірки натискань та оновлення стану здоров’я базуються на сумі часів, що накопичуються з кожного кадру, що гарантує плавність анімації і точність алгоритмів.

Таким чином, поєднання точних математичних формул для розрахунку часу нот, врахування введення користувача з часовими вікнами та динамічної реакції системи через зворотній зв’язок забезпечують високий рівень інтерактивності та приємний ігровий досвід.

Висновки до розділу

У цьому розділі було проведено комплексний аналіз та опис процесу проєктування й реалізації аудіосистеми 2D-гри в середовищі Unity. Розгляд охопив як загальні аспекти предметної області, так і практичні технічні рішення, що забезпечують повноцінну інтеграцію музики та звукових ефектів у геймплей.

На етапі аналізу предметної області було визначено вхідні та вихідні дані системи, сформульовано вимоги до функціонування звуку в межах ігрового середовища. Враховано особливості взаємодії з користувачем, типи аудіоконтенту (фонова музика, звуки дій, діалоги тощо) та технічні обмеження платформи.

Проєктування системи охопило архітектуру аудіомодулів, зокрема роботу аудіоменеджера, тригерів, обробки звукових подій у UI, діалогових систем та ритмічної мінігри. Описано принципи побудови програмних модулів, а також візуалізовано їх взаємозв’язки за допомогою UML-діаграм. Сформована система є гнучкою, розширюваною та придатною до масштабування.

У розділі математичного та алгоритмічного забезпечення докладно розглянуто логіку обробки часу, синхронізацію аудіо з діями гравця, реалізацію ритмічних механік та використання Unity AudioMixer для динамічного керування гучністю. Було продемонстровано застосування логарифмічних перетворень, принципів детекції інпуту, керування станами нот та інтеграцію з графічним інтерфейсом користувача. Усі ці алгоритми дозволили створити інтуїтивно зрозумілу, технічно стійку звукову систему, яка підсилює атмосферу гри.

Загалом, розділ засвідчує, що проєктування та реалізація аудіосистеми потребують ретельного технічного підходу, поєднання знань із програмування, математики, теорії звуку та геймдизайну. Результатом є функціональна, адаптивна система звукового супроводу, яка відіграє важливу роль у формуванні ігрового досвіду.

РОЗДІЛ 3
ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Засоби розробки

Для реалізації проєкту з розробки 2D – гри з інтегрованою аудіосистемою були використані сучасні програмні засоби, які забезпечують повний цикл створення гри: від розробки геймплею та сценаріїв до обробки аудіо та написання коду. Нижче наведено основні інструменти, що були залучені в рамках проєкту.

Unity (версія 2022.3.18f1 LTS) – це популярне кросплатформене середовище розробки ігор, яке забезпечує зручний інтерфейс для візуального компонування сцен, а також має вбудовані засоби роботи з аудіо, фізикою, анімацією та UI. Саме в Unity було реалізовано основну логіку гри, візуальну частину, а також системи інтеграції звукових ефектів та фонової музики. Використання LTS (Long Term Support) версії гарантує стабільність і сумісність із широким спектром пристроїв.

Visual Studio 2022 використовувалась як основне середовище для написання сценаріїв на мові C#. Завдяки глибокій інтеграції з Unity, Visual Studio забезпечує ефективну роботу з автодоповненням коду, дебагінгом, а також дозволяє зручно організовувати проєктну структуру скриптів.

Audacity (версія 3.7.3) – вільний редактор аудіо, який було використано для попередньої обробки звукових файлів: обрізки, нормалізації, зниження шуму, еквалізації та експорту в потрібному форматі. Цей інструмент дозволив адаптувати звукові ефекти під технічні вимоги гри без втрати якості.

BoscaCeoil – це проста, але потужна програма для створення чіптюн-музики. У рамках проєкту вона використовувалась для написання фонових мелодій та музичних вставок у ретро-стилі, які підкреслюють естетику гри та доповнюють наратив. Завдяки візуальному інтерфейсу та підтримці MIDI-експорту, створені треки легко інтегруються в Unity.

Загалом, обраний набір засобів розробки дозволив реалізувати всі етапи створення гри: програмування, створення аудіоконтенту, редагування звуків, написання музики та повноцінну інтеграцію всіх компонентів у ігрове середовище. Інструменти взаємодоповнюють одне одного та створюють ефективний робочий процес, що дозволив досягти поставлених цілей дипломного проєкту.

3.2 Вимоги до технічного та програмного забезпечення

Розробка та запуск 2D-гри в жанрі пригодницького квесту з інтегрованою звуковою системою потребує наявності відповідного технічного та програмного забезпечення як на стороні кінцевого користувача, так і на стороні розробника. Особливу увагу в проєкті приділено аудіосупроводу, який включає динамічні звукові ефекти, діалогові елементи, фонову музику та інтерактивну ритмічну мінігру. Це накладає додаткові вимоги до обробки звуку та загальної стабільності роботи аудіосистеми під час виконання гри. Оскільки однією з задач проєкту є оптимізація під малопотужні пристрої, у цьому підрозділі розглядаються мінімальні технічні характеристики для користувача та розробника.

Для кінцевого користувача, який запускає гру, встановлено такі мінімальні технічні характеристики, які підтверджені тестуванням на реальних пристроях:

- процесор із двома ядрами та чотирма потоками з базовою тактовою частотою 2.5 ГГц;
- оперативна пам’ять об’ємом 4 ГБ DDR3;
- інтегрована відеокарта з 64 МБ відеопам’яті та підтримкою DirectX 11;
- не менше 1 ГБ вільного місця на фізичному накопичувачі;
- наявність клавіатури та комп’ютерної миші;
- дисплей з мінімальною роздільною здатністю 800×600.

Наведена конфігурація дозволяє забезпечити стабільне відтворення звукових ефектів, фонової музики та обробку взаємодій гравця з ігровим світом, включно з наративними зонами та мінігрою, навіть за обмежених апаратних ресурсів.

Для розробника, що працює над проєктом у середовищі Unity, мінімальні технічні вимоги дещо вищі, оскільки необхідно обробляти одночасно велику кількість об’єктів, сцен, звукових подій і візуальних елементів:

– процесор із чотирма ядрами та вісьмома потоками з базовою тактовою частотою 2.8 ГГц;

– оперативна пам’ять об’ємом 12 ГБ DDR3;

– дискретна відеокарта з 4 ГБ відеопам’яті та підтримкою DirectX 11;

– не менше 128 ГБ вільного місця на накопичувачі для зберігання проєкту, редактора Unity, середовища розробки та зовнішніх бібліотек.

Такі вимоги дозволяють ефективно працювати з аудіосистемами, редагувати звукові шари, тестувати ритмічну мінігру, інтегрувати діалогові послідовності та проводити налагодження звукової логіки на різних етапах розробки.

Щодо програмного забезпечення, користувач може запускати гру на таких операційних системах:

– Windows 10 x64 (збірка 19045 і вище);

– macOS;

– Linux.

Це забезпечує кросплатформну сумісність. Для коректної роботи на платформі Windows обов’язкова підтримка DirectX 11 і встановлені драйвери графічного адаптера, що дозволяє гарантувати стабільність виводу графіки та відтворення аудіо.

Для розробника важливо мати налаштоване середовище, що включає Unity Editor (версія 2022.3) та Visual Studio або інший C#-сумісний редактор коду.

Додатково використовуються вбудовані модулі, як-от Audio Mixer, компоненти AudioSource і AudioListener, які безпосередньо впливають на інтерактивність сцени та синхронізацію візуальних і аудіоелементів.

Таким чином, зазначені технічні й програмні вимоги повністю відповідають задачам проєкту – створенню інтерактивної 2D-гри з розвиненою аудіосистемою та ритмічною взаємодією, з можливістю запуску на широкому колі пристроїв.

3.3 Опис програмної реалізації

3.3.1 Архітектура інтеграції аудіо в основну гру

Інтеграція аудіо в ігровий процес реалізована на основі централізованої архітектури, у центрі якої знаходиться модуль AudioManager. Цей компонент є основним координатором для всіх звукових подій гри – від фонової музики до коротких звукових ефектів інтерфейсу, NPC та навколишнього середовища. Завдяки такій централізації вдалося досягти узгодженості у відтворенні звуків, скоротити дублювання коду та забезпечити зручний доступ до аудіофункцій з інших модулів.

3.3.1.1 Загальний огляд системних модулів

У межах основної логіки гри використовується низка скриптів, які відповідають за інтеграцію аудіо до конкретних елементів геймплею. Зокрема:

– SettingsManager – обробка налаштувань гучності, зв’язок із UI та AudioMixer;

– ButtonSound – реалізує звукові ефекти для елементів інтерфейсу;

– PlayerMovement, PlayerFootstepsWithSurfaces, PlayerClimbing – відтворення звуків руху, ходьби, карабкання, прив’язка до анімацій;

– TeleportSystem – запуск аудіоефекту під час переміщення між сценами;

– PlayerWhistle – система свисту з адаптивною зоною дії та звуковими рівнями;

– QuestGiverInteraction – озвучення діалогів NPC на основі реплік.

Ці скрипти реалізовані як незалежні модулі, які через публічні методи взаємодіють із AudioManager, використовуючи підхід слабкого зв’язування.

3.3.1.2 Централізоване управління звуком через AudioManager

Модуль AudioManager реалізований у проєкті як сінглтон, що дозволяє легко звертатися до нього з будь-якого місця коду без потреби створення нових екземплярів. Основні функції AudioManager включають:

– відтворення звукових ефектів (PlaySFX), музики (PlayMusic), діалогів (PlayDialogue), атмосферних звуків (PlayAmbient);
– зупинка та перемикання треків;
– доступ до параметрів AudioMixer для регулювання гучності;
– реалізація ефектів Fade In/Out для плавного переходу між аудіофайлами.

Цей компонент забезпечує логіку пріоритетності між потоками (наприклад, приглушення фонової музики під час діалогів), а також враховує глобальні параметри гучності, які задаються через меню налаштувань.

3.3.1.3 Обробка гучності та використання AudioMixer

Керування гучністю реалізовано через компонент AudioMixer, який дозволяє налаштовувати гучність окремих каналів: Master, Music, SFX, Dialogue. Усі повзунки в меню зв’язуються з параметрами мікшера, що дає змогу змінювати гучність у реальному часі без необхідності перезапуску звуку.

Для зв’язку UI і AudioMixer використовувався скрипт SettingsManager, в якому реалізовано прив’язку повзунків до аудіопараметрів через метод AudioMixer.SetFloat(). Значення зберігаються за допомогою PlayerPrefs, що забезпечує збереження налаштувань між сесіями гри.

Крім цього, AudioMixer дозволяє накладати аудіоефекти (наприклад, lowpass-фільтр для створення приглушеного звучання), які активно використовуються під час зміни ігрових станів – наприклад, при вході до сюжетних зон або при втраті здоров’я в міні-грі.

3.3.2 Реалізація аудіоналаштувань

Аудіоналаштування в грі реалізовані з урахуванням потреб користувача щодо персоналізації гучності різних звукових каналів. Це дозволяє гравцеві індивідуально налаштувати рівень фонового звучання, ефектів і загальної гучності відповідно до своїх уподобань або умов гри. Реалізація цієї функціональності була виконана за допомогою скрипта SettingsManager, який розробляв один із учасників команди, а мною було впроваджено збереження налаштувань гучності та їх прив’язку до параметрів аудіомікшера.

Окрім глобального керування гучністю, значна увага була приділена локальним звуковим ефектам, що підвищують якість взаємодії. Яскравим прикладом є скрипт ButtonSound, розроблений для озвучення елементів користувацького інтерфейсу. Цей самодостатній компонент, що додається безпосередньо до кнопок, реалізує інтерфейси IPointerEnterHandler та IPointerClickHandler для миттєвого відтворення звуків при наведенні курсора та натисканні. Хоча гучність цих ефектів керується глобально через канал «SFX» в AudioMixer, сам механізм їх виклику є локалізованим. Це дозволяє ефективно «оживити» інтерфейс, забезпечуючи чіткий звуковий відгук на дії гравця, не перевантажуючи при цьому центральну систему керування звуком. Таким чином, досягається баланс між глобальними налаштуваннями та локальною інтерактивністю.

Детальна реалізація ключових модулів, що відповідають за взаємодію з інтерфейсом та налаштуваннями можно ознайомитись в Додатку А.

3.3.2.1 Огляд скрипта SettingsManager

SettingsManager виконує роль контролера між користувацьким інтерфейсом (повзунками гучності) та системним компонентом AudioMixer. У цьому скрипті зчитуються та зберігаються поточні значення гучності, які далі застосовуються до відповідних каналів мікшера: Master, Music, SFX.

У реалізації були враховані такі аспекти:

- кожен повзунок (Slider) має окремий обробник подій;
- збереження значень відбувається одразу після зміни повзунка;

– усі параметри мають значення за замовчуванням, якщо попередні не були збережені.

3.3.2.2 Збереження гучності через PlayerPrefs

Щоб забезпечити збереження змін гравця між сесіями, використовується система PlayerPrefs. Для кожного параметра гучності (masterVolume, musicVolume, sfxVolume) створено окремі ключі, до яких записуються значення у лінійній шкалі (від 0 до 1). При запуску гри скрипт зчитує значення з пам’яті та встановлює їх на відповідні повзунки.

Таким чином, навіть після перезапуску гри гравець не втрачає своїх індивідуальних налаштувань.

3.3.2.3 Зв'язок повзунків із параметрами мікшера

Для зв'язку значень повзунків з гучністю аудіопотоків використовується AudioMixer.SetFloat(), який змінює параметри гучності у децибелах (dB). Оскільки децибели – це логарифмічна шкала, значення повзунків (лінійна шкала) конвертуються перед передачею:

$$float\ dB\ =\ Mathf.Log10(volume)\ *\ 20 \qquad (3.1)$$

Це дозволяє коректно адаптувати сприйняття змін гучності до людського слуху. Для випадку, коли значення повзунка дорівнює 0, замість логарифмічного обчислення встановлюється мінімальний рівень гучності, наприклад, –80 dB.

Окрім цього, зміна гучності супроводжується відтворенням короткого тестового звуку (якщо це передбачено в інтерфейсі), щоб гравець міг одразу почути ефект зміни. Такий підхід покращує зручність користування та надає відчуття контролю над ігровим оточенням.

3.3.2.4 Налаштування варіативності звукових ефектів за допомогою RandomizedAudio

Окрім прямого налаштування гучності, для створення більш живого та менш монотонного звукового середовища було розроблено модуль

RandomizedAudio. Цей скрипт є гнучким інструментом для налаштування відтворення випадкових звуків, що дозволяє уникнути механічного повторення. Хоча він використовується для створення атмосфери, за своєю суттю він є системою конфігурації звукових подій.

Основні параметри, що налаштовуються в RandomizedAudio:

– Вибір аудіокліпів: модуль працює з масивом звуків, з якого випадковим чином обирається наступний для відтворення.

– Часові інтервали: за допомогою параметрів minInterval та maxInterval налаштовується випадковий проміжок часу між звуковими ефектами.

– Варіативність звуку: Параметри pitchVariation та volumeVariation дозволяють налаштувати діапазон, у межах якого будуть незначно змінюватися висота тону та гучність кожного звуку, що додає природності.

– Плавність переходів: Можливість увімкнути та налаштувати тривалість ефектів Fade In та Fade Out для плавного з'явлення та зникнення звуків.

Таким чином, RandomizedAudio виступає як розширена система налаштувань, що дозволяє дизайнеру не просто вказати, який звук грати, а й налаштувати, як і коли він буде відтворюватися, створюючи динамічну та непередбачувану звукову картину.

3.3.3 Звукове оформлення дій гравця

Звуковий супровід дій гравця виконує важливу роль у формуванні відчуття занурення в ігровий світ. Кожна дія персонажа – пересування, присідання, підйом по поверхнях або взаємодія з предметами – супроводжується відповідними аудіоефектами, що узгоджуються з візуальним відображенням і станом анімацій. Реалізація цієї системи охоплює три основні скрипти: PlayerMovement, PlayerFootstepsWithSurfaces та PlayerClimbing.

Детальна реалізація ключових модулів, що відповідають за пересування гравця, а також взаємодію з об'єктами можно побачіти в Додатку Б.

3.3.3.1 Скрипт PlayerMovement

Цей скрипт був розроблений іншим учасником команди та відповідає за базову логіку пересування гравця: обробку клавіш, зміну станів, а також

взаємодію з об’єктами. Доповненням в цей скрипт було реалізацією звукових ефектів, які синхронізовані з діями гравця:

– звук кроків – активується лише при русі, враховуючи напрямок та швидкість;

– звук підбору предметів – активується при успішній взаємодії з елементами інвентарю або квестовими об’єктами;

– прив’язка до анімацій – звук кроків активується через Animation Events, що дозволяє точно синхронізувати аудіо з анімацією руху.

Це дозволяє досягти високої точності звукового супроводу навіть при змінах швидкості чи поведінки гравця.

3.3.3.2 Скрипт PlayerFootstepsWithSurfaces

Цей скрипт також було реалізовано іншим учасником команди як частину механіки пересування, включаючи присідання та визначення типу поверхні під гравцем. Моя участь полягала в інтеграції звуків кроків із підтримкою різних типів поверхні, що додає динамічності та глибини ігровому середовищу.

Система враховує тег або фізичний матеріал поверхні, по якій рухається гравець, і відтворює відповідний аудіокліп (наприклад, дерев’яна підлога, плитка, бетон). Для цього використовується Raycast вниз від гравця або зчитування колайдера через OnCollisionStay.

Також до скрипта було додано:

– змінну затримку між кроками – залежно від швидкості пересування (біг, ходьба, присідання);

– перевірку стану руху – щоб уникнути відтворення звуку під час стояння або польоту.

3.3.3.3 Скрипт PlayerClimbing

Скрипт відповідає за логіку підйому по драбинах або поверхнях, і був створений іншим членом команди. З мого боку було реалізовано:

– прив’язку звуку карабкання до анімації підйому;

– врахування ритму руху – звук лунає тільки тоді, коли відбувається зміщення по вертикалі;

– ізоляцію звукового шару – щоб звуки підйому не накладались на звуки кроків або інші дії.

Завдяки цьому підйом по сходах або іншим вертикальним елементам гри сприймається більш реалістично та надає гравцю аудіовідчуття переміщення.

3.3.4 Супровід переміщення між локаціями

Переміщення між локаціями у грі супроводжується не лише візуальними змінами, а й звуковими ефектами, що дозволяють гравцеві чітко зрозуміти момент переходу та підсилити відчуття простору. Для цього було реалізовано відповідне аудіооформлення у межах скрипта TeleportSystem.

Безпосередню логіку телепортації реалізував інший учасник команди, проте саме звуковий супровід був інтегрований автором цієї частини дипломного проєкту. Зокрема, при активації телепорта (виклику функції переміщення на іншу сцену чи в межах сцени), програється короткий звуковий сигнал – це може бути характерний «блімк» або спокійна мелодія fade-out, що позначає завершення поточної дії.

Звуковий ефект викликається через AudioSource або централізовано через AudioManager, що дозволяє у майбутньому легко змінити аудіодоріжку, не змінюючи логіку самого скрипта. Це забезпечує гнучкість та повторне використання коду.

Основні завдання, реалізовані автором у цьому скрипті:

– ініціалізація аудіоджерела для телепортації;

– прив’язка звукового ефекту до дії переміщення;

– забезпечення одноразового запуску звуку на момент телепортації (щоб уникнути дублювання);

– за потреби – плавне загасання (fade out) фонової музики перед завантаженням нової локації.

Таким чином, супровід переміщень між локаціями виконує як інформативну, так і атмосферну функцію, допомагаючи гравцеві не втрачати контекст подій під час зміни сцен.

3.3.5 Аудіореакція NPC та взаємодія через звук

У грі реалізовано два ключові механізми, які забезпечують аудіовзаємодію з неігровими персонажами (NPC): озвучення діалогів та вплив на поведінку ворогів через звук. Обидва підходи працюють на спільну мету – підвищення рівня занурення гравця у світ гри та забезпечення зворотного зв’язку через звукову взаємодію.

3.3.5.1 Озвучення діалогів NPC: QuestGiverInteraction

Ключову роль у цій частині відіграє скрипт QuestGiverInteraction, який реалізує логіку діалогової взаємодії з NPC. Під час активації квесту або виконання умов, система показує діалогові «облачка», а також відтворює індивідуальний звуковий супровід для кожної репліки, заданий у масиві DialogueStep[].

Кожен елемент масиву DialogueStep містить поле AudioClip dialogueSound, що дозволяє призначати унікальний звук для конкретної репліки. Під час виконання діалогу метод ShowDialogueStep() викликає відтворення відповідного аудіо через AudioSource. Таким чином, діалоги не є беззвучними – кожна сцена доповнена короткими вигуками, шумами або «вигаданою мовою» персонажа.

Це рішення дозволяє:

- оживити сцену без повної озвучки;
- надати гравцеві інтуїтивний звуковий зворотний зв’язок;
- зробити діалоги динамічними, з урахуванням анімацій та графіки.

Крім того, реалізовано гнучке керування аудіо через AudioSource, який можна або підключити безпосередньо до NPC, або використати компонент ззовні, залежно від структури сцени. Таке рішення дає змогу розширювати функціональність, підключаючи озвучення як до основних, так і до другорядних персонажів.

3.3.5.2 Вплив на оточення через звук: PlayerWhistle та EnemyController

Однією з особливостей гри є можливість впливати на оточення за допомогою звуку, що реалізовано через механіку свисту. Ця система дозволяє гравцеві стратегічно взаємодіяти з ворогами (NPC), створюючи відволікаючі звукові сигнали.

Скрипт PlayerWhistle був розроблений іншим учасником команди, однак саме автор цієї частини проєкту реалізував логіку звукового супроводу: програвання звуків свисту залежно від обраного рівня гучності (слабкий, середній, сильний). Звукові ефекти згруповані в три масиви: weakWhistleSounds, mediumWhistleSounds, strongWhistleSounds, з яких обирається випадковий кліп.

Звуковий ефект супроводжується візуальним індикатором зони дії (префаб WhistleArea), що масштабно відображає силу сигналу. Після підтвердження дії через клавішу «R» виконується PerformWhistle(), що активує звук через AudioSource та розповсюджує його у сцені.

Відповідна реакція NPC реалізована у скрипті EnemyController, де метод MoveToNoise(Vector2) дозволяє ворогам рухатись у напрямку джерела шуму. Всі NPC у радіусі (визначеному через Physics2D.OverlapCircleAll) отримують цю команду, що створює динамічну реакцію оточення на дії гравця.

Особливості реалізації:

- зона дії звуку залежить від обраного рівня свисту;
- гравець має можливість викликати або відволікати ворогів;
- повна інтеграція звуку з геймплейною механікою;
- імерсивний аудіо-візуальний зворотний зв’язок.

Ця система легко масштабована: у майбутньому її можна розширити звуками від падіння предметів, відкриття дверей тощо, що дозволить ще більше поглибити взаємодію гравця зі світом гри через звук.

3.3.6 Реалізація ритмічної мінігри

Ритмічна мінігра виступає окремою геймплейною секцією у структурі гри, що поєднує аудіо та візуальні елементи, створюючи інтерактивний

музичний виклик для гравця. Вона реалізована як самостійна підсистема з точним обліком часу, синхронізацією з музикою та зворотним зв'язком. Мінігра є не лише розважальним елементом, а й важливою частиною сюжету, дозволяючи гравцеві зануритися у ритм подій, приймати рішення та тренувати реакцію.

Детальна реалізація ключових модулів, що відповідають за рітм-гру можно побачіти в Додатку В.

3.3.6.1 Архітектура модулів

Підсистема ритмічної мінігри реалізована на основі модульної архітектури, де кожен скрипт виконує окрему роль. Основні компоненти:

– StageData – зберігає BPM, масив нот, таймінги та інші параметри рівня. Він виступає джерелом правдивих музичних даних, зокрема часу появи нот (spawnTime), що є критичним для синхронізації.

– NoteSpawner – відповідає за генерацію нот на сцені у потрібний момент згідно з розрахунками на основі формули:

$$noteTime = 60 / BPM * beatNumber + timeOffset \quad (3.2)$$

Компонент враховує timeOffset для компенсації апаратних або затримок аудіо.

– Note – інкапсулює логіку руху ноти, перевірки на влучання (CheckHit) і обробки промаху (OnMiss). Кожна нота рухається у напрямку зони влучання з заданою швидкістю.

– NoteInput – обробляє введення гравця, зіставляє натискання клавіш із активними нотами у списку. Визначає, чи потрапив гравець у «вікно точності» (наприклад, ±0.15 с), і передає результат візуальному зворотному зв'язку.

– HitZoneFeedback – показує результат (влучання/промах) через анімації або зміну спрайтів.

– HealthDisplay – оновлює шкалу здоров'я, яка зменшується при промахах і зберігає прогрес гравця впродовж мінігри.

3.3.6.2 Алгоритмічна основа: таймінги та точність

Одним із головних викликів було забезпечити точне потрапляння нот у заданий ритм. Для цього використовується API Unity Time.deltaTime [13], який гарантує однакову швидкість обробки незалежно від FPS. Усі переміщення нот та зіставлення з часом дій користувача базуються на накопиченні часу у кожному кадрі.

Точність перевіряється шляхом віднімання Time.time від note.spawnTime та визначення, чи входить результат у заданий інтервал. Це дозволяє уникнути проблем із десинхронізацією і забезпечує стабільність механіки.

3.3.6.3 Зворотний зв’язок і система здоров’я

Після кожної дії гравця мінігра реагує негайно:

– При влучанні показується позитивний ефект (зелений спалах, звук підтвердження).

– При промаху – негативний (тремтіння, червоний спалах, звук невдачі).

Одночасно оновлюється HealthDisplay, що інформує гравця про прогрес або наближення до поразки.

3.3.6.4 Інтеграція з AudioManager

Усі звукові події (звук ноти, підтвердження, промах) передаються через AudioManager, що гарантує централізоване керування гучністю та каналами. Це дозволяє:

– Регулювати гучність окремо від решти гри;

– Застосовувати ефекти, наприклад Lowpass при зниженні здоров’я;

– Зберігати узгодженість у стилі звучання.

Мінігра також використовує налаштування гучності, обрані гравцем через AudioSettingsUI, тому повністю інтегрована в загальну звукову архітектуру.

3.3.7 Візуалізація структури: діаграма ієрархії підсистем

Для наочного відображення архітектури програмної реалізації та взаємозв’язків між окремими компонентами аудіосистеми в межах гри було

створено діаграму ієрархії підсистем із назвами скриптів (рис. 3.1). Ця діаграма відображає логічну структуру проєкту, де:

– вказані ключові модулі, що були реалізовані в рамках розробки аудіосистеми;

– виділено скрипти, написані іншими членами команди, в які було інтегровано функціонал відтворення звуку (напр., SettingsManager, PlayerMovement, TeleportSystem);

– позначено індивідуально реалізовані модулі, зокрема підсистему ритмічної мінігри (NoteSpawner, NoteInput, HealthDisplay та інші).

Ця діаграма є важливим інструментом для розуміння розподілу відповідальності в команді, а також допомагає простежити внесок автора саме в частині інтеграції та реалізації звукових механік. Завдяки цьому читач може легко ідентифікувати, які скрипти були змінені, доповнені або створені з нуля в рамках кваліфікаційної роботи, а також як вони співвідносяться з рештою системи.

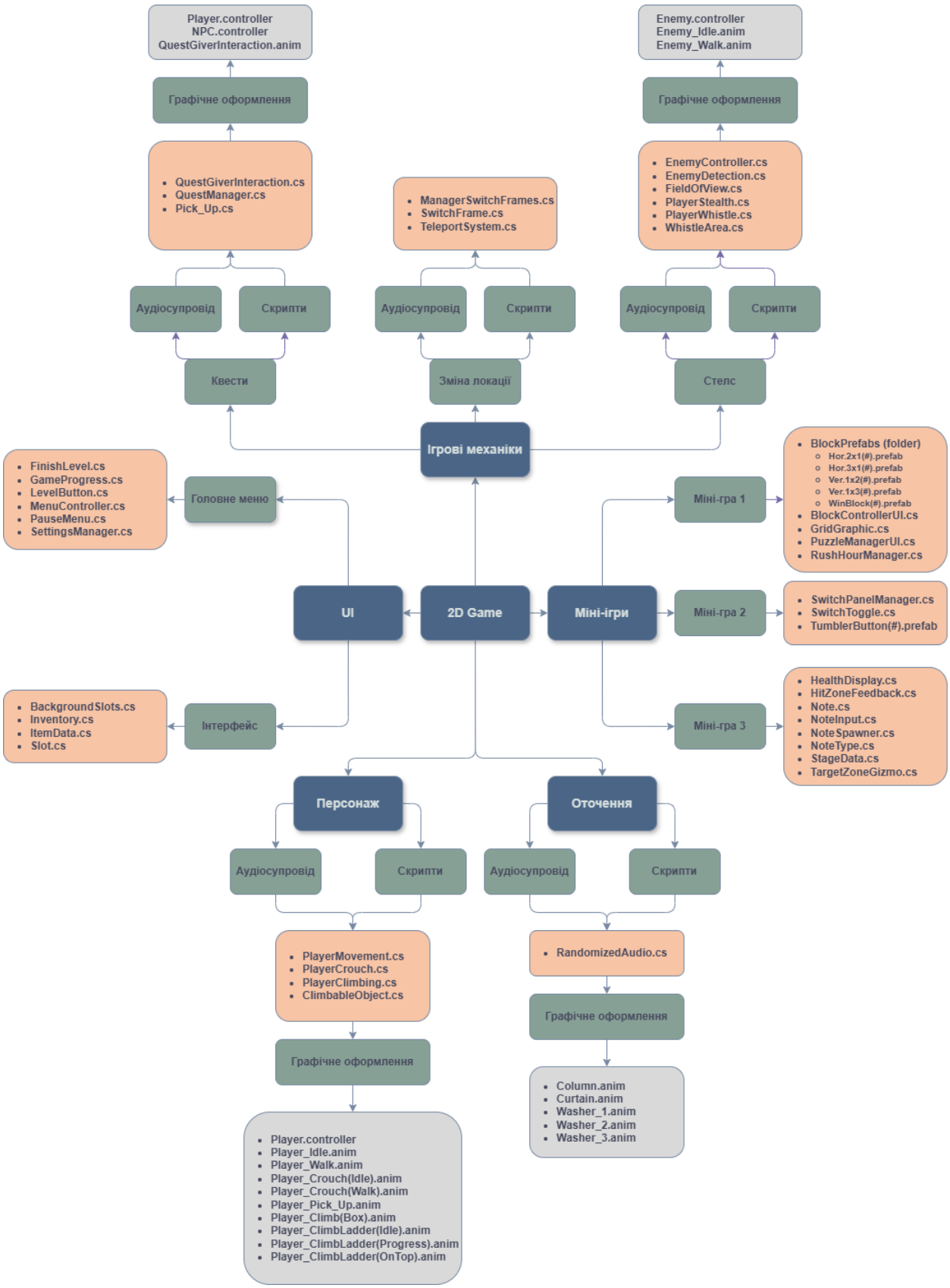

Рисунок 3.1 – Діаграма ієрархії підсистем

3.4 Тестування

3.4.1 Загальна методика тестування

Метою тестування аудіосистеми гри є забезпечення коректного функціонування всіх звукових компонентів, відповідність очікуваній поведінці під час взаємодії з гравцем, а також досягнення емоційного та інтуїтивного зворотного зв’язку. Звуковий супровід має не тільки працювати без збоїв, а й гармонійно вписуватись у загальний ігровий процес, підсилювати наратив і не викликати дискомфорту.

У процесі перевірки були застосовані різні типи тестування, що охоплюють як окремі елементи аудіосистеми, так і її взаємодію з іншими модулями гри.

Модульне тестування полягало в індивідуальній перевірці скриптів, пов’язаних із відтворенням та обробкою звуку. Наприклад, тестувались функції AudioManager, PlayerFootstepsWithSurfaces, PlayerWhistle, SettingsManager. У кожному з них перевірялась коректність виклику звукових подій, передача параметрів у AudioSource, наявність компонентів та виняткова поведінка при відсутності кліпів або неправильному налаштуванні.

Інтеграційне тестування передбачало перевірку взаємодії між модулями аудіосистеми. Зокрема, було протестовано зв’язок між AudioManager і AudioMixer, перевірено, як зміна налаштувань у SettingsManager відображається в мікшері. Також тестувалася взаємодія звукових тригерів з основним контролером сцен, а реакція NPC на звук – з логікою EnemyController.

Сценарне тестування (playtesting) проводилось у формі проходження гри з моделюванням типових ситуацій. Здійснювалась перевірка, чи звучать правильні ефекти у відповідний момент (наприклад, кроки, двері, діалоги, переміщення між локаціями, запуск мініігри). Уважно відстежувалися моменти зміни гучності, плавності переходів, коректності fade-in/fade-out, а також загальна узгодженість між діями гравця та звуковою реакцією.

Юзабіліті-тестування передбачало залучення сторонніх гравців до перевірки сприйняття аудіо. Їм пропонувалось налаштувати звук на свій розсуд, пройти кілька сцен, включно з ритмічною мінігрою, та надати зворотній зв’язок. Зокрема, оцінювались зручність інтерфейсу налаштувань, розбірливість звуків, відповідність гучності та ритму. Цей тип тестування дозволив виявити менш очевидні проблеми, пов’язані з балансом аудіоканалів та сприйняттям на різних типах пристроїв.

Слід враховувати, що тестування звуку має певну суб’єктивність, адже сприйняття гучності, частот і навіть тембру значною мірою залежить від слухача. Також велику роль відіграє середовище тестування – наявність чи відсутність навушників, акустика приміщення, гучність системи. Тому всі перевірки проводились у декількох умовах: на навушниках, вбудованих динаміках ноутбука та зовнішній акустиці.

Таким чином, багаторівнева методика тестування дозволила переконатися в коректності роботи звукової системи та її відповідності ігровим цілям.

3.4.2 Тестування аудіомодулів у рамках сцени

Для перевірки працездатності аудіосистеми в ігровому середовищі було проведено серію тестів у межах реальних сцен гри. Це дозволило оцінити, наскільки стабільно і коректно відтворюються звуки в інтерактивних умовах, а також перевірити взаємодію між окремими модулями. Основна увага приділялась зонам, де використовується тригери, реакція на позицію гравця, а також зміна середовища.

Особливу роль у тестуванні відіграв скрипт PlayerFootstepsWithSurfaces, що відповідає за відтворення звуків кроків залежно від типу поверхні. У процесі тестування було змодельовано переміщення персонажа по різних матеріалах – дерев’яній підлозі, бетонній плитці, килиму. Для кожної поверхні перевірялась відповідність звукового ефекту та його прив’язка до анімації. У разі присідання також перевірялась коректність перемикання темпу звуку.

Було встановлено, що система стабільно розпізнає зміну матеріалу під ногами та відтворює відповідний аудіофайл без затримок.

Тестування звуків у діалогових взаємодіях проводилося з використанням скрипта QuestGiverInteraction. Для кожного NPC перевірялись такі параметри, як відтворення звуку при активації репліки, реакція на тригери входу/виходу, а також відображення звукового супроводу для повторних діалогів. Звуки обирались індивідуально для кожного персонажа з відповідного масиву dialogueSound. У ході перевірки зверталась увага на баланс гучності між діалогом та фоновими ефектами.

Крім того, перевірялась функціональність зонального звуку за допомогою компонентів AudioTriggerZone. У сценах з поділом на внутрішні та зовнішні приміщення, при вході в конкретну зону запускалися відповідні атмосферні треки – наприклад, шум вітру надворі чи приглушене гудіння в коридорі. Тестування проводилось для ситуацій швидкого переходу між зонами та їх перекриття, з акцентом на плавність затухання (fade out) та появи (fade in) звуків. Було підтверджено, що система не допускає конфлікту кількох тригерів і стабільно визначає активну зону.

Окремо перевірялась поведінка AudioManager під час переходу між сценами. Для цього здійснювався перехід з меню до ігрової сцени, між локаціями, а також повернення назад. Було перевірено, чи не виникає накладення треків, чи зберігаються значення гучності, чи активується правильний фоновий трек у новій сцені. Система коректно завершувала старі звуки через fade out, а нові завантажувала з використанням кешованих ресурсів. Це дозволило уникнути затримок та збоїв при зміні середовища.

Таким чином, у результаті сценового тестування було підтверджено функціональність усіх ключових аудіомодулів, їхню здатність адаптуватися до контексту та стабільну роботу у реальному ігровому часі.

3.4.3 Тестування ритмічної мінігри

Ритмічна мінігра є одним із найбільш чутливих до таймінгу елементів гри, тому її тестування вимагало особливої уваги до точності синхронізації

аудіо та візуальних елементів. Основним завданням було перевірити відповідність появи нот заданому ритму, коректність сприйняття введення користувача та стабільність поведінки при зміні технічних умов (наприклад, FPS).

Передусім проводилась перевірка таймінгів нот відповідно до BPM (beats per minute) композиції. На основі об'єкта StageData визначались очікувані моменти появи кожної ноти згідно з формулою:

$$noteTime = 60 / BPM * beatNumber + timeOffset \quad (3.3)$$

У ході тестування зверталась увага на те, чи з’являються ноти у візуальній зоні відповідно до аудіо. Підтверджено, що система NoteSpawner створює об'єкти з точною прив’язкою до музичного ритму.

Особливо ретельно перевірялась механіка попадання та промаху, реалізована у скрипті NoteInput. Гравець мав натискати клавіші у момент, коли нота входить у зону взаємодії (HitZone). Була протестована система вікна точності (tolerance window), яка фіксує влучання при відхиленні ±0.15 сек від ідеального таймінгу. У разі перевищення цього порогу система фіксувала промах. Результати обробки виводились через візуальні та звукові сигнали (зворотний зв’язок), що дозволило легко перевірити логіку роботи під час тестування.

Окреме тестування проводилось у симульованому середовищі з низькою продуктивністю (зменшено FPS до 20–30 кадрів за секунду), щоб перевірити, як система справляється з відліком часу через Time.deltaTime. Було встановлено, що незалежно від частоти кадрів, інтервали появи нот та обробка натискань залишаються стабільними. Це засвідчує, що час у грі відстежується правильно, з урахуванням накопиченої дельти кадрів, що критично для синхронізованого геймплею.

Додаткову увагу приділено системі HitZoneFeedback, яка відповідає за відтворення звуків попадання або промаху. Було протестовано баланс гучності

звуків фідбеку відносно фонової музики, а також візуальні ефекти (спалахи, зміна кольору). Усі звукові сигнали чітко співвідносяться з діями гравця, не затримуються і не викликають артефактів. Це підтверджує успішну інтеграцію аудіозворотного зв’язку в загальну систему.

Загалом, ритмічна мінігра пройшла повний цикл тестування, продемонструвавши високу точність синхронізації та стійкість до технічних змін. Система виявилася надійною та адаптивною, що особливо важливо для механік, де звук є основним орієнтиром.

3.4.4 Тестування системи налаштувань

Тестування системи звукових налаштувань проводилось з метою перевірити коректність взаємодії між користувацьким інтерфейсом, внутрішньою логікою SettingsManager, параметрами AudioMixer та збереженням конфігурації через PlayerPrefs. Уся функціональність тестувалась як у межах окремої сцени з налаштуваннями, так і під час звичайного ігрового процесу.

Першим етапом стала перевірка відповідності значень повзунків (Sliders) у меню налаштувань реальному ефекту на звук. Для цього змінювались значення загальної гучності (Master Volume), гучності музики (Music Volume), звукових ефектів (SFX Volume) та атмосферних звуків (Ambience Volume), після чого тестувалося, чи змінюються відповідні параметри в AudioMixer. Усі зміни коректно передавались через метод AudioMixer.SetFloat(), а аудіо сигнал негайно реагував на зміни рівня, що свідчить про правильну реалізацію логіки прив’язки значень UI до мікшера.

Другим важливим етапом була перевірка механізму збереження значень налаштувань. Після зміни гучності значення записувались у PlayerPrefs через метод PlayerPrefs.SetFloat() у SettingsManager. У результаті перезапуску гри значення зчитувались на старті через PlayerPrefs.GetFloat() і встановлювались для всіх відповідних параметрів. Це дозволило підтвердити, що гравець не втрачає індивідуальні аудіоналаштування після виходу з гри.

Окремо тестувалась функція відтворення тестового звуку при зміні гучності. Під час переміщення повзунка для кожного з параметрів одразу відтворювався відповідний короткий аудіокліп, що дозволяло миттєво оцінити результат налаштування. Такий зворотний зв’язок сприяв підвищенню зручності для користувача і забезпечував швидке налаштування без потреби повертатися до гри.

Фінальна перевірка стосувалась повторного запуску гри: після її перезапуску було підтверджено, що значення всіх налаштувань зберігаються і завантажуються автоматично. Це свідчить про стабільність реалізованої логіки та повну інтеграцію збереження параметрів у систему налаштувань Unity.

Таким чином, система звукових налаштувань показала високу надійність і стабільність у роботі, забезпечуючи інтуїтивну взаємодію з користувачем та відповідність очікуванням. Це дозволяє кожному гравцеві адаптувати звучання гри під власні вподобання, не порушуючи загального балансу звуку.

3.4.5 Тестування звукових механік геймплею

У межах загального тестування аудіосистеми особливу увагу було приділено перевірці звукових механік, які безпосередньо пов’язані з ігровим процесом. Ці механіки не лише супроводжують дії гравця, але й мають функціональний вплив на навколишнє середовище та NPC. Тестування проводилося в декількох тематичних напрямках.

Механіка свисту (PlayerWhistle) перевірялась з погляду коректності запуску, програвання звуку, масштабування зони дії та реакції ворогів. Гравець має змогу вибрати рівень сили свисту, що впливає на радіус зони – ці значення тестувались через візуальний індикатор (WhistleArea). Після активації свисту викликалась функція PerformWhistle(), яка програвала аудіокліп із масиву відповідного рівня гучності (слабкий, середній, сильний), а також шукала NPC у визначеному радіусі. За результатами тестів підтверджено: усі вороги, які перебували в межах зони, реагували та рухались

у напрямку гравця, що доводить правильну роботу логіки EnemyController.MoveToNoise().

Перехід між локаціями перевірявся через систему TeleportSystem, в яку було інтегровано звукове оформлення. Перед перемиканням сцен програвався короткий аудіоефект, що сигналізував про телепортацію. Це забезпечувало гравцеві сенсорний перехід і сприяло сприйняттю безшовного переходу. Було протестовано декілька варіантів телепортацій – між гуртожитком, поверхами, кімнатами - у всіх випадках звук коректно спрацьовував до завантаження нової сцени.

Взаємодія з об'єктами та інтерфейсом також супроводжувалась аудіовідгуком. Наприклад, взаємодія з інвентарем, відкриття дверей, натискання кнопок в меню або активація діалогів через QuestGiverInteraction – кожна з цих дій викликала програвання відповідного звуку через AudioManager або безпосередньо з AudioSource. Було перевірено реакцію на повторні натискання, швидкі перемикання та неочікувану зміну сцен – звукові ефекти при цьому залишались стабільними, без накладання чи конфліктів між джерелами.

Усі результати підтвердили, що звукові механіки геймплею працюють стабільно, взаємодіють з логікою гри й підсилюють занурення гравця у віртуальне середовище. Це забезпечує не лише атмосферу, але й функціональний зв’язок між діями користувача та реакцією світу гри.

3.4.6 Висновки за результатами тестування

У результаті проведеного тестування аудіосистеми гри можна зробити кілька важливих висновків щодо її стабільності, якості реалізації та потенціалу до подальшого розвитку.

Під час тестів були виявлені незначні помилки. Наприклад, на ранніх етапах у деяких сценах спостерігалося накладання кількох звуків при швидкому натисканні UI-кнопок, що згодом було усунуто шляхом обмеження повторного виклику звуку. Також на окремих пристроях із низькою продуктивністю затримки у відтворенні нот у ритмічній мінігрі ставали

помітнішими. Ця проблема була частково вирішена через оптимізацію коду, використання Time.deltaTime і зменшення графічного навантаження у відповідних моментах.

У цілому система показала високу стабільність. Відтворення звуків було надійним у всіх типових сценаріях: у діалогах, під час навігації по локаціях, у мінігрі, при взаємодії з предметами. AudioManager коректно обробляв переходи між сценами та не викликав збоїв, а параметри, збережені в PlayerPrefs, завжди успішно застосовувались після повторного запуску гри.

Особливо цінним стало тестування з залученням гравців. За їхніми відгуками, звучання у грі добре передавало атмосферу, було чітко прив’язане до подій, а звук свисту та діалогів додавали живості. Також було позитивно відзначено, що мінігра зосереджена на ритмі, а не лише на візуальних елементах, що підвищує залученість.

Попри стабільну реалізацію, були визначені перспективи для вдосконалення. Серед них:

– додавання просторового звуку для об’ємного позиціонування джерел;

– розширення варіацій аудіоефектів, щоб уникнути монотонності;

– адаптивне коригування гучності залежно від подій гри (наприклад, у моменті напруження – автоматичне приглушення фонових звуків).

Таким чином, тестування підтвердило, що аудіосистема виконує всі заявлені функції, відповідає технічним і художнім вимогам гри та має потенціал для подальшого розширення з урахуванням отриманого досвіду.

Висновок до розділу

У третьому розділі було детально розглянуто як засоби розробки, так і повну реалізацію аудіосистеми в 2D-пригодницькій грі. На першому етапі було обґрунтовано вибір інструментів – середовища Unity, редакторів коду та аудіо, а також наведено вимоги до технічного та програмного забезпечення для забезпечення стабільної роботи гри на різних пристроях.

Основна частина розділу присвячена аналізу реалізації аудіофункціональності. Була описана централізована архітектура з використанням AudioManager, яка дозволяє ефективно керувати звуками в межах усієї гри. Деталізовано описано модулі, реалізовані як самостійно, так і у співпраці з іншими членами команди, включаючи SettingsManager, PlayerMovement, PlayerWhistle, QuestGiverInteraction та інші.

Окрема увага приділена ритмічній мінігрі – її модульна структура, математична точність синхронізації нот та інтеграція в загальну систему аудіо. Також розглянуто, як було забезпечено узгоджене відтворення звуку під час дій гравця, переходів між локаціями, взаємодії з NPC і використання ігрових механік.

Важливим доповненням до опису стала діаграма ієрархії підсистем, яка дозволяє побачити зв’язки між скриптами, розділити відповідальність у межах команди та визначити особистий внесок автора у розробку аудіосистеми.

Таким чином, у межах третього розділу було реалізовано ключові завдання дипломного проєкту: побудовано масштабовану, модульну, інтегровану аудіосистему, яка відповідає як технічним, так і геймдизайнерським вимогам сучасної 2D-гри.

РОЗДІЛ 4
ОХОРОНА ПРАЦІ

4.1 Регулювання питань охорони праці на законодавчому рівні

Охорона праці в Україні є обов’язковим елементом організації будь-якого виду професійної діяльності та регулюється низкою законодавчих і нормативно-правових актів. Основу правового регулювання становить Закон України «Про охорону праці» [14], який визначає права та обов’язки роботодавців і працівників у галузі забезпечення безпеки, гігієни праці та виробничого середовища. Крім того, до основних документів належать Кодекс законів про працю України, Конституція України, Кодекс цивільного захисту України, а також міжнародні документи – Конвенція МОП №187, Директива Ради ЄС 89/391/ЄЕС, стандарти ISO 45001:2019, ISO 26000 [16] тощо.

Діяльність у межах дипломного проєкту – створення аудіосупроводу та інтеграція звукових ефектів у 2D-гру – передбачає роботу в умовах офісного або дистанційного середовища з використанням комп’ютерної та звукової техніки. Основними факторами, що потребують регулювання з погляду охорони праці, є: тривала робота за комп’ютером, навантаження на слуховий апарат під час обробки звуку в навушниках, психоемоційне та когнітивне перевантаження, а також електробезпека при використанні електронного та аудіообладнання [15][19][20].

Для забезпечення безпеки в таких умовах застосовуються положення Державних санітарних норм і правил, зокрема: ДСН 3.3.6.042-99 – щодо мікроклімату приміщень, ДБН В.2.5-28:2018 – щодо освітлення, ДСанПіН 3.3.6.096-2002 – щодо електромагнітного випромінювання. Також враховуються норми гранично допустимого шумового навантаження згідно з ДСН 3.3.6.037-99 [18].

Регулювання охорони праці в ІТ-сфері, зокрема в ігровій розробці зі звуковою компонентою, є важливою складовою професійної безпеки. З одного

боку, воно забезпечує збереження здоров’я та працездатності розробника, з іншого – сприяє підвищенню ефективності та якості кінцевого продукту. Врахування вимог законодавства в процесі проєктування програмного забезпечення – це не лише вимога часу, але й прояв відповідального підходу до сучасної цифрової праці [15][16].

4.2. Виявлення потенційних небезпек стосовно об’єкта проєктування

Розробка аудіосистеми для 2D-гри в середовищі Unity передбачає тривалу роботу за комп’ютером із використанням спеціалізованого програмного забезпечення (Unity, цифрові аудіостанції, редактори звуку) та аудіообладнання (навушники, мікрофони, звукові карти). Така діяльність пов’язана з низкою потенційно небезпечних і шкідливих виробничих факторів [15],[18],[19] які класифікуються відповідно до їх природи дії (табл. 4.1).

Таблиця 4.1 – Класифікація небезпечних і шкідливих виробничих факторів у проєкті

Група факторів	Конкретні прояви у проєкті	Можливі наслідки
Фізичні	Підвищений рівень шуму в навушниках, недостатня освітленість, перегрів обладнання, висока температура повітря, вібрації від зовнішньої техніки	Погіршення слуху, зору, загальна втома
Хімічні	Можливе використання флюсів та паяння при ремонті техніки	Інгаляція шкідливих речовин, подразнення
Психофізіологічні	Тривала статична поза, розумове перенапруження, монотонність завдань, емоційне вигорання	Порушення постави, хронічна втома, стрес
Електричні	Недостатнє заземлення, використання кількох пристроїв одночасно	Ураження електрострумом, перегрів проводки

Детальніше:

1. Фізичні фактори в основному пов’язані з умовами праці: освітленням, шумовим навантаженням (особливо при роботі з навушниками понад 85 дБ), температурою приміщення [17]. Також варто враховувати тривале перебування в статичній позі.

2. Хімічні фактори можуть з’явитися у разі обслуговування технічного обладнання вручну – наприклад, при ремонті акустичних систем або паянні мікросхем звукових інтерфейсів. Це супроводжується виділенням парів флюсу та інших речовин.

3. Біологічні фактори не мають суттєвого впливу в умовах розробки гри, якщо не йдеться про специфічні лабораторні або польові умови.

4. Психофізіологічні фактори становлять один із головних ризиків: програмісти й звукорежисери часто працюють в умовах монотонної, але інтенсивної діяльності, що викликає втомлюваність, синдром «вигорання», втрату концентрації [15],[19].

5. Електротехнічні фактори пов’язані з великою кількістю обладнання: ПК, монітори, аудіоінтерфейси, студійні монітори, які можуть створювати небезпеку коротких замикань, перевантаження мережі або ураження струмом.

Підсумовуючи, робота над аудіо в грі передбачає переважно фізичні, психофізіологічні та електротехнічні небезпеки [15], які можуть бути усунені або мінімізовані шляхом впровадження організаційних і технічних заходів безпеки, що будуть розглянуті в наступному підрозділі.

4.3 Дослідження ризику реалізації потенційних небезпек на об’єкті проєктування та розробка заходів щодо їх попередження

Процедура оцінювання професійних ризиків є важливою складовою системи управління охороною праці [15]. Її метою є виявлення потенційно небезпечних ситуацій, визначення рівня ризику їх реалізації та розробка

заходів із запобігання чи мінімізації впливу таких загроз. Завданнями оцінки ризиків є:

– ідентифікація джерел небезпеки;
– визначення ймовірності реалізації небезпеки та можливих наслідків;
– класифікація ризиків за рівнем прийнятності;
– розробка технічних, організаційних або профілактичних заходів.

Відповідно до методичних вказівок, для проведення кількісної оцінки ризиків використовується матриця оцінки ризиків, яка базується на двох параметрах: ймовірність події (P) та серйозність наслідків (S). Комбінація цих факторів дозволяє оцінити рівень ризику за шкалою від допустимого до неприпустимого (рис. 4.1).

	Малоймовірна	Можлива	Ймовірна
Низька	Допустимий	Допустимий	Середній
Середня	Допустимий	Середній	Високий
Висока	Середній	Високий	Неприпустимий

Рисунок 4.1 – Приклад - матриця оцінки ризиків

Кольорове кодування: зелений – допустимий; жовтий – середній; помаранчевий – високий; червоний – неприпустимий рівень ризику.

4.3.1 Приклади оцінки типових ризиків

На основі даних, зібраних у попередньому підрозділі, проаналізуємо кілька характерних небезпек, які мають місце в роботі розробника аудіосупроводу в 2D-грі (табл. 4.2).

Таблиця 4.2 – Матриця оцінки ризиків

№	Назва небезпеки	Серйозність наслідків	Ймовірність виникнення	Рівень ризику	Рекомендації
1	Перевантаження слуху при роботі в навушниках	Висока	Середня	Високий	Обмежувач гучності, перерви щогодини, LUFS-моніторинг
2	Психоемоційне виснаження при монотонній роботі	Середня	Висока	Високий	Регламентовані перерви, чергування завдань, профілактика вигорання
3	Порушення постави при незручному положенні тіла	Середня	Висока	Високий	Ергономічне крісло, налаштування монітора, розминка кожні 45–60 хв
4	Ураження електричним струмом від несправної техніки	Висока	Низька	Середній	Технічна перевірка, заземлення, мережеві фільтри

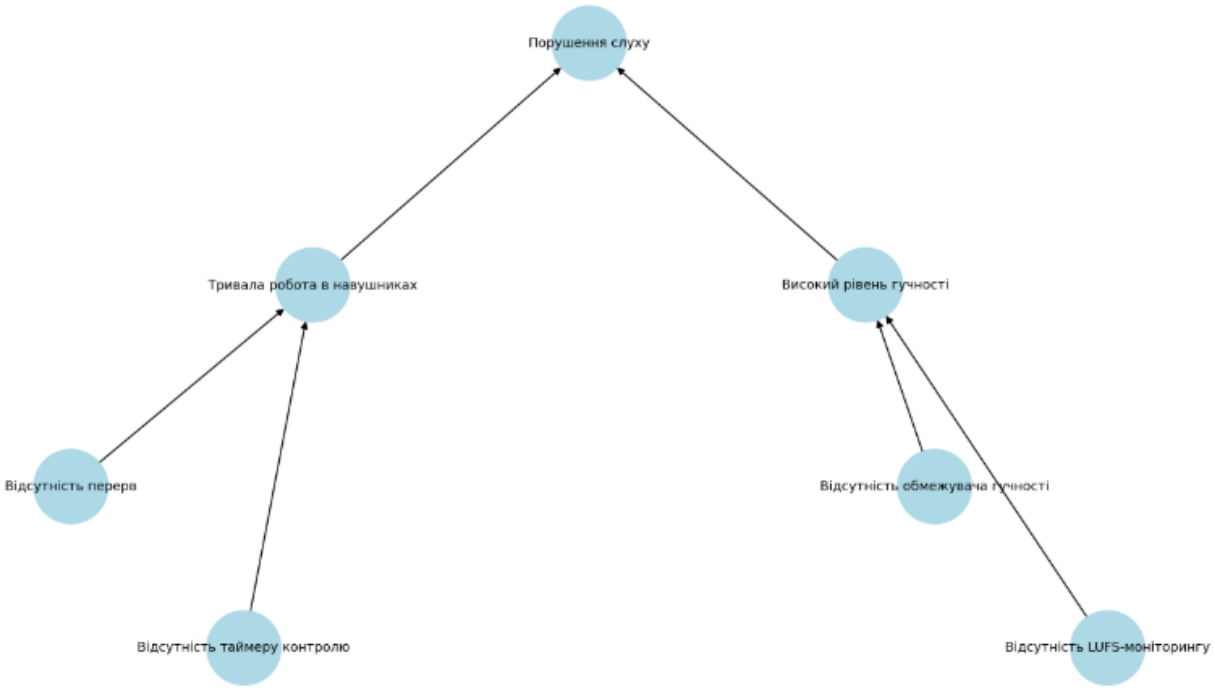

Рисунок 4.2 – Дерево відмов «Порушення слуху»

Опис роботи дерева відмов «Порушення слуху»(рис. 4.2):

Основна подія: Порушення слуху розробника

Причини першого рівня:

– тривала робота в навушниках;
– високий рівень гучності.

Причини другого рівня:

– відсутність перерв;
– відсутність таймеру контролю;
– відсутність обмежувача гучності;
– Відсутність LUFS-моніторингу.

Це дерево демонструє, що фінальна небажана подія виникає через сукупність незалежних факторів, усунення будь-якого з яких вже знижує ризик.

4.3.2 Рекомендовані заходи безпеки

1) Технічні:

– сертифіковані навушники з лімітом гучності (до 85 дБ);

– аудіоінтерфейси з моніторингом LUFS та обмежувачем рівня виходу;

– ергономічне робоче місце (висота столу, положення монітора, опора для ніг).

2) Організаційні:

– перерви щогодини (режим 50/10 або Pomodoro);

– чіткий графік роботи, планування завдань;

– чергування технічних і творчих задач для уникнення монотонності.

3) Санітарно-гігієнічні:

– провітрювання кімнати, температурний режим 20–24°C, вологість 40–60%;

– контроль шумового навантаження на фоні (≤ 40 дБ);

– щоденні вправи для очей і шиї.

4) Превентивні:

– регулярна перевірка технічного стану обладнання;

– проведення первинного інструктажу з охорони праці;

– використання мережевих фільтрів і пристроїв захисту від перенапруги.

Завдяки поетапному аналізу і класифікації типових загроз, а також побудові дерева відмов та матриці ризиків, було запропоновано низку дієвих рішень. Їх реалізація дає змогу знизити ймовірність професійних захворювань, підвищити продуктивність праці та загальний комфорт робочого процесу розробника аудіосистем гри.

Висновки до розділу

Метою розділу «Охорона праці» є аналіз факторів ризику та розробка заходів щодо забезпечення безпечних умов праці на робочому місці розробника, який займається створенням аудіосупроводу в 2D-грі в середовищі Unity. У розділі розглянуто законодавче регулювання охорони праці [14][16] ідентифіковано типові шкідливі та небезпечні фактори, властиві для офісного середовища [15][18][19] IT-фахівця, зокрема звукового

дизайнера, а також досліджено ризик реалізації цих факторів із використанням матриці ризиків та дерева відмов [15][20].

Проведено аналіз чотирьох типових ситуацій, серед яких: перевантаження слухового апарату, психоемоційне виснаження, порушення постави та ризик ураження електрострумом. Для кожного з цих випадків визначено рівень ризику та надано обґрунтовані рекомендації щодо зменшення його до допустимого рівня.

Запропоновані заходи поділено на технічні (ергономічне обладнання, аудіомоніторинг), організаційні (регламентовані перерви, планування задач), санітарно-гігієнічні (провітрювання, контроль мікроклімату) та превентивні (перевірка обладнання, інструктажі). Їх впровадження дозволяє істотно знизити ймовірність розвитку професійних захворювань, підвищити продуктивність, концентрацію та загальний добробут працівників у сфері цифрового звукового дизайну.

Таким чином, поставлена мета розділу досягнута, а запропоновані заходи є ефективними та актуальними для проєктування й організації безпечного робочого середовища у сфері створення аудіо для ігор.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено, спроєктовано та реалізовано інтерактивну аудіосистему для 2D-гри у жанрі пригодницького квесту. Основна увага приділялась не лише технічному відтворенню звуків, але й глибокій інтеграції аудіо в ігрову логіку, з урахуванням динаміки подій, взаємодій із гравцем, сценарних тригерів та наративних зон.

Було доведено, що звук у сучасній грі перестає бути другорядним елементом – він перетворюється на повноцінний засіб керування емоційним сприйняттям, зануренням у світ гри та навіть на механіку впливу на геймплей (як у випадку реалізації системи свисту чи ритмічної мінігри). Такий підхід став можливим завдяки використанню рушія Unity та програмної мови C#, які надали всі необхідні інструменти для створення гнучкої, реактивної та масштабованої аудіосистеми.

Протягом роботи було виконано повний цикл розробки: від аналізу предметної області та вивчення аналогів до створення власної архітектури звукової підсистеми. Основні компоненти системи – AudioManager, SettingsManager, AudioMixer, PlayerFootstepsWithSurfaces, QuestGiverInteraction, модулі мінігри та інші – були інтегровані в єдину структуру з централізованим керуванням, підтримкою гучності, пріоритетності каналів і обробки тригерів.

Особливої уваги заслуговує реалізація ритмічної мінігри, яка поєднала аудіо з точною обробкою інпуту користувача, синхронізацією таймінгів та візуальним зворотним зв’язком. Це стало своєрідною демонстрацією потенціалу звукової взаємодії у геймплеї.

Під час тестування були виявлені окремі незначні проблеми синхронізації в умовах низької продуктивності, однак загальна стабільність системи була підтверджена на різних пристроях і середовищах. Результати юзабіліті-тестування продемонстрували, що більшість гравців позитивно

сприймають інтеграцію звуку, особливо в частині реактивних ефектів, діалогових сигналів і мінігри.

Таким чином, у межах цієї кваліфікаційної роботи було успішно досягнуто поставленої мети – створити повноцінний інтерактивний модуль аудіосупроводу, який органічно доповнює ігровий процес, підтримує наративну структуру гри та збагачує емоційний досвід гравця. Отримані результати можуть бути використані як основа для подальшого розширення проєкту, зокрема, додавання системи динамічної генерації музики, просторового 3D-звучання або інтеграції сторонніх рішень (FMOD, Wwise) з метою підвищення гнучкості налаштувань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Asteroids // Wikipedia. – URL: https://en.wikipedia.org/wiki/Asteroids_(video_game) – (дата звернення 23.05.2025)

2. Osu! // Wikipedia. – URL: https://en.wikipedia.org/wiki/Osu! (дата звернення 23.05.2025)

3. Guitar Hero // Wikipedia. – URL: https://en.wikipedia.org/wiki/Guitar_Hero (дата звернення 23.05.2025)

4. Valiant Hearts: The Great War // Ubisoft. – URL: https://www.ubisoft.com/en-gb/game/valiant-hearts – (дата звернення 23.05.2025)

5. INSIDE & LIMBO Bundle // Steam. – URL: https://store.steampowered.com/bundle/1837/INSIDE__LIMBO/ – (дата звернення 23.05.2025)

6. Unity – Official Site // Unity Technologies. – URL: https://unity.com – (дата звернення 23.05.2025)

7. My Brother Rabbit // Steam. – URL:https://store.steampowered.com/app/855640/My_Brother_Rabbit/ – (дата звернення 23.05.2025)

8. Fran Bow // Steam. – URL: https://store.steampowered.com/app/362680/Fran_Bow/ (дата звернення 23.05.2025)

9. Lowrider Challenge // GTA Wiki. – URL: https://gta.fandom.com/wiki/Lowrider_Challenge (дата звернення 23.05.2025)

10. Undertale // Wikipedia. – URL: https://en.wikipedia.org/wiki/Undertale – (дата звернення 23.05.2025)

11. John Leonard French. The Right Way To Make A Volume Slider In Unity Using Logarithmic Conversion // johnleonardfrench.music. – URL: https://johnleonardfrench.music/the-right-way-to-make-a-volume-slider-in-unity-using-logarithmic-conversion – (дата звернення 16.06.2025)

12. John Leonard French. The Right Way To Make A Volume Slider In Unity Using Logarithmic Conversion // johnleonardfrench.music. – URL: https://johnleonardfrench.music/the-right-way-to-make-a-volume-slider-in-unity-using-logarithmic-conversion – (дата звернення 16.06.2025)

13. Time.deltaTime Explained | Unity For Beginners | Unity Tutorial // YouTube. – URL: https://www.youtube.com/watch?v=8pYq15Lh0x4 – (дата звернення 16.06.2025)

14. Закон України «Про охорону праці» // Верховна Рада України. – URL: https://zakon.rada.gov.ua/laws/show/2694-12 – (дата звернення 17.06.2025)

15. Методичні вказівки до виконання розділу «Охорона праці» для дипломних проєктів бакалавра // ХНУМГ ім. О.М. Бекетова, 2021. – (дата звернення 17.06.2025)

16. ДСТУ ISO 45001:2019. Системи управління охороною здоров’я та безпекою праці // Офіційний сайт ДП «УкрНДНЦ». – URL: https://uas.org.ua/standards/dstu-iso-45001-2019 – (дата звернення 17.06.2025)

17. ДСанПіН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень // МОЗ України. – URL: https://zakon.rada.gov.ua/rada/show/v0042282-99 – (дата звернення 17.06.2025)

18. Бедрій Я.І. Основи охорони праці користувачів ПК. – Тернопіль: Навчальна книга – Богдан, 2014. – (дата звернення 17.06.2025)

19. Піскунова Л.Е. Безпека життєдіяльності. – Київ: Академія, 2014. – (дата звернення 17.06.2025)

20. Додатки 1 і 2 до методичних вказівок // ХНУМГ ім. О.М. Бекетова. – (дата звернення 17.06.2025)

21. SAE International. Guidelines for Sound Pressure Level Monitoring in Headphones. – URL: https://www.sae.org/publications/technical-papers/content/2005-01-1847/ – (дата звернення 17.06.2025)

22. AES Recommendations for Loudness in Game Audio (AES69-2020) // Audio Engineering Society. – URL: https://www.aes.org/standards/ – (дата звернення 17.06.2025)

ДОДАТКИ

Дотаток А

Лістинг коду аудисистеми взаємодії гравця з меню та налаштування звуку

ButtonSound:

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;

public class ButtonSound : MonoBehaviour, IPointerEnterHandler, IPointerClickHandler
{
    public AudioSource; // Посилання на AudioSource
    public AudioClip hoverSound;    // Звук при наведенні
    public AudioClip clickSound;    // Звук при натисканні

    // Событие наведения курсора
    public void OnPointerEnter(PointerEventData eventData)
    {
        if (audioSource != null && hoverSound != null)
        {
            audioSource.PlayOneShot(hoverSound, 1.0f);
        }
    }

    // Подія наведення курсору
    public void OnPointerClick(PointerEventData eventData)
    {
        if (audioSource != null && clickSound != null)
        {
            audioSource.PlayOneShot(clickSound, 1.0f);
        }
    }

    // Автоматично додаємо обробник OnClick
    void Start()
    {
        Button = GetComponent<Button>();
        if (button != null)
        {
            button.onClick.AddListener(() => OnPointerClick(null));
        }
    }
}
```

SoundMixerManager:

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

public class SoundMixerManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
```

```
    {

    }
}
```

SettingsManager

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Audio;
using TMPro;

public class SettingsManager : MonoBehaviour
{
    [Header("Settings UI")]
    public Slider sliderMasterVolume;
    public Slider sliderMusicVolume;
    public Slider sliderSFXVolume;
    public Slider sliderAmbientVolume;
    public TMP_Dropdown dropdownResolution;
    public Toggle toggleFullscreen;
    public Toggle toggleVSync;
    public Button btnSave;
    public Button btnBackFromSettings;

    [Header("Audio")]
    public AudioMixer;

    private Resolution[] resolutions;
    private    float    savedMasterVolume,    savedMusicVolume,    savedSFXVolume,
savedAmbientVolume;
    private int savedResolutionIndex;
    private bool savedFullscreen, savedVSync;
    private bool hasChanges;

    private void Start()
    {
        SetupSettings();
    }

    private void SetupSettings()
    {
        // Перевіряємо, чи це перший запуск
        bool isFirstLaunch = !PlayerPrefs.HasKey("Initialized");

        // Початкові значення для першого запуску
        if (isFirstLaunch)
        {
            // Роздільна здатність: 1920x1080
            savedResolutionIndex = FindResolutionIndex(1920, 1080);
            PlayerPrefs.SetInt("ResolutionIndex", savedResolutionIndex);

            // Звук: 80% (0.8)
            savedMasterVolume = 0.8f;
            savedMusicVolume = 0.8f;
            savedSFXVolume = 0.8f;
            savedAmbientVolume = 0.8f;
            PlayerPrefs.SetFloat("MasterVolume", savedMasterVolume);
            PlayerPrefs.SetFloat("MusicVolume", savedMusicVolume);
            PlayerPrefs.SetFloat("SFXVolume", savedSFXVolume);
            PlayerPrefs.SetFloat("AmbientVolume", savedAmbientVolume);

            // Повноекранний режим: увімкнено
            savedFullscreen = true;
```

```
            PlayerPrefs.SetInt("Fullscreen", 1);

            // V-Sync: вимкнено (за замовчуванням)
            savedVSync = false;
            PlayerPrefs.SetInt("VSync", 0);

            PlayerPrefs.SetInt("Initialized", 1);
            PlayerPrefs.Save();
        }
        else
        {
            // Завантажуємо збережені значення
            savedMasterVolume = PlayerPrefs.GetFloat("MasterVolume", 0.8f);
            savedMusicVolume = PlayerPrefs.GetFloat("MusicVolume", 0.8f);
            savedSFXVolume = PlayerPrefs.GetFloat("SFXVolume", 0.8f);
            savedAmbientVolume = PlayerPrefs.GetFloat("AmbientVolume", 0.8f);
            savedResolutionIndex       =       PlayerPrefs.GetInt("ResolutionIndex",
FindResolutionIndex(1920, 1080));
            savedFullscreen = PlayerPrefs.GetInt("Fullscreen", 1) == 1;
            savedVSync = PlayerPrefs.GetInt("VSync", 0) == 1;
        }

        // Ініціалізація UI
        sliderMasterVolume.value = savedMasterVolume;
        sliderMusicVolume.value = savedMusicVolume;
        sliderSFXVolume.value = savedSFXVolume;
        sliderAmbientVolume.value = savedAmbientVolume;

        resolutions = Screen.resolutions;
        dropdownResolution.ClearOptions();
        int currentResolutionIndex = savedResolutionIndex;
        for (int i = 0; i < resolutions.Length; i++)
        {
            string option = $"{resolutions[i].width}x{resolutions[i].height}";
            dropdownResolution.options.Add(new TMP_Dropdown.OptionData(option));
            if (resolutions[i].width == Screen.currentResolution.width &&
                resolutions[i].height == Screen.currentResolution.height)
            {
                currentResolutionIndex = i;
            }
        }
        dropdownResolution.value = currentResolutionIndex;
        dropdownResolution.RefreshShownValue();

        toggleFullscreen.isOn = savedFullscreen;
        toggleVSync.isOn = savedVSync;

        // Прив'язуємо події
        sliderMasterVolume.onValueChanged.AddListener(OnMasterVolumeChanged);
        sliderMusicVolume.onValueChanged.AddListener(OnMusicVolumeChanged);
        sliderSFXVolume.onValueChanged.AddListener(OnSFXVolumeChanged);
        sliderAmbientVolume.onValueChanged.AddListener(OnAmbientVolumeChanged);
        dropdownResolution.onValueChanged.AddListener(OnResolutionChanged);
        toggleFullscreen.onValueChanged.AddListener(OnFullscreenChanged);
        toggleVSync.onValueChanged.AddListener(OnVSyncChanged);
        btnSave.onClick.AddListener(SaveSettings);

        // Ініціалізація кнопки Save
        hasChanges = false;
        btnSave.interactable = false;

        // Застосовуємо збережені налаштування
        ApplySettings();
    }
```

```
    private int FindResolutionIndex(int width, int height)
    {
        resolutions = Screen.resolutions;
        for (int i = 0; i < resolutions.Length; i++)
        {
            if (resolutions[i].width == width && resolutions[i].height == height)
            {
                return i;
            }
        }
        return 0; // Якщо 1920x1080 не знайдено, використовуємо перше доступне
розширення
    }

    // Обробники змін (звук у реальному часі)
    private void OnMasterVolumeChanged(float value)
    {
        audioMixer.SetFloat("MasterVolume", Mathf.Log10(value) * 20); //
Застосовується в реальному часі
        CheckForChanges();
    }

    private void OnMusicVolumeChanged(float value)
    {
        audioMixer.SetFloat("MusicVolume", Mathf.Log10(value) * 20);
        CheckForChanges();
    }

    private void OnSFXVolumeChanged(float value)
    {
        audioMixer.SetFloat("SFXVolume", Mathf.Log10(value) * 20);
        CheckForChanges();
    }

    private void OnAmbientVolumeChanged(float value)
    {
        audioMixer.SetFloat("AmbientVolume", Mathf.Log10(value) * 20);
        CheckForChanges();
    }

    private void OnResolutionChanged(int index)
    {
        CheckForChanges();
    }

    private void OnFullscreenChanged(bool value)
    {
        CheckForChanges();
    }

    private void OnVSyncChanged(bool value)
    {
        CheckForChanges();
    }

    private void CheckForChanges()
    {
        bool changed = sliderMasterVolume.value != savedMasterVolume ||
                       sliderMusicVolume.value != savedMusicVolume ||
                       sliderSFXVolume.value != savedSFXVolume ||
                       sliderAmbientVolume.value != savedAmbientVolume ||
                       dropdownResolution.value != savedResolutionIndex ||
                       toggleFullscreen.isOn != savedFullscreen ||
                       toggleVSync.isOn != savedVSync;
```

```
        hasChanges = changed;
        btnSave.interactable = changed;
    }

    private void SaveSettings()
    {
        // Зберігаємо поточні значення
        savedMasterVolume = sliderMasterVolume.value;
        savedMusicVolume = sliderMusicVolume.value;
        savedSFXVolume = sliderSFXVolume.value;
        savedAmbientVolume = sliderAmbientVolume.value;
        savedResolutionIndex = dropdownResolution.value;
        savedFullscreen = toggleFullscreen.isOn;
        savedVSync = toggleVSync.isOn;

        // Зберігаємо в PlayerPrefs
        PlayerPrefs.SetFloat("MasterVolume", savedMasterVolume);
        PlayerPrefs.SetFloat("MusicVolume", savedMusicVolume);
        PlayerPrefs.SetFloat("SFXVolume", savedSFXVolume);
        PlayerPrefs.SetFloat("AmbientVolume", savedAmbientVolume);
        PlayerPrefs.SetInt("ResolutionIndex", savedResolutionIndex);
        PlayerPrefs.SetInt("Fullscreen", savedFullscreen ? 1 : 0);
        PlayerPrefs.SetInt("VSync", savedVSync ? 1 : 0);
        PlayerPrefs.Save();

        // Застосовуємо налаштування (для роздільної здатності та V-Sync, звук вже
застосований)
        ApplyNonAudioSettings();

        // Вимикаємо кнопку "Зберегти"
        hasChanges = false;
        btnSave.interactable = false;
    }

    private void ApplySettings()
    {
        // Застосовуємо звук
        audioMixer.SetFloat("MasterVolume", Mathf.Log10(savedMasterVolume) * 20);
        audioMixer.SetFloat("MusicVolume", Mathf.Log10(savedMusicVolume) * 20);
        audioMixer.SetFloat("SoundFXVolume", Mathf.Log10(savedSFXVolume) * 20);
        audioMixer.SetFloat("AmbientFXVolume",   Mathf.Log10(savedAmbientVolume)   *
20);

        // Застосовуємо інші налаштування
        ApplyNonAudioSettings();
    }

    private void ApplyNonAudioSettings()
    {
        // Застосовуємо роздільну здатність
        Resolution = resolutions[savedResolutionIndex];
        Screen.SetResolution(resolution.width, resolution.height, savedFullscreen);

        // Застосовуємо повноекранний режим
        Screen.fullScreen = savedFullscreen;

        // Застосовуємо V-Sync
        QualitySettings.vSyncCount = savedVSync ? 1 : 0;
    }

    public void ResetToSavedSettings()
    {
        // Скидаємо UI
        sliderMasterVolume.value = savedMasterVolume;
        sliderMusicVolume.value = savedMusicVolume;
```

```
        sliderSFXVolume.value = savedSFXVolume;
        sliderAmbientVolume.value = savedAmbientVolume;
        dropdownResolution.value = savedResolutionIndex;
        toggleFullscreen.isOn = savedFullscreen;
        toggleVSync.isOn = savedVSync;

        // Відновлюємо збережені звукові налаштування
        ApplySettings();

        hasChanges = false;
        btnSave.interactable = false;
    }
}
```

RandomizedAudio

```
using UnityEngine;
using System.Collections;

public class RandomizedAudio : MonoBehaviour
{
    [Header("Audio Clips Settings")]
    [Tooltip("Масив звуків, з якого буде випадковий вибір (наприклад, різні варіанти
звуків кроків по сходах).")]
    public AudioClip[] audioClips;

    [Header("Playback Timing Settings")]
    [Tooltip("Мінімальний інтервал (у секундах) між відтворенням звуку.")]
    public float minInterval = 15f;

    [Tooltip("Максимальний інтервал (у секундах) між відтворенням звуку.")]
    public float maxInterval = 30f;

    [Header("Sound Variation Settings")]
    [Tooltip("Варіація гучності (± від базового значення).")]
    public float volumeVariation = 0.2f;

    [Tooltip("Варіація тону (± від базового значення).")]
    public float pitchVariation = 0.1f;

    [Header("Fade Effect Settings")]
    [Tooltip("Увімкнути ефект поступового збільшення гучності (fade-in).")]
    public bool useFadeIn = true;

    [Tooltip("Тривалість збільшення гучності (у секундах).")]
    public float fadeInDuration = 2f;

    [Tooltip("Увімкнути ефект поступового зменшення гучності (fade-out).")]
    public bool useFadeOut = true;

    [Tooltip("Тривалість зменшення гучності (у секундах).")]
    public float fadeOutDuration = 2f;

    private AudioSource;
    private float initialVolume;
    private float initialPitch;
    private float timer;
    private bool waitingForNextSound; // Прапорець для відстеження очікування

    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        initialVolume = audioSource.volume; // Зберігаємо початкову гучність
        initialPitch = audioSource.pitch;   // Зберігаємо початковий тон
```

```
        audioSource.volume = 0f;

        if (audioClips.Length > 0)
        {
            PlayRandomSound(); // Відтворюємо перший звук одразу
        }
    }

    void Update()
    {
        if (waitingForNextSound)
        {
            timer -= Time.deltaTime;
            if (timer <= 0)
            {
                PlayRandomSound();
                waitingForNextSound = false; // Скидаємо прапорець після відтворення
            }
        }

        if (audioSource.isPlaying && useFadeOut)
        {
            float timeLeft = audioSource.clip.length - audioSource.time;
            if (timeLeft <= fadeOutDuration && audioSource.volume > 0)
            {
                float fadeProgress = timeLeft / fadeOutDuration;
                audioSource.volume = Mathf.Lerp(0f, initialVolume, fadeProgress);
            }
        }
        else if (!audioSource.isPlaying && !waitingForNextSound)
        {
            // Звук завершився - починаємо відлік інтервалу
            timer = Random.Range(minInterval, maxInterval);
            waitingForNextSound = true;
        }
    }

    void PlayRandomSound()
    {
        if (audioClips.Length == 0) return;

        audioSource.clip = audioClips[Random.Range(0, audioClips.Length)];
        audioSource.volume = initialVolume; // Встановлюємо повну гучність
        audioSource.pitch    =    initialPitch    +    Random.Range(-pitchVariation,
pitchVariation);

        audioSource.Play();
        LogSoundStart();

        if (useFadeIn)
        {
            audioSource.volume = 0f;
            StartCoroutine(FadeIn());
        }
    }

    void LogSoundStart()
    {
        string clipName = audioSource.clip.name;
        float volume = audioSource.volume;
        float pitch = audioSource.pitch;
        float duration = audioSource.clip.length;
        string fadeInStatus = useFadeIn ? $"Yes (Duration: {fadeInDuration}s)" : "No";
        string fadeOutStatus = useFadeOut ? $"Yes (Duration: {fadeOutDuration}s)" :
"No";
```

```
        Debug.Log($"Sound started: Name = {clipName}, Volume = {volume}, Pitch =
{pitch}, Duration = {duration}s, FadeIn = {fadeInStatus}, FadeOut = {fadeOutStatus}");
    }

    IEnumerator FadeIn()
    {
        float timer = 0f;
        float targetVolume = initialVolume;
        while (timer < fadeInDuration)
        {
            timer += Time.deltaTime;
            float progress = timer / fadeInDuration;
            audioSource.volume = Mathf.Lerp(0f, targetVolume, progress);
            yield return null;
        }
        audioSource.volume = targetVolume;
    }
}


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

public class SoundMixerManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Дотаток Б

Лістинг коду аудисистеми пересування гравця, а також взаємодія з об'єктами.

PlayerMovement

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
[RequireComponent(typeof(AudioSource))]
public class PlayerMovement : MonoBehaviour
{
    [Header("Movement")]
    public float speed = 5f;          // Швидкість руху
    private float moveInput;          // Вхідне значення осі "Horizontal"

    private bool facingRight = true;  // Напрямок, в якому дивиться персонаж
    private bool canMove = true;      // Прапорець, що дозволяє рух

    [Header("Pickup Settings")]
    public AudioClip pickupSound;     // Звук підбору предмета

    private Rigidbody2D rb;           // Компонент Rigidbody2D для фізики
    private Animator;        // Компонент Animator для керування анімаціями
    private AudioSource;  // Компонент AudioSource для відтворення звуку
    private void Awake()
    {
        // Отримуємо компоненти Rigidbody2D, Animator та AudioSource
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
        audioSource = GetComponent<AudioSource>();
    }

    private void Update()
    {
        if (!canMove)
        {
            // Якщо рух заборонено, скидаємо анімацію руху
            if (animator != null)
            {
                animator.SetFloat("HorizontalMove", 0f);
            }
            return;
        }

        // Зчитуємо вхід для горизонтального руху
        moveInput = Input.GetAxis("Horizontal");

        // Передаємо значення руху в аніматор
        if (animator != null)
        {
            animator.SetFloat("HorizontalMove", Mathf.Abs(moveInput));
        }
    }

    private void FixedUpdate()
    {
        if (!canMove) return; // Виходимо, якщо рух заборонено

        // Переміщення персонажа по осі X
        rb.velocity = new Vector2(moveInput * speed, rb.velocity.y);

        // Переворот персонажа, якщо змінився напрямок руху
```

```
        if (!facingRight && moveInput > 0) Flip();
        else if (facingRight && moveInput < 0) Flip();
    }

    /// <summary>
    /// Дозволяє або забороняє рух персонажа.
    /// Якщо заборонено – обнуляється швидкість і скидається анімація.
    /// </summary>
    public void SetCanMove(bool value)
    {
        canMove = value;
        Debug.Log($"SetCanMove called with value: {value}, canMove is now:
{canMove}"); // Відладка

        if (!value)
        {
            rb.velocity = Vector2.zero; // Зупиняємо рух
            if (animator != null)
            {
                animator.SetFloat("HorizontalMove", 0f);
            }
        }
    }

    /// <summary>
    /// Викликається з Animation Event у анімації підбору.
    /// Відтворює звук і повідомляє про підбір предмета.
    /// </summary>
    public void OnPickupItem()
    {
        Debug.Log("OnPickupItem called"); // Відладка
        if (audioSource != null && pickupSound != null)
        {
            audioSource.PlayOneShot(pickupSound); // Відтворюємо звук підбору
        }
        Pickup.OnItemPickedUp(); // Повідомляємо Pickup, що предмет підібрано
    }

    /// <summary>
    /// Перевіряє, чи дивиться персонаж праворуч.
    /// </summary>
    public bool IsFacingRight()
    {
        return facingRight;
    }

    /// <summary>
    /// Переворот персонажа по осі X.
    /// </summary>
    private void Flip()
    {
        facingRight = !facingRight; // Змінюємо напрямок
        Vector3 scaler = transform.localScale;
        scaler.x *= -1; // Віддзеркалюємо масштаб по осі X
        transform.localScale = scaler;
    }

    /// <summary>
    /// Примусовий переворот для сходження по драбині.
    /// </summary>
    public void FlipForLadder()
    {
        Flip();
    }
```

```
    /// <summary>
    /// Повертає, чи дозволено рух персонажу.
    /// </summary>
    public bool GetCanMove()
    {
        return canMove;
    }
}
```

PlayerFootstepsWithSurfaces

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Rigidbody2D))]
public class PlayerFootstepsWithSurfaces : MonoBehaviour
{
    [System.Serializable]
    public class SurfaceFootsteps
    {
        [Tooltip("Тег поверхні (наприклад, 'Wood', 'Carpet', 'Tile').")]
        public string surfaceTag;

        [Tooltip("Масив звуків кроків для цієї поверхні під час звичайної ходьби.")]
        public AudioClip[] footstepClips;

        [Tooltip("Масив звуків кроків для цієї поверхні під час ходьби навприсядки.")]
        public AudioClip[] crouchFootstepClips;
    }

    [Header("Surface Detection Settings")]
    [Tooltip("Шар, на якому знаходяться поверхні (наприклад, 'Ground').")]
    public LayerMask groundLayer;

    [Tooltip("Довжина променя для виявлення поверхні під гравцем.")]
    public float raycastDistance = 1f;

    [Header("Footstep Settings")]
    [Tooltip("Список поверхонь і відповідних звуків кроків.")]
    public SurfaceFootsteps[] surfaceFootsteps;

    [Tooltip("Джерело звуку для відтворення кроків.")]
    public AudioSource footstepAudioSource;

    [Header("Movement Reference")]
    [Tooltip("Скрипт управління рухом персонажа (PlayerMovement). Вкажіть, якщо
знаходиться на іншому об'єкті.")]
    public PlayerMovement;

    [Header("Crouch Reference")]
    [Tooltip("Скрипт управління присіданням персонажа (PlayerCrouch). Вкажіть, якщо
знаходиться на іншому об'єкті.")]
    public PlayerCrouch;

    private int lastPlayedClipIndex = -1;
    private AudioClip[] currentFootstepClips;
    private AudioClip[] currentCrouchFootstepClips;
    private Rigidbody2D rb;
    private bool isMoving = false;
    private bool isCrouching = false;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
```

```
        if (playerMovement == null)
        {
            playerMovement = GetComponent<PlayerMovement>();
            Debug.Log("PlayerMovement auto-assigned from current object: " +
(playerMovement != null));
        }
        else
        {
            Debug.Log("PlayerMovement manually assigned: " +
playerMovement.gameObject.name);
        }

        if (playerCrouch == null)
        {
            playerCrouch = GetComponent<PlayerCrouch>();
            Debug.Log("PlayerCrouch auto-assigned from current object: " +
(playerCrouch != null));
        }
        else
        {
            Debug.Log("PlayerCrouch manually assigned: " +
playerCrouch.gameObject.name);
        }

        currentFootstepClips = new AudioClip[0];
        currentCrouchFootstepClips = new AudioClip[0];

        if (footstepAudioSource == null)
        {
            Debug.LogWarning("Footstep Audio Source is not assigned!");
        }
        else
        {
            Debug.Log("Footstep Audio Source assigned: " +
footstepAudioSource.gameObject.name);
        }

        if (surfaceFootsteps.Length == 0)
        {
            Debug.LogWarning("No surface footstep data assigned!");
        }
    }

    void Update()
    {
        // Перевіряємо, чи рухається персонаж
        bool canMove = playerMovement != null ? playerMovement.GetCanMove() : true;
        isMoving = canMove && Mathf.Abs(rb.velocity.x) > 0.1f;
        Debug.Log("Is Moving: " + isMoving + " | Velocity X: " + rb.velocity.x + " |
Can Move: " + canMove);

        // Перевіряємо, чи персонаж знаходиться у присіданні
        isCrouching = playerCrouch != null ? playerCrouch.IsCrouching : false;
        Debug.Log("Is Crouching: " + isCrouching);

        UpdateSurface();
    }

    void UpdateSurface()
    {
        RaycastHit2D hit = Physics2D.Raycast(transform.position, Vector2.down,
raycastDistance, groundLayer);
        if (hit.collider != null)
        {
            string surfaceTag = hit.collider.tag;
```

```
            Debug.Log("Surface detected: " + surfaceTag);

            bool foundSurface = false;
            foreach (var surface in surfaceFootsteps)
            {
                if (surface.surfaceTag == surfaceTag)
                {
                    currentFootstepClips = surface.footstepClips;
                    currentCrouchFootstepClips = surface.crouchFootstepClips;
                    Debug.Log("Footstep clips set for surface: " + surfaceTag + " |
Normal Clips count: " + currentFootstepClips.Length + " | Crouch Clips count: " +
currentCrouchFootstepClips.Length);
                    foundSurface = true;
                    break;
                }
            }

            if (!foundSurface)
            {
                Debug.LogWarning("No footstep clips found for surface tag: " +
surfaceTag + ". Check Surface Footsteps array in Inspector.");
            }
        }
        else
        {
            Debug.Log("No surface detected under player!");
        }

        if (currentFootstepClips == null || currentFootstepClips.Length == 0)
        {
            currentFootstepClips = new AudioClip[0];
        }
        if (currentCrouchFootstepClips == null || currentCrouchFootstepClips.Length
== 0)
        {
            currentCrouchFootstepClips = new AudioClip[0];
        }
    }

    // Метод, який буде викликатися з Animation Event
    public void PlayFootstep()
    {
        AudioClip[] activeClips = isCrouching ? currentCrouchFootstepClips :
currentFootstepClips;

        if (activeClips.Length == 0 || footstepAudioSource == null)
        {
            Debug.Log("Cannot play footstep: Clips length = " + activeClips.Length +
" | AudioSource = " + (footstepAudioSource != null));
            return;
        }

        if (!isMoving)
        {
            Debug.Log("Not playing footstep: Character is not moving.");
            return;
        }

        int newClipIndex;
        do
        {
            newClipIndex = Random.Range(0, activeClips.Length);
        } while (newClipIndex == lastPlayedClipIndex && activeClips.Length > 1);

        lastPlayedClipIndex = newClipIndex;
```

```
        footstepAudioSource.PlayOneShot(activeClips[newClipIndex], 0.7f);
        Debug.Log("Playing footstep: " + activeClips[newClipIndex].name + " |
Crouching: " + isCrouching);
    }

    void OnDrawGizmos()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawRay(transform.position, Vector2.down * raycastDistance);
    }
}
```

PlayerClimbing

```
using UnityEngine;

[RequireComponent(typeof(PlayerMovement))]
public class PlayerClimbing : MonoBehaviour
{
    [Header("Налаштування карабкання")]
    public float rayDistance = 1f; // Дистанція променя для виявлення поверхні
    public LayerMask climbableLayer; // Шар об'єктів, по яких можна карабкатися
    public KeyCode climbKey = KeyCode.Space; // Клавіша для початку карабкання

    [Header("Звукові ефекти")]
    public AudioClip climbSound; // Звук карабкання

    private PlayerMovement; // Компонент керування рухом
    private bool isClimbing = false; // Прапорець стану карабкання
    private Animator; // Посилання на аніматор
    private Rigidbody2D rb; // Посилання на Rigidbody2D
    private AudioSource; // Посилання на AudioSource

    private void Awake()
    {
        // Ініціалізація компонентів
        playerMovement = GetComponent<PlayerMovement>();
        animator = GetComponent<Animator>();
        rb = GetComponent<Rigidbody2D>();
        audioSource = GetComponent<AudioSource>(); // Отримуємо AudioSource
    }

    private void Update()
    {
        // Якщо гравець вже карабкається – нічого не робимо
        if (isClimbing)
            return;

        // Визначаємо напрямок променя на основі сторони, в яку дивиться гравець
        Vector2 direction = playerMovement.IsFacingRight() ? Vector2.right :
Vector2.left;

        // Випускаємо промінь для виявлення climbable-поверхні
        RaycastHit2D hit = Physics2D.Raycast(
            transform.position,
            direction,
            rayDistance,
            climbableLayer
        );
```

```
        Debug.DrawRay(transform.position, direction * rayDistance, Color.yellow);

        // Починаємо карабкання, якщо поверхню знайдено та натиснута клавіша
        if (hit.collider != null && Input.GetKeyDown(climbKey))
        {
            StartClimbing();
        }
    }

    private void StartClimbing()
    {
        // Блокуємо керування і фізику, активуємо анімацію
        isClimbing = true;
        playerMovement.SetCanMove(false);
        rb.isKinematic = true;
        rb.velocity = Vector2.zero;

        if (animator != null)
        {
            animator.SetTrigger("Climb");
        }
    }

    /// <summary>
    /// Викликається через Animation Event в анімації карабкання.
    /// Відтворює звук карабкання.
    /// </summary>
    public void OnClimbSound()
    {
        Debug.Log("OnClimbSound called"); // Налагодження
        if (audioSource != null && climbSound != null)
        {
            audioSource.PlayOneShot(climbSound); // Відтворюємо звук карабкання
        }
    }

    public void OnClimbAnimationEnd()
    {
        // Завершуємо карабкання, повертаємо керування та фізику
        isClimbing = false;
        rb.isKinematic = false;
        playerMovement.SetCanMove(true);
        Debug.Log("Карабкання завершено, керування повернуто.");
    }

    private void OnAnimatorMove()
    {
        // Застосовуємо root motion для переміщення гравця під час карабкання
        if (isClimbing && animator != null)
        {
            Vector3 delta = animator.deltaPosition;
            delta.z = 0f; // Ігноруємо зміщення по осі Z
            transform.position += delta;
        }
    }
}
```

QuestGiverInteraction

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

```
using System.Collections;

[System.Serializable]
public class DialogueStep
{
    [Tooltip("Зображення для хмаринки NPC.")]
    public Sprite npcBubbleSprite;
    [Tooltip("Анімація NPC для цієї репліки (наприклад, 'Wave' або 'Deny').")]
    public string npcAnimationTrigger;
    [Tooltip("Зображення для хмаринки гравця.")]
    public Sprite playerBubbleSprite;
    [Tooltip("Анімація гравця для цієї репліки.")]
    public string playerAnimationTrigger;
    [Tooltip("Час у секундах для показу цієї репліки перед переходом до
наступної.")]
    public float duration = 1f;
    [Tooltip("Якщо увімкнено, ця репліка завершує квест.")]
    public bool completesQuest = false;
    [Tooltip("Звук, що відтворюється при відображенні цієї репліки.")]
    public AudioClip dialogueSound;
}

public class QuestGiverInteraction : MonoBehaviour
{
    [Header("Налаштування квесту")]
    [Tooltip("Посилання на об'єкт з QuestManager у сцені.")]
    public QuestManager;
    [Tooltip("Номер квесту, який завершує ця взаємодія.")]
    public int questId;

    [Header("Тип взаємодії")]
    [Tooltip("Якщо увімкнено, квест завершується при вході в зону. Якщо вимкнено,
потрібно натиснути E.")]
    public bool isTriggerBased = false;

    [Header("Вимоги (опціонально)")]
    [Tooltip("Предмети, які потрібні для виконання квесту.")]
    public ItemData[] requiredItems;
    [Tooltip("Чи видаляти предмети з інвентаря після виконання квесту.")]
    public bool removeRequiredItemsOnSuccess = true;

    [Header("Нагорода (опціонально)")]
    [Tooltip("Якщо увімкнено, гравець отримає нагороду після виконання.")]
    public bool giveReward = false;
    [Tooltip("Список предметів, які гравець отримає як нагороду.")]
    [SerializeField] private ItemData[] rewards;

    [Header("Наслідки")]
    [Tooltip("Події, які відбудуться після виконання квесту (наприклад, увімкнути
об'єкт).")]
    public UnityEvent onInteractionSuccess;
    [Tooltip("Чи видалити цей об'єкт після виконання квесту.")]
    public bool shouldRemoveObject = false;

    [Header("Налаштування діалогу")]
    [Tooltip("Об'єкт з хмаринкою для NPC.")]
    public GameObject npcBubble;
    [Tooltip("Об'єкт з хмаринкою для гравця.")]
    public GameObject playerBubble;
    [Tooltip("Компонент анімації NPC.")]
    public Animator npcAnimator;
    [Tooltip("Компонент анімації гравця (потрібен об'єкт гравця у сцені).")]
    public Animator playerAnimator;
    [Tooltip("Компонент AudioSource для відтворення звуків діалогу (якщо не
призначений, використовується AudioSource з об'єкта NPC).")]
```

```
    public AudioSource;

    [Header("Основний діалог")]
    [Tooltip("Список кроків основного діалогу між NPC та гравцем.")]
    public DialogueStep[] mainDialogue;
    [Tooltip("Якщо увімкнено, основний діалог програється лише один раз.")]
    public bool playMainDialogueOnce = false;

    [Header("Повторний діалог")]
    [Tooltip("Список кроків діалогу, який програється після першого разу (якщо
основний одноразовий).")]
    public DialogueStep[] repeatDialogue;

    [Header("Діалог при невдачі")]
    [Tooltip("Список кроків діалогу, якщо квест не можна виконати (немає предметів
або квесту).")]
    public DialogueStep[] failDialogue;

    [Header("Керування гравцем")]
    [Tooltip("Якщо увімкнено, гравець не може рухатися або діяти під час діалогу.")]
    public bool lockPlayerDuringDialogue = true;

    private bool isPlayerInRange = false;
    private bool isDialogueActive = false;
    private bool hasPlayedMainDialogue = false;
    private GameObject player;

    private void Awake()
    {
        // Якщо AudioSource не призначений в інспекторі, намагаємося отримати його з
об'єкта NPC
        if (audioSource == null)
        {
            audioSource = GetComponent<AudioSource>();
            if (audioSource == null)
            {
                Debug.LogWarning($"AudioSource відсутній на об'єкті
{gameObject.name} і не призначений в інспекторі. Додайте AudioSource для відтворення
звуків діалогу.");
            }
        }
    }

    private void Start()
    {
        if (npcBubble != null) npcBubble.SetActive(false);
        if (playerBubble != null) playerBubble.SetActive(false);
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInRange = true;
            player = other.gameObject;

            if (!isTriggerBased && !isDialogueActive)
            {
                if (!CanInteract() && failDialogue.Length > 0)
                {
                    StartDialogue(failDialogue);
                }
            }
            else if (CanInteract())
            {
```

```
                PerformInteraction();
            }
        }
    }

    private void OnTriggerExit2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInRange = false;
            if (npcBubble != null) npcBubble.SetActive(false);
            if (playerBubble != null) playerBubble.SetActive(false);
            isDialogueActive = false;
            UnlockPlayerControls();
            player = null;
        }
    }

    private void Update()
    {
        if (!isTriggerBased && isPlayerInRange && Input.GetKeyDown(KeyCode.E) &&
!isDialogueActive)
        {
            if (CanInteract())
            {
                StartDialogue(playMainDialogueOnce && !hasPlayedMainDialogue ?
mainDialogue : repeatDialogue);
            }
            else
            {
                Debug.Log("Взаємодія неможлива!");
                if (failDialogue.Length > 0) StartDialogue(failDialogue);
            }
        }
    }

    private bool CanInteract()
    {
        if (!questManager.IsQuestAvailable(questId)) return false;
        if (requiredItems.Length > 0 && !HasAllRequiredItems()) return false;
        return true;
    }

    private void StartDialogue(DialogueStep[] dialogue)
    {
        if (dialogue.Length == 0)
        {
            if (CanInteract()) PerformInteraction();
            return;
        }

        isDialogueActive = true;
        if (lockPlayerDuringDialogue) LockPlayerControls();
        StartCoroutine(RunDialogue(dialogue));
    }

    private IEnumerator RunDialogue(DialogueStep[] dialogue)
    {
        for (int i = 0; i < dialogue.Length; i++)
        {
            DialogueStep step = dialogue[i];
            ShowDialogueStep(step);
            yield return new WaitForSeconds(step.duration);

            if (step.completesQuest && CanInteract())
```

```
            {
                PerformInteraction();
                break;
            }
        }

        if (npcBubble != null) npcBubble.SetActive(false);
        if (playerBubble != null) playerBubble.SetActive(false);
        isDialogueActive = false;
        if (dialogue == mainDialogue) hasPlayedMainDialogue = true;
        if (lockPlayerDuringDialogue) UnlockPlayerControls();
    }

    private void PerformInteraction()
    {
        questManager.CompleteQuest(questId);
        if (removeRequiredItemsOnSuccess) RemoveAllRequiredItems(requiredItems);
        if (giveReward && rewards != null)
        {
            foreach (var reward in rewards)
            {
                if (reward != null) Inventory.Instance.AddItem(reward);
            }
        }
        onInteractionSuccess?.Invoke();
        if (shouldRemoveObject) Destroy(gameObject);
    }

    private bool HasAllRequiredItems()
    {
        if (requiredItems == null || requiredItems.Length == 0) return true;
        List<ItemData> inventoryItems = new List<ItemData>();
        foreach (Slot in Inventory.Instance.Slots)
        {
            if (slot.HasItem) inventoryItems.Add(slot.CurrentItem);
        }
        foreach (var requiredItem in requiredItems)
        {
            if (!inventoryItems.Contains(requiredItem)) return false;
            inventoryItems.Remove(requiredItem);
        }
        return true;
    }

    private void RemoveAllRequiredItems(ItemData[] items)
    {
        if (items == null) return;
        foreach (var item in items)
        {
            Inventory.Instance.RemoveItem(item);
        }
    }

    private void ShowDialogueStep(DialogueStep step)
    {
        if (npcBubble != null && step.npcBubbleSprite != null)
        {
            npcBubble.SetActive(true);
            SpriteRenderer npcRenderer =
npcBubble.transform.Find("Content")?.GetComponent<SpriteRenderer>();
            if (npcRenderer != null) npcRenderer.sprite = step.npcBubbleSprite;
        }
        if (npcAnimator != null && !string.IsNullOrEmpty(step.npcAnimationTrigger))
            npcAnimator.SetTrigger(step.npcAnimationTrigger);
```

```
        if (playerBubble != null && step.playerBubbleSprite != null)
        {
            playerBubble.SetActive(true);
            SpriteRenderer playerRenderer =
playerBubble.transform.Find("Content")?.GetComponent<SpriteRenderer>();
            if (playerRenderer != null) playerRenderer.sprite =
step.playerBubbleSprite;
        }
        if (playerAnimator != null &&
!string.IsNullOrEmpty(step.playerAnimationTrigger))
            playerAnimator.SetTrigger(step.playerAnimationTrigger);

        // Відтворюємо звук діалогу, якщо він призначений
        if (audioSource != null && step.dialogueSound != null)
        {
            audioSource.PlayOneShot(step.dialogueSound, 0.8f); // Гучність 0.8 для
балансу
        }
    }

    private void LockPlayerControls()
    {
        if (player == null) return;
        PlayerMovement movement = player.GetComponent<PlayerMovement>();
        if (movement != null) movement.enabled = false;
    }

    private void UnlockPlayerControls()
    {
        if (player == null) return;
        PlayerMovement movement = player.GetComponent<PlayerMovement>();
        if (movement != null) movement.enabled = true;
    }
}Якщо AudioSource не заданий у інспекторі – пробуємо отримати його з об’єкта NPC
        if (audioSource == null)
        {
            audioSource = GetComponent<AudioSource>();
            if (audioSource == null)
            {
                Debug.LogWarning($"AudioSource відсутній на об’єкті
{gameObject.name} і не заданий у інспекторі. Додайте AudioSource для відтворення
звуків діалогу.");
            }
        }
    }

    private void Start()
    {
        if (npcBubble != null) npcBubble.SetActive(false);
        if (playerBubble != null) playerBubble.SetActive(false);
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInRange = true;
            player = other.gameObject;

            if (!isTriggerBased && !isDialogueActive)
            {
                if (!CanInteract() && failDialogue.Length > 0)
                {
                    StartDialogue(failDialogue);
                }
```

```
            }
            else if (CanInteract())
            {
                PerformInteraction();
            }
        }
    }

    private void OnTriggerExit2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInRange = false;
            if (npcBubble != null) npcBubble.SetActive(false);
            if (playerBubble != null) playerBubble.SetActive(false);
            isDialogueActive = false;
            UnlockPlayerControls();
            player = null;
        }
    }

    private void Update()
    {
        if (!isTriggerBased && isPlayerInRange && Input.GetKeyDown(KeyCode.E) &&
!isDialogueActive)
        {
            if (CanInteract())
            {
                StartDialogue(playMainDialogueOnce && !hasPlayedMainDialogue ?
mainDialogue : repeatDialogue);
            }
            else
            {
                Debug.Log("Взаємодія неможлива!");
                if (failDialogue.Length > 0) StartDialogue(failDialogue);
            }
        }
    }

    private bool CanInteract()
    {
        if (!questManager.IsQuestAvailable(questId)) return false;
        if (requiredItems.Length > 0 && !HasAllRequiredItems()) return false;
        return true;
    }

    private void StartDialogue(DialogueStep[] dialogue)
    {
        if (dialogue.Length == 0)
        {
            if (CanInteract()) PerformInteraction();
            return;
        }

        isDialogueActive = true;
        if (lockPlayerDuringDialogue) LockPlayerControls();
        StartCoroutine(RunDialogue(dialogue));
    }

    private IEnumerator RunDialogue(DialogueStep[] dialogue)
    {
        for (int i = 0; i < dialogue.Length; i++)
        {
            DialogueStep step = dialogue[i];
            ShowDialogueStep(step);
```

```
            yield return new WaitForSeconds(step.duration);

            if (step.completesQuest && CanInteract())
            {
                PerformInteraction();
                break;
            }
        }

        if (npcBubble != null) npcBubble.SetActive(false);
        if (playerBubble != null) playerBubble.SetActive(false);
        isDialogueActive = false;
        if (dialogue == mainDialogue) hasPlayedMainDialogue = true;
        if (lockPlayerDuringDialogue) UnlockPlayerControls();
    }

    private void PerformInteraction()
    {
        questManager.CompleteQuest(questId);
        if (removeRequiredItemsOnSuccess) RemoveAllRequiredItems(requiredItems);
        if (giveReward && rewards != null)
        {
            foreach (var reward in rewards)
            {
                if (reward != null) Inventory.Instance.AddItem(reward);
            }
        }
        onInteractionSuccess?.Invoke();
        if (shouldRemoveObject) Destroy(gameObject);
    }

    private bool HasAllRequiredItems()
    {
        if (requiredItems == null || requiredItems.Length == 0) return true;
        List<ItemData> inventoryItems = new List<ItemData>();
        foreach (Slot in Inventory.Instance.Slots)
        {
            if (slot.HasItem) inventoryItems.Add(slot.CurrentItem);
        }
        foreach (var requiredItem in requiredItems)
        {
            if (!inventoryItems.Contains(requiredItem)) return false;
            inventoryItems.Remove(requiredItem);
        }
        return true;
    }

    private void RemoveAllRequiredItems(ItemData[] items)
    {
        if (items == null) return;
        foreach (var item in items)
        {
            Inventory.Instance.RemoveItem(item);
        }
    }

    private void ShowDialogueStep(DialogueStep step)
    {
        if (npcBubble != null && step.npcBubbleSprite != null)
        {
            npcBubble.SetActive(true);
            SpriteRenderer npcRenderer =
npcBubble.transform.Find("Content")?.GetComponent<SpriteRenderer>();
            if (npcRenderer != null) npcRenderer.sprite = step.npcBubbleSprite;
        }
```

```
        if (npcAnimator != null && !string.IsNullOrEmpty(step.npcAnimationTrigger))
            npcAnimator.SetTrigger(step.npcAnimationTrigger);

        if (playerBubble != null && step.playerBubbleSprite != null)
        {
            playerBubble.SetActive(true);
            SpriteRenderer playerRenderer =
playerBubble.transform.Find("Content")?.GetComponent<SpriteRenderer>();
            if (playerRenderer != null) playerRenderer.sprite =
step.playerBubbleSprite;
        }
        if (playerAnimator != null &&
!string.IsNullOrEmpty(step.playerAnimationTrigger))
            playerAnimator.SetTrigger(step.playerAnimationTrigger);

        // Відтворюємо звук діалогу, якщо він заданий
        if (audioSource != null && step.dialogueSound != null)
        {
            audioSource.PlayOneShot(step.dialogueSound, 0.8f); // Гучність 0.8 для
балансу
        }
    }

    private void LockPlayerControls()
    {
        if (player == null) return;
        PlayerMovement movement = player.GetComponent<PlayerMovement>();
        if (movement != null) movement.enabled = false;
    }

    private void UnlockPlayerControls()
    {
        if (player == null) return;
        PlayerMovement movement = player.GetComponent<PlayerMovement>();
        if (movement != null) movement.enabled = true;
    }
}
```

TeleportSystem

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class TeleportSystem : MonoBehaviour
{
    [System.Serializable]
    public class TeleportPoint
    {
        public Transform point; // Точка телепортації
        public SwitchFrame targetFrame; // SwitchFrame, пов'язаний з цією точкою
        [HideInInspector] public BoxCollider2D collider; // Колайдер точки
(ініціалізується в Start)

        [Header("Індивідуальні налаштування точки")]
        public AudioClip teleportSound; // Звук телепортації для цієї точки
        public GameObject teleportIcon; // Іконка у світі (наприклад,
SpriteRenderer)
        public float fadeDuration = 1f; // Тривалість затемнення для цієї точки
        public float loadDelay = 1f;    // Затримка завантаження для цієї точки
    }

    [Header("Налаштування активації")]
    public KeyCode activationKey = KeyCode.E;
```

```
    [Header("Точки телепортації")]
    public TeleportPoint teleportPointA;
    public TeleportPoint teleportPointB;

    [Header("UI елементи")]
    public Image fadeImage; // UI-елемент для затемнення екрану

    [Header("Звук")]
    public AudioSource teleportAudioSource; // Загальне джерело звуку

    [Header("Посилання")]
    public Transform playerTransform;
    public PlayerMovement; // Посилання на скрипт керування гравцем
    public ManagerSwitchFrames frameManager; // Посилання на ManagerSwitchFrames

    private bool isTeleporting = false;

    void Start()
    {
        // Шукаємо гравця, якщо він не заданий
        if (playerTransform == null)
        {
            GameObject playerObj = GameObject.FindGameObjectWithTag("Player");
            if (playerObj != null)
            {
                playerTransform = playerObj.transform;
            }
            else
            {
                Debug.LogWarning("Гравця не знайдено! Будь ласка, призначте
playerTransform в інспекторі.");
            }
        }

        // Шукаємо PlayerMovement, якщо не заданий
        if (playerMovement == null && playerTransform != null)
        {
            playerMovement = playerTransform.GetComponent<PlayerMovement>();
            if (playerMovement == null)
            {
                Debug.LogWarning("Скрипт PlayerMovement не знайдено на об'єкті
гравця!");
            }
        }

        // Шукаємо ManagerSwitchFrames, якщо не заданий
        if (frameManager == null)
        {
            frameManager = FindObjectOfType<ManagerSwitchFrames>();
            if (frameManager == null)
            {
                Debug.LogWarning("ManagerSwitchFrames не знайдено на сцені!");
            }
        }

        // Ініціалізуємо колайдери
        if (teleportPointA.point != null) teleportPointA.collider =
teleportPointA.point.GetComponent<BoxCollider2D>();
        if (teleportPointB.point != null) teleportPointB.collider =
teleportPointB.point.GetComponent<BoxCollider2D>();

        // Вимикаємо іконки телепорту
        SetTeleportIconActive(teleportPointA, false);
        SetTeleportIconActive(teleportPointB, false);
```

```
        // Ініціалізуємо fadeImage
        InitializeFadeImage();
    }

    void Update()
    {
        if (isTeleporting || playerTransform == null)
            return;

        // Перевіряємо, чи знаходиться гравець в зоні однієї з точок телепорту
        bool nearA = teleportPointA.collider != null &&
teleportPointA.collider.OverlapPoint(playerTransform.position);
        bool nearB = teleportPointB.collider != null &&
teleportPointB.collider.OverlapPoint(playerTransform.position);

        // Відображаємо іконки для точок, якщо гравець поруч
        SetTeleportIconActive(teleportPointA, nearA);
        SetTeleportIconActive(teleportPointB, nearB);

        // Обробка натискання клавіші телепортації
        if (Input.GetKeyDown(activationKey))
        {
            if (nearA)
            {
                StartTeleport(teleportPointA, teleportPointB);
            }
            else if (nearB)
            {
                StartTeleport(teleportPointB, teleportPointA);
            }
        }
    }

    private void StartTeleport(TeleportPoint fromPoint, TeleportPoint toPoint)
    {
        if (isTeleporting || toPoint.targetFrame == null ||
toPoint.targetFrame.activeFrame == null) return;

        // Активуємо цільову локацію одразу після натискання кнопки
        if (!toPoint.targetFrame.activeFrame.activeSelf)
        {
            toPoint.targetFrame.activeFrame.SetActive(true);
        }

        // Запускаємо корутину телепортації
        StartCoroutine(TeleportCoroutine(fromPoint, toPoint));
    }

    private IEnumerator TeleportCoroutine(TeleportPoint fromPoint, TeleportPoint
toPoint)
    {
        isTeleporting = true;

        // Вимикаємо керування гравцем
        if (playerMovement != null) playerMovement.SetCanMove(false);

        // Відтворюємо звук телепортації (використовуємо звук з точки відправлення)
        PlayTeleportSound(fromPoint);

        // Плавне затемнення екрану (fade out), використовуємо параметри точки
відправлення
        yield return FadeScreen(0f, 1f, fromPoint.fadeDuration);

        // Чекаємо заданий час (імітація завантаження)
```

```
        yield return new WaitForSeconds(fromPoint.loadDelay);

        // Переміщуємо гравця в цільову точку
        if (playerTransform != null) playerTransform.position =
toPoint.point.position;

        // Повідомляємо ManagerSwitchFrames про перехід у нову локацію
        if (frameManager != null && toPoint.targetFrame != null)
frameManager.SwitchToFrame(toPoint.targetFrame);

        // Плавне повернення світла (fade in), використовуємо параметри точки
призначення
        yield return FadeScreen(1f, 0f, toPoint.fadeDuration);

        // Вимикаємо fadeImage, якщо екран повністю прозорий
        if (fadeImage != null && fadeImage.color.a == 0f)
SetUIElementActive(fadeImage, false);

        // Вмикаємо керування гравцем
        if (playerMovement != null) playerMovement.SetCanMove(true);

        isTeleporting = false;
    }

    private IEnumerator FadeScreen(float startAlpha, float endAlpha, float duration)
    {
        if (fadeImage == null) yield break;

        // Вмикаємо fadeImage, якщо він неактивний
        if (!fadeImage.gameObject.activeSelf) SetUIElementActive(fadeImage, true);

        float elapsed = 0f;
        Color = fadeImage.color;
        while (elapsed < duration)
        {
            elapsed += Time.deltaTime;
            color.a = Mathf.Lerp(startAlpha, endAlpha, elapsed / duration);
            fadeImage.color = color;
            yield return null;
        }
        color.a = endAlpha;
        fadeImage.color = color;
    }

    private void PlayTeleportSound(TeleportPoint point)
    {
        if (teleportAudioSource != null && point.teleportSound != null)
        {
            teleportAudioSource.PlayOneShot(point.teleportSound);
        }
    }

    private void SetTeleportIconActive(TeleportPoint point, bool isActive)
    {
        if (point.teleportIcon != null)
        {
            point.teleportIcon.SetActive(isActive);
        }
    }

    private void InitializeFadeImage()
    {
        if (fadeImage != null)
        {
            SetUIElementActive(fadeImage, false);
```

```
            Color = fadeImage.color;
            color.a = 0f;
            fadeImage.color = color;
        }
    }

    private void SetUIElementActive(Image uiElement, bool isActive)
    {
        if (uiElement != null)
        {
            uiElement.gameObject.SetActive(isActive);
        }
    }
}
```

PlayerWhistle

```
using UnityEngine;
using TMPro;

[RequireComponent(typeof(AudioSource))]
public class PlayerWhistle : MonoBehaviour
{
    [Header("Whistle Settings")]
    public LayerMask enemyLayer;         // Шар ворогів
    public TMP_Text hintText;            // Посилання на текст з підказками
    public GameObject whistleAreaPrefab; // Префаб зони свисту
    [Tooltip("Компонент AudioSource для відтворення звуків свисту (якщо не
призначений, використовується AudioSource з об'єкта).")]
    public AudioSource;      // Публічний AudioSource

    [Header("Сила свистів")]
    [Tooltip("Звуки для слабкого свисту (рівень 0).")]
    [SerializeField]
    private AudioClip[] weakWhistleSounds;   // Звуки для слабкого свисту
    [Tooltip("Звуки для середнього свисту (рівень 1).")]
    [SerializeField]
    private AudioClip[] mediumWhistleSounds; // Звуки для середнього свисту
    [Tooltip("Звуки для сильного свисту (рівень 2).")]
    [SerializeField]
    private AudioClip[] strongWhistleSounds; // Звуки для сильного свисту

    private readonly float[] whistleRadii = { 10f, 15f, 20f }; // Фіксовані радіуси
для рівнів свисту
    private GameObject currentWhistleArea; // Поточна зона свисту
    private WhistleArea whistleAreaScript; // Посилання на скрипт зони свисту
    private int whistleLevel = 0;         // Рівень сили свисту (0 = слабкий, 1 =
середній, 2 = сильний)
    private bool isWhistleMode = false;   // Чи активний режим вибору свисту
    private Rigidbody2D rb;              // Rigidbody для перевірки руху

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>(); // Отримуємо Rigidbody2D гравця
        // Якщо AudioSource не призначений в інспекторі, отримуємо його з об'єкта
        if (audioSource == null)
        {
            audioSource = GetComponent<AudioSource>();
            if (audioSource == null)
            {
                Debug.LogWarning($"AudioSource відсутній на об'єкті
{gameObject.name} і не призначений в інспекторі. Додайте AudioSource для відтворення
звуків свисту.");
            }
        }
    }
```

```
    private void Update()
    {
        // Скидаємо свист, якщо гравець рухається
        if (isWhistleMode && rb.velocity.sqrMagnitude > 0.01f) // Якщо гравець
рухається
        {
            CancelWhistle();
        }

        if (Input.GetKeyDown(KeyCode.T)) // Увімкнення або зміна сили свисту
        {
            if (!isWhistleMode)
            {
                ActivateWhistleMode();
            }
            else
            {
                CycleWhistleLevel();
            }
        }

        if (isWhistleMode)
        {
            if (Input.GetKeyDown(KeyCode.Q)) // Скасування свисту
            {
                CancelWhistle();
            }

            if (Input.GetKeyDown(KeyCode.R)) // Підтвердження свисту
            {
                ConfirmWhistle();
            }
        }
    }

    private void ActivateWhistleMode()
    {
        isWhistleMode = true;
        whistleLevel = 0;

        // Створюємо зону свисту
        currentWhistleArea = Instantiate(whistleAreaPrefab, transform.position,
Quaternion.identity);
        whistleAreaScript = currentWhistleArea.GetComponent<WhistleArea>();
        UpdateWhistleArea();

        // Показуємо підказку
        if (hintText != null)
        {
            hintText.gameObject.SetActive(true);
            hintText.text = "'T' - змінити силу свисту; 'Q' - скасувати свист; 'R' -
підтвердити свист";
        }
    }

    private void CycleWhistleLevel()
    {
        whistleLevel = (whistleLevel + 1) % whistleRadii.Length; // Циклічне
перемикання рівня
        UpdateWhistleArea();
    }

    private void UpdateWhistleArea()
    {
```

```
        if (whistleAreaScript != null)
        {
            // Оновлюємо радіус зони
            float radius = whistleRadii[whistleLevel];
            whistleAreaScript.UpdateRadius(radius);
        }
    }

    private void CancelWhistle()
    {
        isWhistleMode = false;

        // Прибираємо зону свисту та підказку
        if (currentWhistleArea != null)
        {
            Destroy(currentWhistleArea);
        }

        if (hintText != null)
        {
            hintText.gameObject.SetActive(false);
        }

        Debug.Log("Підготовку до свисту скасовано.");
    }

    private void ConfirmWhistle()
    {
        isWhistleMode = false;

        // Прибираємо зону свисту та підказку
        if (currentWhistleArea != null)
        {
            Destroy(currentWhistleArea);
        }

        if (hintText != null)
        {
            hintText.gameObject.SetActive(false);
        }

        PerformWhistle();
    }

    private void PerformWhistle()
    {
        float radius = whistleRadii[whistleLevel];

        // Вибираємо масив звуків залежно від рівня свисту
        AudioClip[] selectedSounds = whistleLevel switch
        {
            0 => weakWhistleSounds,
            1 => mediumWhistleSounds,
            2 => strongWhistleSounds,
            _ => null
        };

        // Відтворюємо випадковий звук із вибраного масиву
        if (audioSource != null && selectedSounds != null && selectedSounds.Length >
0)
        {
            AudioClip randomSound = selectedSounds[Random.Range(0,
selectedSounds.Length)];
            if (randomSound != null)
            {
```

```
                audioSource.PlayOneShot(randomSound, 0.8f); // Гучність 0.8 для
балансу
            }
        }

        // Знайти ворогів у радіусі
        Collider2D[] enemies = Physics2D.OverlapCircleAll(transform.position,
radius, enemyLayer);

        foreach (var enemy in enemies)
        {
            EnemyController = enemy.GetComponent<EnemyController>();
            if (enemyController != null)
            {
                enemyController.MoveToNoise(transform.position); // Відправляємо
ворогів на поточну позицію
            }
        }

        Debug.Log($"Свист на радіусі {radius} активовано!");
    }
}
```

Додаток В

Лістинг коду ритм-гри

HealthDisplay

```
using UnityEngine;
using UnityEngine.UI;

public class HealthDisplay : MonoBehaviour
{
    [Tooltip("Масив зображень-сердець, що представляють здоров'я гравця")]
    public Image[] hearts;

    [Tooltip("Прозорість для повного серця (здоров'я збережено)")]
    [Range(0, 1)]
    public float fullAlpha = 1f;

    [Tooltip("Прозорість для втраченого серця (життя втрачено)")]
    [Range(0, 1)]
    public float lostAlpha = 0.5f;

    /// <summary>
    /// Оновлює відображення здоров'я.
    /// <param name="currentHealth">поточна кількість життів (наприклад, 5, 4, 3,
...).</param>
    /// Серця з індексом менше currentHealth будуть повними, інші – напівпрозорими.
    /// </summary>
    public void UpdateHealth(int currentHealth)
    {
        for (int i = 0; i < hearts.Length; i++)
        {
            Color c = hearts[i].color;
            c.a = (i < currentHealth) ? fullAlpha : lostAlpha;
            hearts[i].color = c;
        }
    }
}
```

HitZoneFeedback

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class HitZoneFeedback : MonoBehaviour
{
    [Tooltip("Спрайт, який відображається за замовчуванням")]
    public Sprite defaultSprite;

    [Tooltip("Спрайт, який відображається при успішному попаданні")]
    public Sprite hitSprite;

    [Tooltip("Спрайт, який відображається при промаху")]
    public Sprite missSprite;

    [Tooltip("Тривалість відображення зміненого спрайту (в секундах)")]
    public float feedbackDuration = 0.5f;

    private Image imageComponent;

    void Awake()
```

```
    {
        imageComponent = GetComponent<Image>();
        if (imageComponent != null && defaultSprite != null)
        {
            imageComponent.sprite = defaultSprite;
        }
    }

    public IEnumerator ShowHitFeedback()
    {
        if (imageComponent != null && hitSprite != null && defaultSprite != null)
        {
            imageComponent.sprite = hitSprite;
            yield return new WaitForSeconds(feedbackDuration);
            imageComponent.sprite = defaultSprite;
        }
    }

    public IEnumerator ShowMissFeedback()
    {
        if (imageComponent != null && missSprite != null && defaultSprite != null)
        {
            imageComponent.sprite = missSprite;
            yield return new WaitForSeconds(feedbackDuration);
            imageComponent.sprite = defaultSprite;
        }
    }
}
```

Note

```
using UnityEngine;
using UnityEngine.UI;

public class Note : MonoBehaviour
{
    [Tooltip("Тип ноти, що визначає напрямок або дію ноти")]
    public NoteType;

    private RectTransform destroyZone; // Зона знищення ноти (при пропуску)
    private float speed;
    private NoteSpawner spawner;

    /// <summary>
    /// Ініціалізація ноти з заданими параметрами.
    /// </summary>
    /// <param name="destroyZone">Зона, при досягненні якої нота вважається
промахом</param>
    /// <param name="noteSpeed">Швидкість руху ноти</param>
    /// <param name="spawnerRef">Посилання на NoteSpawner для реєстрації
помилок</param>
    public void Initialize(RectTransform destroyZone, float noteSpeed, NoteSpawner
spawnerRef)
    {
        this.destroyZone = destroyZone;
        speed = noteSpeed;
        spawner = spawnerRef;
    }

    void Update()
    {
        // Рух ноти до зони знищення
        transform.position = Vector3.MoveTowards(transform.position,
destroyZone.position, speed * Time.deltaTime);
```

```
        // Якщо нота наблизилася до DestroyZone (промах), реєструємо помилку
        if (Vector3.Distance(transform.position, destroyZone.position) < 5f)
        {
            spawner.RegisterError();
            Destroy(gameObject);
        }
    }

    void OnDrawGizmos()
    {
        RectTransform rt = GetComponent<RectTransform>();
        if (rt != null)
        {
            Gizmos.color = Color.green;
            Vector3[] corners = new Vector3[4];
            rt.GetWorldCorners(corners);
            for (int i = 0; i < 4; i++)
            {
                Gizmos.DrawLine(corners[i], corners[(i + 1) % 4]);
            }
        }
    }
}
```

NoteInput

```
using UnityEngine;
using UnityEngine.UI;

public class Note : MonoBehaviour
{
    [Tooltip("Тип ноти, що визначає напрямок або дію ноти")]
    public NoteType;

    private RectTransform destroyZone; // Зона знищення ноти (при пропуску)
    private float speed;
    private NoteSpawner spawner;

    /// <summary>
    /// Ініціалізація ноти з заданими параметрами.
    /// </summary>
    /// <param name="destroyZone">Зона, при досягненні якої нота вважається
промахом</param>
    /// <param name="noteSpeed">Швидкість руху ноти</param>
    /// <param name="spawnerRef">Посилання на NoteSpawner для реєстрації
помилок</param>
    public void Initialize(RectTransform destroyZone, float noteSpeed, NoteSpawner
spawnerRef)
    {
        this.destroyZone = destroyZone;
        speed = noteSpeed;
        spawner = spawnerRef;
    }

    void Update()
    {
        // Рух ноти до зони знищення
        transform.position = Vector3.MoveTowards(transform.position,
destroyZone.position, speed * Time.deltaTime);

        // Якщо нота наблизилася до DestroyZone (промах), реєструємо помилку
        if (Vector3.Distance(transform.position, destroyZone.position) < 5f)
        {
```

```
            spawner.RegisterError();
            Destroy(gameObject);
        }
    }

    void OnDrawGizmos()
    {
        RectTransform rt = GetComponent<RectTransform>();
        if (rt != null)
        {
            Gizmos.color = Color.green;
            Vector3[] corners = new Vector3[4];
            rt.GetWorldCorners(corners);
            for (int i = 0; i < 4; i++)
            {
                Gizmos.DrawLine(corners[i], corners[(i + 1) % 4]);
            }
        }
    }
}
```

NoteSpawner

```
using System.Collections;
using UnityEngine;

public class NoteSpawner : MonoBehaviour
{
    [Header("Налаштування префабів та точок")]
    [Tooltip("Масив префабів нот, індекс відповідає значенню NoteType")]
    public GameObject[] notePrefabs;
    [Tooltip("Точка появи нот (позиція спавну)")]
    public RectTransform spawnPoint;

    [Header("Зони")]
    [Tooltip("UI-елемент HitZone, де відбувається перевірка попадання
(використовується в NoteInput)")]
    public RectTransform hitZone;
    [Tooltip("UI-елемент DestroyZone, при досягненні якого нота вважається
промахом")]
    public RectTransform destroyZone;

    [Header("Параметри гри")]
    [Tooltip("Максимальна кількість помилок (життів) до перезапуску етапу")]
    public int maxErrors = 5;

    [Header("Етапи (налаштовуються в інспекторі)")]
    [Tooltip("Масив даних етапів, що визначають послідовність нот та їхній ритм")]
    public StageData[] stages;

    [Header("Налаштування музики")]
    [Tooltip("Компонент AudioSource для відтворення музики")]
    public AudioSource musicSource;
    [Tooltip("Аудіокліп для міні-гри")]
    public AudioClip backgroundMusic;
    [Range(0, 1)]
    [Tooltip("Гучність музики")]
    public float musicVolume = 1.0f;

    [Header("Відображення здоров'я")]
    [Tooltip("Посилання на компонент HealthDisplay для оновлення відображення
життів")]
    public HealthDisplay;
```

```
private int errorCount = 0;
private int currentStage = 0;

void Start()
{
    if (musicSource != null && backgroundMusic != null)
    {
        musicSource.clip = backgroundMusic;
        musicSource.volume = musicVolume;
        musicSource.Play();
    }

    // Оновлюємо відображення здоров'я – спочатку всі серця повні
    if (healthDisplay != null)
    {
        healthDisplay.UpdateHealth(maxErrors);
    }

    StartCoroutine(SpawnStages());
}

IEnumerator SpawnStages()
{
    while (currentStage < stages.Length)
    {
        StageData stage = stages[currentStage];
        Debug.Log("Початок етапу: " + stage.stageName);

        float localBeatInterval = 60f / stage.bpm;
        for (int i = 0; i < stage.noteSequence.Length; i++)
        {
            NoteType = stage.noteSequence[i];
            SpawnNote(noteType, stage.noteSpeed);

            float delay = localBeatInterval;
            if (stage.noteDelays != null && stage.noteDelays.Length > i)
            {
                delay = stage.noteDelays[i];
            }

            yield return new WaitForSeconds(delay);
        }

        yield return new WaitForSeconds(1f); // Пауза між етапами
        currentStage++;
    }

    Debug.Log("Усі етапи завершено!");
}

// Метод приймає швидкість ноти і передає destroyZone для обробки промаху
void SpawnNote(NoteType noteType, float noteSpeed)
{
    int index = (int)noteType;
    if (index < 0 || index >= notePrefabs.Length)
    {
        Debug.LogError("Немає префаба для ноти " + noteType);
        return;
    }

    GameObject noteObj = Instantiate(
        notePrefabs[index],
        spawnPoint.position,
        notePrefabs[index].transform.rotation,
        transform
```

```
        );

        Note = noteObj.GetComponent<Note>();
        if (note != null)
        {
            note.noteType = noteType;
            note.Initialize(destroyZone, noteSpeed, this);
        }
    }

    public void RegisterError()
    {
        errorCount++;
        Debug.Log("<color=red>Помилка! Поточних помилок: " + errorCount +
"</color>");

        // Оновлюємо відображення здоров'я: кількість життів, що залишилася =
maxErrors - errorCount
        if (healthDisplay != null)
        {
            healthDisplay.UpdateHealth(maxErrors - errorCount);
        }

        if (errorCount >= maxErrors)
        {
            Debug.Log("<color=red>Досягнуто максимальної кількості помилок.
Перезапуск етапу...</color>");
            RestartStage();
        }
    }

    void RestartStage()
    {
        StopAllCoroutines();
        foreach (Transform child in transform)
        {
            Destroy(child.gameObject);
        }

        errorCount = 0;
        // Відновлюємо здоров'я - всі серця повні
        if (healthDisplay != null)
        {
            healthDisplay.UpdateHealth(maxErrors);
        }

        StartCoroutine(SpawnStages());
    }
}
```

NoteType

```
public enum NoteType
{
    Up,         // 0
    Down,       // 1
    Left,       // 2
    Right,      // 3
    UpLeft,     // 4
    UpRight,    // 5
    DownLeft,   // 6
    DownRight   // 7
}
```

StageData

```
using System;
using UnityEngine;

[Serializable]
public class StageData
{
    [Tooltip("Назва етапу для зручності в інспекторі")]
    public string stageName;

    [Header("Параметри ритму")]
    [Tooltip("BPM (ударів на хвилину) для даного етапу. Формула: 60/BPM")]
    public float bpm = 120f;
    [Tooltip("Швидкість руху нот для даного етапу")]
    public float noteSpeed = 300f;

    [Header("Послідовність нот")]
    [Tooltip("Послідовність нот для даного етапу (кожен елемент відповідає
напрямку)")]
    public NoteType[] noteSequence;

    [Header("Затримки між нотами")]
    [Tooltip("Масив затримок між нотами (в секундах). Якщо довжина менша за
кількість нот, для решти використовується інтервал, що обчислюється з BPM")]
    public float[] noteDelays;
}
```

TargetZoneGizmo

```
using UnityEngine;

public class TargetZoneGizmo : MonoBehaviour
{
    void OnDrawGizmos()
    {
        RectTransform rt = GetComponent<RectTransform>();
        if (rt != null)
        {
            Gizmos.color = Color.yellow;
            Vector3[] corners = new Vector3[4];
            rt.GetWorldCorners(corners);
            for (int i = 0; i < 4; i++)
            {
                Gizmos.DrawLine(corners[i], corners[(i + 1) % 4]);
            }
        }
    }
}
```