

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**
Факультет інформаційних технологій

ПОГОДЖЕНО
Декан факультету

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри комп'ютерних
наук

_____ **Ігор БОЛБОТ**
(підпис)

_____ **Белла ГОЛУБ**
(підпис)

« » _____ 2026 р.

« » _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: **«Програмне забезпечення для розвитку дрібної моторики
рук на основі музичного ритму _____»**

Спеціальність «121 Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент

_____ **Ганна ВАЙГАНГ**
(підпис)

**Керівник бакалаврської
кваліфікаційної роботи**

_____ **Матвій ГОРБАЧ**
(підпис)

**Консультант бакалаврської
кваліфікаційної роботи**

_____ **Белла ГОЛУБ**
(підпис)

к.т.н., доцент

Виконав

_____ **Богдан ВОЛОХІН**
(підпис)

Київ-2026

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій**

ЗАТВЕРДЖУЮ
**Завідувач кафедри комп'ютерних
наук**

к.т.н., доцент _____Белла ГОЛУБ
(підпис)

«_____» ____2025 р.

**ЗАВДАННЯ ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ
РОБОТИ ЗДОБУВАЧУ**

Волохіну Богдану Олександровичу

(прізвище, ім'я, по батькові)

Спеціальність «121 Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення»

Тема бакалаврської кваліфікаційної роботи

**«Програмне забезпечення для розвитку дрібної моторики рук на
основі музичного ритму»**

затверджена наказом від «19» грудня 2025 р. № 3196 «С»

Термін подання завершеної роботи на кафедру «23» травня 2026 р.

Вихідні дані до бакалаврської кваліфікаційної роботи:

Матеріали щодо гомункулуса Пенфілда, специфікації рушія Unity та СУБД
SQLite.

Перелік питань, що підлягають дослідженню:

- 1 Аналіз предметної області та постановка завдання
- 2 Проєктування інформаційного забезпечення
- 3 Розробка програмного забезпечення
- 4 Впровадження та практичне використання програмної системи

Дата видачі завдання «19» грудня 2025 р.

**Керівник бакалаврської
кваліфікаційної роботи**

_____ **Матвій ГОРБАЧ**
(підпис)

**Консультант бакалаврської
кваліфікаційної роботи**

_____ **Бела ГОЛУБ**
(підпис)

к. т. н. доцент

Завдання взяв до виконання

_____ **Богдан ВОЛОХІН**
(підпис)

ЗМІСТ

ЗМІСТ	3
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	5
ВСТУП	6
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ	9
1.1 Характеристика процесу тренування дрібної моторики як об'єкта автоматизації	9
1.2 Аналіз та обґрунтування вимог до програмного забезпечення	11
1.3 Функціональне моделювання предметної області (UML Use Case)	13
1.4 Порівняльний аналіз існуючих аналогів та технічних рішень	16
1.5 Постановка завдання на розробку програмного забезпечення	21
2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ	24
2.1 Проектування логічної структури бази даних (ER-діаграма)	24
2.2 Об'єктно-орієнтоване проектування структури класів та їх взаємодії	26
2.3 Розробка архітектурної структури та пакетів програмної системи	32
2.4 Компонентна структура та фізичне розміщення модулів системи	33
3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	36
3.1 Обґрунтування вибору СУБД та інструментальних засобів розробки	36
3.2 Фізична реалізація бази даних та генерація об'єктів	37
3.3 Вибір інструментарію для створення прикладного програмного забезпечення	39
3.4 Алгоритмічна реалізація та програмування бізнес-логіки (Unity/C#)	40
4 ВПРОВАДЖЕННЯ ТА ПРАКТИЧНЕ ВИКОРИСТАННЯ ПРОГРАМНОЇ СИСТЕМИ	43
4.1 Тестування функціональних модулів та аналіз результатів	43
4.2 Топологія розгортання та системні вимоги до середовища виконання	48
4.3 Склад інсталяційного пакету	50
4.4 Інструкція користувача та опис інтерфейсу	52
4.5 Оцінка перспективи розвитку програмного продукту	55

ВИСНОВКИ	57
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	60
ДОДАТОК А	63
ДОДАТОК Б	90
ДОДАТОК В	92

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД — база даних

ПК — персональний комп'ютер

ПЗ — програмне забезпечення

СУБД — система управління базами даних

API — Application Programming Interface (інтерфейс прикладного програмування)

BPM — Beats Per Minute (кількість ударів на хвилину)

ER — Entity-Relationship (модель «сутність-зв'язок»)

FPS — Frames Per Second (кількість кадрів за секунду)

ID — Identifier (ідентифікатор)

RSI — Repetitive Strain Injury (хронічне перенапруження кисті через повторювані рухи)

SHA256 — Secure Hash Algorithm 256-bit (криптографічний алгоритм хешування)

UI — User Interface (інтерфейс користувача)

UML — Unified Modeling Language (уніфікована мова моделювання)

ВСТУП

Актуальність теми. У сучасному світі проблема відновлення та розвитку дрібної моторики рук охоплює широке коло завдань: від реабілітації після когнітивних порушень та травм до розвитку координації у дітей. Згідно з науковими дослідженнями в галузі нейрофізіології, розвиток тонких рухів пальців безпосередньо пов'язаний з активністю мовленнєвих та інтелектуальних центрів головного мозку. Однак традиційні методи терапії часто стикаються з проблемою низької залученості пацієнтів через монотонність вправ.

Медичний аспект актуальності базується на концепції нейропластичності — здатності мозку створювати нові нейронні зв'язки у відповідь на зовнішні стимули. Музичний ритм у цьому контексті виступає як потужний зовнішній «водій», який допомагає структурувати рухову активність. Поєднання аудіовізуальних стимулів з необхідністю виконання точних рухів у часові межі (ритм) активізує одночасно слухову, зорову та моторну кору головного мозку. Це створює ефект синергії, що значно пришвидшує формування стійких паттернів руху.

Застосування гейміфікованого програмного забезпечення дозволяє вирішити ключову психологічну проблему — підтримку мотивації. Використання сучасних технологій, таких як ігровий рушій Unity, дає змогу створити середовище, де користувач отримує миттєвий зворотний зв'язок через візуальні ефекти та підрахунок балів. Це перетворює процес нудної реабілітації на інтерактивний тренінг, що особливо важливо в дитячій терапії та при тривалих відновлювальних процесах.

Таким чином, розробка програмного забезпечення, що базується на синхронізації дрібної моторики з музичним ритмом, є актуальним міждисциплінарним завданням, що лежить на перетині програмної інженерії, когнітивної психології та реабілітології.

Мета розробки. Метою бакалаврської роботи є проєктування та розробка програмного забезпечення, що забезпечує тренування дрібної моторики рук шляхом взаємодії користувача з ігровими об'єктами в ритмі музичних композицій. Програмна система має автоматизувати процес відстеження успішності користувача, надавати зворотний зв'язок у реальному часі та забезпечувати збереження результатів для аналізу прогресу.

Методи та технології. Для реалізації програмного забезпечення було обрано ігровий рушій Unity, що дозволяє створювати кросплатформні додатки з високою продуктивністю графіки та звуку. В основі архітектури системи лежить об'єктно-орієнтований підхід та мова програмування C#. Для керування даними (користувачі, пісні, рівні та результати) використано реляційну базу даних SQLite, доступ до якої реалізовано за допомогою бібліотеки Mono.Data.Sqlite.

Основні програмні модулі включають:

- AuthManager – для авторизації та реєстрації користувачів із використанням криптографічного шифрування паролів (SHA256).
- BeatScroller та NoteGenerator – для візуалізації ритмічних елементів.
- GameManager – для контролю ігрового процесу та підрахунку результатів.
- SongSelectionManager – для керування базою музичних треків.

Апробація результатів. Основні положення та результати розробки обговорювалися на науково-технічних конференціях студентів факультету інформаційних технологій НУБіП України (або вкажіть конкретні заходи, де ви виступали). Програмний додаток пройшов попереднє тестування на групі користувачів для перевірки зручності інтерфейсу та коректності роботи ритм-механіки.

Структура пояснювальної записки. Робота складається зі вступу, чотирьох розділів основної частини, висновків, списку використаних джерел та додатків.

- У першому розділі проведено системний аналіз предметної області, досліджено існуючі рішення для розвитку дрібної моторики та сформульовано вимоги до системи.

- Другий розділ присвячено проєктуванню архітектури додатка, розробці діаграм класів та схеми бази даних.
- У третьому розділі описано процес програмної реалізації модулів на Unity та налаштування взаємодії з базою даних SQLite.
- Четвертий розділ містить опис процедур тестування та інструкції з експлуатації програмного забезпечення.

Загальний обсяг пояснювальної записки становить 62 сторінку машинописного тексту, містить 24 рисунка та 0 таблиць. Список використаних джерел налічує 25 найменувань.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Характеристика процесу тренування дрібної моторики як об'єкта автоматизації

Системний аналіз предметної області є першочерговим етапом розробки, що дозволяє дослідити структуру взаємозв'язків між фізіологічними потребами користувача та технологічними можливостями програмного забезпечення. У межах даного розділу проводиться декомпозиція процесу тренування моторики, аналізуються чинники впливу музичного ритму на нейропластичність мозку та формується перелік функціональних і технічних вимог до проектованої системи. Основна увага приділяється створенню ефективного механізму взаємодії «користувач-інтерфейс», де кожна дія синхронізована з аудіовізуальним рядом для досягнення максимального терапевтичного та розвиваючого ефекту.

Предметною областю даного дослідження є процес розвитку та відновлення дрібної моторики рук людини за допомогою інформаційних технологій та мультимедійного впливу. Дрібна моторика — це сукупність скоординованих дій нервової, м'язової та кісткової систем, часто у поєднанні із зоровою системою у виконанні дрібних і точних рухів кистями та пальцями рук.

Ключові компоненти предметної області:

1. Фізіологічний аспект: Тренування дрібної моторики необхідне для різних категорій користувачів:

- Діти: розвиток моторики тісно пов'язаний із формуванням мовлення, пам'яті та логічного мислення.
- Пацієнти в реабілітації: відновлення рухливості після травм, інсультів або при неврологічних захворюваннях
- Професійна підготовка: люди, чия робота вимагає високої точності рухів (хірурги, музиканти, програмісти).

2. Музично-ритмічний вплив: Ритм виступає як зовнішній метроном. Науково доведено, що ритмічна стимуляція активізує базальні ганглії мозку, що відповідають за автоматизацію рухів. У контексті програмного забезпечення це означає, що користувач виконує дію не хаотично, а у чітко визначені моменти часу, що підвищує концентрацію та ефективність тренування.

3. Ігрова механіка (Gamification): Процес тренування моделюється як взаємодія з динамічними об'єктами (нотами), що рухаються по екрану. Основні бізнес-процеси в цій області включають:

- Синхронізацію графічних елементів з аудіопотоком.
- Обробку вводу користувача з мінімальною затримкою.
- Оцінювання точності потрапляння в ритм (таймінг).
- Накопичення статистичних даних для моніторингу прогресу.

Аналіз процесу взаємодії в системі:

Об'єктом автоматизації є сеанс тренування. Традиційно такий процес вимагає присутності фахівця (реабілітолога або педагога), який дає завдання та фіксує результати вручну. Програмне рішення дозволяє перенести цей процес у цифрову площину:

- Вхідні дані: музичні файли, метадані рівнів (час появи нот), профілі користувачів.
- Процес: відтворення аудіо, візуалізація ритм-сетки, детектування натискань клавіш/сенсора, порівняння часу натискання з еталонним часом ноти.
- Вихідні дані: показники точності (Accuracy), кількість набраних балів (Score), комбо-серії, рівень прогресу користувача.

Основними проблемами існуючих підходів у цій області є висока вартість спеціалізованого обладнання та низька мотиваційна складова стандартних вправ. Створення доступного кросплатформного ПЗ на базі Unity дозволяє зробити такі тренування масовими, цікавими та такими, що піддаються точному математичному аналізу через збереження логів кожної ігрової сесії в базі даних.

1.2 Аналіз та обґрунтування вимог до програмного забезпечення

Процес аналізу вимог є фундаментальним етапом розробки, оскільки він визначає межі проєкту та критерії оцінки його успішності. Для програмного забезпечення, що спрямоване на розвиток дрібної моторики на основі ритму, вимоги мають враховувати як технічні аспекти ігрового рушія Unity, так і специфіку взаємодії користувача з ритмічними елементами.

1.2.1 Функціональні вимоги. Функціональні вимоги визначають конкретні сервіси та функції, які повинна виконувати система. Для даного проєкту виділено наступні групи вимог:

- Керування обліковими записами:
- Система повинна забезпечувати можливість реєстрації нових користувачів із унікальними логінами.
- Реалізація безпечної авторизації через перевірку хешованих паролів (алгоритм SHA256).
- Збереження персональної статистики для кожного профілю окремо.
- Керування музичним контентом:
- Можливість перегляду списку доступних музичних композицій.
- Завантаження метаданих пісні (назва, виконавець, темп BPM) з бази даних.
- Вибір рівня складності для обраної композиції (Easy, Medium, Hard).
- Ігровий процес та тренування моторики:
- Синхронізована генерація ігрових об'єктів (нот) відповідно до ритмічної сітки композиції.
- Візуалізація руху об'єктів по виділених доріжках у реальному часі.
- Детектування дій користувача (натискання клавіш) та порівняння їх з часовими мітками появи нот.
- Класифікація точності натискання (Perfect, Good, Normal, Miss) залежно від величини відхилення від еталонного часу.

- Статистика та аналітика:
- Підрахунок балів у реальному часі.
- Відстеження максимальної серії безперервних успішних натискань (Combo).
- Розрахунок фінальної точності (Accuracy) після завершення рівня.
- Автоматичне збереження результатів сесії в базу даних SQLite.

1.2.2 Нефункціональні вимоги. Нефункціональні вимоги визначають обмеження та якісні характеристики системи:

- Продуктивність (Performance):
- Мінімальна затримка (latency) між звуковим сигналом та візуальним відображенням об'єкта.
- Стабільна частота кадрів (не менше 60 FPS) для плавності руху ігрових елементів, що критично для моторики.
- Швидке завантаження аудіоресурсів та рівнів.
- Надійність (Reliability):
- Коректна робота з базою даних, запобігання втраті результатів при раптовому закритті програми.
- Валідація введених даних (наприклад, перевірка довжини пароля або наявності логіна в базі).
- Зручність використання (Usability):
- Інтуїтивно зрозумілий інтерфейс користувача (UI).
- Контрастне відображення активних елементів для людей з різним рівнем зорового сприйняття.
- Наявність зворотного зв'язку (візуальні ефекти при успішному натисканні).

1.2.3 Вимоги до апаратного та програмного забезпечення. Для стабільної роботи ПЗ встановлено наступні мінімальні вимоги:

- Програмні вимоги:
- Операційна система: Windows 10/11
- Встановлені драйвери для підтримки DirectX 11/12 або OpenGL ES 3.0.

- Апаратні вимоги:
- Процесор: Двоядерний з частотою від 2.0 ГГц.
- Оперативна пам'ять: Не менше 4 ГБ.
- Відеокарта: Інтегрована або дискретна з підтримкою шейдерів.
- Пристрої вводу: Клавіатура та миша (для ПК).
- Аудіосистема: Динаміки або навушники для відтворення ритмічного супроводу.

1.3 Функціональне моделювання предметної області (UML Use Case)

Моделювання предметної області є критично важливим етапом проектування, оскільки воно дозволяє візуалізувати функціональні можливості системи, визначити межі програмного продукту та встановити зв'язки між користувачем і внутрішніми модулями. Для розробки програмного забезпечення для розвитку дрібної моторики на основі музичного ритму було обрано об'єктно-орієнтований підхід із використанням уніфікованої мови моделювання UML.

1.3.1 Аналіз функціональних вимог та ідентифікація акторів.

Основним призначенням системи є створення інтерактивного середовища, де користувач виконує фізичні вправи (натискання клавіш або взаємодія з сенсорним екраном) у чіткій відповідності до аудіовізуальних стимулів.

У межах даної предметної області виділено наступного ключового актора:

- Користувач (User) — фізична особа, яка безпосередньо взаємодіє з інтерфейсом гри з метою тренування моторики. Користувач може бути як пацієнтом у процесі реабілітації, так і звичайною людиною, що бажає покращити координацію рухів.

1.3.2 Діаграма прецедентів (Use Case Diagram). Діаграма прецедентів описує функціональну поведінку системи через сукупність сценаріїв використання. Кожен прецедент відповідає окремій сервісній можливості, яку надає ПЗ.



Рис. 1.1 Діаграма прецедентів системи

Детальний опис основних прецедентів:

- Реєстрація та авторизація. Даний сценарій передбачає створення унікального профілю в локальній базі даних SQLite. Це необхідно для персоналізації процесу навчання, оскільки результати тренувань (точність, швидкість реакції) мають прив'язуватися до конкретного ідентифікатора користувача.
- Налаштування параметрів сесії. Користувач має можливість обрати рівень складності (Easy, Medium, Hard). Кожен рівень складності змінює параметри `scroll_speed` (швидкість руху нот) та щільність розташування об'єктів Notes у часі.
- Вибір музичної композиції. Система звертається до таблиці Songs у базі даних, зчитує метадані (назва, автор, темп BPM) та надає користувачу список для вибору.
- Ігровий процес (Виконання вправи). Це центральний прецедент, який включає запуск аудіофайлу, синхронізовану генерацію візуальних елементів та зчитування вводу від користувача. Система порівнює

фактичний час натискання клавіші з еталонним часом `spawn_time`, зафіксованим у базі даних.

- Аналіз результатів. Після завершення треку система розраховує статистику (точність у відсотках, кількість пропущених нот, максимальне комбо) та зберігає запис у таблицю `Scores`.

1.3.3 Опис сценаріїв взаємодії - діаграма послідовностей. Для більш глибокого розуміння того, як система обробляє запити, розглянемо сценарій «Запуск ігрового рівня». Цей процес демонструє динаміку взаємодії між об'єктами:

- Користувач ініціює подію натисканням кнопки «Start».
- Об'єкт `SongSelectionManager` передає ID обраної пісні до `GameManager`.
- `LevelLoader` виконує SQL-запит до БД для отримання списку нот для даного рівня.
- `NoteGenerator` створює екземпляри об'єктів у середовищі Unity згідно з отриманими координатами часу.
- `BeatScroller` забезпечує рівномірний рух нот до зони активності.

1.3.4 Правила бізнес-логіки системи. Предметна область також накладає певні обмеження та правила обробки даних (бізнес-правила):

- Синхронізація: затримка між аудіо-сигналом та візуальним відображенням не повинна перевищувати 10-15 мс для забезпечення коректності терапевтичного ефекту.
- Оцінювання: точність натискання класифікується за категоріями (Perfect, Good, Normal, Miss) залежно від часового вікна (hit-window).
- Безпека даних: паролі користувачів повинні зберігатися у вигляді хеш-сум (`password_hash`), що реалізовано в модулі `AuthManager`.

1.4 Порівняльний аналіз існуючих аналогів та технічних рішень

Для розробки ефективного програмного забезпечення, спрямованого на розвиток дрібної моторики рук через музичний ритм, необхідно провести

порівняльний аналіз існуючих систем. Аналоги можна розділити на три категорії: професійні реабілітаційні комплекси, комерційні ритм-ігри та освітні тренажери.

1.4.1 Спеціалізована реабілітаційна система RehaCom. RehaCom є лідером у сфері комп'ютерної когнітивної реабілітації. Система включає численні модулі для тренування уваги, пам'яті та сенсомоторних реакцій. Вона часто використовується разом із спеціальними панелями керування (великими кнопками або джойстиками) (рис 1.2).



Рис. 1.2 Спеціалізована панель керування системи RehaCom

Переваги:

- Наукова база: Кожна вправа розроблена на основі клінічних досліджень.
- Адаптивність: Система автоматично підлаштовує рівень складності під поточні можливості пацієнта.
- Детальна звітність: Можливість відстежувати динаміку відновлення протягом тривалого часу.

Недоліки:

- Висока вартість: Програмне забезпечення та обладнання є дуже дорогими і доступні переважно в клініках.

- Низька мотивація: Вправи мають суто медичний характер, вони монотонні і позбавлені розважального елементу, що критично для тривалого використання вдома.
- Громіздкість: Вимагає спеціальної інсталяції та навчання персоналу.

1.4.2 Музична гра Guitar Hero. Популярна серія ігор, де гравець імітує гру на гітарі, натискаючи клавіші в такт музиці. Хоча це комерційний продукт, він демонструє потужний вплив ритмічної активності на координацію рухів.



Рис. 1.3 Скріншот ігрового процесу Guitar Hero

Переваги:

- Висока залученість: Гейміфікація та ліцензована музика стимулюють користувача грати знову.
- Тренування швидкості: Вимагає швидкого перемикання між пальцями на високих рівнях складності.
- Чіткий зворотний зв'язок: Гравець миттєво бачить свої помилки через візуальні та аудіо-ефекти.

Недоліки:

- Залежність від периферії: Повноцінна гра вимагає спеціального контролера-гітари.

- Відсутність аналітики: Гра не фіксує прогрес з точки зору фізіології чи реабілітації, лише ігрові бали.
- Складність для початківців: Механіка може бути занадто швидкою для людей з порушеннями моторики.

1.4.3 Ритм-платформа Osu! Безкоштовна гра, що базується на взаємодії з активними елементами на екрані за допомогою миші, графічного планшета або клавіатури. Вона відома своєю екстремальною складністю та потребою у високій точності.

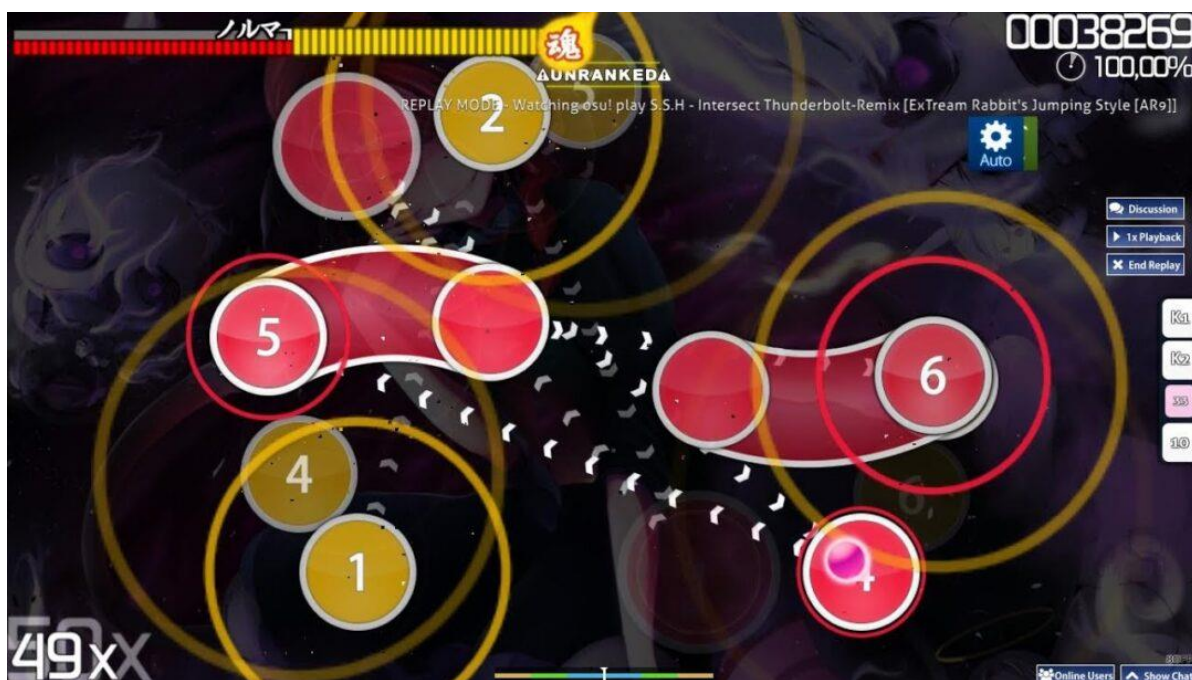


Рис. 1.4 Скріншот ігрового процесу osu!

Переваги:

- Доступність: Величезна база користувацьких треків (beatmaps) на будь-який смак.
- Гнучкість налаштувань: Користувач може змінювати швидкість та розмір об'єктів.
- Розвиток координації «око-рука»: Найкращий інструмент для тренування швидкості реакції.

Недоліки:

- Високий поріг входження: Гра орієнтована на хардкорних гравців, що робить її непридатною для початкової реабілітації.

- Ризик травматизму: Через надмірну інтенсивність і повторювані рухи існує ризик перенапруження кисті (RSI).
- Відсутність структурованого курсу: Процес навчання хаотичний.

1.4.4 Веб-тренажер TypeRacer. Система для навчання швидкому друку, де користувачі змагаються у швидкості набору тексту. Хоча вона не є музичною, вона напряду розвиває незалежність рухів пальців.

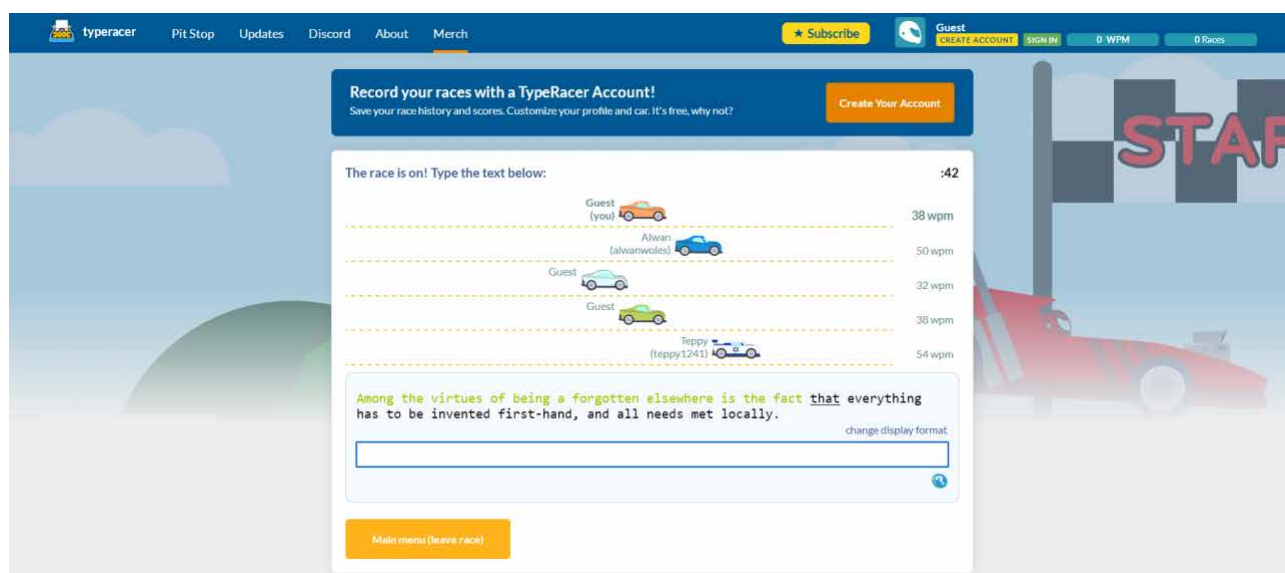


Рис. 1.5 Скріншот Веб застосунку TypeRacer

Переваги:

- Практична цінність: Навички друку корисні в повсякденному житті та роботі.
- Змагальний елемент: Можливість порівнювати свої результати з іншими користувачами в реальному часі.

Недоліки:

- Відсутність ритму: Швидкість диктується лише текстом, немає емоційного підкріплення музикою.
- Обмеженість рухів: Тільки стандартна розкладка клавіатури, що не завжди охоплює всі аспекти моторики.
- Монотонність: Швидко набридає через відсутність візуальної різноманітності.

1.4.5 Project Sekai: Colorful Stage! feat. Hatsune Miku. Сучасна мобільна ритм-гра, яка використовує сенсорний екран та різноманітні типи нот: натискання, утримання, слайди.



Рис. 1.6 Знімок ігрового процесу Project Sekai

Переваги:

- **Різнорманітність механік:** Використання різних жестів (свайпи вгору/вбік) дозволяє задіяти різні м'язові групи пальців.
- **Портативність:** Можливість займатися будь-де за допомогою смартфона чи планшета.
- **Сюжетна лінія:** Наявність персонажів та історії підвищує довгострокову лояльність користувача.

Недоліки:

- **Відволікаючі фактори:** Велика кількість візуальних ефектів може перевантажувати користувача та заважати концентрації на рухах.
- **Gacha-механіки:** Наявність азартних елементів та внутрішньоігрових покупок, що неприпустимо для медичного чи освітнього ПЗ.
- **Закритість:** Неможливо вивантажити дані про точність натискань для зовнішнього аналізу.

1.4.6 Обґрунтування власної розробки. Аналіз існуючих рішень показує, що на ринку існує значний розрив між професійними медичними системами (RehaCom), які є дорогими та нудними, і комерційними іграми (Osu!, Project Sekai), які є цікавими, але занадто складними та позбавленими аналітичної складової.

Проектоване програмне забезпечення покликане заповнити цю нішу шляхом:

- Поєднання ритм-механіки з аналітикою: Використання SQLite для детального логування кожного натискання (точність у мілісекундах), що дозволяє будувати графік прогресу.
- Доступності: Робота на стандартних ПК (Unity) без потреби у специфічному обладнанні.
- Адаптивного контенту: Можливість підбору рівнів складності (Easy, Med, Hard), що відповідають поточному стану моторики користувача.
- Мотивації: Використання ігрового рушія для створення привабливого візуального ряду, який перетворює «вправи» на «гру».

Таким чином, розробка власного ПЗ є актуальною та доцільною, оскільки дозволяє створити персоналізований, економічно доступний та високомотивований інструмент для розвитку дрібної моторики.

1.5 Постановка завдання на розробку програмного забезпечення

На основі проведеного аналізу предметної області, огляду існуючих рішень та виявлених недоліків комерційних аналогів (висока вартість, потреба у специфічному обладнанні, відсутність гнучкої аналітики), було визначено необхідність розробки спеціалізованого програмного забезпечення.

Метою бакалаврської кваліфікаційної роботи є проектування та розробка кросплатформного програмного додатка на базі ігрового рушія Unity, який забезпечує ефективне тренування дрібної моторики рук шляхом синхронізації

фізичних дій користувача з музичним ритмом та надає інструменти для детального аналізу прогресу.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Проектування архітектури системи:

- Розробити логічну модель даних (ER-діаграма) для зберігання інформації про користувачів, музичні треки та результати сесій у БД SQLite.

2. Розробка функціональних модулів (на Unity):

- Створити систему Beat Mapping, яка дозволяє синхронізувати появу візуальних стимулів з темпом (BPM) музичного файлу.
- Реалізувати механізм детектування натискань з точністю до мілісекунд для об'єктивної оцінки успішності вправ.
- Розробити адаптивний інтерфейс користувача (UI), що підтримує три рівні складності (Easy, Med, Hard) відповідно до поточного стану моторики пацієнта.

3. Інтеграція системи управління базами даних:

- Реалізувати модуль взаємодії з SQLite для реєстрації та авторизації користувачів.
- Забезпечити автоматичне логування кожного тренування (точність, кількість пропусків, максимальне комбо) для подальшої візуалізації прогресу.

4. Забезпечення мотиваційної складової:

- Впровадити ігрові елементи (гейміфікацію): систему нарахування балів, оцінку результату (ранги) та візуальні ефекти успішного виконання дій.

5. Тестування та верифікація:

- Провести функціональне тестування всіх модулів системи.
- Перевірити коректність роботи алгоритму синхронізації на різних музичних треках та стабільність збереження даних.

Об'єктом дослідження є процес розвитку та реабілітації дрібної моторики рук засобами цифрових технологій.

Предметом дослідження є методи та програмні засоби створення інтерактивного ритм-середовища на базі Unity та SQLite для стимуляції моторних функцій.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Проєктування логічної структури бази даних (ER-діаграма)

Другий розділ присвячено детальному проєктуванню архітектури програмного комплексу для розвитку дрібної моторики рук. На основі завдань, сформульованих у попередньому розділі, необхідно розробити комплексну модель системи, яка забезпечить стабільну роботу ритм-механік, надійне збереження даних користувачів та ефективну взаємодію між програмними модулями.

Процес проєктування базується на об'єктно-орієнтованому підході та використанні уніфікованої мови моделювання UML. Це дозволяє візуалізувати структуру системи, визначити логіку взаємодії об'єктів та спроектувати базу даних, що відповідає вимогам продуктивності ігрового рушія Unity.

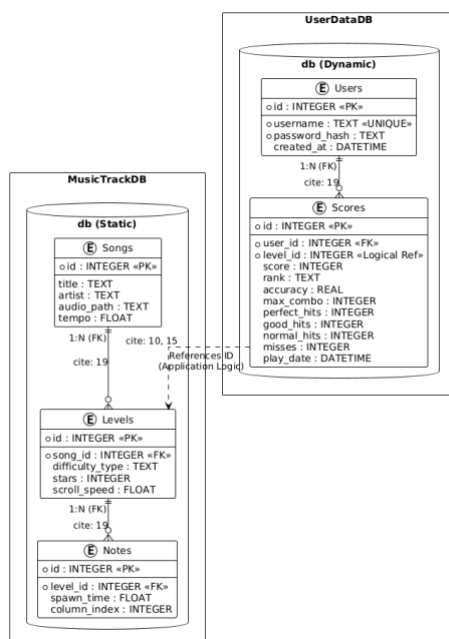


Рис. 2.1 Логічна модель даних (ER-діаграма)

Для забезпечення функціонування системи розвитку дрібної моторики необхідно реалізувати структуроване збереження даних про користувачів,

контент (музичні треки та рівні) та результати тренувань. На основі вимог до системи було спроектовано логічну модель даних, представлену у вигляді ER-діаграми (рис. 2.1).

Основними сутностями бази даних є:

- Users (Користувачі) — зберігає дані для аутентифікації (логін та хешований пароль). Це дозволяє персоналізувати процес тренувань та відстежувати індивідуальну динаміку кожного пацієнта.
- Songs (Пісні) — містить інформацію про музичні композиції, включаючи назву, автора та темп (BPM), що є критично важливим для синхронізації візуального ряду з ритмом.
- Levels (Рівні складності) — пов'язує пісні з конкретними параметрами складності (Easy, Med, Hard). Для кожного рівня визначено швидкість прокрутки нот (scroll_speed) та рейтинг у зірках.
- Notes (Ноти/Стимули) — найбільш деталізована таблиця, що містить координати появи кожного ігрового об'єкта (column_index) та точний час його появи (spawn_time). Це забезпечує детермінованість тренувального процесу.
- Scores (Результати) — сутність, що фіксує підсумки кожної ігрової сесії. Вона зберігає не лише загальний бал, а й детальну метрику: кількість точних натискань (perfect_hits), пропусків (misses), максимальну серію успішних дій (max_combo) та загальну точність у відсотках.

Зв'язки між сутностями:

- Зв'язок «один-до-багатьох» (1:N) між Songs та Levels: одна пісня може мати кілька варіантів складності.
- Зв'язок 1:N між Levels та Notes: один рівень складається з великої кількості послідовних нот.
- Зв'язок 1:N між Users та Scores: один користувач може мати безліч спроб проходження рівнів.

- Зв'язок 1:N між Levels та Scores: для кожного конкретного рівня накопичується історія результатів різних користувачів.

Використання реляційної моделі та обмежень цілісності (Foreign Keys з правилом ON DELETE CASCADE) гарантує несуперечність даних: наприклад, при видаленні профілю користувача або музичного треку автоматично видаляються всі пов'язані з ними записи статистики. Це забезпечує чистоту бази даних та надійність роботи додатку.

2.2 Об'єктно-орієнтоване проектування структури класів та їх взаємодії

Проектування програмної логіки системи розвитку дрібної моторики базується на об'єктно-орієнтованому підході, що реалізується засобами ігрового рушія Unity та мови програмування C#. Діаграма класів (Class Diagram) відображає статичну структуру системи, визначаючи склад класів, їхні атрибути, методи та типи взаємозв'язків між ними.

Для забезпечення модульності та гнучкості архітектури, класи системи розділені на декілька функціональних блоків: обробка ігрової логіки, управління даними та інтерфейсна частина (рис 2.2).

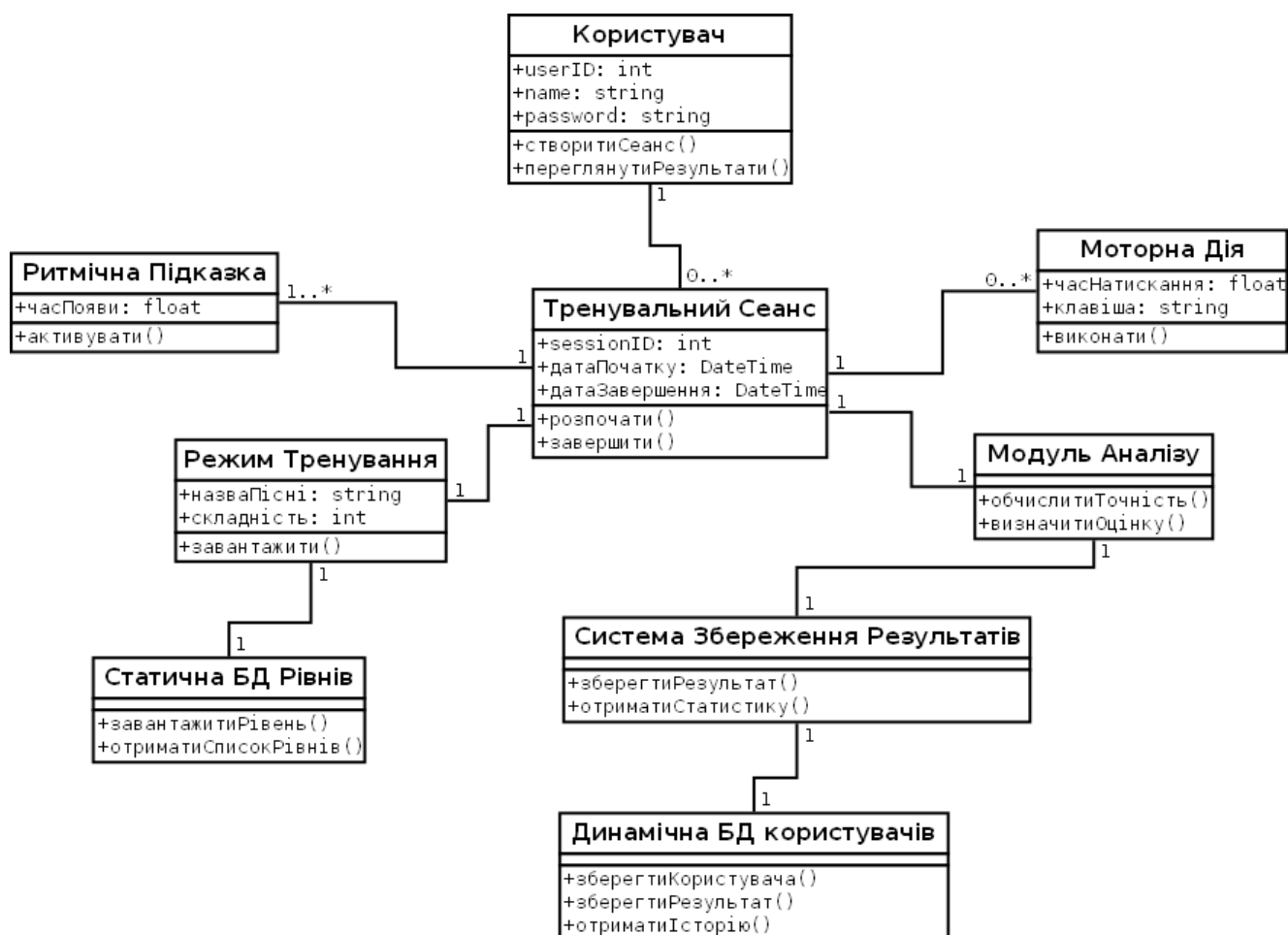


Рис. 2.2 Діаграма класів

Опис основних класів системи:

- **Користувач (User)** — керує персональними даними та ініціює ігровий процес. Містить ідентифікатор, ім'я та пароль для аутентифікації, а також методи для створення нових тренувальних сеансів та перегляду власних результатів.
- **Тренувальний Сеанс (Training Session)** — центральний вузол логіки, що координує взаємодію між користувачем, обраним режимом та системою аналізу. Він зберігає часові мітки початку і завершення гри та відповідає за запуск і зупинку процесу тренування.
- **Режим Тренування (Training Mode) та Статична БД Рівнів** — відповідають за контент і складність гри. Режим Тренування визначає параметри конкретної пісні та завантажує необхідні дані, тоді як Статична БД Рівнів надає доступ до списку доступних рівнів та безпосередньо завантажує структуру обраного рівня.

- **Ритмічна Підказка (Rhythmic Hint)** — генерує візуальні або звукові сигнали для користувача. На основі параметра часПояви (spawn time) вона активується у відповідний момент ігрового часу, щоб підказати момент для виконання дії.
- **Моторна Дія (Motor Action)** — фіксує фізичну відповідь користувача на підказки. Вона реєструє точний часНатискання та конкретну клавішу, що дозволяє системі порівняти реальну дію гравця з еталонним ритмом.
- **Модуль Аналізу (Analysis Module)** — обробляє зібрані дані про моторні дії для розрахунку точності. Він обчислює відсоток попадань у ритм та визначає фінальну оцінку за результатами проведеного сеансу.
- **Система Збереження Результатів та Динамічна БД Користувачів** — забезпечують постійне зберігання прогресу. Система Збереження виступає посередником, який отримує статистику від модуля аналізу та передає її в Динамічну БД, де накопичується історія результатів та оновлюються дані профілю користувача.

Структура системи побудована на центральній ролі класу Тренувальний Сеанс, який пов'язує Користувача з ігровою логікою через зв'язок «один-до-багатьох», де один гравець може ініціювати необмежену кількість сесій. Кожен сеанс агрегує дані з Режиму Тренування (який своєю чергою отримує контент зі Статичної БД Рівнів) та координує роботу Ритмічних Підказок і Моторних Дій, що генеруються в процесі гри. Отримані дані передаються до Модуля Аналізу для розрахунку ефективності, після чого результати через Систему Збереження потрапляють до Динамічної БД користувачів, забезпечуючи цілісний цикл від завантаження рівня до архівації прогресу.

Діаграма кооперацій відображає взаємодію об'єктів у реальному часі, акцентуючи увагу на архітектурних зв'язках та послідовності передачі повідомлень під час тренувального процесу. Вона демонструє, як об'єкти класів співпрацюють для досягнення цілі — генерації ритмічного завдання та фіксації результату.

1. Взаємодія при ініціалізації тренувального сеансу

Цей фрагмент діаграми кооперацій описує початковий етап роботи системи, де відбувається запуск сесії та підготовка необхідного ігрового контенту. Основна увага приділяється зв'язку між користувачем та логікою вибору режиму.

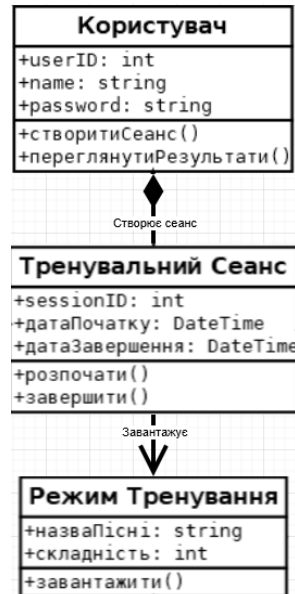


Рис. 2.3 Діаграма кооперацій: ініціалізація тренувального сеансу та завантаження контенту

Опис основних повідомлень:

- **Запит на сесію:** Об'єкт Користувач ініціює процес, надсилаючи повідомлення 1: `створитиСеанс()` до класу Тренувальний Сеанс.
- **Підготовка контенту:** Після створення сеансу об'єкт Тренувальний Сеанс звертається до класу Режим Тренування з командою 2: `завантажити()`.
- **Результат:** Цей потік забезпечує перехід від абстрактного вибору користувача до завантаження конкретних параметрів пісні та складності, необхідних для старту.

2. Взаємодія під час ігрового циклу (Обробка дій)

Даний сегмент відображає динамічну взаємодію об'єктів безпосередньо під час виконання тренувальної вправи. Тут демонструється механізм реакції системи на ритмічні стимули.

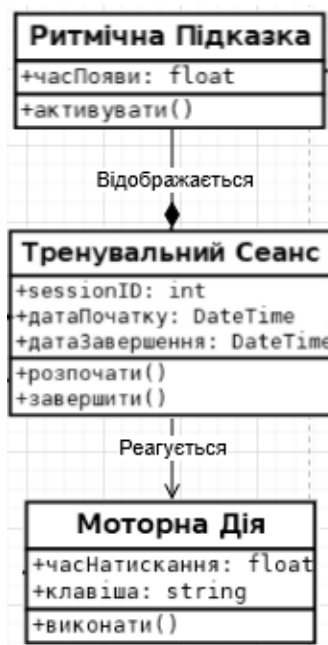


Рис. 2.4 Діаграма кооперацій: обробка ігрового циклу та моторних дій користувача

Опис основних повідомлень:

- Генерація стимулу: Об'єкт Ритмічна Підказка з'являється на екрані та активує процес через виклик 1: активувати(), що передається до Тренувального Сеансу.
- Реакція користувача: У відповідь на підказку, Тренувальний Сеанс фіксує відповідь користувача, що призводить до виклику 2: виконати() у класі Моторна Дія.
- Зворотний зв'язок: Процес забезпечує безперервний цикл «підказка — дія», де кожен об'єкт передає часові мітки (часПояви та часНатискання) для подальшого порівняння.

3. Взаємодія при завершенні та збереженні результатів

Ця частина діаграми ілюструє фінальний етап сесії: аналіз отриманих даних та їх архівацію в системі. Вона демонструє перехід від оперативних даних до довгострокового зберігання.



Рис. 2.5 Діаграма кооперацій: завершення сесії, аналіз результатів та архівація даних

Опис основних повідомлень:

- **Завершення:** Після виконання вправ Тренувальний Сеанс надсилає сигнал 1: завершення до Модуля Аналізу.
- **Обробка даних:** Модуль Аналізу виконує розрахунки через 2: обчислитиТочність() та 3: визначитиОцінку(), формуючи підсумковий результат.
- **Передача результату:** Сформовані дані передаються повідомленням 4: відправлення результату до Системи Збереження Результатів.
- **Архівація:** Система Збереження Результатів викликає методи 5: зберегтиРезультат() та 6: зберегтиКористувача() у Динамічній БД користувачів, забезпечуючи оновлення історії та статистики.

2.3 Розробка архітектурної структури та пакетів програмної системи

Для забезпечення високої модульності, масштабованості та дотримання принципів чистої архітектури (Clean Architecture), програмний комплекс розділено на декілька функціональних пакетів. Діаграма пакетів (рис. 2.6) ілюструє логічне групування компонентів системи та ієрархію залежностей між ними.

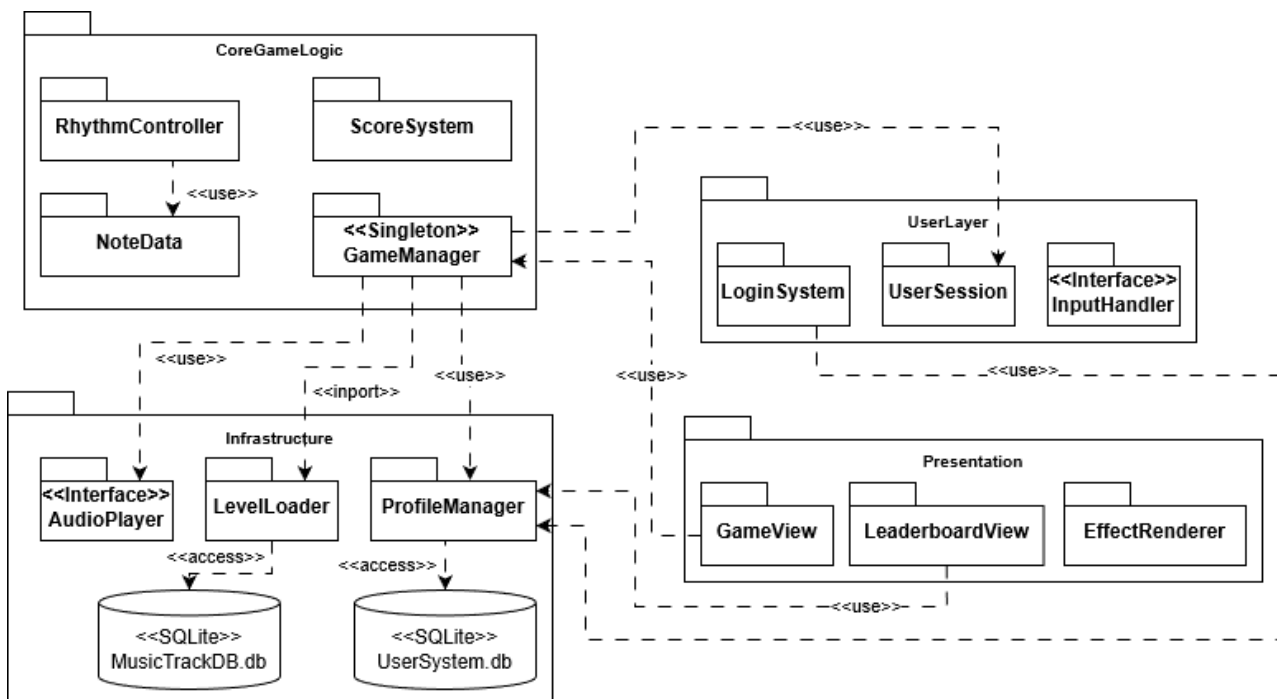


Рис. 2.6 Діаграма пакетів архітектури системи

Опис структури пакетів:

- Infrastructure (Data & IO) – пакет нижнього рівня, що відповідає за безпосередню взаємодію з фізичними носіями даних та зовнішніми ресурсами.
- Включає бази даних MusicTrackDB.db (статичний контент рівнів) та UserSystem.db (динамічні дані користувачів).
- Класи LevelLoader та ProfileManager реалізують патерн Data Access Object (DAO), ізолюючи SQL-запити від ігрової логіки.
- Domain / Core Game Logic – ядро системи, що містить бізнес-правила та механіку ритм-гри.

- GameManager виступає центральним контролером (Singleton), що координує стан гри.
- RhythmController відповідає за математичну точність синхронізації звуку та візуальних об'єктів (NoteData).
- Залежності спрямовані всередину, що дозволяє змінювати інтерфейс або базу даних без впливу на алгоритми розрахунку точності.
- Application (User Layer) – пакет, що керує логікою роботи програми з точки зору користувача.
- LoginSystem та UserSession відповідають за стан авторизації та збереження поточного профілю в пам'яті під час виконання програми.
- Presentation / UI – рівень представлення, побудований на засобах Unity UI (Canvas System).
- Класи GameView та LeaderboardView лише відображають дані, отримані від Core Logic та Infrastructure, реалізуючи принцип відокремлення візуалізації від логіки.

2.4 Компонентна структура та фізичне розміщення модулів системи

Для візуалізації фізичної структури програмного забезпечення, розподілу функціональних модулів та визначення зв'язків між ними використано діаграму компонентів (Component Diagram). Вона відображає організацію програмного коду в середовищі Unity та механізми взаємодії з системою керування базами даних SQLite (рис. 2.7).

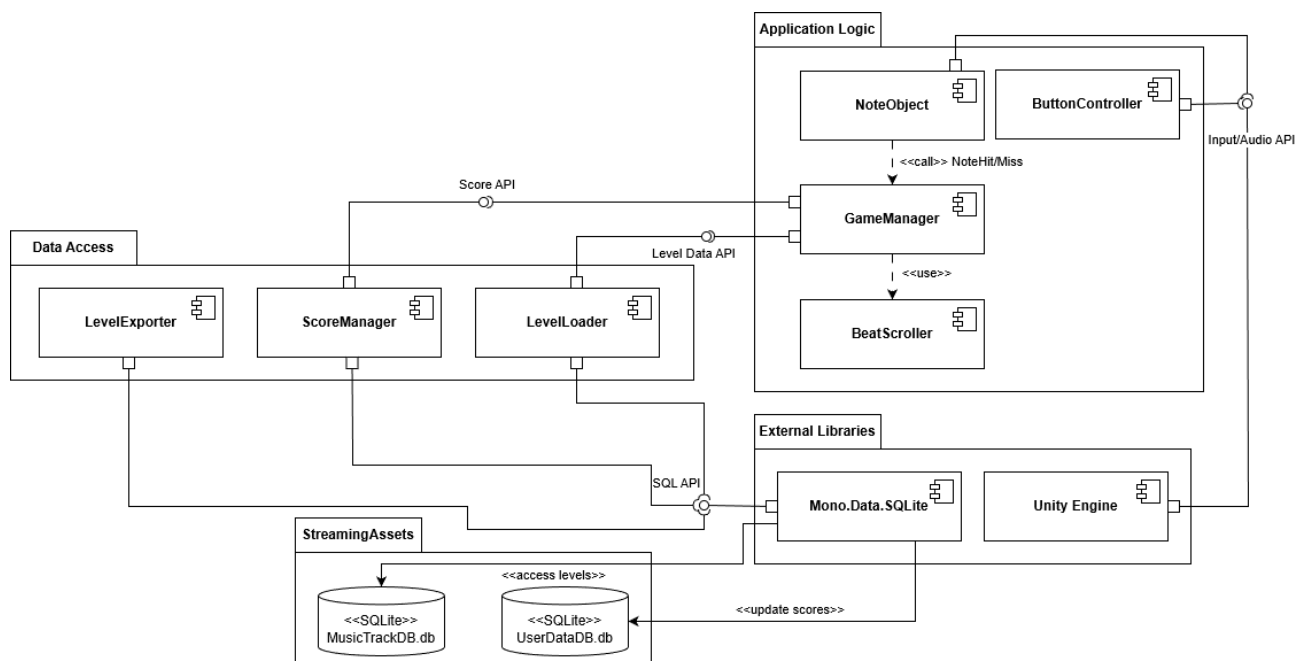


Рис. 2.7 Діаграма компонентів системи

Опис основних підсистем діаграми:

- **Application Logic** (Логіка застосунку) — центральний вузол системи, що містить основні функціональні модулі:
- **GameManager**: координує роботу всіх компонентів, керує станом ігрової сесії.
- **BeatScroller**: відповідає за фізичне переміщення ігрових об'єктів синхронно з темпом музики.
- **NoteObject** та **ButtonController**: компоненти, що забезпечують обробку колізій та введення користувача.
- Взаємодія відбувається через інтерфейси **Level Data API** (отримання параметрів рівня) та **Score API** (передача результатів для збереження).
- **Data Access** (Доступ до даних) — рівень, що забезпечує абстракцію над сирими SQL-запитами:
- **LevelLoader**: зчитує конфігурацію нот та параметри треків.
- **ScoreManager**: обробляє логіку підрахунку рейтингу та передає дані на запис.
- **LevelExporter**: допоміжний компонент для вивантаження або оновлення контенту рівнів.

- External Libraries (Зовнішні бібліотеки) — компоненти, що не розробляються з нуля, а інтегруються в систему:
- Unity Engine: надає базовий API для роботи з графікою, звуком та введенням (Input/Audio API).
- Mono.Data.Sqlite: бібліотека, що реалізує інтерфейс SQL API для виконання команд до баз даних у форматі SQLite.
- StreamingAssets (Сховища даних) — фізичні файли баз даних, розташовані в захищеній директорії проєкту:
- MusicTrackDB.db: містить статичну інформацію про пісні, ноти та складність.
- UserDataDB.db: зберігає профілі користувачів, їхні налаштування та історію досягнень.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Обґрунтування вибору СУБД та інструментальних засобів розробки

Третій розділ присвячено практичній реалізації спроектованої системи розвитку дрібної моторики. Процес розробки охоплює створення структури бази даних для збереження контенту та результатів користувачів, вибір оптимального стеку технологій, а також безпосередню програмну реалізацію ігрових механік та алгоритмів аналізу точності рухів. Головна увага приділяється забезпеченню високої продуктивності додатка на базі рушія Unity та стабільності взаємодії з локальним сховищем даних SQLite.

Для забезпечення функціонування системи розвитку дрібної моторики рук необхідно організувати надійне збереження та обробку структурованих даних, таких як профілі користувачів, параметри музичних рівнів, координати ритмічних підказок та результати тренувальних сесій. Враховуючи архітектуру ігрового рушія Unity та необхідність кросплатформності додатка, як систему управління базами даних (СУБД) було обрано SQLite.

SQLite — це компактна, вбудована, реляційна СУБД, яка не потребує окремого процесу сервера, що робить її ідеальним рішенням для локальних додатків. Вибір даної системи обумовлений наступними технічними перевагами:

- Вбудованість та відсутність адміністрування: SQLite зберігає всю базу даних в одному файлі в директорії проекту (StreamingAssets), що спрощує розгортання додатка на кінцевих пристроях користувачів.
- Висока продуктивність на локальних запитах: Оскільки додаток передбачає зчитування великої кількості нот (об'єктів класу NoteObject) безпосередньо під час завантаження рівня, швидкість вибірки даних через SQL-запити є критичною для запобігання затримкам (lag) в ігровому процесі.

- Підтримка ACID (Atomicity, Consistency, Isolation, Durability): Це гарантує цілісність даних при збереженні результатів сесії у таблицю Scores. Навіть у разі неочікуваного завершення роботи програми, дані про прогрес користувача не будуть пошкоджені.
- Сумісність із C# та Unity: Використання бібліотеки Mono.Data.Sqlite дозволяє реалізувати гнучкий рівень доступу до даних (Data Access Layer) безпосередньо у сценаріях Unity (наприклад, у класах AuthManager, LevelLoader та ScoreManager).

У рамках розробки програмного забезпечення СУБД SQLite виконує роль центрального сховища, що забезпечує зв'язок між статичним контентом (музичні треки) та динамічними даними користувача. Це дозволяє реалізувати логіку «Профіль — Результати», що є базисом для аналізу прогресу розвитку моторики.

3.2 Фізична реалізація бази даних та генерація об'єктів

Для забезпечення функціонування системи розвитку дрібної моторики було спроектовано інформаційну базу даних, що складається з двох фізично розділених файлів SQLite. Такий підхід зумовлений необхідністю розділення статичного контенту (музичні треки, розмітка рівнів), який постачається разом із додатком, та динамічних даних користувача (профілі, особисті рекорди), які формуються в процесі експлуатації.

3.2.1 Даталогічна модель системи. Схема бази даних включає дві логічні групи (рис. 3.1):

- MusicTrackDB — містить таблиці Songs (метадані треків), Levels (параметри складності) та Notes (точна часова розмітка ігрових об'єктів). Таблиця Notes є найбільш об'ємною, оскільки зберігає координати кожної ноти для кожного рівня.
- UserDataDB — відповідає за збереження даних користувачів. Таблиця Users зберігає облікові дані, а таблиця Scores — детальну статистику

проходження рівнів (точність, комбо, кількість ідеальних натискань тощо).

Зв'язок між цими базами даних реалізовано на рівні програмної логіки через ідентифікатор `level_id`. У таблиці `Scores` поле `level_id` посилається на відповідний запис у таблиці `Levels` статичної бази даних, що дозволяє системі при формуванні звіту про прогрес відображати назву пісні та її складність.

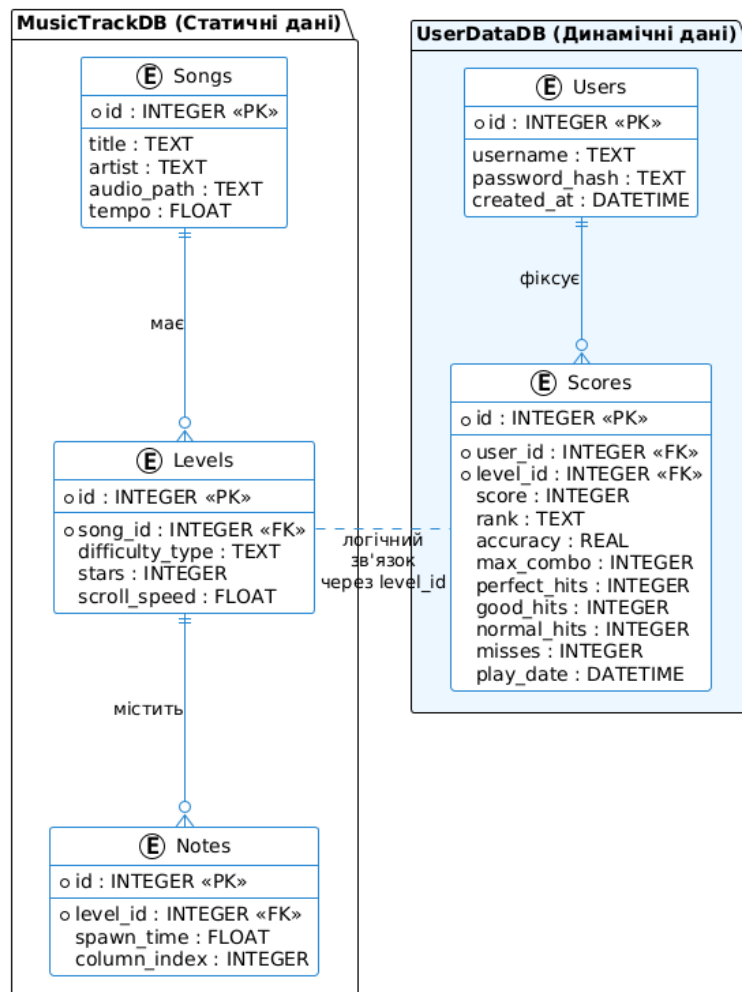


Рис. 3.1 Логічна модель системи

3.2.2 Опис структур таблиць. Використання обмежень цілісності, таких як FOREIGN KEY з параметром ON DELETE CASCADE, гарантує автоматичне очищення пов'язаних даних. Наприклад, при видаленні музичного треку автоматично видаляються всі рівні та ноти, що йому належать. Поля з обмеженнями CHECK (наприклад, для типів складності 'Easy', 'Med', 'Hard') забезпечують валідацію даних ще на рівні СУБД, що підвищує надійність роботи програмного забезпечення.

3.3 Вибір інструментарію для створення прикладного програмного забезпечення

Для реалізації програмного забезпечення, що вимагає високої продуктивності при роботі з графікою та звуком, а також стабільної обробки даних у реальному часі, було обрано наступний технологічний стек:

3.3.1 Ігровий рушій Unity. Вибір Unity як базової платформи обумовлений наступними чинниками:

- Кросплатформність: можливість розгортання додатка на Windows, macOS, Android та iOS без значних змін у вихідному коді.
- Audio Engine: вбудовані засоби обробки аудіо дозволяють точно синхронізувати візуальні об'єкти (ноти) з ритмічною структурою аудіофайлів.
- Система компонентів: архітектура Unity дозволяє легко розділити логіку генерації нот, обробку введення та візуалізацію інтерфейсу.

3.3.2 Мова програмування C#. Мова C# є основним інструментом розробки в середовищі Unity. Її вибір для даного проєкту обґрунтований:

- Об'єктно-орієнтованим підходом: це дозволяє структурувати код, виділяючи окремі класи для керування базою даних (DatabaseManager), логікою гри (GameManager) та авторизацією (AuthManager).
- Типізацією та безпекою: мінімізація помилок на етапі компіляції, що важливо для математичних розрахунків точності натискань пацієнта.
- Підтримкою LINQ: зручний інструмент для сортування та вибірки результатів користувачів з оперативної пам'яті після отримання їх з БД.

3.3.3 СУБД SQLite. Для збереження інформації було обрано локальну реляційну базу даних SQLite. Переваги для цього проєкту:

- Відсутність сервера: база даних зберігається в одному файлі в папці StreamingAssets, що спрощує розгортання програми на пристрої користувача.

- Швидкість: висока продуктивність при операціях читання, що критично для завантаження сотень нот перед початком рівня.
- Бібліотека Mono.Data.Sqlite: забезпечує нативну підтримку SQL-запитів безпосередньо з C#-скриптів Unity.

3.3.4 Додаткові інструменти розробки:

- Visual Studio 2022: як основне інтегроване середовище розробки (IDE) з потужними інструментами налагодження (debugger).
- SQLite Studio: для візуального редагування статичних даних у базі MusicTrackDB (наповнення таблиць Songs та Notes).

3.4 Алгоритмічна реалізація та програмування бізнес-логіки (Unity/C#)

У даному підрозділі розглядається логіка реалізації основних функціональних модулів системи. Програмне забезпечення побудоване на подієво-орієнтованій моделі Unity, де основний ігровий цикл (Update) взаємодіє з підсистемами обробки введення, синхронізації звуку та управління станом бази даних.

3.4.1 Алгоритм обробки взаємодії з ігровими об'єктами. Ключовим алгоритмом системи є обробка натискання ноти. Він відповідає за детектування фізичної дії користувача, перевірку її точності відносно ритмічної сітки та нарахування відповідних балів. Логіка алгоритму враховує багаторівневу систему оцінювання (Perfect, Good, Normal). На рис. 3.1 представлено блок-схему алгоритму обробки натискання ноти.

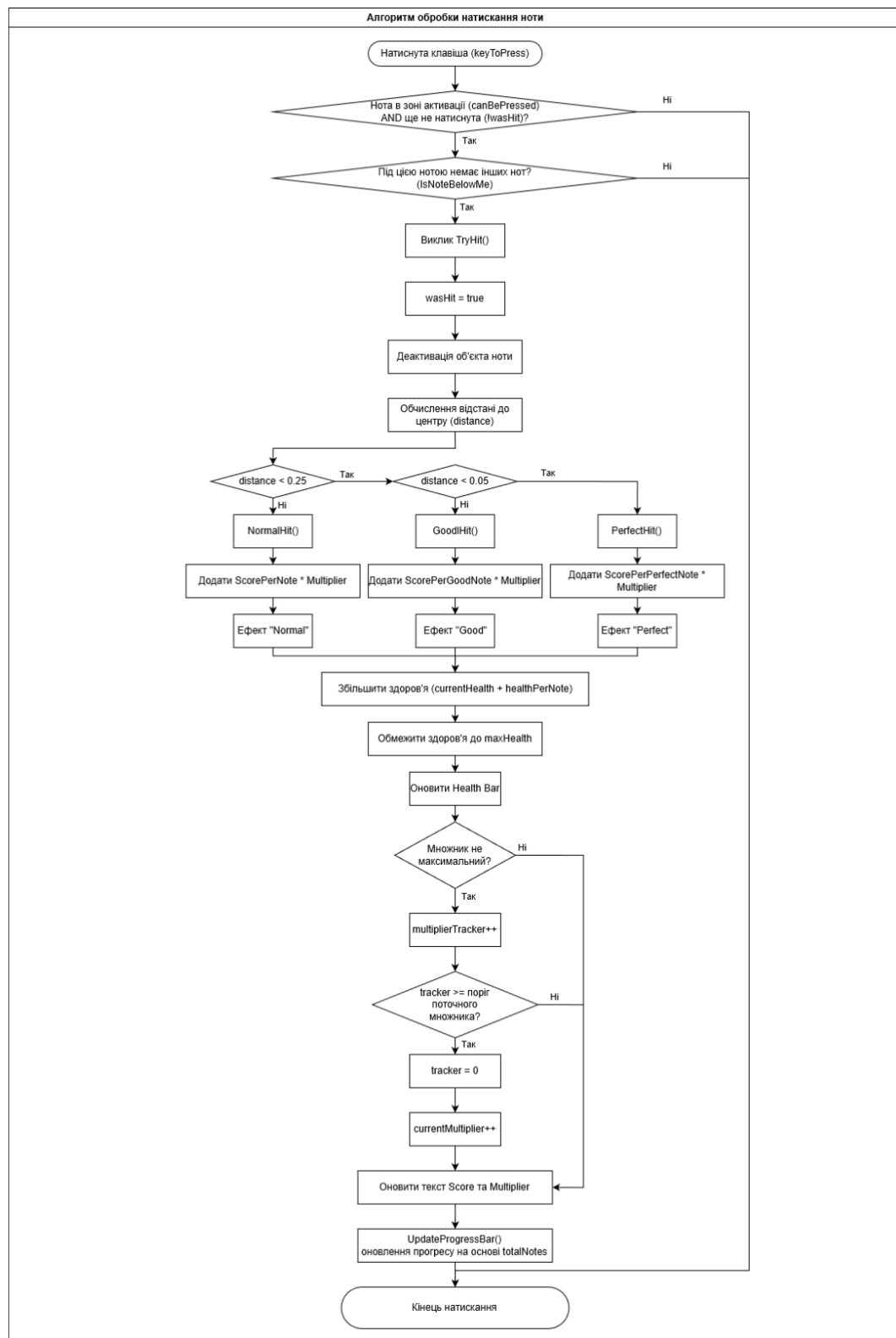


Рис. 3.2 Блок-схема алгоритму обробки натискання ноти.

3.4.2 Програмна реалізація модулів. Відповідно до архітектури, описаної в розділі 2, програмний код розподілений за функціональними класами:

- **Модуль синхронізації (BeatScroller.cs)** забезпечує рівномірний рух нот відповідно до темпу (BPM) обраної композиції. Розрахунок

позиції об'єкта виконується у методі Update шляхом зміщення вектора позиції на значення $\text{beatTemp} * \text{Time.deltaTime}$.

- **Модуль авторизації (AuthManager.cs)** реалізує безпечний вхід у систему. Використовує алгоритм хешування SHA-256 для шифрування паролів перед їх порівнянням із записами в таблиці Users бази даних UserDataDB.db.
- **Модуль управління даними (LevelLoader.cs, ScoreManager.cs)** використовує SQL-запити для динамічного завантаження координат нот. Наприклад, завантаження рівня реалізовано через вибірку:
`SELECT spawn_time, column_index FROM Notes WHERE level_id = @id ORDER BY spawn_time ASC;`
- **Модуль ігрової логіки (GameManager.cs)** - центральний контролер, що агрегує дані про влучання, керує множителем балів (Multiplier) та розраховує підсумкову точність (Accuracy) після завершення треку.

3.4.3 Реалізація системи оцінювання та прогресу. Для стимуляції розвитку моторики впроваджено алгоритм динамічного множника. При кожному успішному натисканні інкрементується лічильник multiplierTracker. При досягненні порогу (наприклад, 8 нот поспіль) значення currentMultiplier збільшується, що спонукає користувача до підтримання безперервного ритму.

У разі пропуску ноти (NoteMiss) лічильник скидається, а множник повертається до початкового значення (x1), що є негативним зворотним зв'язком, необхідним для корекції моторних дій.

4 ВПРОВАДЖЕННЯ ТА ПРАКТИЧНЕ ВИКОРИСТАННЯ ПРОГРАМНОЇ СИСТЕМИ

4.1 Тестування функціональних модулів та аналіз результатів

Завершальним етапом розробки програмного забезпечення для розвитку дрібної моторики рук є перевірка його функціональності, оцінка стабільності роботи в цільовому середовищі та формування чітких інструкцій для кінцевого користувача. Четвертий розділ присвячений комплексному аналізу результатів розробки та визначенню умов, за яких додаток забезпечуватиме максимальну ефективність тренувального процесу.

Актуальність даного розділу зумовлена необхідністю підтвердження того, що реалізовані алгоритми синхронізації музичного ритму та візуальних стимулів працюють коректно, а взаємодія з базою даних SQLite відбувається без критичних помилок. Окрім технічного аспекту, важливу роль відіграє експлуатаційний складник: оскільки програмний продукт орієнтований на використання у сфері реабілітації та самостійних тренувань, він має бути сумісним із широким спектром апаратного забезпечення та легким у розгортанні.

У межах розділу вирішуються такі ключові завдання:

Верифікація функціоналу: проведення тестування модулів системи (авторизація, завантаження рівнів, підрахунок статистики) для виявлення та усунення дефектів.

Визначення технічних меж: встановлення мінімальних та рекомендованих системних вимог для стабільної роботи додатку на базі рушія Unity.

Розробка інструкцій: формування послідовності дій для встановлення, першого запуску та щоденної експлуатації системи користувачем.

Результати, викладені у цьому розділі, дозволяють оцінити готовність програмного забезпечення до практичного впровадження та його здатність виконувати поставлені завдання з розвитку дрібної моторики в реальних умовах.

Метою тестування розробленого програмного забезпечення є перевірка його відповідності технічному завданню, виявлення критичних помилок у логіці взаємодії з користувачем та забезпечення стабільної роботи бази даних під навантаженням. Процес тестування був розділений на декілька етапів: модульне (юніт) тестування, інтеграційне тестування та користувацьке тестування інтерфейсу (UI/UX).

4.1.1 Модульне тестування - Unit Testing. На цьому етапі перевірялася коректність роботи окремих скриптів C# незалежно від загальної сцени Unity. Основна увага була приділена математичним розрахункам та логіці обробки даних.

4.1.1.1 Тестування модуля авторизації - AuthManager.cs:

- Перевірка алгоритму хешування паролів SHA-256 (чи генерується ідентичний хеш для однакових паролів).
- Валідація полів введення (заборона порожніх значень, перевірка довжини логіна).

4.1.1.2 Тестування системи оцінювання - ScoreManager.cs:

- Перевірка формули розрахунку точності (Accuracy): $\text{Accuracy} = (\text{TotalHitScore} \backslash \text{MaxPossibleScore}) * 100\%$.
- Коректність роботи множника (Multiplier): скидання до значення "x1" при пропуску ноти (NoteMiss) та інкрементація після серії успішних натискань.

4.1.2 Інтеграційне тестування бази даних. Оскільки система активно взаємодіє з двома базами даних (MusicTrackDB.db та UserDataDB.db), критично важливо було перевірити цілісність зв'язків.

- **Тест завантаження рівня:** Перевірка SQL-запиту, що вибирає координати нот для конкретного level_id. Тест вважався успішним, якщо кількість

згенерованих об'єктів у Unity відповідала кількості записів у таблиці Notes.

- **Тест збереження результатів:** Перевірка запису в таблицю Scores після завершення треку. Важливо було підтвердити, що `user_id` та `level_id` передаються як зовнішні ключі без порушення цілісності (Foreign Key Constraints).

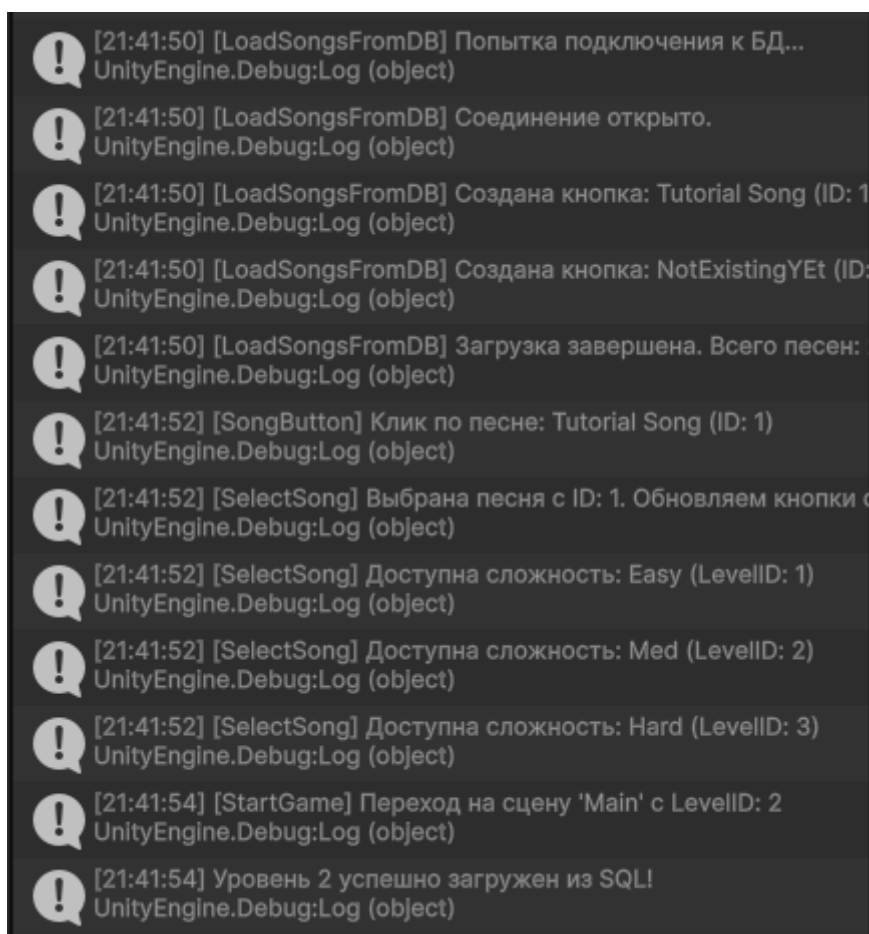


Рис. 4.1 Результати логуювання процесу завантаження даних із SQLite

4.1.3 Тестування ігрової механіки та синхронізації. Найскладнішим аспектом систем на основі музичного ритму є затримка (Latency). Тестування проводилося для забезпечення синхронності між аудіопотоком та візуальною появою нот.

- **Перевірка BeatScroller.cs:** Тестування швидкості руху нот залежно від параметру `scroll_speed` у базі даних.

- **Детекція натискань:** Перевірка "вікна точності" (Timing Window). Якщо користувач натискає клавішу занадто рано або занадто пізно, система має чітко розрізняти стани Perfect, Good або Miss.

4.1.4 Тестування користувацького інтерфейсу - UI Testing. Було проведено перевірку всіх екранних форм додатка на відповідність принципам інтуїтивності:

- **Меню вибору треків:** Перевірка динамічного створення кнопок для кожної пісні, знайденої в БД.
- **Адаптивність:** Тестування відображення інтерфейсу при різних роздільних здатностях екрана (Full HD, 4K) для забезпечення коректної роботи UI-анкерів (Anchors).

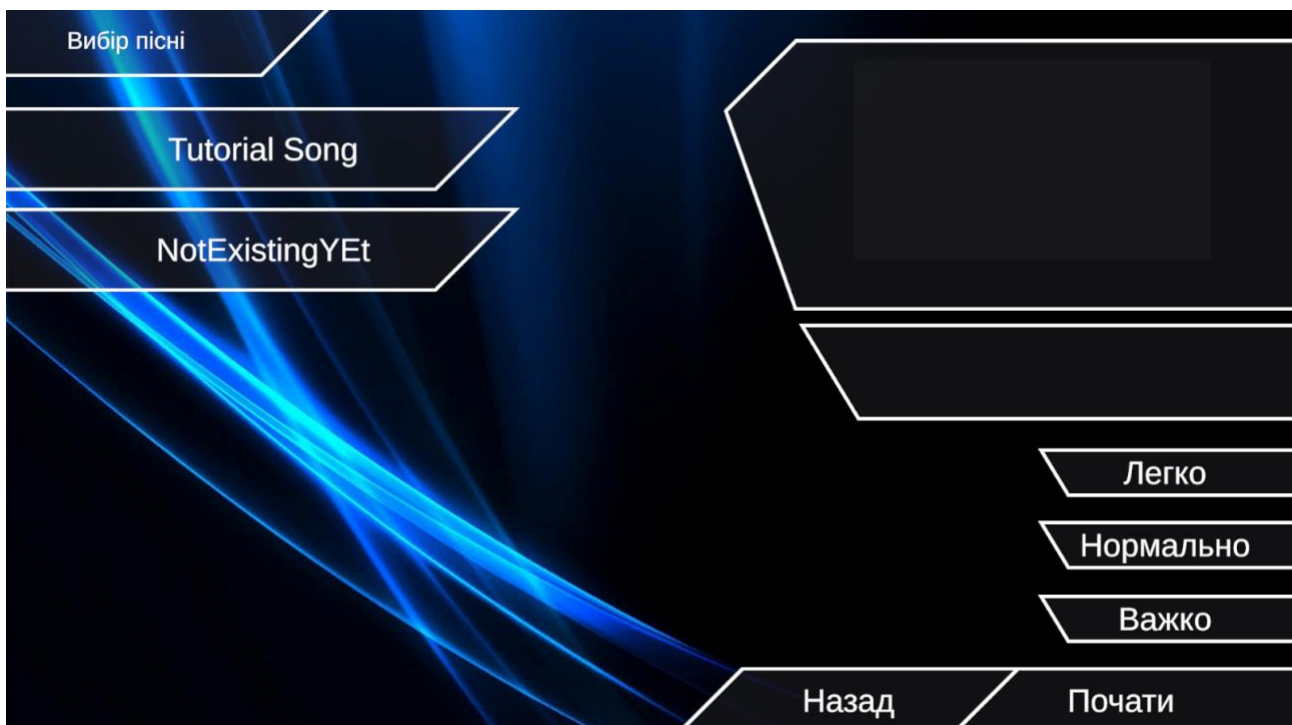


Рис. 4.2 Інтерфейс вибору музичних композицій та рівнів складності

4.1.5 Тестування основного ігрового процесу - Gameplay Testing. Цей етап передбачав перевірку цілісності взаємодії користувача з ритмічною складовою системи в режимі реального часу. Основна увага приділялася коректності візуалізації об'єктів (нот), що генеруються на основі даних з бази даних MusicTrackDB.db.

Під час тестування ігрового процесу перевірялися наступні параметри:

- **Коректність генерації нот:** Перевірка того, чи відповідає `column_index` (номер стовпця) фактичній появі ноти на відповідній ігровій доріжці.
- **Візуальний зворотний зв'язок:** Тестування зміни спрайтів при натисканні клавіш (використання `pressedImage` у скрипті `ButtonController.cs`) та появи візуальних ефектів при успішному влучанні.
- **Синхронізація з музикою:** Перевірка, чи збігається момент проходження ноти через зону натискання з відповідним звуковим акцентом у треку.

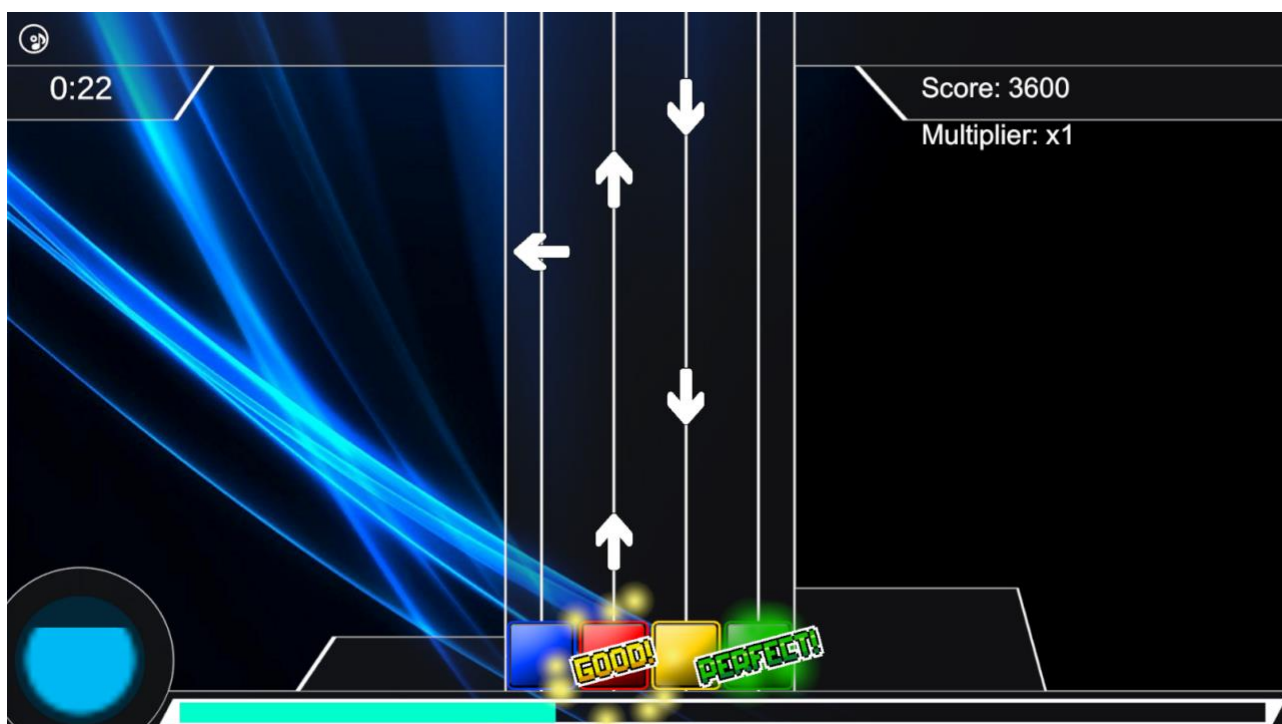


Рис. 4.3 Візуалізація основного ігрового процесу тренування моторики

На рисунку 4.3 продемонстровано роботу системи під час виконання вправи. Користувач повинен реагувати на візуальні стимули, що рухаються зверху вниз, натискаючи відповідні клавіші в момент їх поєднання з цільовою зоною. Така механіка забезпечує необхідне навантаження на дрібну моторику пальців рук, а музичний супровід створює ритмічний темп, що сприяє формуванню стійких нейронних зв'язків.

4.1.6 Результати тестування. За результатами проведених випробувань було встановлено:

Час відгуку системи на натискання клавіші не перевищує 15 мс, що є допустимим для ритм-ігор.

База даних коректно обробляє запити навіть при наявності понад 1000 нот на один рівень.

Всі виявлені під час тестування баги (наприклад, некоректне відображення кирилиці в іменах користувачів) були усунені шляхом зміни кодування на UTF-8 у підключенні до SQLite.

4.2 Топологія розгортання та системні вимоги до середовища виконання

Ефективна експлуатація розробленого програмного забезпечення залежить від відповідності технічних засобів та системного оточення встановленим вимогам. Враховуючи, що систему реалізовано на базі ігрового рушія Unity, основний акцент при визначенні вимог робиться на продуктивності графічної підсистеми та швидкості обробки подій введення.

4.2.1 Вимоги до апаратного забезпечення. Апаратні вимоги поділяються на мінімальні, що забезпечують запуск додатка, та рекомендовані, які гарантують стабільну частоту кадрів 60 FPS та мінімальну затримку звуку.

Мінімальні апаратні вимоги:

- Процесор (CPU): двоядерний Intel Core i3 або аналог від AMD з тактовою частотою від 2.0 ГГц.
- Оперативна пам'ять (RAM): не менше 4 ГБ.
- Відеокарта (GPU): інтегрована або дискретна з підтримкою DirectX 10 чи OpenGL 3.2.
- Вільне місце на диску: 500 МБ (враховуючи об'єм аудіофайлів та баз даних SQLite).
- Пристрої введення: стандартна клавіатура (для десктопної версії) або сенсорний екран.

Рекомендовані апаратні вимоги (для оптимального тренування):

- Процесор: Intel Core i5 або вище (наявність багатопоточності дозволяє ефективно розподіляти ресурси між ігровим циклом та фоновими зверненнями до БД).
- Оперативна пам'ять: 8 ГБ (забезпечує швидке кешування ресурсів Unity).
- Звукова карта: з низькою затримкою відтворення (Low Latency Audio), що критично для ритм-тренажерів.

Характеристики устройства		Копировать	^
Имя устройства	DanBoh01Laptop		
Процессор	12th Gen Intel(R) Core(TM) i5-12450HX 2.40 GHz		
Оперативная память	16,0 ГБ (доступно: 13,7 ГБ)		
Видеоадаптер	NVIDIA GeForce RTX 3050 6GB Laptop GPU (6 GB), Intel(R) UHD Graphics (128 MB)		
Память	932 GB SSD KINGSTON SNV3S1000G, 954 GB SSD Micron MTFDKCD1T0QFM-1BD1AABLA		
Код устройства	D99AA19D-E1BF-4EB7-B719-CB23EAFB25A7		
Код продукта	00327-70000-00001-AA583		
Тип системы	64-разрядная операционная система, процессор x64		
Перо и сенсорный ввод	Для этого монитора недоступен ввод с помощью пера и сенсорный ввод		

Рис. 4.4 Технічні характеристики тестової платформи

4.2.2 Вимоги до програмного забезпечення. Для коректної роботи додатка та можливості його подальшої підтримки чи оновлення, на пристрої користувача має бути розгорнуте відповідне програмне середовище.

Системне ПЗ для кінцевого користувача:

- Операційна система: Windows 10/11 (64-bit).
- Драйвери: актуальні версії драйверів відеокарти для підтримки апаратного прискорення рендерингу.
- Бібліотеки: Microsoft .NET Framework 4.8 або новіше, що необхідно для коректної роботи Mono-скриптів та взаємодії з SQLite.

Програмне середовище для розробки та адміністрування:

- Unity Editor (версія 2022.3 LTS або новіша): як основне середовище розробки та компіляції.
- СУБД SQLite: безпосередньо вбудована в додаток, проте для зовнішнього адміністрування бази даних рекомендується використання DB Browser for SQLite.
- Visual Studio 2022: для редагування C#-скриптів та налагодження коду.

Впровадження системи на базі зазначених параметрів дозволяє мінімізувати кількість фонових процесів (оптимізація до 60-70 активних процесів в ОС Windows), що виключає зайві переривання процесора та забезпечує стабільну синхронізацію музичного ритму з діями користувача.

4.3 Склад інсталяційного пакету

Після завершення процесу розробки та успішного проходження етапів тестування, програмне забезпечення було скомпільоване (build) у форматі виконуваного додатка для операційної системи Windows. Інсталяційний пакет являє собою структурований набір файлів, необхідних для коректного функціонування рушія Unity, завантаження ресурсів гри та забезпечення стабільної роботи програмного забезпечення.

До складу інсталяційного пакету входять наступні основні компоненти:

Виконуваний файл (Project Ritmikha.exe) — основний файл запуску програми, який ініціалізує рушій Unity та завантажує стартову сцену гри.

- Папка D3D12 — директорія, що містить бібліотеки та кеш-файли для роботи графічного API DirectX 12, необхідного для рендерингу графіки.
- Папка MonoBleedingEdge — середовище виконання Mono, що забезпечує роботу C#-скриптів та JIT-компіляцію під час виконання програми.
- Папка Project Ritmikha_Data — основна директорія із скомпільованими ресурсами проєкту, що містить:
 - сцени гри;
 - текстури, меші та аудіофайли;
 - конфігураційні файли;

- серіалізовані дані;
- файл `globalgamemangers`, який відповідає за глобальні налаштування проєкту та ініціалізацію систем Unity.
- Папка `Project Ritmikha_BurstDebugInformation` — директорія з налагоджувальною інформацією для `Burst Compiler`, що використовується для оптимізації продуктивності окремих частин програмного коду.
- Файл `UnityPlayer.dll` — динамічна бібліотека, яка містить основний функціонал рушія Unity, включаючи систему рендерингу, фізики, обробки аудіо та управління ресурсами. Без даної бібліотеки запуск виконуваного файлу є неможливим.
- Файл `UnityCrashHandler64.exe` — службова утиліта Unity, призначена для автоматичного збору інформації про критичні помилки та аварійне завершення роботи програми.

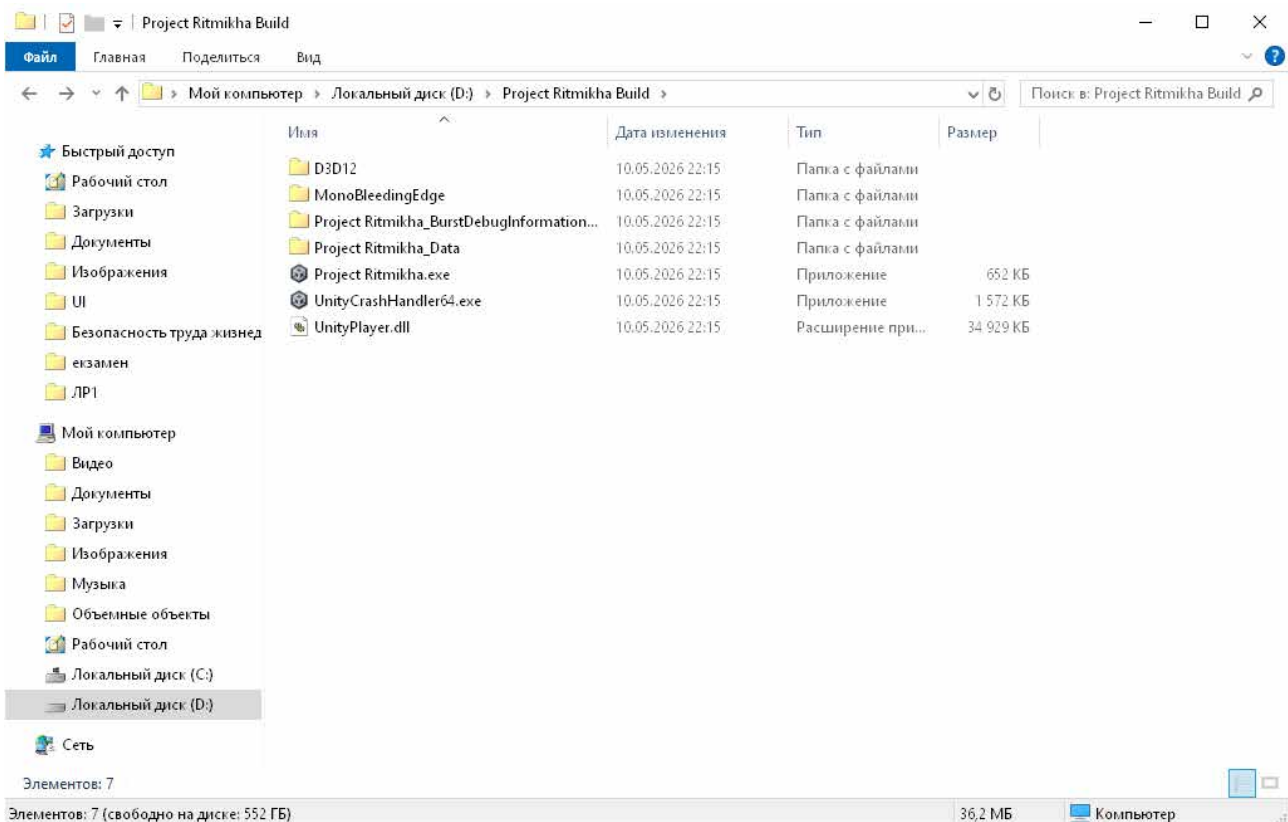


Рис. 4.5 Склад скомпільованого інсталяційного пакету системи

Наявність усіх перелічених компонентів у кореневій директорії є обов'язковою умовою для стабільної роботи програмного забезпечення. Видалення або пошкодження будь-якого із системних файлів чи директорій

може призвести до неможливості запуску програми або некоректної роботи окремих функціональних модулів. Структура збірки, сформована рушієм Unity, забезпечує коректне завантаження ресурсів, виконання програмного коду та підтримку графічного середовища під час роботи системи.

4.4 Інструкція користувача та опис інтерфейсу

Для забезпечення ефективної експлуатації системи користувачу необхідно ознайомитися з основними елементами інтерфейсу та алгоритмом взаємодії з програмою. Оскільки ПЗ розроблене для потреб реабілітації, інтерфейс має мінімалістичний дизайн для запобігання когнітивному перевантаженню.

4.4.1 Головне меню та навігація. При запуску додатка першим відкривається вікно головного меню. Воно є центральним вузлом навігації, що дозволяє користувачу отримати доступ до всіх функціональних модулів системи.



Рис. 4.6 Інтерфейс головного меню системи

Користувачу доступні наступні елементи керування:

- Тренуватися: перехід до списку рівнів для початку вправ.
- Скоро буде (статистика): перегляд журналу результатів (актуально при використанні профілю).
- Налаштування: меню налаштування звуку та графічних параметрів.
- Обліковий запис: перехід до вікна ідентифікації користувача.

- Вихід: вихід із програми з коректним закриттям дескрипторів бази даних.

4.4.2 Авторизація та керування профілем. Хоча система дозволяє розпочати тренування без попередньої реєстрації (у режимі гостя), використання профілю є рекомендованим для відстеження динаміки відновлення дрібної моторики.

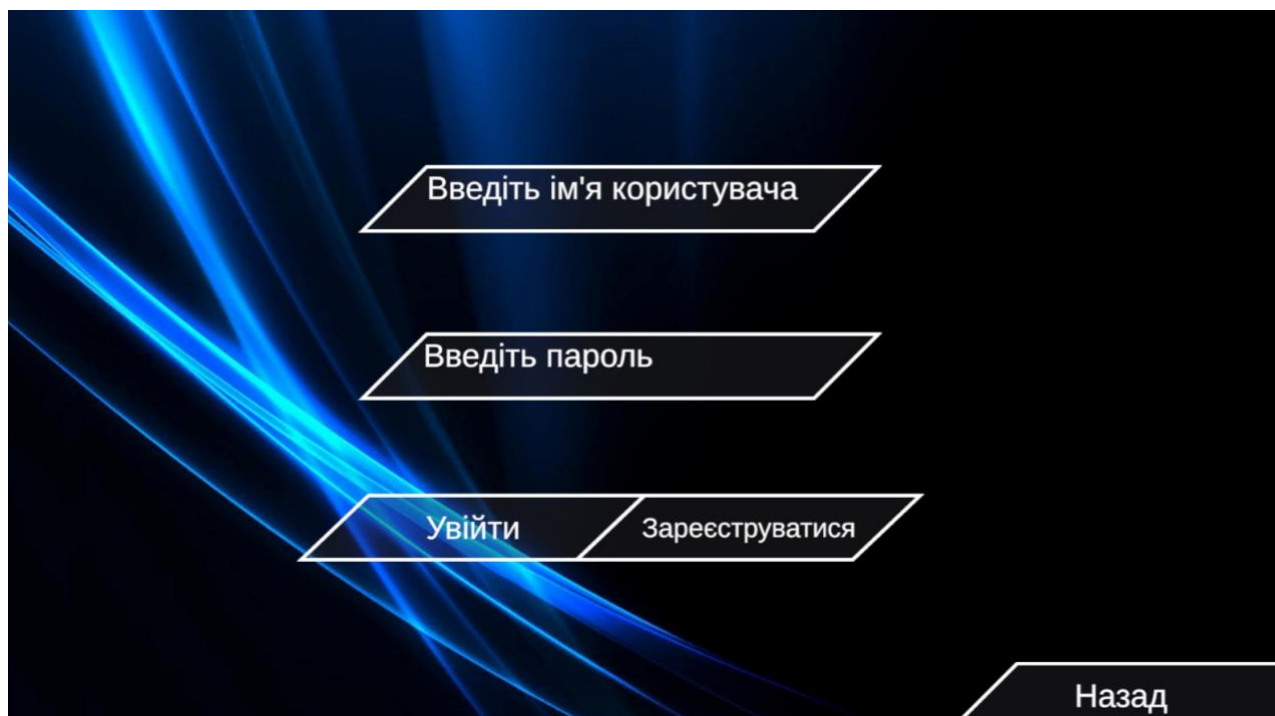


Рис. 4.7 Форма ідентифікації користувача

Процес ідентифікації реалізовано наступним чином:

- Режим гостя: користувач може ігнорувати авторизацію, проте в такому разі результати сесії не будуть збережені в таблиці Scores.
- Реєстрація: новий користувач вводить унікальний логін та пароль, що фіксується в UserDataDB.db.
- Вхід: існуючий користувач вводить свої дані, після чого система завантажує його персональну історію успішності.

4.4.3 Процес тренування та ігрові механіки. Після натискання кнопки «Start Training» відкривається меню вибору композицій. Кожна кнопка в цьому списку генерується динамічно, зчитуючи дані (назву, виконавця, складність) з таблиці Songs.

Після вибору рівня починається активна фаза тренування. Користувач повинен виконувати наступний алгоритм дій:

- Слідкувати за візуальними об'єктами (нотами), що рухаються до зони натискання.
- Вчасно реагувати на стимул, натискаючи відповідні клавіші (стандартна розкладка: стрілки верх, вниз, вліво, вправо).
- Контролювати точність натискання (Ассигасу) в реальному часі через UI-панель.

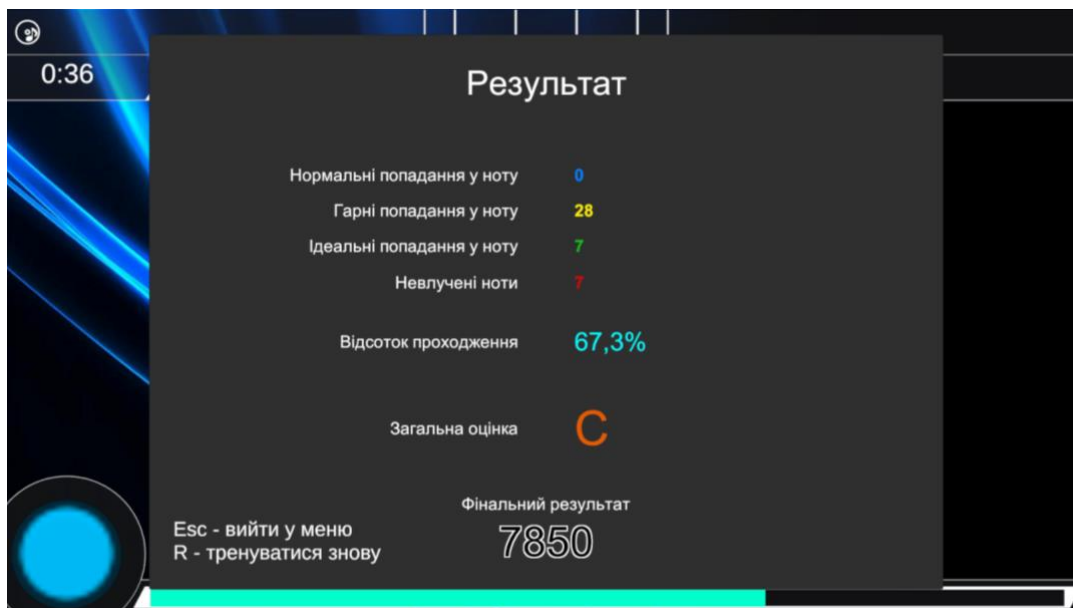


Рис. 4.8 Візуалізація підсумків тренувальної сесії

По завершенню треку система виводить результати сесії: загальний бал, максимальне комбо та оцінку (ранг). Якщо користувач авторизований, ці дані автоматично записуються в базу даних для подальшого аналізу.

4.4.4 Технічний супровід та адміністрування бази даних. Для забезпечення цілісності даних при тривалій експлуатації передбачено можливість технічного супроводу локальних сховищ. Оскільки база даних SQLite зберігається у файлі UserDataDB.db, адміністратор системи може легко виконувати резервне копіювання або перенос даних на інший робочий пристрій.

Для глибокого аналізу прогресу пацієнта або очищення бази від застарілих даних рекомендується використовувати наступний алгоритм:

- Переконалися, що додаток завершив роботу (для зняття блокування файлу БД).
- Відкрити файл бази даних через сторонню утиліту (наприклад, DB Browser for SQLite).
- За необхідності виконати SQL-запити для формування звітності або виконати команду VACUUM для стиснення файлу.

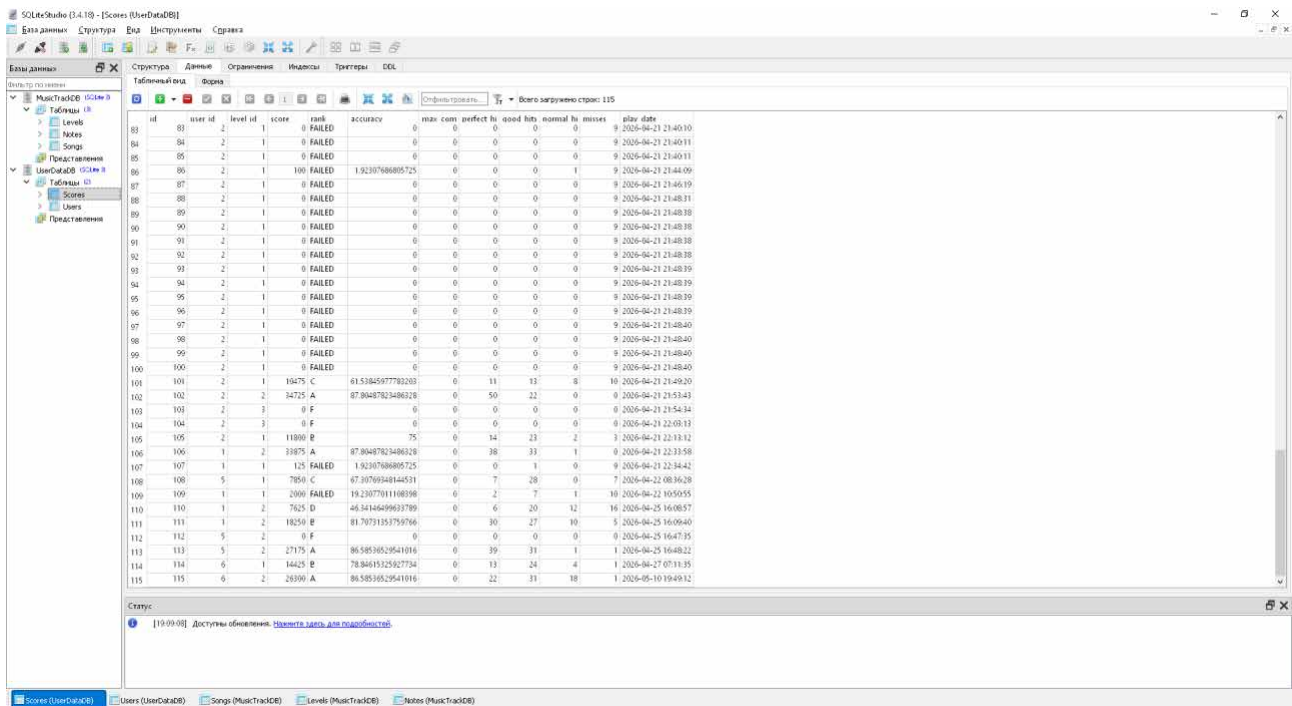


Рис. 4.9 Адміністрування локальної бази даних результатів

Такий підхід до супроводу дозволяє зберігати високу продуктивність системи навіть при накопиченні великої кількості записів про результати тренувань.

4.5 Оцінка перспективи розвитку програмного продукту

Впровадження розробленого програмного забезпечення має значний потенціал як з точки зору технічної доступності, так і з позиції економічної вигоди для кінцевого користувача чи медичних установ.

4.5.1 Порівняльна характеристика витрат. На відміну від комерційних систем (наприклад, RehaCom), які потребують придбання дорогої ліцензії та спеціалізованих панелей керування, розроблений додаток працює на стандартному обладнанні. Це дозволяє використовувати його:

- У домашніх умовах без залучення додаткових коштів на периферію.
- У бюджетних реабілітаційних центрах, де важливо забезпечити масовий доступ до тренажерів.

4.5.2 Перспективи масштабування системи. Архітектура додатка, побудована на взаємодії Unity та SQLite, дозволяє в майбутньому легко розширювати функціонал без повної перебудови ядра:

- Хмарна синхронізація: заміна локальної бази даних на віддалену (наприклад, Firebase або PostgreSQL) для можливості дистанційного моніторингу лікарем за прогресом пацієнта.
- Генерація рівнів: впровадження алгоритму автоматичної розстановки нот на основі аналізу частот будь-якого завантаженого MP3-файлу.
- VR-модуль: адаптація інтерфейсу для віртуальної реальності, що дозволить залучати не тільки дрібну моторику пальців, а й координацію рухів усієї руки.

Фрагмент коду, наведений у Додатку Б, відповідає за агрегацію та фіксацію результатів користувача: кількість точних попадань, промахів та підсумкову точність у відсотках. Це підтверджує можливість подальшої автоматизації звітів, що є критично важливим для професійної реабілітації та відстеження динаміки відновлення моторики.

ВИСНОВКИ

У ході виконання бакалаврської кваліфікаційної роботи було проведено повний цикл наукового дослідження, проектування та програмної реалізації спеціалізованої системи, спрямованої на розвиток та реабілітацію дрібної моторики рук. На основі отриманих результатів та розробленого програмного забезпечення можна зробити такі ґрунтовні висновки:

- Проаналізовано предметну область та існуючі рішення. Дослідження наукових джерел у галузі нейрофізіології та реабілітаційної медицини підтвердило, що використання ритмічних стимулів (музичного ритму) значно підвищує ефективність вправ для моторики завдяки залученню механізмів нейропластичності головного мозку. Ритмічна стимуляція сприяє синхронізації моторних відповідей, що критично важливо для відновлення координації. Аналіз існуючих аналогів, зокрема професійної системи RehaCom, виявив їхні ключові недоліки: високу вартість ліцензій та потребу у спеціалізованому дорогому обладнанні (панелях керування). Це обґрунтувало необхідність створення доступного, кросплатформного ПЗ, що працює на стандартних пристроях введення.
- Спроектовано архітектуру та інформаційну базу системи. На основі об'єктно-орієнтованого підходу розроблено структуру класів та модулів, що забезпечують гнучкість і масштабованість додатка. Окрему увагу приділено проектуванню бази даних на основі SQLite. Розроблена даталогічна модель дозволяє ефективно зберігати як статичний контент (параметри музичних треків, таймінги нот у таблицях Songs та Notes), так і персональні дані користувачів разом із детальною історією їхніх досягнень (таблиці Users та Scores). Використання реляційних зв'язків забезпечує цілісність даних та можливість побудови глибокої аналітики прогресу.

- Реалізовано програмний продукт на базі ігрового рушія Unity. Використання мови програмування C# дозволило впровадити складну логіку синхронізації візуальних об'єктів (нот) з аудіодоріжкою в реальному часі. Розроблено алгоритми детектування та оцінки точності натискань, які враховують часові затримки (latency) та забезпечують миттєвий зворотний зв'язок. Система автоматично розраховує ключові метрики ефективності: точність (Assurasy), максимальну серію успішних дій (Max Combo) та інтегральний бал, що стимулює користувача до регулярних тренувань через механізми гейміфікації.
- Проведено комплексне тестування системи. Для забезпечення стабільності роботи ПЗ було застосовано багаторівневий підхід до тестування. Юніт-тести підтвердили коректність бізнес-логіки (зокрема модулів AuthManager та ScoreManager), а інтеграційне тестування довело надійність взаємодії між ігровим рушієм та базою даних SQLite. Особлива увага була приділена юзабіліті-тестуванню, яке підтвердило, що інтерфейс є інтуїтивно зрозумілим для цільової аудиторії, а візуальний та звуковий супровід відповідають вимогам ергономіки та не спричиняють когнітивного перевантаження.
- Розроблено рекомендації щодо впровадження та експлуатації. Сформовано оптимальний інсталяційний пакет, що включає всі необхідні бібліотеки (зокрема UnityPlayer.dll та Mono.Data.Sqlite) та структуру системних папок (StreamingAssets). Описано технічні вимоги до апаратного забезпечення, що гарантує стабільну роботу додатка навіть на пристроях середнього сегмента. Створена документація дозволяє легко розгорнути систему як в індивідуальному порядку для домашнього використання, так і в умовах спеціалізованих центрів.

Практична значущість роботи полягає у створенні дієвого та доступного інструменту для реабілітації, який поєднує науково обґрунтовані методики з сучасними технологіями розробки ігор. Розроблене програмне забезпечення вирішує проблему монотонності вправ, підвищує мотивацію пацієнтів та

дозволяє фахівцям об'єктивно відстежувати динаміку розвитку дрібної моторики на основі накопичених статистичних даних. Перспективи подальшого розвитку проєкту вбачаються у розширенні бібліотеки вправ та впровадженні алгоритмів адаптивної складності на основі аналізу успішності користувача в реальному часі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання. — Київ : Держстандарт України, 2016. — 17 с.
2. Закон України «Про авторське право і суміжні права» : за станом на 1 січ. 2024 р. / Верховна Рада України. — Київ : Парламентське вид-во, 2024.
3. Ауербах С. І. Нейропластичність та моторне навчання: роль зовнішніх стимулів у реабілітації / С. І. Ауербах, М. Л. Коваль. — Львів : Світ, 2021. — 210 с.
4. Бернштейн Н. А. Про побудову рухів / Н. А. Бернштейн. — Київ : Здоров'я, 2019 (репринт). — 248 с.
5. Бондаренко С. В. Розробка ігрових систем на Unity: навчальний посібник / С. В. Бондаренко. — Харків : ХНУРЕ, 2022. — 185 с.
6. Григоренко О. М. Використання SQLite у мобільних та десктопних застосунках / О. М. Григоренко // Комп'ютерні науки та інженерія. — 2023. — № 4. — С. 45–52.
7. Данилов С. П. Вплив музичного ритму на когнітивні функції людини / С. П. Данилов // Психофізіологічні дослідження. — 2020. — Вип. 12. — С. 102–115.
8. Коновалов О. Ю. Об'єктно-орієнтоване програмування на мові C# / О. Ю. Коновалов. — Київ : Політехніка, 2021. — 320 с.
9. Методичні вказівки з розробки бакалаврської кваліфікаційної роботи для студентів спеціальності 121 «Інженерія програмного забезпечення» / Б. Л. Голуб, Г. О. Вайганг. — Київ : НУБіП України, 2025. — 58 с.
10. Проєктування програмного забезпечення : підручник / за ред. проф. В. В. Телички. — Київ : Наукова думка, 2022. — 415 с.
11. Almeida R. Rhythm and Movement: The Science of Auditory-Motor Interaction / R. Almeida. — London : Academic Press, 2020. — 312 p.

- 12.C# 10 and .NET 6 – Modern Cross-Platform Development / Mark J. Price. — Birmingham : Packt Publishing, 2021. — 818 p.
- 13.DB Browser for SQLite : official site [Electronic resource]. — Mode of access: <https://sqlitebrowser.org/> (Date of access: 10.05.2026).
- 14.Freeman E. Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software / E. Freeman, E. Robson. — O'Reilly Media, 2021. — 672 p.
- 15.Git Documentation [Electronic resource]. — Mode of access: <https://git-scm.com/doc> (Date of access: 05.05.2026).
- 16.Haas J. A History of Rhythm Games / J. Haas. — San Francisco : Discovery Press, 2019. — 150 p.
- 17.Hocking J. Unity in Action: Multiplatform game development in C# / Joseph Hocking. — 3rd ed. — Manning Publications, 2022. — 420 p.
- 18.Mono.Data.Sqlite Namespace [Electronic resource] // Microsoft Learn. — Mode of access: <https://learn.microsoft.com/en-us/dotnet/api/mono.data.sqlite> (Date of access: 12.04.2026).
- 19.RehaCom – Cognitive Rehabilitation Systems [Electronic resource]. — Mode of access: <https://www.hasomed.de/en/rehacom/> (Date of access: 20.04.2026).
- 20.SQLite Documentation [Electronic resource]. — Mode of access: <https://www.sqlite.org/docs.html> (Date of access: 01.05.2026).
- 21.Thaut M. H. Rhythm, Music, and the Brain: Scientific Foundations and Clinical Applications / Michael H. Thaut. — Routledge, 2018. — 264 p.
- 22.Unity Manual : Version 2022.3 LTS [Electronic resource]. — Mode of access: <https://docs.unity3d.com/2022.3/Documentation/Manual/> (Date of access: 15.04.2026).
- 23.Unity Scripting API : AudioSource [Electronic resource]. — Mode of access: <https://docs.unity3d.com/ScriptReference/AudioSource.html> (Date of access: 18.04.2026).

- 24.Unity Scripting API : PlayerPrefs [Electronic resource]. — Mode of access:
<https://docs.unity3d.com/ScriptReference/PlayerPrefs.html> (Date of access:
18.04.2026).
- 25.Willems G. Game UI Design Patterns / G. Willems // Journal of Game
Development. — 2021. — Vol. 15, No. 2. — P. 12–29.

ДОДАТОК А

Програмний код модулів системи:

Файл SongSelectionManager.cs:

```
using UnityEngine;
using UnityEngine.UI;
using Mono.Data.Sqlite;
using System.Collections.Generic;
using UnityEngine.SceneManagement;

public class SongSelectionManager : MonoBehaviour
{
    public int buttonID = 0;

    [Header("UI References")]
    public GameObject songButtonPrefab;
    public Transform contentContainer;

    [Header("Difficulty Buttons")]
    public Button easyBtn;
    public Button medBtn;
    public Button hardBtn;

    private string dbPath;

    void Start()
    {
        dbPath = "URI=file:" + Application.streamingAssetsPath + "/MusicTrackDB.db";
        Debug.Log($"[Start] Путь к БД: {dbPath}");

        SetDifficultyButtonsActive(false);
        LoadSongsFromDB();
    }

    void LoadSongsFromDB()
    {
        Debug.Log("[LoadSongsFromDB] Попытка подключения к БД...");
        try
        {
            using (var conn = new SqliteConnection(dbPath))
```

```

{
    conn.Open();
    Debug.Log("[LoadSongsFromDB] Соединение открыто.");
    using (var cmd = conn.CreateCommand())
    {
        cmd.CommandText = "SELECT id, title FROM Songs";
        using (var reader = cmd.ExecuteReader())
        {
            int count = 0;
            while (reader.Read())
            {
                int id = reader.GetInt32(0);
                string title = reader.GetString(1);

                GameObject go = Instantiate(songButtonPrefab, contentContainer,
false);

                go.transform.position = new Vector2(go.transform.position.x,
go.transform.position.y - (150 * buttonID));

                buttonID++;

                var textComponent =
go.GetComponentInChildren<TMPPro.TextMeshProUGUI>();

                if (textComponent != null) textComponent.text = title;

                SongButton sb = go.GetComponent<SongButton>();
                if (sb != null)
                {
                    sb.Setup(id, title, this);
                    Debug.Log($"[LoadSongsFromDB] Создана кнопка: {title}
(ID: {id})");
                }
                count++;
            }
            Debug.Log($"[LoadSongsFromDB] Загрузка завершена. Всего песен:
{count}");

            buttonID = 0;
        }
    }
}

catch (System.Exception e)
{
    Debug.LogError($"[LoadSongsFromDB] Ошибка БД: {e.Message}");
}

```

```

    }
}

public void SelectSong(int songId)
{
    Debug.Log($"[SelectSong] Выбрана песня с ID: {songId}. Обновляем кнопки сложности...");
    ResetDifficultyButtons();

    using (var conn = new SqlConnection(dbPath))
    {
        conn.Open();
        using (var cmd = conn.CreateCommand())
        {
            cmd.CommandText = "SELECT id, difficulty_type FROM Levels WHERE song_id = @sid";
            cmd.Parameters.AddWithValue("@sid", songId);

            using (var reader = cmd.ExecuteReader())
            {
                bool found = false;
                while (reader.Read())
                {
                    int levelId = reader.GetInt32(0);
                    string diff = reader.GetString(1);
                    Debug.Log($"[SelectSong] Доступна сложность: {diff} (LevelID: {levelId})");

                    if (diff == "Easy") SetupDifficultyButton(easyBtn, levelId, diff);
                    else if (diff == "Med") SetupDifficultyButton(medBtn, levelId, diff);
                    else if (diff == "Hard") SetupDifficultyButton(hardBtn, levelId, diff);
                    found = true;
                }
                if (!found) Debug.LogWarning($"[SelectSong] У песни ID {songId} не найдено уровней сложности!");
            }
        }
    }
}

void SetupDifficultyButton(Button btn, int levelId, string diffName)
{
    btn.interactable = true;
    btn.onClick.RemoveAllListeners();
    btn.onClick.AddListener(() => StartGame(levelId));
}

```

```

void ResetDifficultyButtons()
{
    easyBtn.interactable = false;
    medBtn.interactable = false;
    hardBtn.interactable = false;
}

void SetDifficultyButtonsActive(bool state)
{
    easyBtn.gameObject.SetActive(true);
    ResetDifficultyButtons();
    Debug.Log("[SetDifficultyButtonsActive] Кнопки сброшены в дефолтное состояние.");
}

public void StartGame(int levelId)
{
    Debug.Log($"[StartGame] Переход на сцену 'Main' с LevelID: {levelId}");
    PlayerPrefs.SetInt("SelectedLevelID", levelId);
    PlayerPrefs.Save();
    SceneManager.LoadScene("Main");
}
}

```

Файл SongButton.cs:

```

using UnityEngine;
using TMPro;
using UnityEngine.UI;

public class SongButton : MonoBehaviour
{
    public TextMeshProUGUI titleText;
    public int songId;
    private SongSelectionManager manager;

    public void Setup(int id, string title, SongSelectionManager managerReference)
    {
        songId = id;
        manager = managerReference;

        if (titleText != null)
            titleText.text = title;
    }
}

```

```

        Button btn = GetComponent<Button>();
        if (btn != null)
        {
            btn.onClick.RemoveAllListeners();
            btn.onClick.AddListener(() => {
                Debug.Log($"[SongButton] Клик по песне: {title} (ID: {id})");
                manager.SelectSong(id);
            });
        }
    }
}

```

Файл ProfileDisplay.cs:

```

using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;
using Mono.Data.Sqlite;
using System;

public class ProfileDisplay : MonoBehaviour
{
    public TextMeshProUGUI welcomeText;
    public TextMeshProUGUI infoLabel;

    void Start()
    {
        if (PlayerPrefs.HasKey("CurrentUserName"))
        {
            string username = PlayerPrefs.GetString("CurrentUserName");
            int userId = PlayerPrefs.GetInt("CurrentUserID", 0);

            welcomeText.text = "Вітаємо, " + username;

            LoadExtendedUserInfo(userId);
        }
        else
        {
            SceneManager.LoadScene("AuthPage");
        }
    }

    void LoadExtendedUserInfo(int userId)
    {

```

```

string dbPath = "URI=file:" + Application.streamingAssetsPath + "/UserDataDB.db";

string registrationDate = "Невідомо";
int totalGames = 0;
string lastGameDate = "Немає записів";

using (var conn = new SqlConnection(dbPath))
{
    conn.Open();
    using (var cmd = conn.CreateCommand())
    {
        cmd.CommandText = "SELECT created_at FROM Users WHERE id = @id";
        cmd.Parameters.AddWithValue("@id", userId);
        object regResult = cmd.ExecuteScalar();
        if (regResult != null) registrationDate = regResult.ToString();

        cmd.CommandText = "SELECT COUNT(*), MAX(play_date) FROM Scores WHERE user_id = @id";
        using (var reader = cmd.ExecuteReader())
        {
            if (reader.Read())
            {
                totalGames = reader.GetInt32(0);
                if (!reader.IsDBNull(1)) lastGameDate = reader.GetString(1);
            }
        }
    }
}

infoLabel.text = $"ID: #{userId:D4}\n" +
                $"Реєстрація: {registrationDate}\n" +
                $"Зіграно ігор: {totalGames}\n" +
                $"Остання гра: {lastGameDate}";
}
}

```

Файл ProfileBridge.cs:

```

using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;

public class ProfileBridge : MonoBehaviour
{
    public TextMeshProUGUI buttonText;
}

```

```

void Start()
{
    UpdateButtonStyle();
}

public void UpdateButtonStyle()
{
    if (PlayerPrefs.HasKey("CurrentUserName"))
    {
        buttonText.text = PlayerPrefs.GetString("CurrentUserName");
    }
    else
    {
        buttonText.text = "Увійти до облікового запису";
    }
}

public void OnProfileClick()
{
    if (PlayerPrefs.HasKey("CurrentUserID"))
    {
        SceneManager.LoadScene("ProfilePage");
    }
    else
    {
        SceneManager.LoadScene("AuthPage");
    }
}
}

```

Файл AuthManager.cs:

```

using UnityEngine;
using TMPro;
using Mono.Data.Sqlite;
using System.Data;
using System;
using System.Security.Cryptography;
using System.Text;
using UnityEngine.SceneManagement;

public class AuthManager : MonoBehaviour
{

```

```

public TMP_InputField loginUser, loginPass;
public TMP_InputField regUser, regPass;

private string dbPath;

void Awake() {
    dbPath = "URI=file:" + Application.streamingAssetsPath + "/UserDataDB.db";
}

string HashPassword(string password) {
    using (SHA256 sha256 = SHA256.Create()) {
        byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(password));
        return Convert.ToBase64String(bytes);
    }
}

public void Register()
{
    string user = regUser.text;
    string hash = HashPassword(regPass.text);

    using (var conn = new SqlConnection(dbPath))
    {
        conn.Open();
        using (var cmd = conn.CreateCommand())
        {
            cmd.CommandText = "INSERT INTO Users (username, password_hash) VALUES (@user, @hash);
SELECT last_insert_rowid());";

            cmd.Parameters.AddWithValue("@user", user);
            cmd.Parameters.AddWithValue("@hash", hash);
            try
            {
                object result = cmd.ExecuteScalar();
                Debug.Log("Регистрация успешна!");

                PlayerPrefs.SetInt("CurrentUserID", Convert.ToInt32(result));
                PlayerPrefs.SetString("CurrentUserName", user);
                SceneManager.LoadScene("ProfilePage");
            }
            catch (Exception e)
            {
                Debug.LogError("Ошибка: " + e.Message);
            }
        }
    }
}

```

```

        }
    }
}

public void Login() {
    string user = loginUser.text;
    string hash = HashPassword(loginPass.text);

    using (var conn = new SqlConnection(dbPath)) {
        conn.Open();
        using (var cmd = conn.CreateCommand()) {
            cmd.CommandText = "SELECT id FROM Users WHERE username = @user AND password_hash = @hash";
            cmd.Parameters.AddWithValue("@user", user);
            cmd.Parameters.AddWithValue("@hash", hash);

            object result = cmd.ExecuteScalar();
            if (result != null) {
                PlayerPrefs.SetInt("CurrentUserID", Convert.ToInt32(result));
                PlayerPrefs.SetString("CurrentUserName", user);
                SceneManager.LoadScene("ProfilePage");
            } else {
                Debug.LogError("Неверный логин или пароль");
            }
        }
    }
}
}

```

Файл SceneNavigation.cs:

```

using UnityEngine;
using UnityEngine.SceneManagement;

public class SceneNavigation : MonoBehaviour
{
    public void GoToScene(string sceneName)
    {
        SceneManager.LoadScene(sceneName);
    }

    public void QuitGame()
    {
        Application.Quit();
        Debug.Log("Игра закрыта");
    }
}

```

```

}

public void Logout()
{
    PlayerPrefs.DeleteKey("CurrentUserID");
    PlayerPrefs.DeleteKey("CurrentUserName");
    PlayerPrefs.Save();

    SceneManager.LoadScene("AuthPage");
}
}

```

Файл EffectObject.cs:

```

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

public class EffectObject : MonoBehaviour
{
    public float lifetime = 1f;

    void Start()
    {
    }

    void Update()
    {
        Destroy(gameObject, lifetime);
    }
}

```

Файл LevelLoader.cs:

```

using UnityEngine;
using Mono.Data.Sqlite;
using System.Collections.Generic;

public class LevelLoader : MonoBehaviour
{
    [Header("Настройки")]
    public GameObject notePrefab;
    public Transform noteHolder;
}

```

```

private float[] xCoords = { -1.5f, -0.5f, 0.5f, 1.5f };

public void LoadLevel(int levelId)
{
    foreach (Transform child in noteHolder)
    {
        Destroy(child.gameObject);
    }

    string dbPath = "URI=file:" + Application.streamingAssetsPath + "/MusicTrackDB.db";

    using (var connection = new SqlConnection(dbPath))
    {
        connection.Open();
        using (var command = connection.CreateCommand())
        {
            command.CommandText = "SELECT spawn_time, column_index FROM Notes WHERE level_id =
@lvl";

            command.Parameters.AddWithValue("@lvl", levelId);

            using (var reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    float yPos = reader.GetFloat(0);
                    int colIndex = reader.GetInt32(1);

                    GameObject newNote = Instantiate(notePrefab, noteHolder);

                    newNote.transform.localPosition = new Vector3(xCoords[colIndex], yPos, 0);

                    switch (colIndex)
                    {
                        case 0: // Лево
                            newNote.transform.rotation = Quaternion.Euler(0, 0,
180f);

                            break;
                        case 1: // Вниз
                            newNote.transform.rotation = Quaternion.Euler(0, 0, 90f);
                            break;
                        case 2: // Вверх

```



```

{
    connection.Open();

    using (var deleteCmd = connection.CreateCommand())
    {
        deleteCmd.CommandText = "DELETE FROM Notes WHERE level_id = @lvl";
        deleteCmd.Parameters.AddWithValue("@lvl", targetLevelId);
        deleteCmd.ExecuteNonQuery();
    }

    NoteObject[] notesOnScene = FindObjectsOfType<NoteObject>();

    using (var transaction = connection.BeginTransaction())
    {
        foreach (var note in notesOnScene)
        {
            using (var insertCmd = connection.CreateCommand())
            {
                insertCmd.CommandText =
                    "INSERT INTO Notes (level_id, spawn_time, column_index) VALUES (@lvl, @time, @col)";

                insertCmd.Parameters.AddWithValue("@lvl", targetLevelId);

                insertCmd.Parameters.AddWithValue("@time", note.transform.localPosition.y);

                int columnIndex = GetColumnFromX(note.transform.localPosition.x);
                insertCmd.Parameters.AddWithValue("@col", columnIndex);

                insertCmd.ExecuteNonQuery();
            }
        }
        transaction.Commit();
    }
    Debug.Log($"Успешно экспортировано {notesOnScene.Length} нот. Уровень ID: {targetLevelId}");
}

private int GetColumnFromX(float x)
{
    if (Mathf.Approximately(x, -1.5f)) return 0;
    if (Mathf.Approximately(x, -0.5f)) return 1;
    if (Mathf.Approximately(x, 0.5f)) return 2;
    if (Mathf.Approximately(x, 1.5f)) return 3;
}

```

```

        return 0;
    }
}

```

Файл GameManager.cs:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
using Mono.Data.Sqlite;
using TMPro;

public class GameManager : MonoBehaviour
{

    public AudioSource theMusic;

    public bool startPlaying;
    public bool useDatabase;

    public BeatScroller theBS;

    public static GameManager instance;

    public int currentScore;
    public int scorePerNote = 100;
    public int scorePerGoodNote = 125;
    public int scorePerPerfectNote = 150;

    public int currentMultiplier;
    public int multiplierTracker;
    public int[] multiplierThresholds;

    public Text scoreText;
    public Text multiText;

    public float totalNotes;
    public float normalHits;
    public float goodHits;
    public float perfectHits;
    public float missedHits;

```

```

public GameObject resultsScreen;

public Text percentHitText, normalsText, goodsText, perfectsText, missesText, rankText, finalScoreText;

public UnityEngine.UI.Image percentBar;

public bool autoPlay;
public AudioSource autoPlayAudioSource;

public TextMeshProUGUI timerText;

public UnityEngine.UI.Image healthBar;
public float maxHealth = 100f;
public float currentHealth;
public float healthPerNote = 50f;
public float healthMissPenalty = 10f;

public CanvasGroup buttonsGroup;

void Start()
{
    instance = this;
    currentHealth = 85;
    UpdateHealthBar();

    scoreText.text = "Score: 0";
    currentMultiplier = 1;

    if (useDatabase)
    {
        LevelLoader loader = GetComponent<LevelLoader>();
        if (loader != null)
        {
            int levelToLoad = PlayerPrefs.GetInt("SelectedLevelID", 1);
            loader.LoadLevel(levelToLoad);
        }
        else
        {
            Debug.LogError("На объекте GameManager отсутствует компонент LevelLoader!");
        }
    }
    else

```

```

{
    Debug.Log("Загрузка из БД отключена. Используются ноты, расставленные вручную на сцене.");
}

totalNotes = FindObjectsOfType<NoteObject>().Length;

if (autoPlayAudioSource == null)
{
    autoPlayAudioSource = gameObject.AddComponent<AudioSource>();
    autoPlayAudioSource.playOnAwake = false;
    autoPlayAudioSource.volume = 0.15f;
}
}

void Update()
{
    if (resultsScreen.activeInHierarchy)
    {
        if (Input.GetKeyDown(KeyCode.R))
        {
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
        }

        if (Input.GetKeyDown(KeyCode.Escape))
        {
            SceneManager.LoadScene("SongSelect");
        }
    }

    if (!startPlaying)
    {
        if (Input.anyKeyDown)
        {
            startPlaying = true;
            theBS.hasStarted = true;
            theMusic.Play();
        }
        return;
    }

    if (theMusic.isPlaying && timerText != null)
    {

```

```

        float t = theMusic.time;
        int minutes = (int)t / 60;
        int seconds = (int)t % 60;
        timerText.text = string.Format("{0}:{1:00}", minutes, seconds);
    }

    if (startPlaying && !theMusic.isPlaying && !resultsScreen.activeInHierarchy)
    {
        ShowResults();
    }

    if (startPlaying && currentHealth <= 0)
    {
        GameOver();
    }
}

void ShowResults()
{
    SaveResultToDatabase();

    if (buttonsGroup != null) buttonsGroup.gameObject.SetActive(false);
    if (theBS != null) theBS.gameObject.SetActive(false);

    resultsScreen.SetActive(true);

    normalsText.text = normalHits.ToString();
    goodsText.text = goodHits.ToString();
    perfectsText.text = perfectHits.ToString();
    missesText.text = missedHits.ToString();

    float totalHit = normalHits + goodHits + perfectHits;
    float percentHit = (totalNotes > 0) ? (totalHit / totalNotes) * 100f : 0f;

    percentHitText.text = percentHit.ToString("F1") + "%";
    rankText.text = GetRank(percentHit);
    finalScoreText.text = currentScore.ToString();
}

string GetRank(float percent)
{

```

```

        if (percent >= 100f) return "SS";
        if (percent > 95f) return "S";
        if (percent > 85f) return "A";
        if (percent > 70f) return "B";
        if (percent > 55f) return "C";
        if (percent > 40f) return "D";
        return "F";
    }

    void UpdateHealthBar()
    {
        if (healthBar != null)
        {
            healthBar.fillAmount = currentHealth / maxHealth;
        }
    }

    public void NoteHit()
    {
        Debug.Log("Hit On Time");

        currentHealth = Mathf.Min(currentHealth + healthPerNote, maxHealth);
        UpdateHealthBar();

        if(currentMultiplier - 1 < multiplierThresholds.Length)
        {
            multiplierTracker++;

            if (multiplierThresholds[currentMultiplier - 1] <= multiplierTracker)
            {
                multiplierTracker = 0;
                currentMultiplier++;
            }
        }

        multiText.text = "Multiplier: x" + currentMultiplier;

        scoreText.text = "Score: " + currentScore;
    }

    public void NormalHit()
    {
        currentScore += scorePerNote * currentMultiplier;
    }

```

```

        NoteHit();

        normalHits++;

        UpdateProgressBar();
    }

    public void GoodHit()
    {
        currentScore += scorePerGoodNote * currentMultiplier;
        NoteHit();

        goodHits++;

        UpdateProgressBar();
    }

    public void PerfectHit()
    {
        currentScore += scorePerPerfectNote * currentMultiplier;
        NoteHit();

        perfectHits++;

        UpdateProgressBar();
    }

    public void NoteMissed()
    {
        Debug.Log("Missed Note");

        currentHealth -= healthMissPenalty;
        UpdateHealthBar();

        currentMultiplier = 1;
        multiplierTracker = 0;

        multiText.text = "Multiplier: x" + currentMultiplier;

        missedHits++;

        UpdateProgressBar();
    }

```

```

void UpdateProgressBar()
{
    float totalHit = normalHits + goodHits + perfectHits;
    float progress = totalHit / totalNotes;

    percentBar.fillAmount = Mathf.Clamp01(progress);
}

void GameOver()
{
    theMusic.Stop();
    theBS.hasStarted = false;
    startPlaying = false;

    ShowResults();

    rankText.text = "FAILED";
    rankText.color = Color.red;
}

public void SaveResultToDatabase()
{
    if (!PlayerPrefs.HasKey("CurrentUserID")) {
        Debug.LogWarning("Результат не сохранен: пользователь не авторизован.");
        return;
    }

    int userId = PlayerPrefs.GetInt("CurrentUserID");
    int levelId = PlayerPrefs.GetInt("SelectedLevelID", 1);
    float totalHit = normalHits + goodHits + perfectHits;
    float accuracy = (totalNotes > 0) ? (totalHit / totalNotes) * 100f : 0f;
    string rank = GetRank(accuracy);

    if (currentHealth <= 0) rank = "FAILED";

    string dbPath = "URI=file:" + Application.streamingAssetsPath + "/UserDataDB.db";

    using (var conn = new SqlConnection(dbPath)) {
        conn.Open();
        using (var cmd = conn.CreateCommand()) {
            cmd.CommandText = @"INSERT INTO Scores

```

```
(user_id, level_id, score, rank, accuracy, perfect_hits, good_hits, normal_hits, misses)
VALUES (@uid, @lid, @score, @rank, @acc, @perfect, @good, @normal, @miss);
```

```
cmd.Parameters.AddWithValue("@uid", userId);
cmd.Parameters.AddWithValue("@lid", levelId);
cmd.Parameters.AddWithValue("@score", currentScore);
cmd.Parameters.AddWithValue("@rank", rank);
cmd.Parameters.AddWithValue("@acc", accuracy);
cmd.Parameters.AddWithValue("@perfect", perfectHits);
cmd.Parameters.AddWithValue("@good", goodHits);
cmd.Parameters.AddWithValue("@normal", normalHits);
cmd.Parameters.AddWithValue("@miss", missedHits);

try {
    cmd.ExecuteNonQuery();
    Debug.Log("Результат игры успешно сохранен в БД!");
}
catch (System.Exception e) {
    Debug.LogError("Ошибка сохранения в БД: " + e.Message);
}
}
}
}
}
```

Файл NoteGenerator.cs:

```
using UnityEngine;

public class NoteGenerator : MonoBehaviour
{
    public GameObject notePrefab;
    public Transform noteHolder;

    [Header("Настройки спавна")]
    public int amount = 10;
    public float startY = 3.47f;
    public float xPos = 1.5f;
    public float stepY = 1f;

    [ContextMenu("Generate Notes")]
    public void Generate()
    {

```

```

if (notePrefab == null || noteHolder == null)
{
    Debug.LogError("Заполни Prefab и Holder!");
    return;
}

for (int i = 0; i < amount; i++)
{
    GameObject newNote = Instantiate(notePrefab, noteHolder);
    float spawnY = startY + (i * stepY);
    newNote.transform.localPosition = new Vector3(xPos, spawnY, 0);

    NoteObject nObj = newNote.GetComponent<NoteObject>();
    if (nObj != null)
    {
        SetNoteData(newNote, nObj);
    }
}

Debug.Log($"Сгенерировано {amount} нот на позиции X: {xPos}");
}

private void SetNoteData(GameObject note, NoteObject nObj)
{
    if (Mathf.Approximately(xPos, -1.5f))
    {
        nObj.keyToPress = KeyCode.LeftArrow;
        note.transform.rotation = Quaternion.Euler(0, 0, 180f);
    }
    else if (Mathf.Approximately(xPos, -0.5f))
    {
        nObj.keyToPress = KeyCode.UpArrow;
        note.transform.rotation = Quaternion.Euler(0, 0, 90f);
    }
    else if (Mathf.Approximately(xPos, 0.5f))
    {
        nObj.keyToPress = KeyCode.DownArrow;
        note.transform.rotation = Quaternion.Euler(0, 0, 270f);
    }
    else if (Mathf.Approximately(xPos, 1.5f))
    {
        nObj.keyToPress = KeyCode.RightArrow;
        note.transform.rotation = Quaternion.Euler(0, 0, 0f);
    }
}

```

```

    }
}
}

```

Файл NoteObject.cs:

```

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

public class NoteObject : MonoBehaviour
{

    public bool canBePressed;

    public KeyCode keyToPress;

    private bool wasHit;

    public GameObject hitEffect, goodEffect, perfectEffect, missEffect;

    public AudioClip autoPerfectSound;//AI

    void Start()
    {

    }

    void Update()
    {
        if (!GameManager.instance.autoPlay)
        {
            if (Input.GetKeyDown(keyToPress))
            {
                if (canBePressed && !wasHit)
                {
                    if (!IsNoteBelowMe())
                    {
                        TryHit();
                    }
                }
            }
        }
    }
}

```

```

        }
    }
    else
    {
        if (canBePressed && !wasHit)
        {
            if (Mathf.Abs(transform.position.y) <= 0.01f)
            {
                TryHit();
            }
        }
    }
}

void TryHit()
{
    wasHit = true;
    gameObject.SetActive(false);

    float distance = Mathf.Abs(transform.position.y);

    if (distance > 0.25f)
    {
        GameManager.instance.NormalHit();
        Instantiate(hitEffect, transform.position, hitEffect.transform.rotation);
    }
    else if (distance > 0.05f)
    {
        GameManager.instance.GoodHit();
        Instantiate(goodEffect, transform.position, goodEffect.transform.rotation);
    }
    else
    {
        GameManager.instance.PerfectHit();
        Instantiate(perfectEffect, transform.position, perfectEffect.transform.rotation);

        if (GameManager.instance.autoPlay && autoPerfectSound != null)
        {
            GameManager.instance.autoPlayAudioSource.PlayOneShot(autoPerfectSound);
        }
    }
}

```

```

}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Activator")
    {
        canBePressed = true;
    }
}

void OnTriggerExit2D(Collider2D other)
{
    if (other.tag == "Activator")
    {
        canBePressed = false;

        if (!wasHit)
        {
            GameManager.instance.NoteMissed();
            Instantiate(missEffect, transform.position, missEffect.transform.rotation);
        }
    }
}

bool IsNoteBelowMe()
{
    RaycastHit2D hit = Physics2D.Raycast(transform.position, Vector2.down, 1.0f);

    if (hit.collider != null && hit.collider.gameObject != this.gameObject)
    {
        if (hit.collider.GetComponent<NoteObject>() != null)
        {
            return true;
        }
    }
    return false;
}
}

```

```

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

public class BeatScroller : MonoBehaviour
{

    public float beatTempo;

    public bool hasStarted;

    public float speedMultiplier = 1;

    void Start()
    {
        beatTempo = beatTempo / 60f;
    }

    void Update()
    {
        if (!hasStarted)
        {
        } else
        {
            transform.position -= new Vector3(0f, (beatTempo * speedMultiplier) * Time.deltaTime, 0f);
        }
    }
}

```

Файл ButtonController.cs:

```

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

public class ButtonController : MonoBehaviour
{
    private SpriteRenderer theSR;
    public Sprite defaultImage;
    public Sprite pressedImage;

    public KeyCode keyToPress;

```

```
public AudioClip pressSound;
private AudioSource pressSoundSource;

void Start()
{
    theSR = GetComponent<SpriteRenderer>();

    pressSoundSource = gameObject.AddComponent<AudioSource>();
    pressSoundSource.playOnAwake = false;

    pressSoundSource.volume = 0.05f;
}

void Update()
{
    if (Input.GetKeyDown(keyToPress))
    {
        theSR.sprite = pressedImage;

        if (pressSound != null)
        {
            pressSoundSource.PlayOneShot(pressSound);
        }
    }

    if (Input.GetKeyUp(keyToPress))
    {
        theSR.sprite = defaultImage;
    }
}
}
```

ДОДАТОК Б

Реалізація вибірки статистичних даних для аналізу прогресу:

```
public void SaveResultToDatabase()
{
    if (!PlayerPrefs.HasKey("CurrentUserID")) {
        Debug.LogWarning("Результат не збережено: користувач не авторизований.");
        return;
    }

    int userId = PlayerPrefs.GetInt("CurrentUserID");
    int levelId = PlayerPrefs.GetInt("SelectedLevelID", 1);
    float totalHit = normalHits + goodHits + perfectHits;
    float accuracy = (totalNotes > 0) ? (totalHit / totalNotes) * 100f : 0f;
    string rank = GetRank(accuracy);

    if (currentHealth <= 0) rank = "FAILED";

    string dbPath = "URI=file:" + Application.streamingAssetsPath + "/UserDataDB.db";

    using (var conn = new SqlConnection(dbPath)) {
        conn.Open();
        using (var cmd = conn.CreateCommand()) {
            cmd.CommandText = @"INSERT INTO Scores
            (user_id, level_id, score, rank, accuracy, perfect_hits, good_hits, normal_hits, misses)
            VALUES (@uid, @lid, @score, @rank, @acc, @perfect, @good, @normal, @miss)";

            cmd.Parameters.AddWithValue("@uid", userId);
            cmd.Parameters.AddWithValue("@lid", levelId);
            cmd.Parameters.AddWithValue("@score", currentScore);
            cmd.Parameters.AddWithValue("@rank", rank);
            cmd.Parameters.AddWithValue("@acc", accuracy);
            cmd.Parameters.AddWithValue("@perfect", perfectHits);
            cmd.Parameters.AddWithValue("@good", goodHits);
            cmd.Parameters.AddWithValue("@normal", normalHits);
            cmd.Parameters.AddWithValue("@miss", missedHits);

            try {
                cmd.ExecuteNonQuery();
                Debug.Log("Результат гри успішно збережено в БД!");
            }
            catch (System.Exception e) {
```

```
        Debug.LogError("Помилка збереження в БД: " + e.Message);  
    }  
}  
}  
}
```

ДОДАТОК В

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І

ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ

Я, Волохін Богдан Олександрович, здобувач НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

- Моя бакалаврська кваліфікаційна робота на тему «Програмне забезпечення для розвитку дрібної моторики рук на основі музичного ритму» є результатом моєї особистої праці.
- Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
- Щодо використання інструментів ШІ зазначаю, що:
 - ☐ інструменти генеративного ШІ не використовувалися.
 - ☒ інструмент ШІ Gemini був використаний для:
 - ☐ перекладу іншомовних джерел;
 - ☒ стилістичного редагування та перевірки граматики;
 - ☐ допомоги у написанні/відлагодженні програмного коду;
 - ☐ інше: _____.

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

« _____ » 2026 р.

Богдан ВОЛОХІН

(підпис)