**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

**Факультет інформаційних технологій**

| | |
|---|---|
| **ПОГОДЖЕНО** | **ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ** |
| **Декан факультету** | **Завідувач кафедри комп'ютерних наук** |
| _____ **Ігор БОЛБОТ** | _____ **Белла ГОЛУБ** |
| (підпис) | (підпис) |
| «___» _____ 2026 р. | «___» _____ 2026 р. |

# БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

**на тему: «Ігровий застосунок з реалізацією механізму нескінченного бігу»**

Спеціальність «122 Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

**Гарант освітньої програми**
д.е.н., професор _____ **Роман РУДЕНСЬКИЙ**
(підпис)

**Керівник бакалаврської кваліфікаційної роботи**
к.т.н, доцент _____ **Ганна ВАЙГАН**
(підпис)

**Виконав** _____ **Вадим СУПРУНЮК**
(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)

«___» _____ 2026 р.

# ЗАВДАННЯ ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Супрунюку Вадиму Ростиславовичу
(прізвище, ім'я, по батькові)

Спеціальність «122 Комп'ютерні науки»
Освітньо-професійна програма «Комп'ютерні науки»

Тема бакалаврської кваліфікаційної роботи
«Ігровий застосунок з реалізацією механізму нескінченного бігу»
затверджена наказом від «06» січня 2026 р. № 46 «С»

Термін подання завершеної роботи на кафедру «_22__» травня 2026 р.

Вихідні дані до роботи: опис ігрових механік жанру «Endless Runner»; вимоги до тривимірного ігрового застосунку з динамічною генерацією перешкод та системою збереження рекордів; структура модулів керування транспортними засобами, генерації середовища та користувацького інтерфейсу; технології Unity, C#, Universal Render Pipeline, SQLite, TextMeshPro та Unity UI. .

Перелік питань, що підлягають дослідженню: аналіз предметної області та обгрунтування вибору технологій; проєктування та архітектура ігрового застосунку; програмна реалізація та оптимізація ігрового застосунку; тестування та демонстрація роботи застосунку

Перелік графічних документів (за необхідності).

Дата видачі завдання «_02_» ___лютого_____ 2025 р.

**Керівник бакалаврської кваліфікаційної роботи** _____ **Ганна ВАЙГАНГ**
(підпис)

**Завдання взяв до виконання** _____ **Вадим СУПРУНЮК**
(підпис)

# ЗМІСТ

**ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ** **5**

**ВСТУП** **7**

**1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБГРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ** **11**

1.1 Аналіз жанру ігрових застосунків «Endless Runner» 11

1.2 Порівняльний аналіз ігрових рушіїв та вибір Unity 15

1.3 Обґрунтування вибору системи керування базами даних SQLite 18

1.4 Огляд графічних та звукових ресурсів розробки 21

1.5 Постановка завдання 23

**2 ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА ІГРОВОГО ЗАСТОСУНКУ** **25**

2.1 Функціональні вимоги до гри та сценарії використання Use Case 25

2.2 Логічна модель даних 28

2.3 Діаграма пакетів та діаграма класів 30

2.4 Діаграма кооперацій та діаграма компонентів 34

**3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ОПТИМІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ** **38**

3.1 Реалізація механіки нескінченної генерації перешкод та середовища 38

3.2 Розробка штучного інтелекту для транспортних засобів 42

3.3 Налаштування графіки, конвеєру рендеренгу, фільтрів та візуальних ефектів 45

3.4 Реалізація інтерфейсу користувача(UI) та інтеграція кастомних шрифтів 48

3.5 Робота зі звуковим супроводом та аудіо-ефектами 52

3.6 Інтеграція SQLite для виведення таблиці рекордів 54

**4. ТЕСТУВАННЯ ТА ДЕМОНСТРАЦІЯ РОБОТИ ЗАСТОСУНКУ** **58**

4.1 Методика та результати тестування ігрових механік і геймплею 58

4.2 Вимоги до апаратного та програмного забезпечення 62
4.3 Склад інсталяційного пакету та процес розгортання 65
**ВИСНОВКИ** **67**
**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ** **69**
**ДОДАТОК А** **72**
**ДОДАТОК Б** **87**

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) – програмний інтерфейс застосунку.

Asset – цифровий ресурс ігрового середовища (модель, текстура, звук тощо).

C# – об’єктно-орієнтована мова програмування платформи .NET.

Draw Calls – кількість викликів рендерингу графічних об’єктів.

Game Over – стан завершення ігрової сесії.

JSON (JavaScript Object Notation) – текстовий формат обміну структурованими даними.

Leaderboard – таблиця рекордів користувачів.

Low-Poly – стиль тривимірної графіки з малою кількістю полігонів.

PCM (Pulse Code Modulation) – імпульсно-кодова модуляція аудіосигналу.

Prefab – шаблон ігрового об’єкта в середовищі Unity.

Profiler – інструмент аналізу продуктивності програмного забезпечення.

SDF (Signed Distance Field) – технологія рендерингу шрифтів на основі поля знакових відстаней.

SFX (Sound Effects) – звукові ефекти.

SQLite – вбудована реляційна система керування базами даних.

URP (Universal Render Pipeline) – універсальний конвеєр рендерингу Unity.

WAV (Waveform Audio File Format) – формат зберігання аудіофайлів.

XML (eXtensible Markup Language) – розширювана мова розмітки.

БД – база даних.

ГП – графічний процесор (Graphics Processing Unit, GPU).

ДВГ – двовимірна графіка (Two-Dimensional Graphics, 2D).

К/с – кількість кадрів за секунду (Frames Per Second, FPS).

КД – користувацький досвід взаємодії із системою (User Experience, UX).

КІ – користувацький інтерфейс (User Interface, UI).

МСП – мова структурованих запитів (Structured Query Language, SQL).

МСР – механізм збору сміття (Garbage Collector, GC).

ООП – об’єктно-орієнтоване програмування.

Патерн Object Pool – шаблон проєктування для повторного використання об’єктів у пам’яті.

ПЗ – програмне забезпечення.

ПР – програмний рушій.

СУБД – система керування базами даних.

ТВГ – тривимірна графіка (Three-Dimensional Graphics, 3D).

УММ – уніфікована мова моделювання (Unified Modeling Language, UML).

ЦП – центральний процесор (Central Processing Unit, CPU).

ШІ – штучний інтелект (Artificial Intelligence, AI).

# ВСТУП

Актуальність теми. Сучасний етап розвитку інформаційних технологій характеризується стрімким поширенням інтерактивних цифрових систем, серед яких комп'ютерні ігри займають окреме місце як складні програмно-технічні комплекси, що поєднують засоби тривимірної графіки, системи реального часу, алгоритми керування даними та механізми взаємодії користувача із програмним середовищем. Індустрія розробки комп'ютерних ігор є одним із найбільш динамічних напрямів сучасної програмної інженерії, а створення ігрових застосунків потребує використання сучасних підходів до архітектурного проєктування, оптимізації програмного коду та управління ресурсами пам'яті [1, 2, 3. 4].

Одним із популярних напрямів аркадних ігор є жанр «Endless Runner», особливістю якого є безперервний рух ігрового персонажа або транспортного засобу в динамічно згенерованому середовищі. Популярність такого типу застосунків пояснюється високою динамікою ігрового процесу, простотою взаємодії користувача із системою та можливістю реалізації ігрового процесу на різних апаратних платформах. Разом із цим реалізація механізму нескінченного бігу створює значне навантаження на програмну систему, оскільки потребує постійного створення, обробки та оновлення великої кількості тривимірних об'єктів сцени в режимі реального часу [1, 2].

Важливою проблемою під час розробки таких систем є забезпечення стабільної продуктивності та плавності відображення графіки. Постійна генерація та видалення об'єктів може спричиняти надмірне навантаження на механізми керування пам'яттю ігрового рушія, що негативно впливає на частоту кадрів та загальну стабільність роботи застосунку. Для вирішення цієї проблеми застосовуються сучасні методи оптимізації, зокрема патерни об'єктно-орієнтованого програмування та механізми повторного використання ресурсів оперативної пам'яті, одним із яких є патерн Object Pool [1, 3].

Крім забезпечення продуктивності, повноцінний ігровий застосунок повинен містити механізми збереження результатів користувача, систему взаємодії із графічним інтерфейсом, підтримку аудіосупроводу та модулі обробки ігрових подій. Для реалізації таких функцій доцільним є використання сучасних програмних платформ і засобів розробки. Одним із найбільш поширених середовищ створення інтерактивних тривимірних застосунків є Unity, який забезпечує підтримку мови програмування C#, засобів моделювання сцен, інтегрованих механізмів профілювання продуктивності та можливостей роботи з локальними базами даних [6 – 9].

Використання локальної реляційної бази даних SQLite дозволяє реалізувати компактну та ефективну систему збереження результатів користувача без необхідності використання окремого серверного програмного забезпечення. Це забезпечує автономність роботи застосунку, швидкий доступ до даних та спрощує структуру програмної системи [9, 20 – 22].

Отже, розробка ігрового застосунку з реалізацією механізму нескінченного бігу є актуальним завданням у галузі комп’ютерних наук, оскільки поєднує питання архітектурного проєктування програмного забезпечення, оптимізації систем реального часу, процедурної генерації тривимірного середовища, інтеграції баз даних та реалізації сучасного користувацького інтерфейсу.

Мета дослідження полягає у проєктуванні, моделюванні та програмній реалізації тривимірного ігрового застосунку жанру «Endless Runner» у середовищі Unity із використанням механізмів процедурної генерації об’єктів, оптимізації використання оперативної пам’яті та локального збереження ігрових результатів.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- провести аналіз предметної області та сучасних підходів до розробки ігрових застосунків жанру «Endless Runner»;
- здійснити порівняльний аналіз сучасних ігрових рушіїв та обґрунтувати вибір середовища розробки;
- спроєктувати архітектуру програмного забезпечення ігрового застосунку;

- реалізувати механізми процедурної генерації ігрового середовища та взаємодії ігрових об’єктів;
- реалізувати механізми оптимізації використання оперативної пам’яті на основі патерна Object Pool та інтегрувати локальну базу даних SQLite;
- виконати тестування програмного застосунку та оцінити ефективність його роботи.

Об’єкт дослідження – процес розробки та оптимізації тривимірних ігрових застосунків у середовищах реального часу.

Предмет дослідження – методи та програмні засоби реалізації механізму нескінченного бігу, процедурної генерації ігрового середовища, оптимізації використання ресурсів пам’яті та збереження даних у тривимірному ігровому застосунку.

Методи дослідження. У роботі використано методи об’єктно-орієнтованого аналізу та програмування, методи UML-моделювання для проєктування архітектури системи, методи реляційного моделювання даних для створення структури бази даних SQLite, алгоритми процедурної генерації об’єктів, а також емпіричні методи тестування та оцінювання продуктивності програмного забезпечення.

Практичне значення одержаних результатів полягає у створенні функціонального ігрового застосунку з механізмом нескінченного бігу, який може бути використаний як основа для подальшого розвитку інтерактивних програмних систем, дослідження методів оптимізації ігрових застосунків та вивчення сучасних технологій розробки програмного забезпечення у галузі комп’ютерних наук.

Структура та обсяг роботи. Бакалаврська кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. У першому розділі проведено аналіз предметної області, розглянуто особливості жанру «Endless Runner», виконано порівняльний аналіз сучасних ігрових рушіїв та обґрунтовано вибір технологій розробки. У другому розділі наведено процес проєктування архітектури програмного забезпечення, побудову

UML-діаграм та логічної моделі даних. Третій розділ присвячений програмній реалізації ігрового застосунку, механізмам процедурної генерації, оптимізації продуктивності, інтеграції графічного інтерфейсу та локальної бази даних SQLite. У четвертому розділі подано результати тестування програмного застосунку, описано процес розгортання та оцінено ефективність функціонування розробленої системи.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБГРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ

## 1.1 Аналіз жанру ігрових застосунків «Endless Runner»

Жанр «Endless Runner» належить до класу аркадних ігор із безперервним ігровим процесом, у яких рух керованого об’єкта виконується автоматично, а основна взаємодія користувача із системою зосереджується на своєчасному реагуванні на зміну ігрового середовища. Активний розвиток цього жанру розпочався разом із поширенням мобільних платформ та розвитком кросплатформних ігрових рушіїв, що дозволило створювати динамічні інтерактивні застосунки з високою частотою оновлення сцени та мінімальним порогом входження для користувача [1–4]. У сучасних умовах ігри такого типу активно використовуються не лише як розважальні продукти, а і як приклад реалізації систем реального часу з процедурною генерацією об’єктів та інтерактивною взаємодією користувача із програмним середовищем.

Ключовою особливістю жанру є відсутність фіксованого завершення рівня. Ігрова сесія триває до моменту виникнення критичної події, зазвичай зіткнення керованого об’єкта з перешкодою або іншим елементом сцени. Унаслідок цього основне навантаження під час виконання застосунку припадає не на складні сюжетні сценарії чи побудову великих статичних карт, а на постійну генерацію ігрового середовища, оновлення позицій об’єктів, перевірку колізій та підтримання стабільної продуктивності в режимі реального часу.

Динамічний характер геймплею вимагає безперервного відтворення ігрового світу без помітних затримок, що зумовлює підвищені вимоги до оптимізації алгоритмів генерації середовища та ефективного використання ресурсів оперативної пам’яті.

Як показує аналіз сучасних реалізацій ігор цього типу, механіка нескінченного руху безпосередньо впливає на архітектуру програмної системи,

організацію пам’яті та логіку взаємодії окремих модулів застосунку. Основні особливості жанру та їх вплив на програмну реалізацію наведено в табл. 1.1.

Таблиця 1.1

Ключові особливості жанру «Endless Runner» та їх вплив на програмну реалізацію

| Особливість жанру | Характеристика ігрової механіки | Вплив на архітектуру та програмну реалізацію |
|---|---|---|
| Процедурна генерація середовища | Ігровий простір формується динамічно під час виконання застосунку шляхом комбінування окремих блоків середовища, перешкод і транспортних засобів | Потребує реалізації алгоритмів генерації об’єктів, випадкового вибору префабів та точного позиціонування сегментів сцени |
| Нескінченність ігрового простору | Ігрова сесія не має фіксованої кінцевої точки та триває до виникнення умови завершення гри | Вимагає постійного оновлення сцени, динамічного керування пам’яттю та деактивації невидимих об’єктів |
| Висока динаміка геймплею | Швидкість руху та інтенсивність появи перешкод поступово зростають під час гри | Необхідна стабільна частота кадрів, швидка обробка фізичних взаємодій та оптимізація рендерингу |
| Циклічність ігрових сесій | Після завершення гри користувач може швидко розпочати нову сесію без тривалого очікування | Потрібне швидке скидання стану сцени та повторна ініціалізація ігрових компонентів без повного перезавантаження ресурсів |
| Конкурентна мотивація | Ігровий процес орієнтований на покращення результатів та повторне проходження для досягнення більшого рахунку | Потребує інтеграції системи збереження результатів та реалізації таблиці рекордів |
| Багатооб’єктність сцени | У сцені одночасно присутня значна кількість рухомих та статичних об’єктів | Вимагає використання механізмів оптимізації пам’яті та повторного використання об’єктів на основі патерна Object Pool |
| Безперервність взаємодії | Гравець постійно взаємодіє з динамічним середовищем у режимі реального часу | Потребує швидкої обробки подій, перевірки колізій та синхронізації між компонентами системи |
| Простота керування | Основна взаємодія реалізується через обмежену кількість дій, зокрема зміну смуги руху та ухилення від перешкод | Вимагає створення інтуїтивного користувацького інтерфейсу та швидкого реагування системи на дії користувача |

Дані, наведені в табл. 1.1, демонструють, що реалізація застосунків жанру «Endless Runner» безпосередньо пов’язана із завданнями оптимізації продуктивності та ефективного керування ресурсами системи. На відміну від покрокових або сюжетно орієнтованих ігор, у яких основна увага приділяється

сценарній логіці, у раннерах критичне значення мають швидкість обробки подій, стабільність графічного конвеєра та мінімізація затримок під час генерації нових елементів сцени.

Для тривимірних застосунків цього типу додатковим фактором навантаження є необхідність одночасного опрацювання значної кількості моделей, текстур, анімацій та фізичних взаємодій. Постійне створення і видалення об’єктів за допомогою стандартних механізмів рушія призводить до інтенсивної роботи механізму збору сміття, що може спричиняти короткочасні затримки відображення графіки та зниження частоти кадрів [1, 6, 12]. З цієї причини сучасні ігрові застосунки активно використовують технології повторного використання ресурсів пам’яті та шаблони оптимізації життєвого циклу об’єктів.

На рисунку 1.1 наведено загальну структурну модель функціонування ігрового застосунку жанру «Endless Runner».

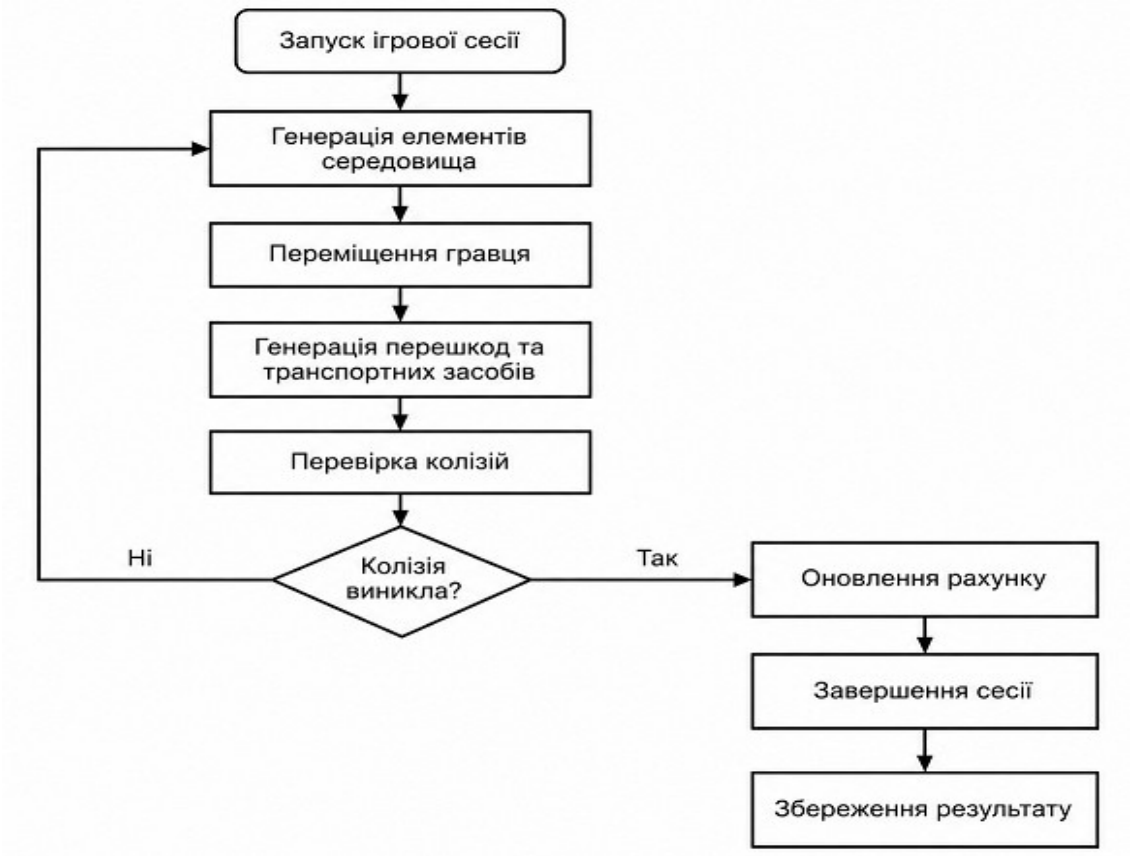



Рис. 1.1 Загальна схема функціонування ігрового застосунку жанру «Endless Runner»

Наведена схема відображає циклічний характер роботи системи, при якому більшість процесів виконується безперервно до моменту завершення ігрової сесії. Такий підхід вимагає чіткої синхронізації між модулями генерації об’єктів, системою фізичних взаємодій, графічним конвеєром та підсистемою керування пам’яттю.

Розроблюваний застосунок реалізує тривимірний «Endless Runner», у якому користувач керує автомобілем на багатосмуговій трасі, уникаючи зіткнень із транспортом і перешкодами, що генеруються динамічно. Така концепція поєднує механіку раннера з елементами симуляції дорожнього руху та алгоритмами штучного інтелекту транспортного трафіку.

Програмна реалізація побудована на модульній архітектурі, де генерація середовища, керування транспортом, обробка колізій і збереження результатів функціонують незалежно. Для оптимізації роботи з динамічними об’єктами використовується патерн Object Pool, який мінімізує операції створення та видалення елементів сцени під час геймплею [1, 7].

Важливим аспектом застосунків цього жанру є також забезпечення повторної зацікавленості користувача. Саме тому більшість сучасних реалізацій використовують системи збереження рекордів, таблиці результатів та механізми повторного запуску гри без тривалого перезавантаження сцени. У розроблюваному застосунку для цього інтегровано локальну базу даних SQLite, що забезпечує автономне збереження ігрових результатів та швидкий доступ до таблиці рекордів без використання зовнішнього серверного програмного забезпечення [20–22].

Отже, аналіз жанру «Endless Runner» показав, що подібні застосунки поєднують високі вимоги до продуктивності систем реального часу, механізмів процедурної генерації та ефективного керування пам’яттю. Це обумовлює необхідність використання сучасних ігрових рушіїв, інструментів профілювання та оптимізованих архітектурних рішень, здатних забезпечити стабільну роботу програмного продукту в умовах постійної зміни ігрового середовища.

## 1.2 Порівняльний аналіз ігрових рушіїв та вибір Unity

Одним із ключових етапів розробки інтерактивного програмного забезпечення є вибір технологічної платформи, яка забезпечує необхідну функціональність, стабільну продуктивність та підтримку сучасних засобів програмування. Для тривимірного ігрового застосунку з процедурною генерацією середовища, системою колізій, інтеграцією локальної бази даних та механізмами оптимізації пам’яті ігровий рушій визначає архітектуру програмної системи та особливості реалізації графічного конвеєра [6–9].

Серед сучасних кросплатформних засобів розробки найбільш поширеними є Unreal Engine, Godot Engine та Unity. Unreal Engine орієнтований на створення масштабних тривимірних проєктів із фотореалістичною графікою та використовує мову C++ і систему Blueprints. Попри високий рівень продуктивності, для low-poly аркадного застосунку його функціональність є частково надлишковою через високі системні вимоги та значний розмір фінального білду [6].

Godot Engine є відкритим ігровим рушієм із підтримкою GDScript і C#. Середовище є зручним для невеликих проєктів, однак його інструменти профілювання та підтримка складних тривимірних систем поступаються Unity і Unreal Engine [3].

Unity є кросплатформним середовищем розробки, що широко використовується для створення мобільних і десктопних ігор. Використання мови C# забезпечує підтримку об’єктно-орієнтованого підходу та інтеграцію бібліотек .NET [8, 18, 19]. Рушій містить інтегровані засоби профілювання продуктивності, підтримує Universal Render Pipeline та має готові модулі для реалізації користувацького інтерфейсу, фізичних взаємодій і роботи з базами даних [12–17].

Для детального аналізу можливостей зазначених платформ було проведено їх порівняння за основними технічними критеріями, результати якого наведено в табл. 1.2.

Таблиця 1.2

Порівняльна характеристика ігрових рушіїв за технічними критеріями

| Критерій порівняння | Unreal Engine | Godot Engine | Unity |
|---|---|---|---|
| Основна мова програмування | C++, Blueprints | GDScript, C# | C# |
| Тип ліцензування | Комерційна модель із роялті | Відкрите програмне забезпечення | Комерційна та персональна ліцензії |
| Парадигма керування пам’яттю | Ручне керування пам’яттю та власний Garbage Collector | Підрахунок посилань (Reference Counting) | Автоматичне керування пам’яттю через .NET Garbage Collector |
| Підтримка тривимірної графіки | Висока, орієнтація на AAA-проєкти | Середня | Висока |
| Продуктивність у low-poly 3D-проєктах | Надлишкова функціональність та високі системні вимоги | Достатня для невеликих проєктів | Висока при оптимальному використанні ресурсів |
| Інтеграція локальних СУБД | Потребує сторонніх C++ модулів або плагінів | Реалізується через додаткові плагіни | Наявна підтримка бібліотек .NET та SQLite |
| Інструменти профілювання та оптимізації | Професійні, але складні у використанні | Базові засоби аналізу продуктивності | Інтегровані профайлери та засоби оптимізації |
| Підтримка процедурної генерації | Висока | Середня | Висока |
| Розмір фінального білду | Значний (>100 МБ) | Дуже малий (15–30 МБ) | Помірний (30–50 МБ) |
| Підтримка документації та спільноти | Дуже велика | Середня | Дуже велика |
| Швидкість розробки аркадних ігор | Середня | Висока | Висока |
| Зручність реалізації UI | Висока, але потребує додаткового налаштування | Базова система UI | Розвинуті інструменти UI та TextMesh Pro |
| Доцільність використання для жанру «Endless Runner» | Частково надлишковий функціонал | Обмежений інструментарій для 3D | Оптимальне співвідношення продуктивності та швидкості розробки |

Наведені в таблиці 1.2 дані демонструють, що Unity забезпечує найбільш збалансоване поєднання функціональності, продуктивності та швидкості розробки для застосунків жанру «Endless Runner». Особливо важливими для

реалізації розроблюваної системи є підтримка мови C#, наявність інтегрованого профайлера та можливість ефективної роботи з великою кількістю динамічних об’єктів сцени.

На рисунку 1.2 наведено загальну схему взаємодії основних компонентів Unity під час виконання ігрового застосунку.

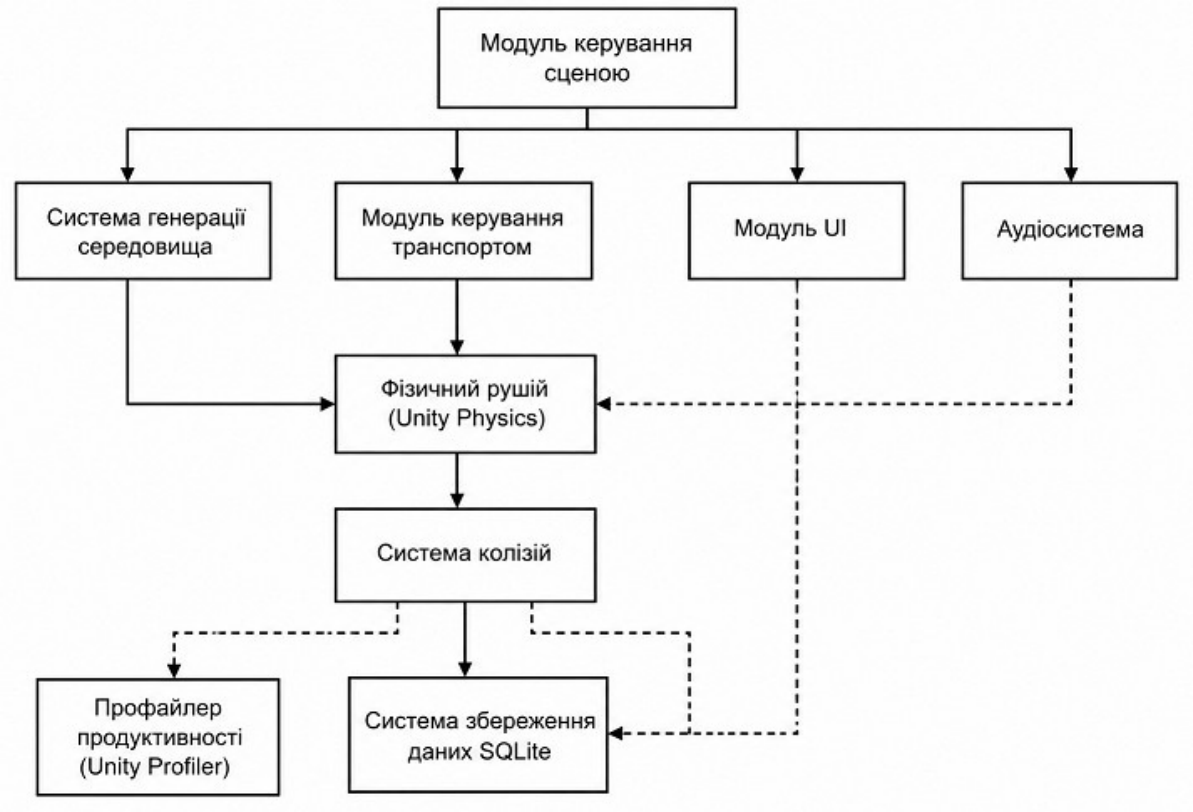



Рис. 1.2 Схема взаємодії компонентів Unity у розроблюваному застосунку

Наведена схема відображає модульний принцип організації програмного забезпечення, при якому окремі функціональні компоненти взаємодіють між собою через внутрішні механізми рушія Unity. Такий підхід дозволяє ізолювати критичні підсистеми, спростити тестування та забезпечити подальше масштабування програмного продукту.

Важливою перевагою Unity для розроблюваного застосунку є підтримка сучасних механізмів оптимізації продуктивності. Використання профайлера Unity дозволяє аналізувати навантаження на центральний процесор, графічний конвеєр та систему керування пам’яттю в режимі реального часу [29]. Це є особливо важливим для застосунків із процедурною генерацією середовища, де

одночасно виконується велика кількість операцій створення, деактивації та повторного використання об’єктів сцени.

Додатковою перевагою Unity є підтримка технології TextMesh Pro для реалізації масштабованого користувацького інтерфейсу [15], а також можливість інтеграції локальної бази даних SQLite через бібліотеки платформи .NET [20–22]. Це дозволило реалізувати систему збереження результатів користувача без використання зовнішніх серверних компонентів, що спрощує архітектуру програмного застосунку та забезпечує автономність його роботи.

Отже, результати проведеного аналізу показали, що Unity найбільш повно відповідає вимогам розроблюваного ігрового застосунку. Рушій забезпечує ефективну підтримку тривимірної графіки, процедурної генерації середовища, оптимізації використання пам’яті та інтеграції локальної бази даних, що дозволяє реалізувати стабільний та функціонально завершений програмний продукт жанру «Endless Runner».

## 1.3 Обґрунтування вибору системи керування базами даних SQLite

Під час проєктування ігрового застосунку важливим завданням є вибір засобу збереження даних користувача. Для застосунків жанру «Endless Runner» основними персистентними даними є результати ігрових сесій, на основі яких формується локальна таблиця рекордів [20–23]. У середовищі Unity для збереження даних найчастіше використовуються PlayerPrefs, текстові формати JSON/XML та локальні реляційні СУБД. PlayerPrefs забезпечує збереження простих параметрів у форматі «ключ-значення», однак є малоефективним для роботи з таблицями рекордів через відсутність засобів сортування та структурованого зберігання даних. Формати JSON і XML дозволяють зберігати складні структури, проте потребують повного зчитування та перезапису файлу під час оновлення інформації, що створює додаткове навантаження на систему.

SQLite є вбудованою реляційною системою керування базами даних, яка працює без окремого серверного програмного забезпечення та підтримує SQL-запити. База даних зберігається у вигляді одного локального файлу, що забезпечує компактність структури та високу швидкість роботи із записами [20–23]. Для визначення найбільш доцільного підходу було проведено порівняння основних засобів збереження даних, результати якого наведено в таблиці 1.3.

Таблиця 1.3

Порівняльна характеристика методів збереження даних у середовищі Unity

| Критерій оцінювання | PlayerPrefs | JSON/XML | SQLite |
|---|---|---|---|
| Тип та структурованість даних | Пари «ключ-значення» для простих типів даних | Структуровані текстові файли та масиви об’єктів | Реляційна структура таблиць із підтримкою типів даних |
| Підтримка складних структур даних | Низька | Середня | Висока |
| Масштабованість | Обмежена | Середня, ускладнюється зі збільшенням файлу | Висока, підтримується додавання нових записів і полів |
| Швидкість вибірки та сортування | Низька, сортування виконується в програмному коді | Середня, потребує десеріалізації даних | Висока, використовується SQL та конструкція ORDER BY |
| Надійність збереження даних | Середня | Середня, можливий ризик пошкодження файлу | Висока, підтримуються транзакції ACID |
| Підтримка SQL-запитів | Відсутня | Відсутня | Наявна |
| Навантаження на оперативну пам’ять | Мінімальне | Підвищене під час десеріалізації файлів | Мінімальне |
| Вимоги до системних ресурсів | Мінімальні | Середні | Мінімальні, використовується вбудована бібліотека |
| Зручність інтеграції з Unity | Висока | Висока | Висока |
| Доцільність використання для таблиці рекордів | Низька | Середня | Висока |

Наведені в табл. 1.3 результати демонструють, що SQLite є найбільш доцільним рішенням для реалізації системи збереження результатів у розроблюваному застосунку. Використання реляційної структури дозволяє

виконувати сортування, фільтрацію та вибірку записів безпосередньо засобами SQL, що зменшує навантаження на програмну логіку Unity та спрощує обробку даних.

На рисунку 1.3 наведено загальну схему взаємодії ігрового застосунку з локальною базою даних SQLite.

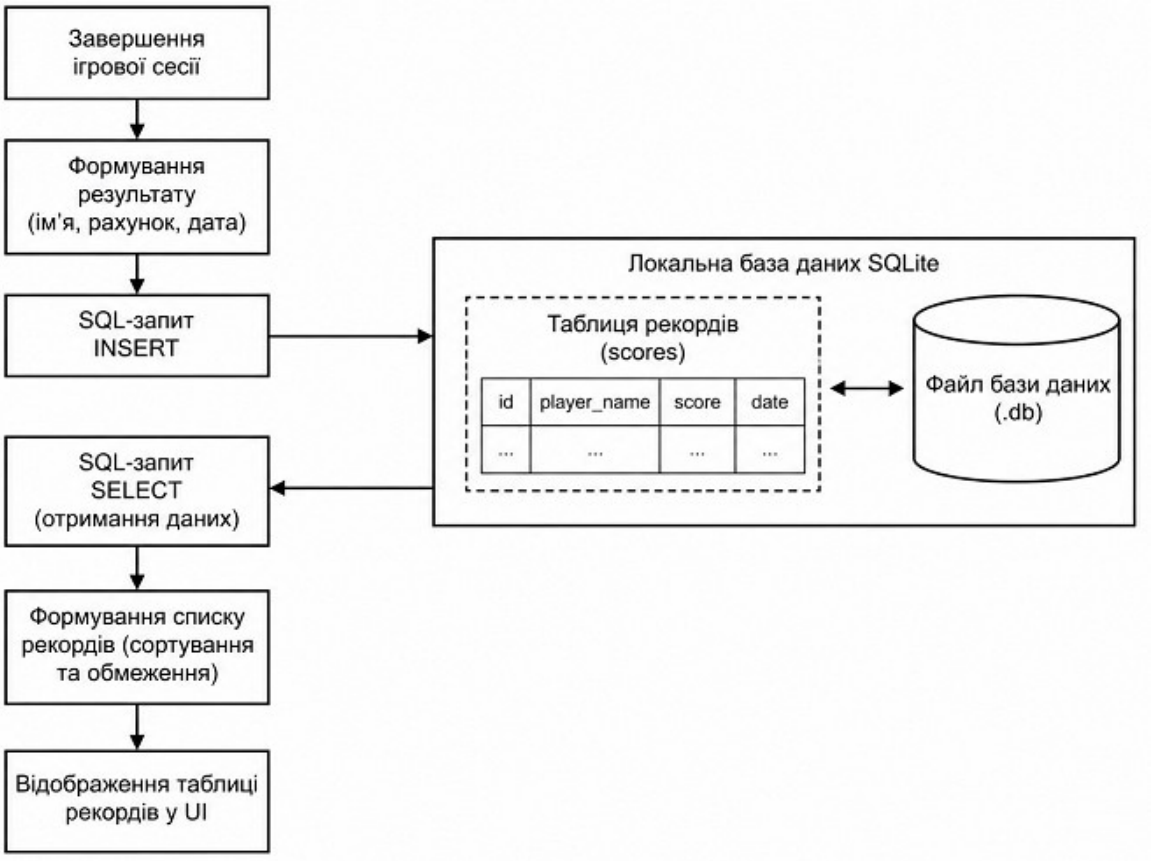



Рис. 1.3. Схема взаємодії ігрового застосунку з базою даних SQLite

Наведена схема демонструє послідовність обробки даних після завершення ігрової сесії. Результат користувача передається до бази даних за допомогою SQL-запиту, після чого збережені записи використовуються для формування таблиці рекордів та відображення її в інтерфейсі застосунку.

Використання SQLite у розроблюваному застосунку забезпечує компактність архітектури та не потребує налаштування окремого серверного середовища. Доступ до бази даних реалізується через бібліотеки платформи .NET, що спрощує інтеграцію з Unity та забезпечує підтримку стандартних механізмів роботи з SQL-запитами [18–23].

Для зберігання результатів у застосунку використовується одна таблиця, яка містить ім’я користувача, отриманий рахунок та дату створення запису. Така структура є достатньою для реалізації локальної таблиці рекордів і дозволяє уникнути надлишкового ускладнення структури бази даних. Сортування результатів здійснюється за допомогою SQL-конструкції ORDER BY, що забезпечує швидке формування списку найкращих результатів без додаткової обробки масивів у програмному коді.

Таким чином, використання SQLite є раціональним рішенням для реалізації підсистеми збереження даних у розроблюваному ігровому застосунку. Обрана СУБД забезпечує необхідний рівень продуктивності, підтримує ефективну роботу з локальними даними та дозволяє реалізувати надійну систему збереження результатів користувача без використання зовнішніх серверних компонентів.

## 1.4 Огляд графічних та звукових ресурсів розробки

Візуальна та звукова складові є важливими елементами сучасного ігрового застосунку, оскільки безпосередньо впливають на сприйняття ігрового процесу, зручність взаємодії користувача із системою та загальну продуктивність програмного забезпечення. Для застосунків жанру «Endless Runner» особливого значення набувають швидкість зчитування візуальної інформації, чіткість елементів інтерфейсу та стабільність роботи графічної й аудіосистеми під час динамічного оновлення сцени.

Під час розробки застосунку було обрано low-poly стилістику тривимірної графіки. Використання моделей із невеликою кількістю полігонів дозволяє зменшити навантаження на графічний процесор, скоротити кількість викликів рендерингу та забезпечити стабільну частоту кадрів без використання складних механізмів оптимізації [6, 12, 16]. Такий підхід також відповідає загальній стилістиці аркадних ігор та забезпечує високу читабельність об’єктів під час швидкого руху сцени.

Для формування міського середовища, реалізації користувацького інтерфейсу та озвучення ігрових подій було використано декілька зовнішніх мультимедійних ресурсів. Основні характеристики використаних графічних, текстових та звукових компонентів наведено в табл. 1.4.

Таблиця 1.4

Характеристика та призначення мультимедійних ресурсів ігрового застосунку

| Назва ресурсу \ пакету | Тип ресурсу | Функціональне призначення в ігровому застосунку |
|---|---|---|
| KayKit : City Builder | 3D-моделі (Low-Poly) | Побудова модульних блоків міського середовища: двосмугові дорожні покриття, хмарочоси, житлові будинки, ліхтарні стовпи, світлофори та базові моделі цивільних автомобілів для ШІ-трафіку. |
| KayKit : HelloweenBits | 3D-моделі (Low-Poly) | Кастомізація ігрового простору, додавання тематичних декорацій (елементи стилізованого оточення), що дозволяє урізноманітнити процедурну генерацію та зробити візуальний ряд більш унікальним. |
| Rubik Mono One Regular | Шрифт (TrueType Font) | Використовується як основна текстова гарнітура для елементів графічного інтерфейсу користувача (меню, лічильник набраних балів, таблиця рекордів) через систему TextMesh Pro. |
| Звукові ефекти(SFX) | Аудіофайли (.wav, .ogg) | Звукова індикація ігрових подій: кліки по кнопках інтерфейсу, звуки прискорення, зіткнення автомобіля з перешкодою (спрацьовування тригера Game Over). |

Наведені в табл. 1.4 ресурси формують цілісну мультимедійну систему застосунку та забезпечують узгоджене поєднання графічного інтерфейсу, тривимірного оточення та звукового супроводу.

Окрему увагу в ході розробки було приділено інтеграції шрифту Rubik Mono One Regular. Традиційні вбудовані шрифти Unity (Legacy Text) використовують растровий метод рендерингу, через що при масштабуванні тексту в інтерфейсі виникає ефект розмиття (пікселізації). Для уникнення цієї

проблеми шрифт було імпортовано та конвертовано у формат Signed Distance Field (SDF) за допомогою інструменту TextMesh Pro. Метод SDF базується на збереженні векторних відстаней до меж символів у спеціальній текстурній карті, завдяки чому текст інтерфейсу (зокрема динамічний лічильник балів) залишається максимально чітким та читабельним при будь-якій роздільній здатності екрана чи масштабуванні UI-елементів.

Аудіоресурси проєкту пройшли етап попередньої оптимізації. Фонова музика та звукові ефекти були стиснуті за допомогою кодеків Ogg Vorbis (для тривалих треків) та PCM/WAV (для коротких звуків подій). Це дозволило досягти мінімального використання оперативної пам'яті аудіосистемою (Audio Clip Compression) та запобігти затримкам відтворення звуку в моменти критичних ігрових подій.

Таким чином, використані графічні та звукові ресурси забезпечують необхідний рівень візуальної цілісності, продуктивності та зручності взаємодії користувача із застосунком. Обрані мультимедійні компоненти відповідають архітектурним особливостям розроблюваної системи та дозволяють реалізувати стабільний ігровий процес із динамічним оновленням тривимірного середовища.

## 1.5 Постановка завдання

На основі проведеного аналізу предметної області, дослідження особливостей жанру «Endless Runner», порівняння сучасних ігрових рушіїв та засобів збереження даних сформовано основні вимоги до розроблюваного програмного застосунку.

Метою розробки є проєктування, моделювання, програмна реалізація та тестування тривимірного ігрового застосунку жанру «Endless Runner» у середовищі Unity із використанням механізмів процедурної генерації об’єктів, оптимізації використання оперативної пам’яті та локального збереження результатів користувача за допомогою SQLite.

Розроблюваний застосунок повинен забезпечувати безперервний рух транспортного засобу із можливістю маневрування між смугами руху та взаємодії з динамічними перешкодами. Ігрове середовище має формуватися процедурно під час виконання програми шляхом генерації дорожніх сегментів, транспортних засобів та елементів міського оточення. Для створення динамічного ігрового процесу необхідно реалізувати алгоритми руху транспортного трафіку та систему обробки колізій між об’єктами сцени.

Важливою вимогою до програмної системи є забезпечення стабільної продуктивності під час активного оновлення ігрового середовища. Для цього необхідно реалізувати механізм повторного використання динамічних об’єктів на основі патерна Object Pool, що дозволить мінімізувати кількість операцій створення та видалення елементів сцени під час виконання гри. Архітектура програмного забезпечення повинна базуватися на принципах об’єктно-орієнтованого програмування та підтримувати модульну структуру взаємодії компонентів системи.

Для реалізації системи збереження результатів необхідно інтегрувати локальну базу даних SQLite, яка забезпечує автономне зберігання інформації без використання зовнішнього серверного програмного забезпечення. База даних повинна підтримувати швидке виконання операцій запису, сортування та вибірки результатів користувачів для формування таблиці рекордів.

Інтерфейс застосунку має забезпечувати зручну взаємодію користувача із системою та містити головне меню, ігрову панель із лічильником балів, вікно таблиці рекордів і панель завершення гри. Для реалізації текстових елементів необхідно використовувати систему TextMesh Pro із підтримкою масштабованого SDF-відображення шрифтів. Аудіосистема застосунку повинна підтримувати фоновий музичний супровід та звукові ефекти для відтворення ігрових подій.

Вхідними даними для реалізації проєкту є набори low-poly тривимірних моделей KayKit, текстові ресурси інтерфейсу, шрифти у форматі TrueType та аудіофайли у форматах WAV і Ogg Vorbis. Результатом розробки має стати

функціональний ігровий застосунок для операційної системи Windows, реалізований у вигляді автономного виконуваного програмного продукту з інтегрованою локальною базою даних та повним набором необхідних мультимедійних ресурсів.

# 2 ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА ІГРОВОГО ЗАСТОСУНКУ

## 2.1 Функціональні вимоги до гри та сценарії використання Use Case

Проєктування ігрового застосунку передбачає визначення його функціональних меж, основних дій користувача та реакцій системи на ці дії. Для формалізації функціональних вимог використано UML-діаграму прецедентів, яка дозволяє подати взаємодію гравця із застосунком на рівні сценаріїв використання без деталізації програмного коду [6, 8].

Основним актором системи є гравець, який взаємодіє із застосунком через користувацький інтерфейс. Його дії охоплюють запуск ігрової сесії, керування автомобілем, перегляд таблиці рекордів, збереження результату після завершення гри, перезапуск сесії та вихід із застосунку. Центральним сценарієм є «Грати в гру», оскільки саме він об’єднує ключові механіки застосунку: маневрування автомобілем, нарахування балів, перевірку колізій та фіксацію завершення ігрової сесії.

У розробленому застосунку завершення ігрової сесії відбувається після фіксації зіткнення автомобіля гравця з перешкодою або транспортним засобом штучного інтелекту. У момент виникнення колізії система автоматично припиняє подальше оновлення ігрового процесу, блокує керування автомобілем та активує панель Game Over. На цій панелі користувач отримує інформацію про набраний рахунок, а також можливість зберегти власний результат або повторно розпочати гру без повернення до головного меню.

Механізм збереження рекордів реалізовано через локальну базу даних SQLite, що забезпечує швидке виконання операцій запису та вибірки даних без використання зовнішнього серверного середовища. Після введення користувачем ігрового імені результат автоматично записується до таблиці

рекордів разом із відповідним значенням рахунку та датою створення запису. Перегляд таблиці рекордів здійснюється через окремий інтерфейсний модуль, який формує SQL-запит до бази даних, отримує найкращі результати та відображає їх у графічному інтерфейсі із використанням компонентів TextMesh Pro.

Основні функціональні сценарії застосунку подано в табл. 2.1.

Таблиця 2.1

Функціональні сценарії використання ігрового застосунку

| Прецедент | Ініціатор | Зміст сценарію | Результат виконання |
|---|---|---|---|
| Переглянути головне меню | Гравець | Відкриття стартового екрана після запуску застосунку | Користувач отримує доступ до основних дій |
| Грати в гру | Гравець | Запуск ігрової сесії з керуванням автомобілем | Починається активний ігровий процес |
| Маневрувати автомобілем | Гравець | Зміна положення автомобіля на дорозі для уникнення перешкод | Автомобіль переміщується між смугами руху |
| Набирати бали | Система | Автоматичне збільшення рахунку під час проходження дистанції | Оновлюється поточний результат користувача |
| Фіксувати Game Over | Система | Виявлення колізії та зупинення ігрової сесії | Відображається панель завершення гри |
| Зберегти рекорд | Гравець | Передавання імені користувача та рахунку до бази даних | Результат записується в таблицю рекордів |
| Перезапустити гру | Гравець | Повторний запуск ігрової сесії після завершення | Стан гри скидається, починається нова сесія |
| Переглянути | Гравець | Отримання | На екрані |

| | | | |
|---|---|---|---|
| таблицю рекордів | | збережених результатів із бази даних | відображаються найкращі результати |
| Вийти з гри | Гравець | Завершення роботи застосунку | Програма закривається |

Як видно з табл. 2.1, функціональна структура застосунку орієнтована на просту та зрозумілу взаємодію гравця із системою. Основна логіка зосереджена навколо ігрової сесії, а додаткові сценарії забезпечують збереження результатів, перегляд рекордів і повторне проходження.

Графічне подання зазначених сценаріїв наведено на рис. 2.1. Діаграма демонструє зв’язки між актором «Гравець» та основними прецедентами системи. Зв’язки «include» відображають обов’язкові дії, що виконуються під час гри, а зв’язки «extend» показують сценарії, які активуються після завершення ігрової сесії.

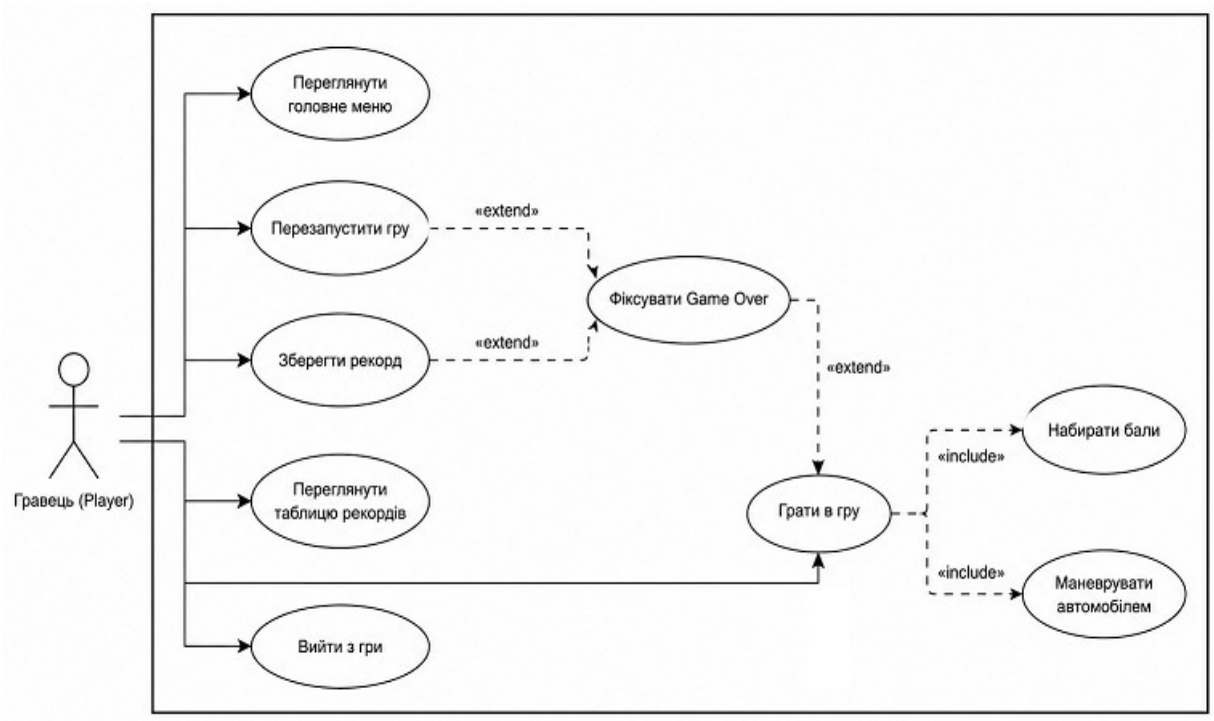



Рис. 2.1 Діаграма прецедентів (Use Case Diagram) ігрового застосунку

Подана діаграма підтверджує, що розроблюваний застосунок має чітко визначені функціональні межі. Взаємодія користувача не перевантажена

зайвими діями, а основні сценарії відповідають логіці жанру «Endless Runner»: швидкий запуск гри, безперервний ігровий процес, фіксація результату, збереження рекорду та можливість повторного проходження.

## 2.2 Логічна модель даних

Для забезпечення збереження результатів користувачів у розроблюваному застосунку використано локальну реляційну систему керування базами даних SQLite. Логічна модель даних призначена для зберігання інформації про завершені ігрові сесії та формування таблиці рекордів, яка використовується в інтерфейсі застосунку. Основними вимогами до структури бази даних є висока швидкість виконання операцій читання і запису, мінімальне навантаження на оперативну пам’ять та компактність файлового сховища [20–23].

Оскільки функціональність системи збереження орієнтована на фіксацію результатів користувачів, логічну модель даних побудовано на основі однієї сутності – таблиці Leaderboard. Такий підхід забезпечує спрощення структури бази даних, мінімізує кількість SQL-операцій та дозволяє швидко виконувати запис і вибірку результатів без додаткового навантаження на ігровий рушій Unity. Специфікацію атрибутів таблиці наведено в табл. 2.2.

Таблиця 2.2

Специфікація атрибутів сутності Leaderboard

| Назва поля (атрибуту) | Тип даних у СУБД | Ключ / обмеження | Опис функціонального призначення атрибуту |
|---|---|---|---|
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | Унікальний ідентифікатор запису ігрової сесії. Значення автоматично збільшується під час додавання нового рекорду. |
| player_name | TEXT | NOT NULL | Ім’я або нікнейм користувача, введений у полі Input Field після завершення гри. |
| score | INTEGER | NOT NULL | Підсумковий результат користувача, що використовується для формування таблиці рекордів та |

| | | | |
|---|---|---|---|
| | | | сортування записів. |
| date_achieved | TEXT | NOT NULL | Дата та час збереження результату ігрової сесії. |

Використання декількох взаємопов’язаних таблиць для локального ігрового застосунку є недоцільним, оскільки потребувало б виконання додаткових операцій об’єднання даних (JOIN) та ускладнювало б програмну логіку. Однотаблична структура забезпечує компактне зберігання інформації, зменшує обсяг операцій читання й запису та дозволяє швидко формувати таблицю рекордів.

Сутність Leaderboard містить інформацію про ім’я користувача, отриманий рахунок та дату збереження результату. Кожен запис відповідає окремій завершеній ігровій сесії, результати якої були збережені через інтерфейс Game Over.

Для формування таблиці рекордів використовується SQL-запит із сортуванням результатів за спаданням значення поля score та обмеженням кількості записів:

```
SELECT player_name, score, date_achieved
FROM Leaderboard
ORDER BY score DESC
LIMIT 10;
```

Використання SQL-сортування дозволяє отримувати вже впорядкований набір даних без додаткової обробки масивів у програмному коді Unity. Це зменшує навантаження на ігровий рушій та забезпечує швидке оновлення інтерфейсу таблиці рекордів.

Графічне представлення логічної структури бази даних наведено на рис. 2.2. ER-діаграма відображає структуру таблиці Leaderboard, її атрибути та ключові обмеження.

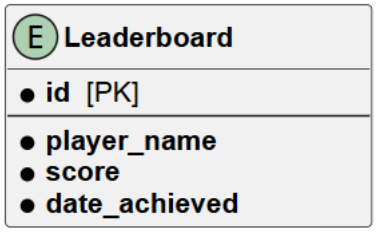



Рис. 2.2 Логічна модель даних (ER-діаграма) бази даних

Розроблена модель даних інтегрується з програмними компонентами Unity через бібліотеки платформи .NET та модулі SQLite [18–23]. Ініціалізація структури таблиці виконується автоматично під час першого запуску застосунку. Файл бази даних зберігається у локальній директорії Application.persistentDataPath, що забезпечує збереження інформації після повторного запуску або оновлення застосунку.

Використання SQLite також забезпечує підтримку транзакцій ACID, що гарантує цілісність даних навіть у разі аварійного завершення роботи програми. Завдяки компактній однотабличній структурі та використанню SQL-запитів система збереження результатів працює стабільно та не створює додаткового навантаження на основний ігровий процес.

## 2.3 Діаграма пакетів та діаграма класів

Програмна архітектура розроблюваного ігрового застосунку побудована на принципах об’єктно-орієнтованого програмування та модульного розподілу функціональності в середовищі Unity мовою C#. Такий підхід дозволяє відокремити логіку ігрового процесу, компоненти інтерфейсу, механізми процедурної генерації та систему збереження даних, що спрощує підтримку програмного коду та забезпечує стабільну роботу застосунку [1, 5, 7].

Для високорівневого опису структури програмного забезпечення було розроблено UML-діаграму пакетів, яка відображає взаємодію основних логічних модулів системи. Архітектура застосунку містить пакети Core Systems, Gameplay Modules, UI Components, Data Layer та External Plugins & Utilities. Подібний

розподіл дозволяє ізолювати окремі функціональні підсистеми та мінімізувати кількість залежностей між ними.

Пакет Core Systems містить центральні класи керування станом гри, обробкою подій завершення сесії та взаємодією між основними компонентами застосунку. Gameplay Modules об’єднує механізми процедурної генерації траси, системи керування автомобілем, генерації транспортного трафіку та допоміжні компоненти випадкової модифікації об’єктів. UI Components відповідає за роботу графічного інтерфейсу, формування таблиці рекордів та взаємодію користувача із системою. Data Layer забезпечує роботу з локальною базою даних SQLite, а пакет External Plugins & Utilities містить зовнішні бібліотеки та допоміжні утиліти, зокрема TextMesh Pro та компоненти SQLite.

На рис. 2.3 наведено UML-діаграму пакетів, яка відображає структуру модулів застосунку та взаємозв’язки між ними.

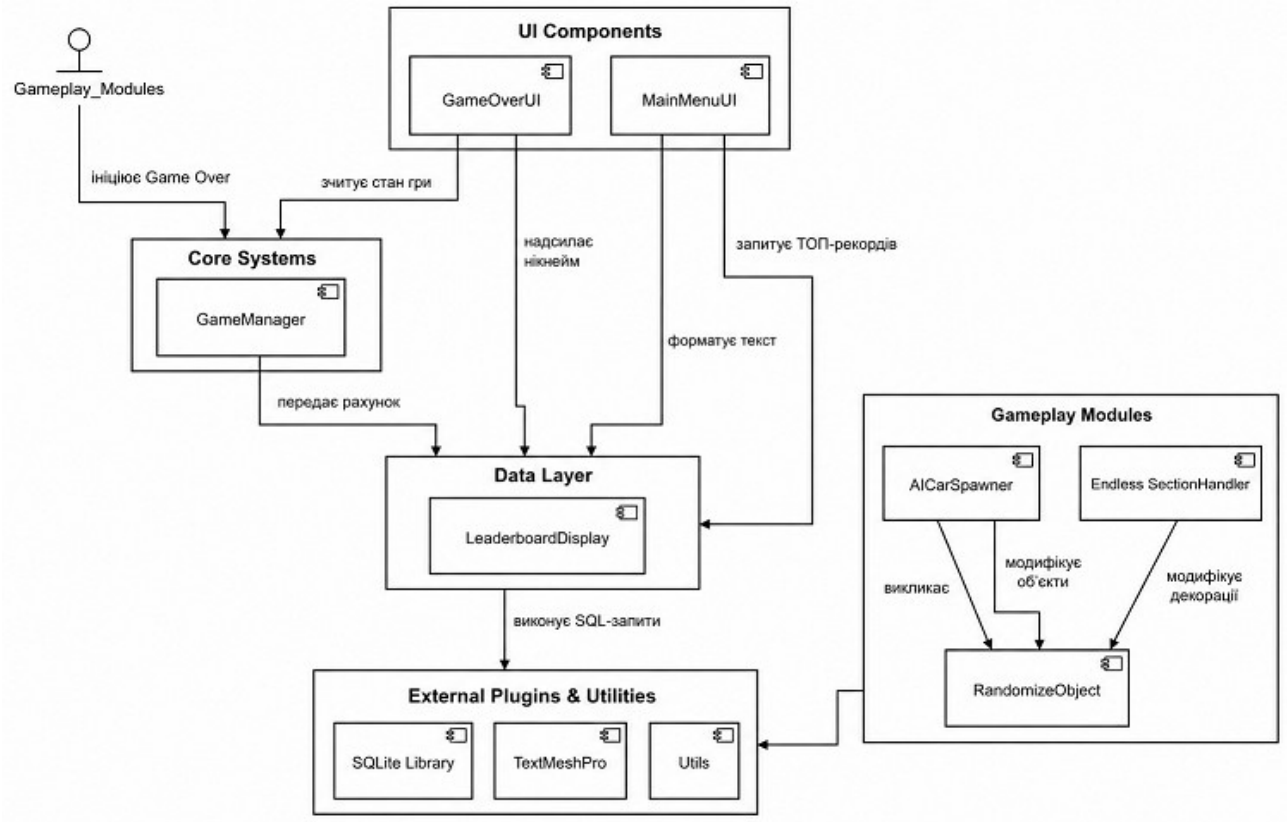



Рис. 2.3 Діаграма пакетів  ігрового застосунку

Подана діаграма демонструє, що взаємодія між пакетами побудована за ієрархічним принципом. Центральні системні компоненти координують ігровий

процес та передають дані до підсистеми збереження, тоді як інтерфейсні модулі виконують переважно функції відображення інформації та взаємодії з користувачем. Такий підхід дозволяє уникнути циклічних залежностей між компонентами та забезпечує модульність програмної архітектури.

Для деталізації внутрішньої структури програмного забезпечення було розроблено UML-діаграму класів, яка відображає основні класи системи, їх атрибути, методи та взаємозв'язки. Більшість класів застосунку успадковують базовий клас MonoBehaviour, що забезпечує інтеграцію компонентів у життєвий цикл Unity та підтримку обробки подій Update, Start і FixedUpdate [8, 12].

Діаграма класів також дозволяє формалізувати взаємодію між компонентами системи, визначити рівень залежностей між модулями та відобразити розподіл функціональної відповідальності між програмними класами.

Ключові програмні класи та їх функціональне призначення наведено в табл. 2.3.

Таблиця 2.3

Основні класи програмної системи

| Назва класу | Базовий клас | Функціональне призначення |
|---|---|---|
| GameManager | MonoBehaviour | Керування станом ігрової сесії та логікою завершення гри |
| CarHandler | MonoBehaviour | Керування рухом автомобіля гравця та обробка колізій |
| AICarSpawner | MonoBehaviour | Генерація транспортного трафіку та робота з пулом автомобілів |
| EndlessLevelHandler | MonoBehaviour | Процедурна генерація дорожніх сегментів |
| RandomizeObject | MonoBehaviour | Випадкова модифікація параметрів об'єктів середовища |
| LeaderboardDisplay | MonoBehaviour | Відображення таблиці рекордів у UI |
| DatabaseManager | MonoBehaviour | Робота з базою даних SQLite та виконання SQL-запитів |
| UIHandler | MonoBehaviour | Керування елементами графічного інтерфейсу |
| AIHandler | MonoBehaviour | Логіка руху автомобілів штучного інтелекту |
| ExplodeHandler | MonoBehaviour | Реалізація ефектів зіткнення та вибуху |
| Utils | Static class | Допоміжні математичні та сервісні методи |

Як показано в табл. 2.3, кожен клас реалізує окрему функціональну відповідальність, що відповідає принципам модульного проєктування. Основна логіка ігрового процесу зосереджена в класах GameManager та CarHandler, тоді як генерація середовища та транспортного трафіку виконується незалежними компонентами Gameplay Modules.

Взаємодія між класами побудована за принципом розподілу функцій між окремими компонентами системи. Керуючі класи координують загальний стан гри та обмін даними між підсистемами, тоді як допоміжні модулі виконують локальні операції, пов’язані з генерацією об’єктів, обробкою фізичних взаємодій, відображенням інтерфейсу та роботою з базою даних. Такий підхід спрощує підтримку програмного коду, підвищує масштабованість архітектури та дозволяє ізолювати критичні елементи продуктивності від компонентів візуалізації й збереження даних.

На рис. 2.4 наведено UML-діаграму класів, яка відображає структуру програмних компонентів та взаємозв’язки між ними.

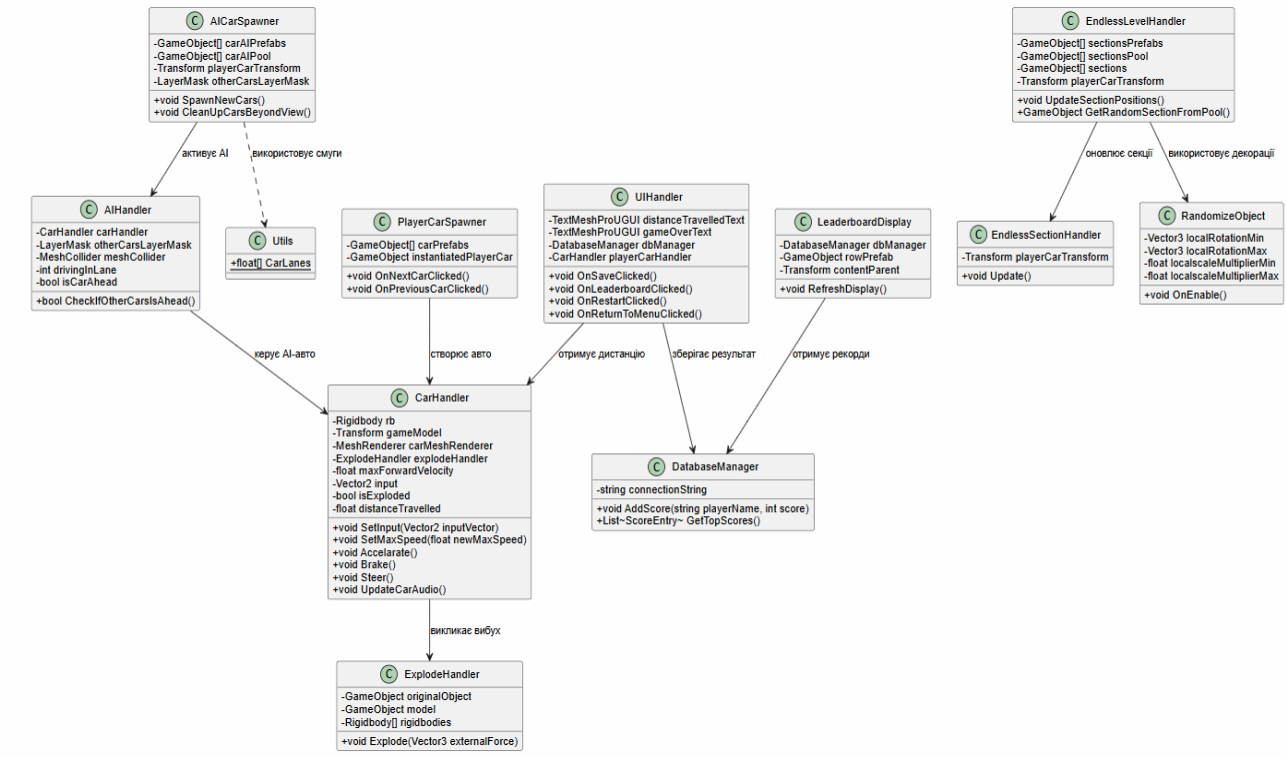



Рис. 2.4 Діаграма класів програмних скриптів гри

Діаграма демонструє взаємодію між основними класами застосунку. Клас CarHandler координує керування автомобілем користувача та взаємодіє з компонентами AIHandler і ExplodeHandler. Генерація транспортних засобів та дорожніх сегментів виконується класами AICarSpawner і EndlessLevelHandler із використанням механізмів Object Pool. Компоненти UIHandler і LeaderboardDisplay забезпечують взаємодію користувача з інтерфейсом та отримання результатів із локальної бази даних через DatabaseManager.

Важливим елементом архітектури є ізоляція підсистеми роботи з даними від основної логіки геймплею. Класи інтерфейсу та системи керування звертаються до DatabaseManager лише під час запису або отримання результатів, що дозволяє уникнути надлишкової залежності між компонентами. Аналогічно модулі генерації середовища функціонують автономно та взаємодіють між собою через події та посилання на компоненти Unity.

Таким чином, побудована архітектура застосунку забезпечує модульність, масштабованість та підтримку стабільної роботи системи під час виконання процедурної генерації середовища та обробки великої кількості динамічних об’єктів. UML-діаграми пакетів і класів дозволяють формалізувати структуру програмного забезпечення та відобразити взаємодію основних компонентів розроблюваного застосунку.

## 2.4 Діаграма кооперацій та діаграма компонентів

Для аналізу динамічної взаємодії між програмними компонентами та опису фізичної структури програмного забезпечення під час проєктування застосунку використано UML-діаграму кооперацій і UML-діаграму компонентів. Використання цих типів моделей дозволяє відобразити як механізм обміну повідомленнями між об’єктами системи під час виконання ігрового процесу, так і структуру програмних модулів, бібліотек та зовнішніх ресурсів, що формують архітектуру застосунку [1, 5, 7].

Діаграма кооперацій використовується для моделювання взаємодії між об’єктами системи під час виконання певного сценарію. На відміну від діаграми послідовності, основна увага приділяється не часовому порядку виконання операцій, а структурі зв’язків між об’єктами та повідомленням, якими вони обмінюються. Для розроблюваного застосунку основним сценарієм взаємодії є завершення ігрової сесії після виникнення колізії, зупинення генерації об’єктів та збереження результату користувача до бази даних SQLite.

Основними учасниками кооперації є компоненти PlayerCollision, CarHandler, AICarSpawner, GameOverPanel та DatabaseManager. Після виявлення зіткнення компонент PlayerCollision передає повідомлення про виникнення колізії до CarHandler. Далі система зупиняє ігровий процес, припиняє генерацію транспортного трафіку та активує панель завершення гри. У разі підтвердження збереження результату компонент DatabaseManager формує SQL-запит до локальної бази даних SQLite та виконує запис інформації про завершену сесію.

Графічне представлення взаємодії між основними об’єктами наведено на рис. 2.5.

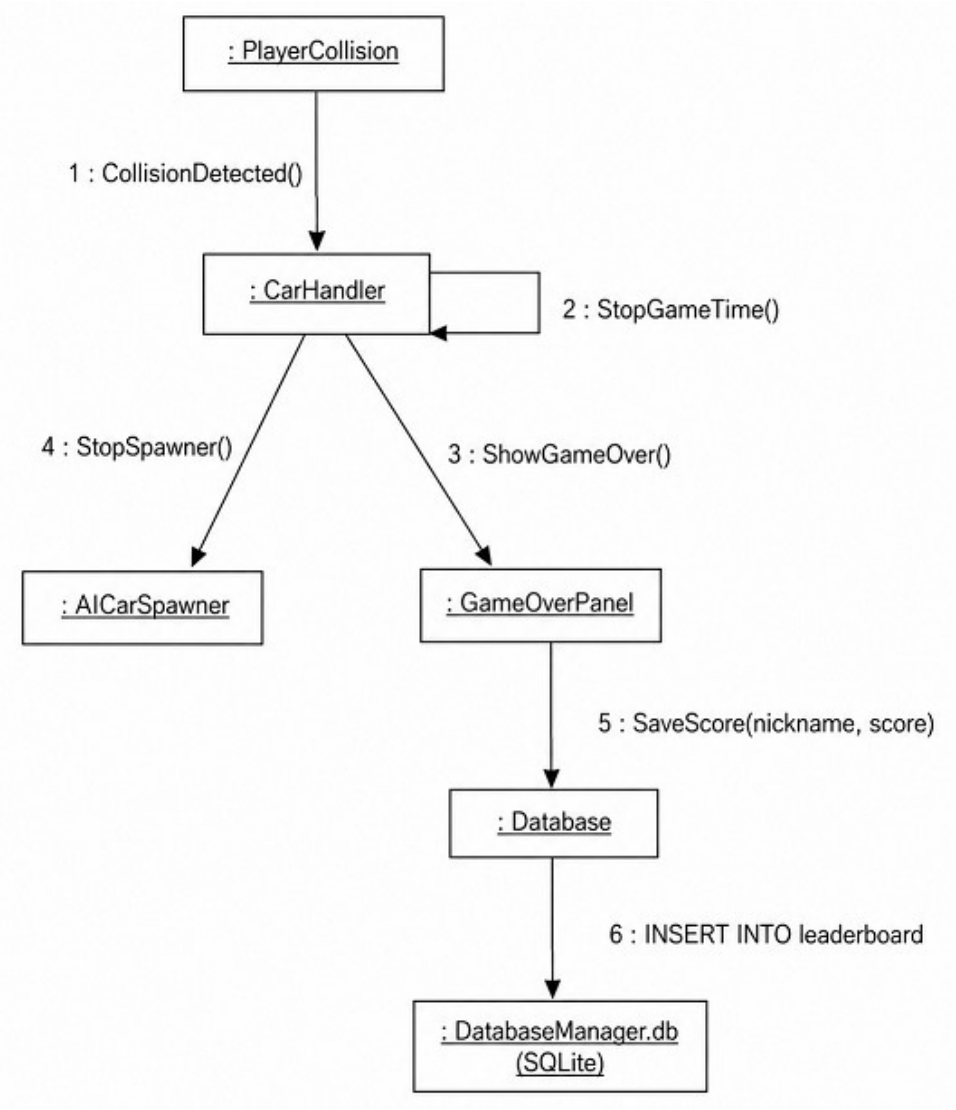



Рис. 2.5 Діаграма кооперацій ігрового циклу застосунку

Наведена діаграма демонструє послідовність обміну повідомленнями між компонентами системи після виникнення події зіткнення. Центральним координатором взаємодії виступає клас CarHandler, який керує переходом гри до стану Game Over та ініціює виклик інших підсистем. Подібний підхід дозволяє ізолювати логіку завершення гри від механізмів генерації середовища та роботи з базою даних.

Для аналізу фізичної структури програмного забезпечення та взаємозалежностей між програмними модулями було розроблено UML-діаграму компонентів. Вона відображає структуру програмних пакетів, зовнішніх

бібліотек, ресурсних модулів та інтерфейсних компонентів, що використовуються під час роботи застосунку.

Архітектура застосунку містить декілька основних компонентів: Presentation Layer, Core Gameplay Components, Data Access Components, Graphics & Resource Packs та External Utilities. Компонент Presentation Layer відповідає за реалізацію користувацького інтерфейсу та взаємодію користувача із системою. Core Gameplay Components містить логіку керування автомобілем, генерації транспортного трафіку та процедурного формування рівня. Data Access Components забезпечує роботу з локальною базою даних SQLite та реалізацію SQL-запитів. Graphics & Resource Packs містить тривимірні моделі, префаби та допоміжні ресурси, а External Utilities включає бібліотеки TextMesh Pro, SQLite та допоміжні утиліти.

Основні залежності між компонентами системи наведено в табл. 2.4.

Таблиця 2.4

Матриця взаємозв’язків компонентів системи

| Компонент | Залежний компонент | Тип взаємодії | Призначення взаємодії |
|---|---|---|---|
| Presentation Layer | Core Gameplay Components | Залежність | Передавання команд запуску та перезапуску гри |
| Presentation Layer | TextMesh Pro | Використання | Формування текстових елементів інтерфейсу |
| Core Gameplay Components | Graphics & Resource Packs | Агрегація | Використання low-poly моделей та префабів |
| Core Gameplay Components | Data Access Components | Залежність | Передавання результатів користувача |
| Data Access Components | SQLite Library | Інтеграція | Виконання SQL-запитів та робота з базою даних |
| Data Access Components | DatabaseManager.db | Фізичний запис | Збереження результатів у локальному файлі БД |
| Gameplay Modules | External Utilities | Використання | Допоміжні математичні та сервісні функції |

Наведена в табл. 2.4 структура залежностей демонструє, що взаємодія між компонентами побудована за принципом модульності та мінімізації жорстких зв’язків. Ігрові модулі взаємодіють із системою збереження даних лише через спеціалізований компонент Data Access Components, що дозволяє ізолювати SQL-логіку від механізмів геймплею.

На рис. 2.6 наведено UML-діаграму компонентів, яка відображає фізичну структуру програмного забезпечення та взаємодію між його модулями.

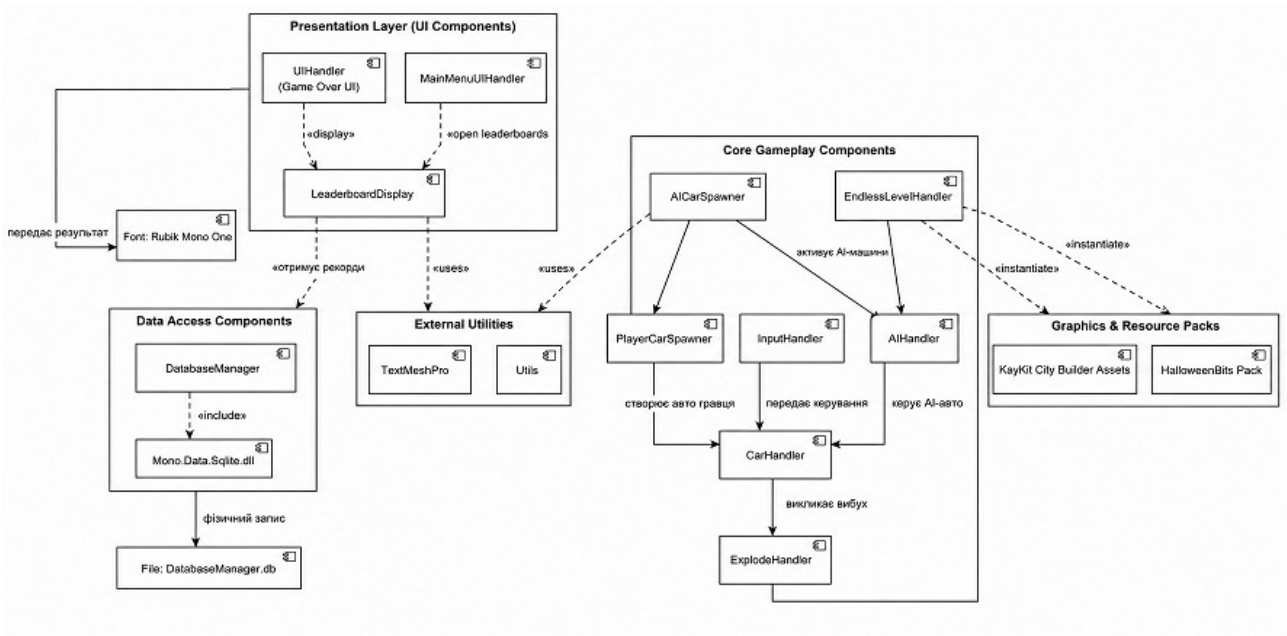



Рис.2.6 Діаграма компонентів ігрового застосунку

Діаграма компонентів демонструє, що архітектура застосунку організована за принципом розподілу функціональності між незалежними модулями. Центральне місце займає ядро геймплею, яке взаємодіє з підсистемою інтерфейсу, графічними ресурсами та модулем доступу до даних. Такий підхід забезпечує гнучкість програмної структури, спрощує масштабування функціональності та дозволяє підтримувати стабільну роботу застосунку навіть під час інтенсивного оновлення ігрового середовища.

# 3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ОПТИМІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ

## 3.1 Реалізація механіки нескінченної генерації перешкод та середовища

Одним із ключових технічних завдань під час розробки ігрових застосунків жанру «Endless Runner» є забезпечення безперервного формування ігрового середовища без критичного навантаження на оперативну пам’ять і центральний процесор. Постійне створення та видалення великої кількості тривимірних об’єктів під час активного геймплею призводить до збільшення кількості викликів збирача сміття середовища Mono/.NET, що негативно впливає на стабільність частоти кадрів та викликає мікрозатримки відображення [1, 7, 12].

Для усунення зазначеної проблеми в застосунку реалізовано механізм нескінченної генерації середовища на основі патерна Object Pool. Його використання дозволяє створювати необхідні об’єкти лише один раз під час ініціалізації сцени та повторно використовувати їх у процесі гри шляхом активації та деактивації компонентів. Такий підхід суттєво зменшує кількість операцій виділення пам’яті та забезпечує стабільну продуктивність під час тривалих ігрових сесій.

Центральним компонентом реалізації системи генерації перешкод є клас AICarSpawner, який відповідає за створення та повторне використання транспортних засобів штучного інтелекту. Під час запуску сцени в методі Start() виконується ініціалізація пулу автомобілів carAIPool. Кожен елемент пулу створюється на основі відповідного префабу та переводиться в неактивний стан за допомогою методу SetActive(false). Після цього запускається асинхронна корутина UpdateLessOftenCO(), яка циклічно виконує перевірку положення об’єктів та оновлення ігрового середовища.

Фрагмент програмної реалізації ініціалізації пулу об’єктів та корутини оновлення наведено на рис. 3.1.

```
void Start()
{
    playerCarTransform = GameObject.FindGameObjectWithTag("Player").transform;


    int prefabsIndex = 0;

    for (int i = 0; i < carAIPool.Length; i++)
    {
        carAIPool[i] = Instantiate(carAIPrefabs[prefabsIndex]);
        carAIPool[i].SetActive(false);

        prefabsIndex++;

        if (prefabsIndex > carAIPrefabs.Length - 1)
            prefabsIndex = 0;
    }

    StartCoroutine(UpdateLessOftenCO());
}

1 reference
IEnumerator UpdateLessOftenCO()
{
    while (true)
    {
        CleanUpCarsBeyondView();
        SpawnNewCars();
        yield return wait;
    }

}
```

Рис. 3.1 – Програмна реалізація ініціалізації пулу та асинхронної корутини оновлення середовища

Використання корутин Unity замість стандартного методу Update() дозволяє зменшити навантаження на процесор шляхом виконання логіки генерації через визначені часові інтервали. У реалізованому алгоритмі корутина періодично викликає методи CleanUpCarsBeyondView() та SpawnNewCars(). Перший метод відповідає за пошук і деактивацію автомобілів, які залишили область видимості камери, а другий – за активацію нових об’єктів перед автомобілем користувача.

Для забезпечення стабільності роботи алгоритму генерації додатково реалізовано механізм контролю просторового розташування активних об’єктів. Перед повторною активацією автомобіля або елемента середовища система виконує перевірку можливого перетину колайдерів та аналізує доступність смуги руху. Це дозволяє уникнути накладання транспортних засобів, появи

перешкод у недопустимих координатах та виникнення критичних помилок фізичної моделі під час активної фази геймплею. Завдяки цьому генерація об’єктів відбувається рівномірно, а ігрове середовище зберігає логічну послідовність і стабільну динаміку руху.

Основні функції модулів генерації середовища наведено в табл. 3.1.

Таблиця 3.1

Функціональне призначення компонентів генерації середовища

| Компонент | Основне призначення | Реалізована функція |
|---|---|---|
| AICarSpawner | Генерація транспортного трафіку | Активація та деактивація автомобілів із пулу |
| EndlessLevelHandler | Формування дорожніх сегментів | Безперервне оновлення траси |
| RandomizeObject | Випадкова модифікація об’єктів | Зміна параметрів декорацій |
| Utils | Допоміжні математичні методи | Обчислення позицій та діапазонів |
| AIHandler | Логіка руху транспорту ШІ | Керування швидкістю та напрямком руху |

Як показано в табл. 3.1, генерація ігрового середовища реалізована через взаємодію декількох незалежних компонентів. Такий підхід дозволяє розділити функції між окремими модулями та зменшити складність програмного коду.

Для запобігання накладанню транспортних засобів один на одного перед активацією нового елемента пулу виконується перевірка області розміщення за допомогою колайдерів та маски шарів otherCarsLayerMask. Після успішного проходження перевірки об’єкт переміщується в нову позицію попереду автомобіля гравця та активується.

Окрему роль у формуванні динамічного середовища виконує компонент RandomizeObject, який забезпечує процедурну модифікацію параметрів декорацій та елементів сцени. Під час кожної активації об’єкта автоматично викликається метод OnEnable(), у якому випадковим чином змінюються координати, кути повороту або масштаб елементів середовища. Це дозволяє уникнути візуальної повторюваності траси та підвищує реіграбельність застосунку.

Візуальний результат роботи системи нескінченної генерації наведено на рис. 3.2.


Рис. 3.2 Візуальне відображення згенерованого нескінченного ігрового середовища

Наведене зображення демонструє формування безперервного дорожнього середовища із транспортними засобами та елементами міської інфраструктури. Генерація нових об’єктів виконується динамічно під час руху автомобіля користувача, при цьому деактивовані елементи повторно використовуються системою пулінгу.

Застосування механізму Object Pool у поєднанні з асинхронними корутинами Unity забезпечило стабільну роботу застосунку під час тривалих ігрових сесій. Використання повторного перевикористання об’єктів дозволило мінімізувати кількість операцій виділення пам’яті та уникнути мікрозатримок, пов’язаних із роботою збирача сміття. Реалізована система генерації забезпечує плавність геймплею та підтримує стабільну продуктивність навіть за значної кількості динамічних об’єктів на сцені.

## 3.2 Розробка штучного інтелекту для транспортних засобів

Однією з ключових складових ігрового процесу в застосунках жанру «Endless Runner» є система рухомих перешкод, яка формує динамічне середовище та забезпечує зростання складності гри. У розробленому застосунку цю функцію виконує підсистема транспортних засобів під керуванням штучного інтелекту, що створює автономний дорожній трафік та змушує користувача постійно аналізувати ситуацію на трасі, змінювати смуги руху й уникати зіткнень. На відміну від складних систем навігації з використанням алгоритмів пошуку шляху або NavMesh, реалізований механізм штучного інтелекту базується на спрощеній моделі лінійного руху транспортних засобів уздовж фіксованих дорожніх смуг [6, 8, 12]. Такий підхід є доцільним для аркадного застосунку, оскільки дозволяє зменшити навантаження на систему фізики та забезпечити стабільну роботу гри навіть за великої кількості одночасно активних об’єктів.

Центральним компонентом реалізації транспортного трафіку є клас AICarSpawner, який відповідає за генерацію автомобілів, вибір позицій появи та повторне використання об’єктів із пулу. Під час виконання методу SpawnNewCars() система виконує пошук доступного об’єкта в масиві carAIPool та перевіряє можливість його безпечного розміщення на одній із дорожніх смуг.

Фрагмент програмної реалізації алгоритму генерації автомобілів штучного інтелекту наведено на рис. 3.3.

```
void SpawnNewCars()
{
    if (Time.time - timeLastCarSpawned < 2)
        return;

    GameObject carToSpawn = null;

    //Find a car to spawn
    foreach (GameObject aiCar in carAIPool)
    {
        if (aiCar.activeInHierarchy)
            continue;

        carToSpawn = aiCar;
        break;

    }
```

Рис. 3.3 Програмна реалізація алгоритму безпечного спавну автомобілів штучного інтелекту

Як показано на рис. 3.3, алгоритм перевіряє часовий інтервал між появою нових автомобілів та виконує пошук неактивного елемента в пулі об’єктів. Після вибору відповідного екземпляра система може використовувати його для повторної активації без створення нового GameObject у пам’яті.

Перед активацією транспортного засобу виконується просторовий аналіз області появи. Для цього використовуються колайдери та маска шарів otherCarsLayerMask, що дозволяє уникнути накладання моделей або появи автомобілів у вже зайнятих координатах. У разі виявлення перешкоди генерація об’єкта відкладається до наступного циклу виконання корутини.

Після успішної перевірки автомобіль переводиться в активний стан та починає рух вздовж траси. Керування транспортними засобами здійснюється через компонент AIHandler, який модифікує положення об’єкта за допомогою Transform або Rigidbody у методі FixedUpdate(). Швидкість руху автомобілів ШІ змінюється відносно швидкості автомобіля користувача, що дозволяє створювати різні типи дорожніх ситуацій. Частина транспортних засобів рухається попутно, імітуючи повільний трафік, тоді як інші можуть рухатися назустріч, підвищуючи складність гри та швидкість реакції користувача.

Основні логічні стани транспортних засобів штучного інтелекту наведено в табл. 3.2.

Таблиця 3.2

Життєвий цикл транспортних засобів штучного інтелекту

| Стан об’єкта | Умова переходу | Викликаний метод | Результат виконання |
|---|---|---|---|
| Очікування в пулі | Ініціалізація сцени або деактивація | SetActive(false) | Об’єкт перебуває в неактивному стані |
| Перевірка позиції | Спрацювання корутини генерації | SpawnNewCars() | Аналіз вільної дорожньої смуги |
| Активація | Успішна перевірка координат | SetActive(true) | Автомобіль з’являється на сцені |
| Активний рух | Виконання FixedUpdate() | AIHandler | Автомобіль рухається вздовж траси |
| Реєстрація колізії | Зіткнення з автомобілем гравця | OnCollisionEnter() | Ініціалізація стану Game Over |
| Деактивація | Вихід за межі видимості | CleanUpCarsBeyondView() | Об’єкт повертається до пулу |

Наведена в табл. 3.2 модель життєвого циклу демонструє, що всі транспортні засоби функціонують за принципом повторного використання об’єктів. Такий підхід забезпечує стабільну витрату пам’яті та мінімізує кількість операцій створення і видалення GameObject.

Візуальний результат роботи системи транспортного трафіку наведено на рис. 3.4.


Рис. 3.4 Візуалізація функціонування транспортних засобів під керуванням ШІ на гральних смугах

На рис. 3.4 показано приклад функціонування автомобілів штучного інтелекту на різних смугах руху. Генерація транспортних засобів виконується динамічно під час руху користувача, а повторне використання елементів пулу дозволяє підтримувати стабільну продуктивність застосунку навіть за значної щільності трафіку.

Реалізована система штучного інтелекту дозволила створити динамічне дорожнє середовище без використання складних навігаційних алгоритмів та ресурсоємних механізмів пошуку шляху. Використання лінійних траєкторій руху, просторової перевірки колізій та механізму Object Pool забезпечило стабільну роботу транспортного трафіку та підтримання високої швидкодії застосунку під час виконання ігрового процесу.

## 3.3 Налаштування графіки, конвеєру рендеренгу, фільтрів та візуальних ефектів

Візуальна складова є важливим елементом сприйняття ігрового застосунку, оскільки впливає на комфорт взаємодії користувача та швидкість розпізнавання об’єктів під час динамічного геймплею. Для жанру «Endless Runner» особливого значення набувають стабільність частоти кадрів і чіткість відображення перешкод, тому під час розробки було приділено увагу налаштуванню конвеєра рендерингу, освітлення та ефектів постобробки [12, 16].

Для реалізації графічної підсистеми використано Universal Render Pipeline, який забезпечує ефективніше використання ресурсів GPU порівняно зі стандартним Built-in Render Pipeline. Це дозволяє підтримувати стабільну продуктивність навіть при одночасному відображенні значної кількості динамічних об’єктів сцени.

Під час налаштування URP було оптимізовано параметри освітлення, згладжування та тіней. Для зменшення навантаження на графічну підсистему обмежено кількість додаткових джерел світла, знижено дистанцію відображення тіней та відключено надлишкові ефекти рендерингу. Основні параметри графічного профілю наведено в табл. 3.3.

Таблиця 3.3

Основні параметри графічного профілю Universal Render Pipeline

| Параметр | Значення | Призначення |
|---|---|---|
| HDR | Активовано | Покращення динамічного діапазону освітлення |
| MSAA | Disabled | Зменшення навантаження на GPU |
| Shadow Resolution | 1024 | Оптимізація якості тіней |
| Additional Lights | Per Pixel | Підтримка локального освітлення |
| Additional Lights Limit | 2 | Обмеження кількості джерел світла |

| | | |
|---|---|---|
| Shadow Distance | 50 | Контроль дальності відображення тіней |
| Cascade Count | 1 | Зменшення витрат рендерингу |
| Upscaling Filter | Automatic | Автоматична адаптація масштабування |

Наведені в табл. 3.3 параметри дозволили досягти балансу між якістю зображення та стабільною продуктивністю застосунку. Особливу увагу було приділено оптимізації системи тіней та освітлення, оскільки саме ці компоненти найбільше впливають на навантаження графічного процесора.

Профіль налаштувань Universal Render Pipeline, використаний у застосунку, наведено на рис. 3.5.

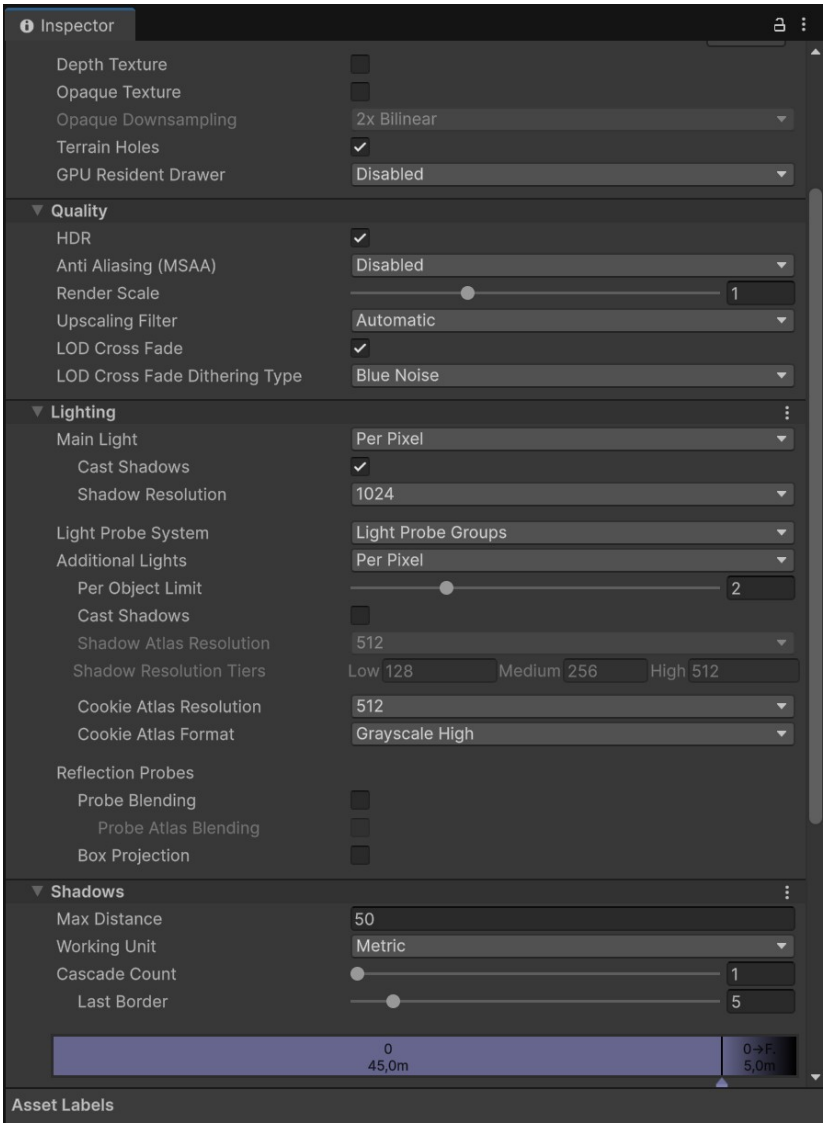


Рис. 3.5 Профіль налаштувань конвеєру Universal Render Pipeline розроблюваної гри

Для забезпечення стилістичної цілісності low-poly середовища всі матеріали об’єктів були переведені на стандартні шейдери URP. Освітлення сцени реалізовано за допомогою джерела типу Directional Light, яке формує м’яке рівномірне освітлення дорожнього середовища та забезпечує достатню контрастність між транспортними засобами й елементами траси. Додатково використовувались Reflection Probes та налаштування Global Illumination для покращення сприйняття матеріалів без суттєвого збільшення навантаження на GPU.

Для фінальної художньої обробки зображення використано систему постобробки URP Volume. Використання ефектів постобробки дозволило підвищити візуальну виразність сцени без необхідності застосування складних моделей освітлення. Основні візуальні ефекти наведено в табл. 3.4.

Таблиця 3.4

Використані ефекти постобробки зображення

| Ефект | Призначення | Візуальний результат |
|---|---|---|
| Bloom | Підсвічування яскравих об’єктів | Формування світіння навколо освітлених елементів |
| Color Grading | Корекція кольорової палітри | Посилення контрастності та насиченості сцени |
| Vignette | Затемнення країв екрана | Концентрація уваги користувача на центрі сцени |
| Motion Blur | Розмиття руху | Підсилення відчуття швидкості |
| Tone Mapping ACES | Кінематографічна корекція кольору | Формування більш природної палітри освітлення |

Застосування ефекту Bloom дозволило створити м’яке світіння навколо яскравих об’єктів сцени, зокрема фар автомобілів та елементів інтерфейсу. Для корекції кольорової палітри використано режим ACES Tone Mapping, який забезпечує більш кінематографічне відображення кольорів та покращує контрастність low-poly середовища. Ефект Motion Blur використовується для підсилення відчуття швидкості руху, а Vignette концентрує увагу користувача на центральній області екрана.

Фінальний візуальний результат після застосування конвеєра URP та ефектів постобробки наведено на рис. 3.6.


Рис 3.6 Фінальний візуальний вигляд гри після застосування фільтрів та ефектів пост-обробки

Наведене зображення демонструє підсумковий вигляд головної сцени застосунку після налаштування освітлення, матеріалів та системи постобробки. Використання low-poly графіки у поєднанні з URP дозволило забезпечити високу чіткість елементів середовища та стабільну продуктивність навіть під час активного руху сцени.

Реалізована графічна підсистема забезпечує ефективне використання ресурсів графічного процесора та підтримує стабільну частоту кадрів під час виконання ігрового процесу. Використання Universal Render Pipeline, оптимізованих налаштувань освітлення та системи постобробки дозволило

сформувати цілісний візуальний стиль застосунку та підвищити якість сприйняття ігрового середовища користувачем.

## 3.4 Реалізація інтерфейсу користувача(UI) та інтеграція кастомних шрифтів

Інтерфейс користувача є одним із ключових компонентів інтерактивного програмного забезпечення, оскільки забезпечує взаємодію користувача з функціональними можливостями системи та впливає на загальне сприйняття ігрового процесу. Для застосунків жанру «Endless Runner» важливо забезпечити мінімальне перевантаження екранного простору, оскільки надлишок графічних елементів ускладнює сприйняття динамічної сцени та знижує комфорт керування транспортним засобом. Тому під час розробки інтерфейсу основну увагу було приділено компактності елементів, високій читабельності тексту та швидкому доступу до основних функцій гри [8, 12, 17].

Для реалізації графічного інтерфейсу використано систему Unity UI (uGUI) у поєднанні з TextMeshPro, який забезпечує високоякісний рендеринг тексту та підтримку кастомних шрифтів. Архітектура інтерфейсу побудована за модульним принципом, що дозволяє окремо керувати станами головного меню, ігрової сцени та панелі завершення гри. Перемикання між екранами виконується централізовано через програмний модуль GameManager, який активує або деактивує відповідні UI-панелі залежно від поточного стану застосунку.

Під час активної ігрової сесії на екрані відображаються лише найважливіші елементи інтерфейсу – лічильник набраних балів та інформація про поточний стан гри. Після фіксації зіткнення транспортного засобу система переводить застосунок у стан Game Over, зупиняє ігровий процес та активує відповідну панель взаємодії з користувачем.

Зовнішній вигляд вікна завершення ігрової сесії наведено на рис. 3.7.


Рис. 3.7 Графічний інтерфейс вікна завершення ігрової сесії (Game Over)

Як показано на рис. 3.7, панель завершення гри містить поле відображення результату, елемент введення нікнейму користувача та набір функціональних кнопок для збереження рекорду, повторного запуску гри, відкриття таблиці лідерів і повернення до головного меню. Така структура інтерфейсу забезпечує швидку взаємодію користувача із системою без необхідності переходу між додатковими екранами.

Основні компоненти інтерфейсу користувача та їх функціональне призначення наведено в табл. 3.5.

Таблиця 3.5

Основні компоненти інтерфейсу користувача застосунку

| Компонент UI | Призначення |
|---|---|
| Main Menu | Головне меню запуску гри та навігації |
| Score Counter | Відображення поточного результату |
| Game Over Panel | Панель завершення ігрової сесії |
| Input Field | Введення нікнейму користувача |
| Leaderboards Panel | Відображення таблиці рекордів |
| Buttons UI | Керування функціями застосунку |

Програмна обробка натискань кнопок реалізована через систему UnityEvent, що дозволяє зв’язувати UI-компоненти з відповідними методами програмної логіки без використання складних механізмів ручної обробки подій. Такий підхід забезпечує гнучкість модифікації інтерфейсу та спрощує підтримку програмного коду.

Важливим етапом реалізації UI стала інтеграція кастомного шрифту Rubik Mono One Regular, який використовується для відображення заголовків, кнопок та динамічних текстових елементів інтерфейсу. Використання стандартних растрових шрифтів Unity призводить до втрати чіткості тексту під час масштабування або зміни роздільної здатності екрана. Для усунення цього недоліку шрифт було конвертовано у формат TMP Font Asset за допомогою внутрішнього інструменту TextMeshPro Font Asset Creator.

У процесі генерації шрифтового ресурсу застосовано технологію Signed Distance Field (SDF), яка забезпечує векторoподібне відображення символів незалежно від масштабу тексту. На відміну від класичних растрових шрифтів, метод SDF формує текстурний атлас відстаней до меж символів, що дозволяє шейдерам TextMeshPro відтворювати гладкі контури літер без ефекту пікселізації.

Параметри згенерованого SDF-атласу кастомного шрифту наведено на рис. 3.8.


Рис.3.8 Параметри SDF-атласу кастомного шрифту в середовищі Unity

Використання TextMeshPro також дозволило реалізувати додаткові стилістичні ефекти тексту, зокрема контур, зовнішню тінь та підсилення контрастності символів. Це забезпечило високу читабельність інтерфейсу навіть на фоні динамічного low-poly середовища та під час швидкого переміщення сцени.

Для адаптації інтерфейсу під різні роздільні здатності екранів використано систему Canvas Scaler, яка автоматично масштабує UI-елементи залежно від параметрів дисплея. Завдяки цьому інтерфейс коректно відображається як у стандартній Full HD роздільній здатності, так і на Ultra HD екранах без втрати якості тексту або деформації елементів керування.

Реалізована система користувацького інтерфейсу забезпечує зручну навігацію між функціональними модулями застосунку, стабільне відображення графічних елементів та високу чіткість текстового контенту. Інтеграція TextMeshPro та технології Signed Distance Field дозволила сформувати стилістично цілісний інтерфейс, який відповідає загальній low-poly концепції гри та забезпечує комфортну взаємодію користувача із застосунком.

## 3.5 Робота зі звуковим супроводом та аудіо-ефектами

Звуковий супровід є важливою складовою інтерактивного середовища комп’ютерної гри, оскільки забезпечує додатковий канал взаємодії користувача з програмною системою та суттєво впливає на емоційне сприйняття ігрового процесу. Для динамічних застосунків жанру «Endless Runner» аудіосистема виконує не лише декоративну, а й інформаційну функцію, сигналізуючи про зміну стану гри, небезпечні ситуації та взаємодію об’єктів сцени [6, 9, 12].

У розробленому застосунку підсистема звукового супроводу реалізована засобами вбудованого аудіорушія Unity із використанням компонентів AudioListener, AudioSource та AudioMixer. Такий підхід дозволяє забезпечити централізоване керування звуковими потоками, гнучке налаштування параметрів відтворення та оптимальне використання ресурсів під час виконання гри.

Компонент AudioListener виконує функцію віртуального приймача звуку та розташований на головній ігровій камері. Усі звукові сигнали, що генеруються AudioSource-компонентами, передаються через AudioListener до аудіопідсистеми операційної системи. Для відтворення звукових ефектів до ігрових об’єктів транспортних засобів, елементів інтерфейсу та тригерів колізій було додано AudioSource, який забезпечує динамічне відтворення аудіокліпів залежно від подій геймплею.

Під час реалізації аудіосистеми використано режим тривимірного просторового звучання (3D Spatial Audio), що дозволяє враховувати положення джерела звуку відносно камери користувача. Для більшості рухомих транспортних засобів активовано логарифмічне затухання гучності залежно від дистанції до гравця. Це створює додатковий ефект просторової присутності та покращує орієнтацію користувача в ігровому середовищі.

Основні звукові події та їх функціональне призначення наведено в табл. 3.6.

Таблиця 3.6

Основні звукові ефекти ігрового застосунку

| Тип звукового ефекту | Призначення |
|---|---|
| Звук двигуна автомобіля | Формування відчуття безперервного руху |
| Скрип шин | Супровід різкого маневрування |
| Звуковий сигнал (Horn) | Попередження про небезпечну ситуацію |
| Звук зіткнення | Фіксація аварії та завершення гри |
| Звуки кнопок UI | Підтвердження взаємодії з інтерфейсом |
| Фоновий аудіосупровід | Формування атмосфери ігрового процесу |

Як показано в табл. 3.6, звукові ефекти охоплюють як елементи ігрового процесу, так і взаємодію користувача з інтерфейсом. Такий підхід забезпечує

цілісність аудіосередовища та підсилює зворотний зв’язок між користувачем і системою.

Для централізованого керування звуковими потоками застосовано Audio Mixer, у якому створено окремі групи для звукових ефектів та фонового супроводу. Це дозволяє незалежно регулювати рівень гучності окремих категорій аудіо та спрощує подальше масштабування функціональності застосунку. Крім того, використання Audio Mixer забезпечує підтримку цифрової обробки сигналу, включаючи зміну рівня гучності, компресію та фільтрацію звуку.

Програмне відтворення аудіо інтегроване безпосередньо в логіку основних класів застосунку. Для циклічних звуків, таких як робота двигуна автомобіля, використовується режим Loop, який забезпечує безперервне відтворення аудіокліпу протягом ігрової сесії. Для короткочасних ефектів застосовано метод PlayOneShot(), який дозволяє відтворювати окремі звуки без переривання основного аудіопотоку.

Використання PlayOneShot() є доцільним з погляду оптимізації пам’яті, оскільки цей метод не створює додаткових екземплярів AudioSource та не потребує динамічного дублювання аудіокомпонентів у сцені. Завдяки цьому зменшується ризик надлишкового використання оперативної пам’яті та виключається накопичення невикористовуваних аудіооб’єктів під час тривалих ігрових сесій.

Для зменшення навантаження на систему всі аудіофайли було попередньо оптимізовано. Короткі звукові ефекти імпортовано у форматі WAV із мінімальною затримкою відтворення, тоді як тривалі фонові композиції використовують стиснення Ogg Vorbis. Такий підхід дозволив зменшити розмір ресурсів проєкту та забезпечити стабільну роботу аудіопідсистеми без втрати якості звучання.

Реалізована система звукового супроводу забезпечує синхронізацію аудіоефектів із подіями ігрового процесу, формує цілісну атмосферу застосунку та підвищує рівень занурення користувача у віртуальне середовище. Інтеграція

просторового звучання, централізованого керування аудіопотоками та оптимізованих методів відтворення дозволила створити стабільну та ефективну аудіопідсистему, яка відповідає вимогам сучасних інтерактивних застосунків.

## 3.6 Інтеграція SQLite для виведення таблиці рекордів

Одним із важливих компонентів ігрових застосунків жанру «Endless Runner» є система збереження та відображення результатів користувачів. Наявність локальної таблиці рекордів підвищує мотивацію до повторного проходження гри та забезпечує накопичення статистики ігрових сесій. Для реалізації цієї підсистеми в застосунку використано реляційну СКБД SQLite, інтегровану безпосередньо в середовище Unity [20–23].

Взаємодія з базою даних реалізована засобами бібліотек Mono.Data.Sqlite та System.Data платформи .NET. База даних представлена локальним файлом, який автоматично створюється в директорії Application.persistentDataPath під час першого запуску гри. Це забезпечує автономність роботи застосунку без використання зовнішнього серверного програмного забезпечення.

Програмна архітектура підсистеми побудована на використанні класу DatabaseManager, який відповідає за відкриття з'єднання, виконання SQL-запитів та передавання результатів до графічного інтерфейсу Unity UI. Основною функцією класу є вибірка найкращих результатів із таблиці Leaderboard та їх відображення у вікні рекордів. Фрагмент програмної реалізації алгоритму вибірки даних із бази SQLite наведено на рис. 3.9.

```
public List<ScoreEntry> GetTopScores()
{
    List<ScoreEntry> scores = new List<ScoreEntry>();
    using (var connection = new SqliteConnection(connectionString))
    {
        connection.Open();
        using (var command = connection.CreateCommand())
        {
            command.CommandText = "SELECT name, score FROM Leaderboard ORDER BY score DESC LIMIT 10;";
            using (IDataReader reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    scores.Add(new ScoreEntry(reader.GetString(0), reader.GetInt32(1)));
                }
            }
        }
    }
    return scores;
}
```

Рис. 3.9 Програмний алгоритм вибірки та зчитування результатів із бази даних SQLite

Як показано на рис. 3.9, отримання результатів виконується через SQL-запит із сортуванням за спаданням значення score та обмеженням кількості записів оператором LIMIT 10. Використання такого підходу дозволяє мінімізувати обсяг вибірки та скоротити навантаження на оперативну пам’ять під час формування таблиці рекордів.

Основні SQL-операції, реалізовані в підсистемі SQLite, наведено в табл. 3.7.

Таблиця 3.7

Основні SQL-запити підсистеми таблиці рекордів

| Призначення операції | SQL-запит |
|---|---|
| Створення таблиці рекордів | CREATE TABLE IF NOT EXISTS Leaderboard (...) |
| Додавання нового результату | INSERT INTO Leaderboard (player_name, score, date_achieved) VALUES (...) |
| Отримання ТОП-10 результатів | SELECT name, score FROM Leaderboard ORDER BY score DESC LIMIT 10 |
| Видалення записів | DELETE FROM Leaderboard |
| Оновлення структури БД | ALTER TABLE Leaderboard ... |

Виконання запиту здійснюється через об’єкт SqliteCommand, після чого результати зчитуються за допомогою IDataReader. У процесі виконання циклу reader.Read() програма послідовно отримує текстове значення нікнейму користувача та числовий результат ігрової сесії. Отримані дані перетворюються у структури типу ScoreEntry та записуються до динамічного списку List<ScoreEntry>, який надалі використовується для формування елементів інтерфейсу.

Важливим аспектом реалізації є коректне керування підключеннями до бази даних. Для уникнення блокування файлу SQLite використано конструкцію

using, яка автоматично закриває з’єднання та звільняє ресурси після завершення виконання SQL-команд. Це дозволяє забезпечити стабільну роботу застосунку навіть при багаторазовому відкритті таблиці рекордів протягом однієї ігрової сесії.

Після завершення обробки даних виконується динамічне формування елементів таблиці рекордів у середовищі Unity UI. Для цього використовується набір TextMeshProUGUI-компонентів, які заповнюються отриманими значеннями нікнейму та рахунку. Відображення записів здійснюється у вертикальному контейнері з автоматичним компонуванням елементів інтерфейсу.

Візуальне відображення таблиці рекордів користувачів наведено на рис. 3.10.


Рис. 3.10 Візуальне відображення таблиці рекордів користувачів у графічному інтерфейсі гри

Як показано на рис. 3.10, інтерфейс таблиці рекордів реалізований у єдиному стилі з іншими UI-компонентами застосунку. Для відображення тексту використано кастомний шрифт Rubik Mono One через систему TextMeshPro, що забезпечує високу чіткість символів незалежно від роздільної здатності екрана.

Під час реалізації таблиці рекордів особливу увагу приділено швидкодії підсистеми. Оскільки вибірка даних виконується локально та містить обмежену кількість записів, затримки при відкритті вікна рекордів практично відсутні. Крім того, використання SQLite дозволяє гарантувати цілісність збережених даних і підтримує виконання транзакцій відповідно до стандарту ACID.

Реалізована підсистема SQLite забезпечує стабільне збереження результатів користувачів, швидке формування таблиці рекордів та ефективну інтеграцію з графічним інтерфейсом Unity. Використання локальної реляційної бази даних дозволило створити автономний механізм обробки результатів без додаткових серверних компонентів та забезпечити високу продуктивність роботи застосунку.

# 4. ТЕСТУВАННЯ ТА ДЕМОНСТРАЦІЯ РОБОТИ ЗАСТОСУНКУ

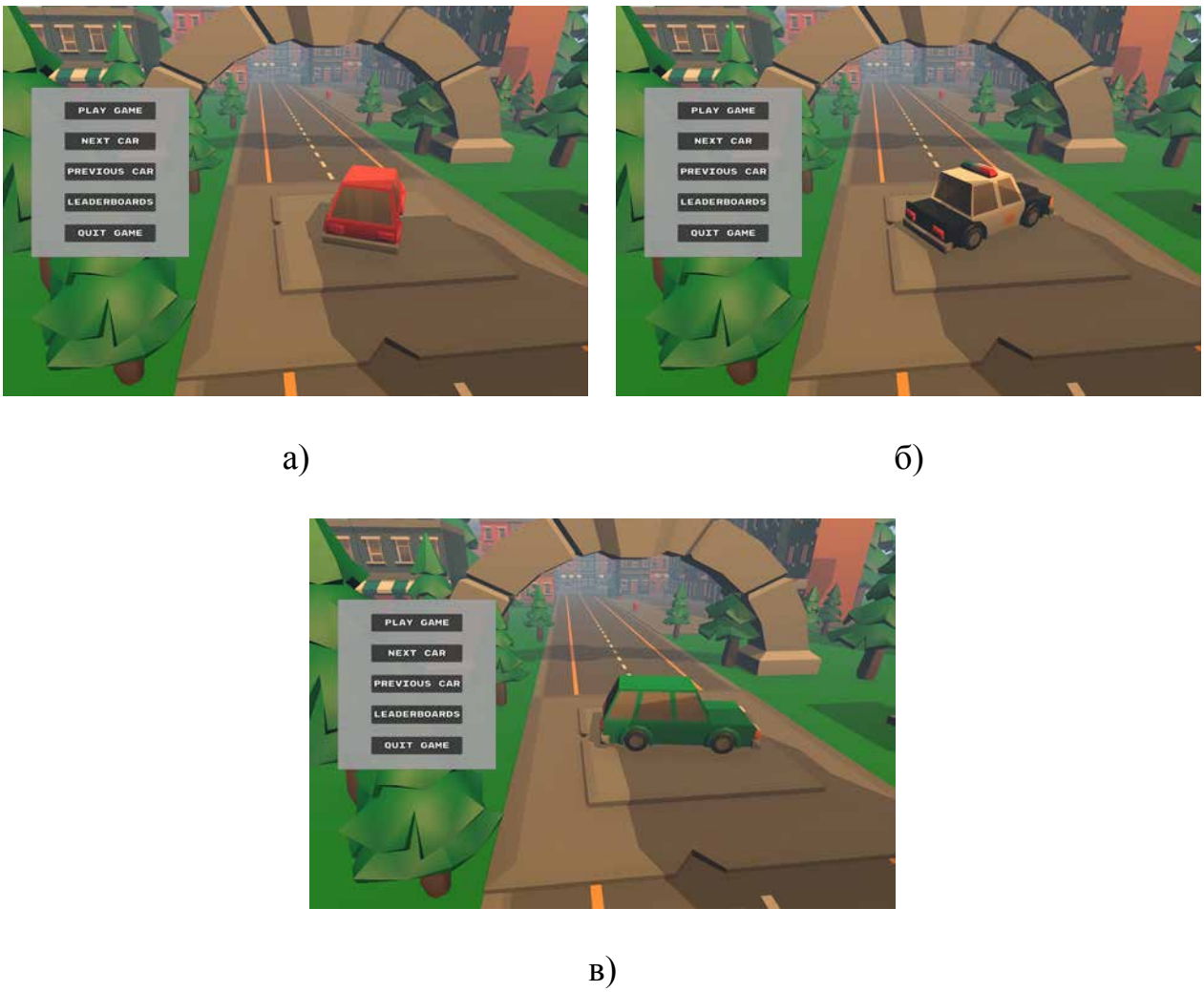
## 4.1 Методика та результати тестування ігрових механік і геймплею

Завершальним етапом розробки ігрового застосунку стало комплексне тестування програмних компонентів, механік геймплею, графічного інтерфейсу та підсистеми збереження результатів. Основною метою тестування була перевірка стабільності функціонування застосунку під час тривалих ігрових сесій, оцінювання коректності взаємодії між програмними модулями та підтвердження відповідності реалізованої системи функціональним вимогам, сформованим на етапі проєктування [8, 12, 20].

Тестування виконувалося на персональному комп’ютері під керуванням операційної системи Windows із використанням вбудованих засобів профілювання Unity Profiler та Runtime Debug Tools. Перевірка охоплювала механіку генерації середовища, функціонування транспортного трафіку, систему обробки колізій, UI-компоненти, відображення таблиці рекордів та роботу аудіопідсистеми. Особлива увага приділялася стабільності частоти кадрів, відсутності витоків пам’яті та коректності виконання SQL-запитів до SQLite.

Початковим етапом тестування стала перевірка роботи головного меню застосунку та механізму вибору транспортного засобу. Після запуску програми користувач переходить до стартового екрана, де доступні основні елементи керування: запуск гри, перемикання автомобілів, відкриття таблиці рекордів та завершення роботи застосунку. Перемикання між моделями транспортних засобів виконується без перезавантаження сцени шляхом динамічної активації відповідних префабів.

На рис. 4.1 наведено інтерфейс головного меню з реалізованою системою вибору автомобіля.




а) б)

в)

Рис. 4.1 Головне меню гри та система вибору транспортних засобів

Як показано на рис. 4.1, інтерфейс побудований у єдиному стилі з використанням технології TextMeshPro та кастомного шрифту Rubik Mono One. Під час тестування підтверджено коректне перемикання моделей автомобілів, відсутність затримок при оновленні сцени та стабільне відображення UI-елементів на різних роздільних здатностях екрана.

Наступним етапом стало тестування безпосереднього ігрового процесу. Перевірка виконувалася під час тривалих сесій із безперервною генерацією дорожніх сегментів, транспортних засобів та декоративних елементів середовища. Було підтверджено коректну роботу алгоритму Object Pool, який забезпечує повторне використання об’єктів без створення додаткових екземплярів у пам’яті.

Основні результати тестування механік геймплею наведено в табл. 4.1.

Таблиця 4.1

Результати тестування ігрових механік застосунку

| Компонент тестування | Очікуваний результат | Фактичний результат |
|---|---|---|
| Генерація дорожніх сегментів | Безперервне формування траси | Працює стабільно |
| Object Pool | Відсутність затримок і GC-сплесків | Підтверджено |
| Керування автомобілем | Плавне переміщення між смугами | Працює коректно |
| Генерація AI-трафіку | Випадкове створення перешкод | Виконується стабільно |
| Підрахунок очок | Безперервне оновлення рахунку | Працює без помилок |
| Collision Detection | Коректне визначення зіткнень | Підтверджено |
| Audio System | Синхронне відтворення SFX | Затримки відсутні |
| SQLite-запити | Збереження та вибірка рекордів | Працює стабільно |

Візуальне відображення основної ігрової сцени під час тестування наведено на рис. 4.2.


Рис. 4.2 Ігровий процес під час виконання тестової сесії

У процесі тестування підтверджено коректну роботу механіки переміщення автомобіля між смугами руху. Перехід між позиціями виконується за допомогою плавної інтерполяції координат, що забезпечує природний характер руху транспортного засобу без ефекту миттєвого телепортування. Під час багаторазових маневрів не було зафіксовано порушень фізичної моделі або некоректної взаємодії з колайдерами сцени.

Додатково перевірялася стабільність генерації транспортного трафіку та декоративних елементів середовища. Під час тестових запусків алгоритм випадкового розміщення об’єктів не створював конфліктів позиціонування або накладання моделей. Система очищення та повторної активації елементів пулу функціонувала без накопичення невикористовуваних об’єктів у пам’яті.

Окремий етап тестування був присвячений перевірці механіки завершення гри та взаємодії з локальною базою даних SQLite. Після зіткнення автомобіля гравця з перешкодою система коректно переводила гру до стану Game Over, блокувала подальший рух транспортного засобу та активувала модальне вікно завершення сесії.

Інтерфейс завершення гри наведено на рис. 4.3.


Рис. 4. Вікно завершення гри з відображенням результату та засобами збереження рекорду

Як показано на рис. 4.3, користувач отримує можливість ввести власний нікнейм та зберегти результат у локальній базі даних SQLite. Під час тестування було перевірено сценарії як із записом результату, так і без його збереження. В усіх випадках SQL-запити виконувалися коректно, а нові записи миттєво відображалися в таблиці рекордів після повторного відкриття відповідного вікна.

Під час проведення тестування також здійснювався моніторинг продуктивності застосунку. Аналіз показав стабільне використання оперативної пам’яті без різких стрибків навантаження, що підтверджує ефективність реалізованої системи Object Pool та оптимізованого конвеєру рендерингу Universal Render Pipeline. Частота кадрів залишалася стабільною навіть під час тривалого функціонування системи генерації середовища та AI-трафіку.

Проведене тестування підтвердило повну працездатність усіх основних функціональних компонентів гри, стабільність взаємодії між програмними модулями та коректну інтеграцію графічної, звукової та інформаційної підсистем застосунку. Отримані результати свідчать про відповідність

розробленого програмного продукту поставленим функціональним і технічним вимогам.

## 4.2 Вимоги до апаратного та програмного забезпечення

Стабільність роботи ігрового застосунку, швидкість завантаження сцен та коректне функціонування графічної й аудіопідсистем безпосередньо залежать від характеристик програмного та апаратного забезпечення цільового пристрою. Під час проєктування та тестування гри було виконано оптимізацію графічних ресурсів, системи генерації об’єктів та механізмів роботи з пам’яттю, що дозволило суттєво знизити навантаження на центральний процесор і графічний адаптер навіть під час тривалих ігрових сесій [8, 12].

Розроблений застосунок орієнтований на запуск у середовищі операційної системи Microsoft Windows із підтримкою сучасних графічних API та бібліотек Unity Runtime. Для забезпечення коректної роботи Universal Render Pipeline, системи постобробки та бібліотек SQLite необхідною є підтримка DirectX 11 або новіших версій графічного інтерфейсу. Додатково застосунок потребує наявності системних компонентів Microsoft Visual C++ Redistributable, які забезпечують коректне завантаження нативних бібліотек Mono.Data.Sqlite та роботу середовища виконання .NET.

Основні вимоги до програмного забезпечення наведено в табл. 4.2.

Таблиця 4.2

Вимоги до програмного забезпечення для запуску гри

| Компонент | Характеристика |
|---|---|
| Операційна система | Microsoft Windows 10 / Windows 11 (64-bit) |
| Середовище виконання | Unity Runtime (.NET / Mono) |
| Графічний API | DirectX 11 / DirectX 12 / Vulkan |
| Драйвери GPU | Актуальні драйвери з підтримкою Shader Model 5.0 |
| Системні бібліотеки | Microsoft Visual C++ Redistributable |

| | |
|---|---|
| Підсистема баз даних | SQLite через Mono.Data.Sqlite |

Як показано в табл. 4.2, програмна конфігурація застосунку не потребує встановлення додаткових серверних компонентів або зовнішніх СУБД, оскільки вся система збереження даних функціонує локально на основі SQLite. Це дозволяє суттєво спростити процес розгортання та запуску гри на цільовому пристрої користувача.

Під час формування апаратних вимог враховувались особливості реалізованого low-poly стилю, використання конвеєра Universal Render Pipeline, технології Object Pool та оптимізованих аудіоресурсів. Завдяки невеликій кількості полігонів моделей і мінімізації навантаження на GPU застосунок демонструє стабільну продуктивність навіть на комп’ютерах початкового рівня.

Додатково слід зазначити, що використання патерна Object Pool та оптимізованої системи генерації об’єктів дозволило суттєво знизити навантаження на оперативну пам’ять і центральний процесор під час активної фази геймплею. Повторне використання ігрових об’єктів замість їх постійного створення та видалення забезпечує стабільну частоту кадрів і мінімізує виникнення мікрозатримок, пов’язаних із роботою збирача сміття середовища .NET. Це особливо важливо для застосунків жанру «Endless Runner», у яких швидкість оновлення сцени безпосередньо впливає на комфорт ігрового процесу та коректність відображення динамічних перешкод.

Мінімальні та рекомендовані апаратні характеристики наведено в табл. 4.3.

Таблиця 4.3

Апаратні вимоги ігрового застосунку

| Компонент | Мінімальні вимоги | Рекомендовані вимоги |
|---|---|---|
| Центральний процесор (CPU) | Intel Core i3 / AMD Ryzen 3, 2 ядра, від 2.0 ГГц | Intel Core i5 / AMD Ryzen 5, 4 ядра, від 3.0 ГГц |

| | | |
|---|---|---|
| Оперативна пам’ять (RAM) | 4 ГБ | 8 ГБ |
| Відеокарта (GPU) | Інтегроване GPU або 1 ГБ VRAM з підтримкою DX11 | NVIDIA GTX 1050 / AMD RX 560 або вище |
| Відеопам’ять | 1 ГБ VRAM | 2 ГБ VRAM і більше |
| Дисковий простір | 200 МБ | SSD-накопичувач, 200 МБ |
| Аудіопідсистема | Сумісний аудіоадаптер | Аудіоадаптер із підтримкою багатоканального звуку |

Проведене тестування підтвердило, що навіть на мінімальній конфігурації застосунок підтримує стабільну частоту кадрів без критичних просідань продуктивності. Найбільше навантаження на систему створюють операції рендерингу тіней, постобробки та одночасне відображення великої кількості транспортних засобів AI-трафіку. Водночас застосування Universal Render Pipeline, обмеження роздільної здатності Shadow Maps та оптимізація корутин дозволили уникнути перевантаження графічної підсистеми.

Окремо слід відзначити невеликий розмір фінального програмного продукту. Компактність застосунку забезпечується використанням низькополігональних моделей, оптимізованих текстур, стиснення аудіофайлів у форматах WAV та Ogg Vorbis, а також застосуванням локальної бази даних SQLite, що фізично представлена одним файлом невеликого обсягу. Завдяки цьому інсталяція гри не створює значного навантаження на дискову підсистему та не потребує складних процедур розгортання.

## 4.3 Склад інсталяційного пакету та процес розгортання

Фінальним етапом розробки програмного продукту стало формування інсталяційного пакету та перевірка коректності розгортання застосунку на цільових пристроях користувачів. Під час компіляції проєкту в середовищі Unity

під платформу Windows x64 було виконано оптимізацію графічних ресурсів, текстур, аудіофайлів і бібліотек, що дозволило суттєво зменшити розмір кінцевого програмного продукту без втрати функціональності та якості візуального відображення. Загальний обсяг усіх файлів застосунку після фінальної збірки становить близько 30 МБ, що забезпечує швидке завантаження, мінімальні вимоги до дискового простору та зручність поширення програмного продукту [8, 12].

Архітектура застосунку побудована за принципом автономного локального виконання без використання зовнішніх серверних компонентів. Усі необхідні ресурси, бібліотеки та база даних SQLite постачаються разом із виконуваним файлом гри. Завдяки цьому програмний продукт не потребує складної інсталяції, додаткових налаштувань системи або постійного мережевого з’єднання. Основні компоненти інсталяційного пакету наведено в табл. 4.4.

Таблиця 4.4

Склад інсталяційного пакету ігрового застосунку

| Компонент | Призначення |
|---|---|
| RoadWay.exe | Основний виконуваний файл застосунку |
| RoadWay_Data | Каталог із ресурсами Unity, сценами, шейдерами, текстурами, аудіофайлами та UI-компонентами |
| sqlite3.dll | Динамічна бібліотека для взаємодії з SQLite |
| MonoBleedingEdge | Системні бібліотеки середовища виконання Mono/.NET |
| game_scores.db | Локальна база даних SQLite для збереження рекордів |
| Resources та StreamingAssets | Допоміжні каталоги з конфігураційними й мультимедійними ресурсами |

Як показано в табл. 4.4, усі компоненти програмного продукту розгортаються локально та функціонують як єдина автономна система. Особливістю реалізованої архітектури є автоматичне створення файлу бази

даних SQLite у разі його відсутності під час першого запуску гри, що дозволяє уникнути помилок ініціалізації та спрощує перенесення застосунку.

Процес розгортання гри було максимально спрощено для кінцевого користувача. Після завантаження архіву достатньо розпакувати його в будь-яку директорію локального накопичувача та запустити файл RoadWay.exe. Під час старту застосунку Unity автоматично ініціалізує всі необхідні бібліотеки, завантажує сцени, графічні ресурси та аудіокомпоненти, після чого користувач переходить до головного меню гри.

Для перевірки працездатності виконано тестові запуски на різних конфігураціях Windows із контролем коректності ініціалізації SQLite, Universal Render Pipeline та швидкості відкриття головної сцени.

Результати тестування процесу розгортання наведено в табл. 4.5.

Таблиця 4.5

Результати тестування інсталяційного пакету

| Параметр перевірки | Результат |
|---|---|
| Запуск виконуваного файлу | Успішний |
| Завантаження ресурсів Unity | Коректне |
| Ініціалізація SQLite | Виконано без помилок |
| Автоматичне створення БД | Підтверджено |
| Відображення UI-компонентів | Коректне |
| Робота аудіосистеми | Стабільна |
| Час запуску гри | До 5 секунд |
| Підтримка портативного запуску | Підтверджено |

Проведене тестування підтвердило стабільність роботи інсталяційного пакету та відсутність критичних помилок під час розгортання застосунку. Завдяки використанню локальної архітектури Unity та SQLite гра не потребує записів у системний реєстр Windows, встановлення серверних компонентів або надання прав адміністратора. Це забезпечує високу мобільність програмного

продукту та можливість його використання на широкому спектрі персональних комп’ютерів без додаткового налаштування системного середовища.

# ВИСНОВКИ

У бакалаврській кваліфікаційній роботі вирішено актуальне прикладне завдання з проєктування та програмної реалізації тривимірного ігрового застосунку жанру «Endless Runner» із використанням процедурної генерації середовища, механізмів оптимізації продуктивності та локального збереження результатів користувачів. У процесі виконання роботи досягнуто поставленої мети та реалізовано всі визначені завдання, пов’язані з аналізом предметної області, побудовою архітектури системи, програмною реалізацією функціональних модулів і тестуванням готового програмного продукту.

Проведений аналіз особливостей жанру «Endless Runner» показав, що застосунки такого типу характеризуються високою інтенсивністю оновлення сцени, значною кількістю динамічних об’єктів та підвищеними вимогами до стабільності роботи систем реального часу. На основі порівняльного аналізу сучасних ігрових рушіїв обґрунтовано вибір середовища Unity, яке забезпечує підтримку тривимірної графіки, модульної архітектури, засобів профілювання продуктивності та інтеграції локальної бази даних SQLite. Використання мови програмування C# дозволило реалізувати структуровану архітектуру програмного забезпечення з розподілом функціональності між окремими модулями.

У ході проєктування побудовано UML-діаграми та розроблено логічну модель даних на основі локальної таблиці Leaderboard. Використання SQLite забезпечило компактне автономне зберігання рекордів без необхідності підключення зовнішнього серверного середовища та дозволило реалізувати швидке сортування і вибірку результатів засобами SQL.

Практична реалізація застосунку охопила створення механіки безперервного руху транспортного засобу, процедурної генерації дорожнього полотна, перешкод і транспортного трафіку, а також системи перевірки колізій та завершення ігрової сесії. Для оптимізації роботи з великою кількістю динамічних об’єктів застосовано патерн Object Pool, який дозволив мінімізувати

витрати оперативної пам’яті та зменшити навантаження на механізм збору сміття Unity. Це забезпечило стабільну частоту кадрів і плавність ігрового процесу під час активного оновлення сцени.

Окрему увагу приділено реалізації графічної та мультимедійної складових застосунку. Використання low-poly моделей, конвеєра Universal Render Pipeline, технології TextMesh Pro та оптимізованих аудіоресурсів дозволило сформувати цілісне візуальне й звукове середовище зі збереженням високої продуктивності на комп’ютерах початкового рівня. Реалізований користувацький інтерфейс забезпечує зручний доступ до основних функцій застосунку, включаючи вибір автомобіля, запуск гри, перегляд таблиці рекордів та збереження результатів після завершення ігрової сесії.

Проведене тестування підтвердило коректність роботи всіх функціональних модулів застосунку, стабільність процедурної генерації середовища, правильність роботи системи колізій, підсистеми збереження результатів та аудіосупроводу. Під час тестових запусків не було виявлено критичних помилок, пов’язаних із роботою SQLite, завантаженням графічних ресурсів або функціонуванням основних сценаріїв взаємодії користувача із системою.

Розроблений застосунок є автономним програмним продуктом для операційної системи Windows, не потребує складного встановлення та може функціонувати у portable-режимі після розпакування архіву. Отримані результати мають практичну цінність для подальшого розвитку тривимірних ігрових систем реального часу, оптимізації процедурної генерації середовища та інтеграції локальних баз даних у застосунках, створених на базі Unity.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Матвєєв М. І., Кучук Г. А. Оптимізація мобільної гри в середовищі Unity. Системи управління, навігації та зв'язку. 2023. № 2(72). С. 131–134. DOI: 10.26906/SUNZ.2023.2.131.
2. Mikhailutsa O. M., Pozhuyev A. V., Wolik S. A. Analysis of procedural generation methods for 3D game content. Applied Questions of Mathematical Modelling. 2021. Vol. 4, No. 2.2. P. 128–136. DOI: 10.32782/KNTU2618-0340/2021.4.2.2.13.
3. Bezditnyi V., Chebanyuk O. Software Engineering Fundamentals to Design Application for Modern Game Engines. CEUR Workshop Proceedings. 2024. Vol. 3806. P. 89–99. URL: https://ceur-ws.org/Vol-3806/S_46_Bezditnyi_Chebanyuk.pdf.
4. Moiseienko N. V., Moiseienko M. V., Kuznetsov V. S., Rostalny B. A., Kiv A. E. Teaching computer game development with Unity engine: a case study. CEUR Workshop Proceedings. 2023. Vol. 3482. P. 237–251. URL: https://ceur-ws.org/Vol-3482/paper330.pdf.
5. Leshchuk S. O. Teaching programming for future information technology specialists through end-to-end game design. CEUR Workshop Proceedings. 2025. Vol. 3949. URL: https://ceur-ws.org/Vol-3949/paper17.pdf.
6. Bond J. G. Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#. 3rd ed. Boston : Addison-Wesley Professional, 2022. ISBN 978-0136619949.
7. Gregory J. Game Engine Architecture. 4th ed. Boca Raton : CRC Press, 2025. ISBN 978-1032443065.
8. Albahari J. C# 12 in a Nutshell: The Definitive Reference. Sebastopol : O'Reilly Media, 2024. ISBN 978-1098147440.
9. Freeman A. Pro ASP.NET Core 9. New York : Apress, 2025. DOI: 10.1007/979-8-8688-0756-5.
10. Sung K., Smith G. Basic Math for Game Development with Unity 3D. Berkeley : Apress, 2023. DOI: 10.1007/978-1-4842-9885-5.

11. Sung K., Smith G. Vectors. In: Basic Math for Game Development with Unity 3D. Berkeley : Apress, 2023. DOI: 10.1007/978-1-4842-9885-5_4.
12. Unity Technologies. Unity 6 User Manual. URL: https://docs.unity3d.com/Manual/
13. Unity Technologies. Unity Scripting API. URL: https://docs.unity3d.com/ScriptReference/
14. Unity Technologies. Object pooling tutorial. Unity Learn. URL: https://learn.unity.com/tutorial/introduction-to-object-pooling
15. Unity Technologies. TextMesh Pro Documentation. URL: https://docs.unity3d.com/Packages/com.unity.textmeshpro@latest
16. Unity Technologies. Universal Render Pipeline Documentation. URL: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@latest
17. Unity Technologies. Addressables Package Documentation. URL: https://docs.unity3d.com/Packages/com.unity.addressables@latest
18. Microsoft Corporation. C# documentation. Microsoft Learn. URL: https://learn.microsoft.com/dotnet/csharp/
19. Microsoft Corporation. .NET documentation. Microsoft Learn. URL: https://learn.microsoft.com/dotnet/
20. Microsoft Corporation. SQLite and EF Core documentation. Microsoft Learn. URL: https://learn.microsoft.com/ef/core/providers/sqlite/
21. SQLite Development Team. SQLite Documentation. URL: https://www.sqlite.org/docs.html
22. SQLite Development Team. SQLite C Interface Documentation. URL: https://www.sqlite.org/c3ref/intro.html
23. DB Browser for SQLite Team. DB Browser for SQLite Documentation. URL: https://sqlitebrowser.org/
24. Kay Lousberg. KayKit City Builder Pack. Itch.io. URL: https://kaylousberg.itch.io/kaykit-city-builder
25. Kay Lousberg. KayKit Halloween Bits Pack. Itch.io. URL: https://kaylousberg.itch.io/kaykit-halloween-bits

26. Google Fonts. Rubik Mono One Font Family. URL: https://fonts.google.com/specimen/Rubik+Mono+One
27. Unity Technologies. Unity Asset Store Publisher Portal Documentation. URL: https://docs.unity3d.com/Manual/AssetStore.html
28. Microsoft Corporation. .NET Garbage Collection Documentation. Microsoft Learn. URL: https://learn.microsoft.com/dotnet/standard/garbage-collection/
29. Unity Technologies. Profiling in Unity. Unity Manual. URL: https://docs.unity3d.com/Manual/Profiler.html
30. Unity Technologies. Audio in Unity. Unity Manual. URL: https://docs.unity3d.com/Manual/Audio.html

# ДОДАТОК А

## Лістинг кодів скриптів

### AICarSpawner:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class AICarSpawner : MonoBehaviour
{
    [SerializeField]
    GameObject[] carAIPrefabs;
    GameObject[] carAIPool = new GameObject[20];
    Transform playerCarTransform;

    float timeLastCarSpawned = 0;
    WaitForSeconds wait = new WaitForSeconds(0.2f);
    [SerializeField]
    LayerMask otherCarsLayerMask;
    Collider[] overlappedCheckCollider = new Collider[1];

    // Start is called before the first frame update
    void Start()
    {
        playerCarTransform = GameObject.FindGameObjectWithTag("Player").transform;
        int prefabsIndex = 0;

        for (int i = 0; i < carAIPool.Length; i++)
        {
            carAIPool[i] = Instantiate(carAIPrefabs[prefabsIndex]);
            carAIPool[i].SetActive(false);

            prefabsIndex++;

            if (prefabsIndex > carAIPrefabs.Length - 1)
                prefabsIndex = 0;
        }
        StartCoroutine(UpdateLessOftenCO());
    }
    IEnumerator UpdateLessOftenCO()
    {
        while (true)
        {
            CleanUpCarsBeyondView();
            SpawnNewCars();
            yield return wait;
        }
    }
    void SpawnNewCars()
    {
        if (Time.time - timeLastCarSpawned < 2)
            return;

        GameObject carToSpawn = null;

        //Find a car to spawn
        foreach (GameObject aiCar in carAIPool)
        {
            if (aiCar.activeInHierarchy)
                continue;

            carToSpawn = aiCar;
            break;
        }
        //No car available to spawn
        if(carToSpawn == null)
            return;

        Vector3 spawnPosition = new Vector3(0, playerCarTransform.transform.position.y,
playerCarTransform.transform.position.z + 100);

        if(Physics.OverlapBoxNonAlloc(spawnPosition, Vector3.one *2, overlappedCheckCollider,
Quaternion.identity, otherCarsLayerMask) >0)
```

```
            return;

        carToSpawn.transform.position = spawnPosition;
        carToSpawn.SetActive(true);
        timeLastCarSpawned = Time.time;

    }
    void CleanUpCarsBeyondView()
    {
        foreach (GameObject aiCar in carAIPool)
        {
            //Scip active cars
            if (!aiCar.activeInHierarchy)
                continue;

            //Check if AI car is too far ahead
            if (aiCar.transform.position.z - playerCarTransform.position.z > 200)
                aiCar.SetActive(false);
            //Check if AI car is too far behind
            if (aiCar.transform.position.z - playerCarTransform.position.z < -20)
                aiCar.SetActive(false);
        }
    }
}
```

# AIHandler:

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

public class AIHandler : MonoBehaviour
{
    [SerializeField]
    CarHandler carHandler;
    [SerializeField]
    LayerMask otherCarsLayerMask;
    [SerializeField]
    MeshCollider meshCollider;
    [Header("SFX")]
    [SerializeField]
    AudioSource honkHornAS;

    //Collision detection
    RaycastHit[] raycastHits = new RaycastHit[1];
    bool isCarAhead = false;
    float carAheadDistance = 0;

    //Lanes
    int drivingInLane = 0;

    //Timing
    WaitForSeconds wait = new WaitForSeconds(0.2f);

    private void Awake()
    {
        if(CompareTag("Player"))
        {
            Destroy(this);
            return;
        }
    }
    // Start is called before the first frame update
    void Start()
    {
        StartCoroutine(UpdateLessOftenCO());
    }
    // Update is called once per frame
    void Update()
    {
        float accelerationInput = 1.0f;
        float steerInput = 0.0f;

        if (isCarAhead)
        {
            accelerationInput = -1;
            if(carAheadDistance < 10 && !honkHornAS.isPlaying)
```

```
            {
                honkHornAS.pitch = Random.Range(0.2f, 0.8f);
                honkHornAS.volume = Random.Range(0.1f, 0.3f);
                honkHornAS.Play();
            }
        }

        float desiredPositionX = Utils.CarLanes[drivingInLane];
        float difference = desiredPositionX - transform.position.x;

        if (Mathf.Abs(difference) > 0.05f)
            steerInput = 1.0f * difference;

        steerInput = Mathf.Clamp(steerInput, -1.0f, 1.0f);
        carHandler.SetInput(new Vector2(steerInput, accelerationInput));
    }
    IEnumerator UpdateLessOftenCO()
    {
        while (true)
        {
            isCarAhead =  CheckIfOtherCarsIsAhead();
            yield return wait;
        }
    }

    bool CheckIfOtherCarsIsAhead()
    {
        meshCollider.enabled = false;
        int numberOfHits = Physics.BoxCastNonAlloc(transform.position, Vector3.one * 0.25f,
transform.forward, raycastHits, Quaternion.identity, 2, otherCarsLayerMask);
        meshCollider.enabled = true;

        if (numberOfHits > 0)
        {
            carAheadDistance = (transform.position - raycastHits[0].point).magnitude;
            return true;
        }
        return false;
    }
    //Events
    private void OnEnable()
    {
        //Set radom speed
        carHandler.SetMaxSpeed(Random.Range(5, 10));
        //Set random lane
        drivingInLane = Random.Range(0, Utils.CarLanes.Length);

    }
}
```

## CarHandler:

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

public class CarHandler : MonoBehaviour
{
    [SerializeField]
    Rigidbody rb;
    [SerializeField]
    Transform gameModel;
    [SerializeField]
    MeshRenderer carMeshRenderer;
    public MeshRenderer CarMeshRenderer => carMeshRenderer;

    [SerializeField]
    ExplodeHandler explodeHandler;
    [Header("SFX")]
    [SerializeField]
    AudioSource carEngineAS;
    [SerializeField]
    AnimationCurve carPitchAnimationCurve;
    [SerializeField]
    AudioSource carSkidAS;
    [SerializeField]
    AudioSource carCrashAS;
```

```
    //max values
    float maxSteerVelocity = 2;
    float maxForwardVelocity = 30;
    float carMaxSpeedPercentage = 0;

    //multipliers
    float accelerationMultiplier = 3;
    float breaksMultiplier = 15;
    float steeringMultiplier = 5;

    //Input
    Vector2 input = Vector2.zero;

    // Emissive property
    int _EmissiveColor = Shader.PropertyToID("_EmissiveColor");
    Color emissiveColor = Color.white;
    float emissiveColorMultiplier = 0f;

    // Explode state
    bool isExploded = false;
    bool isPlayer = true;

    //Stats
    float carStartPositionZ;
    float distanceTravelled = 0;
    public float DistanceTravelled => distanceTravelled;

    //Events
    public event Action<CarHandler> OnPlayerCrushed;

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    void Start()
    {
        isPlayer = CompareTag("Player");

            if (isPlayer)
                carEngineAS.Play();
        carStartPositionZ = transform.position.z;
    }
    // Update is called once per frame
    void Update()
    {
        if (isExploded)
        {
            FadeOutCarAudio();
            return;
        }
        // Rotate the car model "turning"
        gameModel.transform.rotation = Quaternion.Euler(0, rb.linearVelocity.x * 5, 0);

        if (carMeshRenderer != null)
        {
            float desiredCarEmissiveColorMultiplier = 0f;

            if (input.y < 0)
                desiredCarEmissiveColorMultiplier = 4.0f;

            emissiveColorMultiplier = Mathf.Lerp(emissiveColorMultiplier,
desiredCarEmissiveColorMultiplier, Time.deltaTime * 4);
            carMeshRenderer.material.SetColor(_EmissiveColor, emissiveColor *
emissiveColorMultiplier);
        }
        UpdateCarAudio();

        //Update distance travelled
        distanceTravelled = transform.position.z - carStartPositionZ;
    }
    private void FixedUpdate()
    {
        //Is exploded
        if (isExploded)
        {
            //apply drag
            rb.linearDamping = rb.linearVelocity.z * 0.1f;
            rb.linearDamping = Mathf.Clamp(rb.linearDamping, 1.5f, 10);

            //Move towards after a the car has exploaded
```

```
            rb.MovePosition(Vector3.Lerp(transform.position, new Vector3(0, 0, transform.position.z),
Time.deltaTime * 0.5f));

            return;
        }
        //Apply Acceleration and Brakes
        if (input.y > 0)
            Accelarate();
        else
            rb.linearDamping = 0.2f;

            if (input.y < 0)
                Brake();

        Steer();

        //Force the car to not go backwards
        if (rb.linearVelocity.z < 0)
            rb.linearVelocity = Vector3.zero;
    }
    void Accelarate()
    {
        rb.linearDamping = 0;

        if (rb.linearVelocity.z >= maxForwardVelocity)
            return;

        rb.AddForce(rb.transform.forward * accelerationMultiplier * input.y);
    }
    void Brake()
    {
        if (rb.linearVelocity.z <= 0)
            return;

        rb.AddForce(rb.transform.forward * breaksMultiplier * input.y);
    }
    void Steer()
    {
        if (Mathf.Abs(input.x) > 0)
        {
            //Move the car sideways
            float speedBaseSteerLimit = rb.linearVelocity.z / 5.0f;
            speedBaseSteerLimit = Mathf.Clamp01(speedBaseSteerLimit);

            rb.AddForce(rb.transform.right * steeringMultiplier * input.x * speedBaseSteerLimit);

            //Normalize the X Velocity
            float normalizedX = rb.linearVelocity.x / maxSteerVelocity;

            //Ensure that we don't allow it on get bigger than 1 in magnitued
            normalizedX = Mathf.Clamp(normalizedX, -1.0f, 1.0f);

            //Make sure we stay within the turn speed limit
            rb.linearVelocity = new Vector3(normalizedX * maxSteerVelocity, 0, rb.linearVelocity.z);

        }
        else
        {
            //Auto center car
            rb.linearVelocity = Vector3.Lerp(rb.linearVelocity, new Vector3(0, 0,
rb.linearVelocity.z), Time.fixedDeltaTime * 3);
        }
    }
    void UpdateCarAudio()
    {
        if (!isPlayer)
            return;

        carMaxSpeedPercentage = rb.linearVelocity.z / maxForwardVelocity;
        carEngineAS.pitch = carPitchAnimationCurve.Evaluate(carMaxSpeedPercentage);

        if (input.y < 0 && carMaxSpeedPercentage > 0.2f)
        {
            if (!carSkidAS.isPlaying)
                carSkidAS.Play();

            carSkidAS.volume = Mathf.Lerp(carSkidAS.volume, 2.0f, Time.deltaTime * 10);
        }
```

```
            else
            {
                carSkidAS.volume = Mathf.Lerp(carSkidAS.volume, 1.0f, Time.deltaTime * 30);
            }
        }
        void FadeOutCarAudio()
        {
            if (!isPlayer)
                return;

            carEngineAS.volume = Mathf.Lerp(carEngineAS.volume, 0, Time.deltaTime * 10);
            carSkidAS.volume = Mathf.Lerp(carSkidAS.volume, 0, Time.deltaTime * 10);
        }

        public void SetInput(Vector2 inputVector)
        {
            inputVector.Normalize();

            input = inputVector;
        }
        public void SetMaxSpeed(float newMaxSpeed)
        {
            maxForwardVelocity = newMaxSpeed;
        }
        IEnumerator SlowDownTimeCO()
        {
            while (Time.timeScale > 0.2f)
            {
                Time.timeScale -= Time.deltaTime * 0.5f;
                yield return null;
            }
            yield return  new WaitForSeconds(0.5f);

            while (Time.timeScale <= 1.0f)
            {
                Time.timeScale += Time.deltaTime;
                yield return null;
            }

            Time.timeScale = 1.0f;
        }

        //Events
        private void OnCollisionEnter(Collision collision)
        {
            if (!isPlayer)
            {
                if (collision.transform.root.CompareTag("Untagged"))
                    return;

                if(collision.transform.root.CompareTag("CarAI"))
                    return;
                Debug.Log("БАБАХ! Врізався в: " + collision.gameObject.name);
            }

            Vector3 velocity = rb.linearVelocity;
            explodeHandler.Explode(velocity * 45);

            isExploded = true;

            carCrashAS.volume = carMaxSpeedPercentage;
            carCrashAS.volume = Mathf.Clamp(carCrashAS.volume, 0.25f, 1.0f);

            carCrashAS.pitch = carMaxSpeedPercentage;
            carCrashAS.pitch = Mathf.Clamp(carCrashAS.pitch, 0.3f, 1.0f);

            carCrashAS.Play();

            //Trigger  event
            OnPlayerCrushed?.Invoke(this);

            StartCoroutine(SlowDownTimeCO());
        }
}
```

## PlayerCarSpawner:

```
using UnityEngine;
```

```
using Unity.Cinemachine;

public class PlayerCarSpawner : MonoBehaviour
{
    [SerializeField] GameObject[] carPrefabs;

    [Header("Camera")]
    [SerializeField] CinemachineCamera cinemachineCamera;
    [Header("Menu")]
    [SerializeField] bool isMainMenu = false;
    GameObject instantiatedPlayerCar = null;
    int carIndex = 0;
    static GameObject selectedCarPrefab = null;
    Quaternion menuRotation = Quaternion.identity;
    bool rotationInitialized = false;

    void Start()
    {
        SpawnCar();
    }

    void Update()
    {
        if (isMainMenu && instantiatedPlayerCar != null)
        {
            instantiatedPlayerCar.transform.Rotate(Vector3.up * 20f * Time.deltaTime);
            menuRotation = instantiatedPlayerCar.transform.rotation;
        }
    }
    void SpawnCar()
    {
        if (instantiatedPlayerCar != null) Destroy(instantiatedPlayerCar);

        GameObject prefabToSpawn;
        if (isMainMenu)
        {
            prefabToSpawn = carPrefabs[carIndex];
            selectedCarPrefab = prefabToSpawn;

            if (!rotationInitialized)
            {
                menuRotation = prefabToSpawn.transform.rotation;
                rotationInitialized = true;
            }

            instantiatedPlayerCar = Instantiate(prefabToSpawn, prefabToSpawn.transform.position,
menuRotation);
            PrepareCarForMenu(instantiatedPlayerCar);
        }
        else
        {
            prefabToSpawn = (selectedCarPrefab != null) ? selectedCarPrefab : carPrefabs[0];

            instantiatedPlayerCar = Instantiate(prefabToSpawn, prefabToSpawn.transform.position,
prefabToSpawn.transform.rotation);
        }

        if (cinemachineCamera != null)
            cinemachineCamera.Follow = instantiatedPlayerCar.transform;
    }
    void PrepareCarForMenu(GameObject car)
    {
        if (car.TryGetComponent<CarHandler>(out var handler))
            handler.enabled = false;

        if (car.TryGetComponent<Rigidbody>(out var rb))
            rb.isKinematic = true;
    }
    public void OnNextCarClicked()
    {
        if (!isMainMenu) return;
        carIndex++;
        if (carIndex >= carPrefabs.Length) carIndex = 0;
        SpawnCar();
    }

    public void OnPreviousCarClicked()
    {
```

```
        if (!isMainMenu) return;
        carIndex--;
        if (carIndex < 0) carIndex = carPrefabs.Length - 1;
        SpawnCar();
    }
}
```

## DatabaseManager:

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

public class DatabaseManager : MonoBehaviour
{
    private string connectionString;
    void Awake()
    {
        connectionString = "URI=file:" + Application.persistentDataPath + "/GameDatabase.db";
        CreateSchema();
    }

    private void CreateSchema()
    {
        using (var connection = new SqliteConnection(connectionString))
        {
            connection.Open();
            using (var command = connection.CreateCommand())
            {
                command.CommandText = "CREATE TABLE IF NOT EXISTS Leaderboard (id INTEGER PRIMARY KEY
AUTOINCREMENT, name TEXT, score INTEGER);";
                command.ExecuteNonQuery();
            }
        }
    }

    public void AddScore(string playerName, int score)
    {
        using (var connection = new SqliteConnection(connectionString))
        {
            connection.Open();
            using (var command = connection.CreateCommand())
            {
                command.CommandText = "INSERT INTO Leaderboard (name, score) VALUES (@name, @score);";
                command.Parameters.Add(new SqliteParameter("@name", playerName));
                command.Parameters.Add(new SqliteParameter("@score", score));
                command.ExecuteNonQuery();
            }
        }
    }
    public List<ScoreEntry> GetTopScores()
    {
        List<ScoreEntry> scores = new List<ScoreEntry>();
        using (var connection = new SqliteConnection(connectionString))
        {
            connection.Open();
            using (var command = connection.CreateCommand())
            {
                command.CommandText = "SELECT name, score FROM Leaderboard ORDER BY score DESC LIMIT
10;";
                using (IDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        scores.Add(new ScoreEntry(reader.GetString(0), reader.GetInt32(1)));
                    }
                }
            }
        }
        return scores;
    }
}

[System.Serializable]
public class ScoreEntry
{
    public string playerName;
```

```
    public int score;
    public ScoreEntry(string name, int s)
    {
        playerName = name;
        score = s;
    }
}
```

## EndlessLevelHandler:

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

public class EndlessLevelHandler : MonoBehaviour
{
    [SerializeField]
    GameObject[] sectionsPrefabs;
    GameObject[] sectionsPool = new GameObject[20];
    GameObject[] sections = new GameObject[10];
    Transform playerCarTransform;
    WaitForSeconds waitFor100ms = new WaitForSeconds(0.1f);
    const float sectionLength = 26;

    // Start is called before the first frame update
    void Start()
    {
        playerCarTransform = GameObject.FindGameObjectWithTag("Player").transform;

        int prefabIndex = 0;

        for (int i = 0; i < sectionsPool.Length; i++)
        {
            sectionsPool[i] = Instantiate(sectionsPrefabs[prefabIndex]);
            sectionsPool[i].SetActive(false);

            prefabIndex++;

            if (prefabIndex > sectionsPrefabs.Length - 1)
                prefabIndex = 0;
        }

        for(int i = 0; i < sections.Length; i++)
        {
            GameObject randomSection = GetRandomSectionFromPool();

            randomSection.transform.position = new Vector3(sectionsPool[i].transform.position.x, -10,
i * sectionLength);
            randomSection.SetActive(true);

            sections[i] = randomSection;
        }
        StartCoroutine(UpdatelessOftenCO());
    }
    IEnumerator UpdatelessOftenCO()
    {
        while (true)
        {
            UpdateSectionPositions();
            yield return waitFor100ms;
        }

    }
    void UpdateSectionPositions()
    {
        for (int i = 0; i < sections.Length; i++)
        {
            if (sections[i].transform.position.z - playerCarTransform.position.z < -sectionLength)
            {
                Vector3 lastSectionPosition = sections[i].transform.position;
                sections[i].SetActive(false);

                sections[i] = GetRandomSectionFromPool();

                sections[i].transform.position = new Vector3(lastSectionPosition.x, -10,
lastSectionPosition.z + sections.Length * sectionLength);
                sections[i].SetActive(true);
            }
```

```
        }
    }
    GameObject GetRandomSectionFromPool()
    {
        int randomIndex = Random.Range(0, sectionsPool.Length);

        bool isNewSectionFound = false;

        while (!isNewSectionFound)
        {
            if (!sectionsPool[randomIndex].activeInHierarchy)
               isNewSectionFound = true;

            else
            {
                randomIndex++;
                if (randomIndex > sectionsPool.Length - 1)
                    randomIndex = 0;
            }
        }
        return sectionsPool[randomIndex];
    }
}
```

## EndlessSectionHandler:

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

public class EndlessSectionHandler : MonoBehaviour
{
    Transform playerCarTransform;

    // Start is called before the first frame update
    void Start()
    {
      playerCarTransform = GameObject.FindGameObjectWithTag("Player").transform;
    }

    // Update is called once per frame
    void Update()
    {
        float distanceToPlayer = transform.position.z - playerCarTransform.position.z;

        float lerpPercentage = 1.0f - ((distanceToPlayer - 100)) / 150.0f;
        lerpPercentage = Mathf.Clamp01(lerpPercentage);

        transform.position = Vector3.Lerp(new Vector3(transform.position.x, -10,
transform.position.z), new Vector3(transform.position.x, 0, transform.position.z),lerpPercentage);
    }

}
```

## CarPartHandler:

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

public class CarPartHandler : MonoBehaviour
{
    AudioSource bounceAS;
    void Awake()
    {
        bounceAS = GetComponent<AudioSource>();
    }
    void OnCollisionEnter(Collision collision)
    {
        if (!bounceAS.isPlaying)
        {
            bounceAS.pitch = collision.relativeVelocity.magnitude * 0.5f;

            bounceAS.pitch = Mathf.Clamp(bounceAS.pitch, 0.5f, 1.0f);

            bounceAS.Play();
        }
    }
}
```

## ExplodeHandler:

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

public class ExplodeHandler : MonoBehaviour
{
    [SerializeField]
    GameObject originalObject;
    [SerializeField]
    GameObject model;
    Rigidbody[] rigidbodies;

    private void Awake()
    {
        rigidbodies = model.GetComponentsInChildren<Rigidbody>(true);
    }

    // Start is called before the first frame update
    void Start()
    {
       // Explode(Vector3.forward);
    }

    public void Explode(Vector3 externalForce)
    {
        originalObject.SetActive(false);

        foreach (Rigidbody rb in rigidbodies)
        {
            rb.transform.parent = null;

            rb.GetComponent<MeshCollider>().enabled = true;

            rb.gameObject.SetActive(true);
            rb.isKinematic = false;
            rb.interpolation = RigidbodyInterpolation.Interpolate;
            rb.AddForce(Vector3.up * 200 + externalForce, ForceMode.Force);
            rb.AddTorque(Random.insideUnitSphere * 0.5f, ForceMode.Impulse);

            //Change the tag so other objects can explode after being hit by a carpart
            rb.gameObject.tag = "CarPart";
        }
    }
}
```

## InputHandler:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;
public class inputHandler : MonoBehaviour
{
    [SerializeField]
    CarHandler carHandler;
    private void Awake()
    {
        if (!CompareTag("Player"))
        {
            Destroy(this);
            return;
        }
    }
    private void Update()
    {
        Vector2 input = Vector2.zero;

        input.y = Input.GetAxis("Vertical");
        input.x = Input.GetAxis("Horizontal");

        carHandler.SetInput(input);

        if (Input.GetKeyDown(KeyCode.P))
        {
            //Restore time scale
            Time.timeScale = 1.0f;
            SceneManager.LoadScene(SceneManager.GetActiveScene().name);
```

```
        }
    }
}
```

## LeaderboardDisplay:

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

public class LeaderboardDisplay : MonoBehaviour
{
    [Header("Settings")]
    public DatabaseManager dbManager;
    public GameObject rowPrefab;
    public Transform contentParent;

    public void RefreshDisplay()
    {
        Debug.Log("1. RefreshDisplay викликано!");

        if (contentParent == null)
        {
            Debug.LogError("ПОМИЛКА: Не призначено Content Parent в інспекторі!");
            return;
        }

        foreach (Transform child in contentParent)
        {
            Destroy(child.gameObject);
        }

        if (dbManager == null)
            dbManager = Object.FindAnyObjectByType<DatabaseManager>();

        if (dbManager == null)
        {
            Debug.LogError("2. ПОМИЛКА: DatabaseManager не знайдено на сцені!");
            return;
        }

        List<ScoreEntry> topScores = dbManager.GetTopScores();
        Debug.Log($"3. Отримано записів з бази: {topScores.Count}");

        foreach (ScoreEntry entry in topScores)
        {
            Debug.Log($"4. Створюю рядок для: {entry.playerName}");
            GameObject row = Instantiate(rowPrefab, contentParent);

            // Скидаємо масштаб, бо іноді UI префаби стають величезними або мікроскопічними
            row.transform.localScale = Vector3.one;

            TMP_Text[] texts = row.GetComponentsInChildren<TMP_Text>();

            if (texts.Length >= 2)
            {
                texts[0].text = entry.playerName;
                texts[1].text = entry.score.ToString("D6");
            }
            else
            {
                Debug.LogWarning("У префабі ScoreRow знайдено менше 2-х компонентів TMP_Text!");
            }
        }
    }
}
```

## MainMenuUIHandler:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenuUIHandler : MonoBehaviour
{
    [Header("Panels")]
    [SerializeField]
    private GameObject leaderboardPanel;

    [Header("Scripts")]
```

```
    [SerializeField]
    private LeaderboardDisplay leaderboardDisplay;

    public void OnStartGameClicked()
    {
        SceneManager.LoadScene("Stage");
    }

    public void OnLeaderboardClicked()
    {
        if (leaderboardPanel != null)
        {
            leaderboardPanel.SetActive(true);

            if (leaderboardDisplay != null)
            {
                leaderboardDisplay.RefreshDisplay();
            }
            else
            {
                Debug.LogWarning("LeaderboardDisplay не призначено в інспекторі!");
            }
        }
    }

    public void OnCloseLeaderboardClicked()
    {
        if (leaderboardPanel != null)
        {
            leaderboardPanel.SetActive(false);
        }
    }
    public void OnQuitGameClicked()
    {
        Debug.Log("Вихід із гри...");
        Application.Quit();
    }
}
```

## UIHandler:

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

public class UIHandler : MonoBehaviour
{
    [Header("Original UI")]
    [SerializeField]
    TextMeshProUGUI distanceTravelledText;
    [SerializeField]
    TextMeshProUGUI gameOverText;
    [SerializeField]
    CanvasGroup gameOverCanvasGroup;
    [SerializeField]
    GameObject gameOverPanel;

    [Header("New Leaderboard UI")]
    [SerializeField]
    TMP_InputField playerNameInput;
    [SerializeField]
    Button saveButton;
    [SerializeField]
    GameObject leaderboardPanel;

    CarHandler playerCarHandler;
    DatabaseManager dbManager;

    void Awake()
    {
        dbManager = Object.FindAnyObjectByType<DatabaseManager>();
    }
```

```
    void Start()
    {

        GameObject player = GameObject.FindGameObjectWithTag("Player");
        if (player != null)
        {
            playerCarHandler = player.GetComponent<CarHandler>();
            playerCarHandler.OnPlayerCrushed += PlayerCarHandler_OnPlayerCrushed;
        }

        gameOverCanvasGroup.interactable = false;
        gameOverCanvasGroup.alpha = 0;
        gameOverCanvasGroup.blocksRaycasts = false;

        if (leaderboardPanel != null) leaderboardPanel.SetActive(false);
    }

    void Update()
    {
        if (playerCarHandler != null)
            distanceTravelledText.text =
playerCarHandler.DistanceTravelled.ToString("000000");
    }

    IEnumerator StartGameAnimationCO()
    {
        yield return new WaitForSecondsRealtime(1.0f);
        while (gameOverCanvasGroup.alpha < 1.0f)
        {
            gameOverCanvasGroup.alpha = Mathf.MoveTowards(gameOverCanvasGroup.alpha, 1.0f,
Time.unscaledDeltaTime * 2);
            yield return null;
        }
        gameOverCanvasGroup.interactable = true;
        gameOverCanvasGroup.blocksRaycasts = true;
    }
    void PlayerCarHandler_OnPlayerCrushed(CarHandler obj)
    {
        gameOverText.text = $"DISTANCE {distanceTravelledText.text}";
        StartCoroutine(StartGameAnimationCO());
    }

    public void OnSaveClicked()
    {
        string name = playerNameInput.text;
        if (!string.IsNullOrEmpty(name) && dbManager != null)
        {
            int distance = (int)playerCarHandler.DistanceTravelled;
            dbManager.AddScore(name, distance);

            saveButton.interactable = false;
            playerNameInput.interactable = false;
            Debug.Log($"<color=green>SUCCESS:</color> Saved {name} with score {distance}");
        }
    }

    public void OnLeaderboardClicked()
    {
        if (leaderboardPanel != null)
        {
            if (gameOverPanel != null) gameOverPanel.SetActive(false);
            leaderboardPanel.SetActive(true);

            LeaderboardDisplay display =
leaderboardPanel.GetComponent<LeaderboardDisplay>();
            if (display != null) display.RefreshDisplay();
        }
    }
```

```
    public void OnCloseLeaderboardClicked()
    {
        if (leaderboardPanel != null)
        {
            leaderboardPanel.SetActive(false);
            if (gameOverPanel != null) gameOverPanel.SetActive(true);
        }
    }

    public void OnRestartClicked()
    {
        Time.timeScale = 1.0f;
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }


    public void OnReturnToMenuClicked()
    {
        Time.timeScale = 1.0f;

        SceneManager.LoadScene("Main menu");
    }
}
```

## RandomizeObject:

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

public class RandomizeObject : MonoBehaviour
{
    [SerializeField]
    Vector3 localRotationMin = Vector3.zero;
    [SerializeField]
    Vector3 localRotationMax = Vector3.zero;
    [SerializeField]
    float localscaleMultiplierMin = 0.8f;
    [SerializeField]
    float localscaleMultiplierMax = 1.5f;

    Vector3 localScaleOriginal = Vector3.one;

    private void Start()
    {
        localScaleOriginal = transform.localScale;
    }

    void OnEnable()
    {
        transform.localRotation = Quaternion.Euler(Random.Range(localRotationMin.x,
localRotationMax.x), Random.Range(localRotationMin.y, localRotationMax.y),
Random.Range(localRotationMin.z, localRotationMax.z));

        transform.localScale = transform.localScale * Random.Range(localscaleMultiplierMin,
localscaleMultiplierMax);
    }
}
```

## Utils:

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

public static class Utils
{
    static float[] carLanes = { -0.3f, 0.3f};
    public static float[] CarLanes => carLanes;

}
```

# ДОДАТОК Б

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

**Декларація про академічну доброчесність та використання ШІ**

Я, Супрунюк Вадим Ростиславович, здобувач НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота на тему «Ігровий застосунок з реалізацією механізму нескінченного бігу» є результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що (обрати необхідне):

- [ ] інструменти генеративного ШІ не використовувалися.
- [ ] інструменти ШІ (вказати назву: ChatGPT, Gemini, Claude, DeepL, _____інші (вказати назву)) були використані 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. _____ **Вадим СУПРУНЮК**

(підпис)