МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА[1]

Бабаліч Максим Юрійович
(прізвище, ім'я, по батькові здобувача)

на тему Розробка гри «Гонки»
(повна назва теми)

за матеріалами праць провідних спеціалістів з розробки ПЗ та проектування БД
(повна назва бази дослідження)

науковий керівник к.т.н., доцент _____ Хоцкіна В. Б.
(наук. ступінь, вчене звання) *(підпис)* *(прізвище, ініціали)*

Робота допущена до захисту в ЕК
Протокол засідання кафедри
від 11.06.2025 № 12
Завідувач кафедри _____
(підпис)
д.т.н., професор Зеленський О.С.
Наук. ступень, вчене звання *Ініціали, прізвище*

Кривий Ріг – 2025

[1] Розробка гри «Гонки»

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет Інформаційних технологій
Кафедра Інформатики і прикладного програмного забезпечення
Спеціальність Інженерія програмного забезпечення
Форма навчання Денна

«ЗАТВЕРДЖУЮ»
Завідувач кафедри _____ Зеленський О.С.
(підпис) (Прізвище, ініціали)
«11» червня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ[2]

1. Тема роботи «Розробка гри «Гонки»»
Керівник роботи к.т.н., доцент Хоцкіна В. Б.
затверджені наказом закладу вищої освіти від «04» квітня 2025 р. № 222-ст

2. Строк подання здобувачем роботи до «09» червня 2025 р.

3. Зміст кваліфікаційної роботи, об’єкт, предмет та мета дослідження:

Розділ 1. Постановка задачі

Розділ 2. Розробка алгоритму розв’язання задачі

Розділ 3. Розробка бази даних задачі

Розділ 4. Розробка програмного забезпечення

Об’єкт дослідження: гра «Гонки»

Предмет дослідження: гра

Мета кваліфікаційної роботи: створення гри «Гонки»

5. Дата видачі завдання «04» квітня 2025 р.

[2] Розробка гри «Гонки»

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів МДР	Строк виконання етапів роботи	Відмітка керівника про виконання етапів (дата, підпис)
1	Підготовка розділу 1	04.04.2025-13.04.2025	
2	Підготовка розділу 2	14.04.2025-26.04.2025	
3.	Підготовка розділу 3	27.04.2025-15.05.2025	
4	Підготовка розділу 4	16.05.2025-08.06.2025	
5	Реєстрація завершеної кваліфікаційної роботи	09.06.2025	Реєстраційний №___ «09»червня 2025 р.
6	Отримання відгуку від наукового керівника	10.06.2025	
7	Подання кваліфікаційної роботи на перегляд завідувачу кафедри	11.06.2025	
8	Отримання зовнішньої рецензії	12.06.2025	
9	Попередній захист кваліфікаційної роботи на кафедрі	13.06.2025	
10	Підготовка до захисту в ЕК	16.06.2025-21.06.2025	

Завдання підготував науковий керівник _____ Хоцкіна В. Б.

(підпис) (прізвище та ініціали)

Завдання одержав _____ Бабаліч М. Ю.

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

на бакалаврську роботу

«Розробка гри «Гонки»»

Бабаліч М. Ю.

Кваліфікаційна робота за спеціальністю 121 – Інженерія програмного забезпечення – Державний університет економіки і технологій. – Кривий Ріг, 2025.

У межах кваліфікаційної роботи реалізовано 2D-гру у жанрі Гонки з використанням мови програмування C++ та графічного API OpenGL, який застосовано для побудови візуальної частини гри.

Проєкт включає ключові компоненти ігрового процесу: керування транспортним засобом, обробку зіткнень, динамічну трасу з перешкодами, а також базові елементи інтерфейсу.

Архітектура гри побудована з урахуванням принципів об'єктно-орієнтованого програмування, що забезпечує зручність підтримки та подальшого розширення функціоналу.

Також підготовлено інсталятор, який дозволяє користувачам легко та безперешкодно встановлювати гру на пристрої з операційною системою Windows.

У результаті реалізовано завершену настільну 2D-гру, яка відповідає основним очікуванням користувачів щодо зручності використання, стабільності функціонування та привабливого візуального оформлення.

Ключові слова: C++, OPENGL, 2D-ГРА, ГОНКИ, ДВОМІРНА ГРАФІКА, ІГРОВА МЕХАНІКА, ІНТЕРФЕЙС КОРИСТУВАЧА, ІНСТАЛЯТОР, WINDOWS.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

2D-гра	Гра, в якій усі елементи (транспорт, траса, об’єкти, перешкоди) відображаються в двовимірному просторі, без використання об’ємних моделей.
OpenGL	Універсальний інструмент для розробки графіки, який функціонує на різних операційних системах. У 2D-іграх ця бібліотека дозволяє реалізувати відображення графічних елементів у реальному часі, включаючи анімацію, роботу з текстурами, кольорами та геометричними об’єктами.
Інсталятор	Спеціальний програмний модуль, який дозволяє користувачу встановити гру на комп’ютер з операційною системою Windows у кілька простих кроків.
Гонки	Жанр комп’ютерних ігор являють собою спеціалізовані програми, які відтворюють процес керування транспортом — найчастіше автомобілями — на визначених маршрутах.
Логіка гри	Сукупність внутрішніх механізмів та програмних структур, які визначають поведінку елементів гри у відповідь на дії користувача. Саме вона формує правила, за якими функціонує віртуальний світ, включаючи рух об’єктів, зіткнення, досягнення мети тощо.
UI (інтерфейс користувача)	Набір візуальних компонентів, що забезпечують інтерактивну взаємодію між гравцем і грою. До нього належать такі елементи, як кнопки, меню, панелі індикації, спливаючі підказки тощо, які полегшують навігацію та управління в ігровому середовищі.

ЗМІСТ

ВСТУП

Індустрія відеоігор стрімко розвивається, і створення ігрових застосунків стає все більш актуальним напрямом у сфері програмної інженерії. Гоночні ігри вирізняються високим попитом завдяки поєднанню активного ігрового процесу, елементів суперництва та виразного візуального оформлення.

Використання сучасних технологій дозволяє створювати ігрові проекти на широкому спектрі платформ, що дозволяє розробникам вибирати інструменти та методи створення контенту. Ігри з двовимірною графікою заслуговують особливої уваги, оскільки вони залишаються популярними серед гравців завдяки своїй динамічності та зручності у сприйнятті, незважаючи на те, наскільки простими вони є порівняно з 3D-іграми.

Крім того, двовимірні ігри ідеально підходять для початківців у розробці ігор, оскільки вони дозволяють зосередитися на основних елементах логіки та дизайну без надмірної складності тривимірної графіки. Вони дозволяють швидко отримати видимий результат, що підвищує мотивацію та розуміння процесу розробки ігрового програмного забезпечення.

Серед засобів реалізації графічної частини ігор вагоме місце займає OpenGL — універсальний інтерфейс програмування, який забезпечує ефективне створення двовимірної та тривимірної графіки на різних платформах. Завдяки широким можливостям і високій продуктивності, він є ефективним інструментом для створення графічної частини ігор.

Розробка ігор також допомагає вивчати алгоритми, архітектуру програмного забезпечення, графічні API та принципи об’єктно-орієнтованого програмування. Розробники стикаються з багатьма завданнями, включаючи створення циклу рендерингу та створення взаємодії з користувачем, під час розробки навіть простого проекту. Це робить ігрову розробку ідеальним полігоном, який допомагає закріпити теоретичні знання.

Оптимізація продуктивності є важливою частиною створення ігор. У процесі реалізації проєкту ефективність роботи з пам’яттю, швидкість

рендерингу та чуйність управління є важливими факторами, які прямо впливають на якість користувацького досвіду. Ці елементи поглиблюють розуміння апаратного рівня та характеристик програмної реалізації графіки.

У рамках цієї кваліфікаційної роботи розроблено 2D-гру у жанрі Гонки з використанням мови програмування C++ та графічного API OpenGL. У грі реалізовано основну ігрову логіку, систему керування транспортом, виявлення зіткнень та виведення графіки. Крім того, було розроблено установник, що забезпечує легкий і швидкий процес інсталяції гри на комп’ютери під керуванням операційної системи Windows.

Цей проєкт є корисним прикладом практичного застосування знань у галузі програмування, розробки графіки та побудови структури програмного забезпечення з орієнтацією на користувацький досвід.

РОЗДІЛ 1
ПОСТАНОВКА ЗАДАЧІ

1.1. Основні визначення та характеристика задачі

У XXI столітті відеоігри стали важливою складовою цифрового простору та глобальної індустрії розваг. Вони охоплюють широкий спектр напрямків – від освітніх ігор до складних багатокористувацьких симуляторів – і активно застосовуються не лише для дозвілля, а й для розвитку логіки, уваги та координації. Розробка сучасних комп'ютерних ігор вимагає комплексного підходу, що охоплює знання у програмуванні, комп'ютерній графіці, моделюванні фізичних процесів, створенні інтерфейсів і розумінні поведінки користувача.

Серед різноманіття жанрів особливу популярність здобули гонки, які належать до активних аркадних ігор, що симулюють керування транспортними засобами. Гравець отримує змогу керувати транспортним засобом із метою якнайшвидшого проходження траси. В результаті гравець стає більш залученим у ігровий процес, відчуваючи безперервність дій.

Ігри гоночного типу вирізняються такими ключовими елементами, як плавне управління, реалістична або стилізована графіка, чітка структура рівнів, а також баланс між складністю та задоволенням від гри. Все це формує вимоги до технічної реалізації таких проєктів: необхідна оптимізована графіка, стабільна логіка руху, обробка зіткнень та зручний інтерфейс для взаємодії з користувачем.

Актуальність теми зумовлена не лише популярністю жанру серед користувачів, а й можливістю використання під час розробки різноманітних технічних підходів, серед яких — OpenGL. Це графічний програмний інтерфейс, який широко застосовується для створення як двовимірної, так і тривимірної графіки. Серед ключових переваг OpenGL можна виділити його здатність працювати на різних платформах, ефективне керування графічними елементами та високу продуктивність при рендерингу.

Метою даної роботи є створення 2D-гри у жанрі гонки, де всі графічні елементи відображаються у двовимірному просторі. Розробка гри виконується із застосуванням мови програмування C++ та бібліотеки OpenGL, що відповідає за графічне відображення ігрової сцени, траси, транспорту та елементів інтерфейсу. Крім того, гра повинна включати базову логіку: запуск і завершення гонки, виведення результатів, обробку дій гравця та взаємодію з об'єктами на трасі.

Особливу увагу також приділено створенню інсталяційного модуля, який дозволить користувачеві без зайвих складнощів встановити гру на персональний комп’ютер під керуванням операційної системи Windows.

Таким чином, завдання проєкту полягає не лише у створенні працездатної гри, а й у реалізації практичних навичок алгоритмічного мислення, побудови структури програмного продукту, інтеграції сучасних графічних рішень і дотриманні принципів зручності для користувача.

1.2. Аналіз існуючих на ринку аналогічних ігор

Розробка ігрового проєкту передбачає не лише реалізацію програмної складової, а й детальне ознайомлення з уже існуючими аналогами в обраній сфері.Аналіз уже реалізованих ігор у жанрі гонки дає змогу виявити сильні та слабкі сторони конкурентів, зрозуміти сучасні тенденції, вимоги користувачів і можливості покращення власного продукту.

Розгляд аналогічних проєктів допомагає оцінити рівень складності реалізації окремих функцій, якість графіки, особливості геймплею, а також варіанти інтерактивної взаємодії з користувачем. Такий аналіз є необхідним етапом, що дозволяє сформувати чітке уявлення про ринок, виявити найкращі практики та уникнути повторення поширених недоліків.

Почнемо огляд гри OVERDRIFT FESTIVAL, хоч ця гра є і 3-Д жанру, але можна взяти за основу сюжет, вибір автомобілів, різність трас та таке інше.

Особливості гри:

- Відкритий світ, дослідження території;

- Різні заїзди;
- Автопарк та кастомізація обраних авто;
- Керування передається так, як і в реальності.

При скачуванні гри з сайту, можна звернути увагу на високу оцінку та позитивні відгуки користувачів, що дійсно говорить про високий рівень гри.

Ігрове головне меню демонструє широкий функціонал, доступний гравцю перед початком заїздів. Інтерфейс поділено на кілька розділів, серед яких — налаштування транспортного засобу (CUSTOMIZATION), його покращення (UPGRADES) і тонке технічне налаштування (TUNING). Також присутні опції перегляду карти світу, вибору серверів, створення власного сервера, а також перегляду рейтингів і досягнень.

Справа на екрані відображено детальну інформацію про вибраний автомобіль: технічні показники, зокрема потужність двигуна, вагу, налаштування підвіски, рівень керованості та інші характеристики, які суттєво впливають на ефективність дрифту. Додатково, гравець може обмінювати лайки на внутрішньоігрові бонуси.

Меню виконане в яскравому стилі з динамічним візуальним оформленням, що передає атмосферу автоспортивного фестивалю (рис. 1.1).

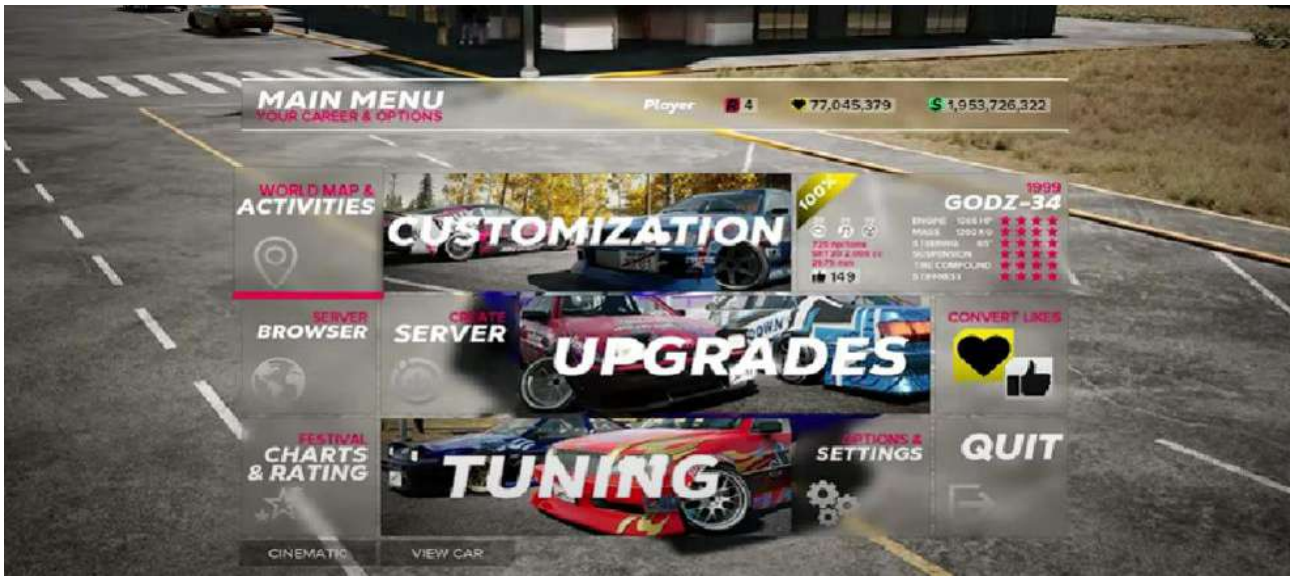

Рис. 1.1. Ігрове меню OVERDRIFT FESTIVAL

Вибір авто відбувається гортанням вправо або вліво переліками авто, на картці презентації написано назву та переваги. Гра має біля 40 видів різних авто з можливістю тюненгу (рис. 1.2).

Рис. 1.2. Вибір авто для гри в OVERDRIFT FESTIVAL

Також, гра має велику кількість трас: гірські траси, індустріальні місця та звичайні траси з автомобільним трафіком (рис. 1.3).

Рис. 1.3. Мапа гри OVERDRIFT FESTIVAL

Початок ігрового процесу реалізовано як динамічний дрифт-заїзд на локації, стилізованій під великий морський порт. Простора відкрита територія з промисловими кранами, вантажними контейнерами та видом на морське узбережжя створює ефект масштабності та свободи руху. Двоє учасників змагання — спортивні автомобілі з яскравим тюнінгом — демонструють майстерність дрифту, залишаючи густі клуби диму позаду себе.

Перший транспортний засіб має агресивний зовнішній вигляд із поєднанням темного корпусу та кислотних елементів, що підкреслює його змагальну натуру. Другий автомобіль вирізняється яскравим бірюзовим кольором із декоративними елементами, що надає йому індивідуальності. Обидві машини активно взаємодіють на трасі, створюючи напружену атмосферу суперництва.

Попри нескладну будову, траса відкриває перед гравцем широкий простір для реалізації дрифтових елементів, чітких поворотів і точного контролю заносу автомобіля. У поєднанні зі звуковим супроводом та ефектами взаємодії коліс із поверхнею створюється ефект повної присутності, що стимулює гравця до постійної активності та вдосконалення навичок керування. (рис. 1.4).

Рис. 1.4. Початок гри в OVERDRIFT FESTIVAL

Тепер розглянемо вже 2-Д гру «Гонки» під назвою «NitroRally». В цю гру можна грати як онлайн на сайти так і завантажити на ПК. Відгуків ніяких на сайті не має, є лише опис гри та меню інших ігор на прикладі цієї. Опис гри виглядає цікавим, є можливість обрання трас, є турбо надув, треба вписуватися в повороти та пройти як можна швидше два кола за суперників і отримати перемогу. При запуску гри з'являється меню: початок, налаштування, інструкція, найкращий час, гараж (рис. 1.5).

Рис. 1.5. Меню гри NitroRally

Після натиску «Старт» з'являється карта, де можна обрати місце, тобто карту на якій буде відбуватися гонка. Також є можливість або повернутися до меню, або обрати перегляд найкращого часу. Карта зображена у вигляді карти Землі, на суходолі знаходяться червоні крапочки, це позначені місця, які можна обрати для гонки, зеленим світиться місце, яке за початкових умов було обрано (рис. 1.6).

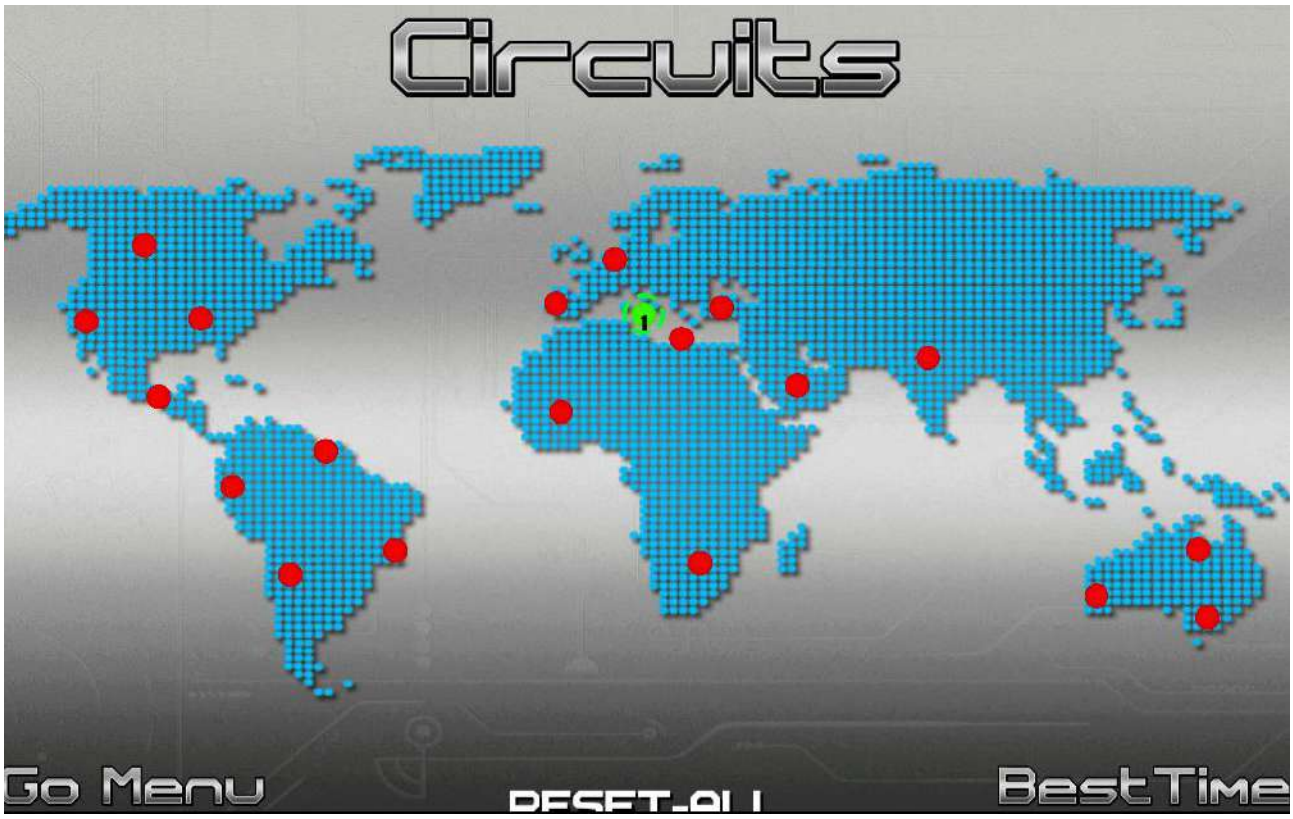

Рис. 1.6. Карта гри NitroRally

Після вибору місця з’являється сама траса з авто. У грі з видом зверху гравець керує маленьким авто, що рухається по звивистій ґрунтовій доріжці. На екрані відображається кількість пройдених кіл, час проходження траси та стартова позиція машини. Оточення виконане у стилі природного середовища — із зеленими насадженнями, камінням, водоймами та іншими деталями рельєфу. Керування відбувається клавішами на ПК: вправо або вліво, або можна керувати з допомогою стрілочок, які зображені на екрані за допомогою мишки (рис. 1.7).

Рис. 1.7. Траса з авто в грі NitroRally

У грі присутній музичний супровід, який підсилює загальний настрій і занурює гравця в атмосферу динамічної їзди. Коли автомобіль врізається в перешкоду, чути реалістичні звуки зіткнення, що додають відчуття справжності. Якщо ж машина їде стабільно і без аварій протягом певного часу, активується режим прискорення, що дозволяє збільшити швидкість.

У випадку, коли транспортний засіб блокується після зіткнення, система візуалізації генерує ефект пилового туману, що імітує зупинку руху. Під час дрифту на асфальті з’являються чорні сліди шин, які підкреслюють гостроту маневру (рис. 1.8).

Рис. 1.8. Гра NitroRally

Так як в час, який поставлений був для проходження всіх кіл, не вкладено, то висвітилося меню з інформацією про програш та можливістю або перегравати, або обрати наступну гру (рис. 1.9).

Рис. 1.9. Завершення гри NitroRally

В цілому гра NitroRally гарна для наслідування створення гри «Гонка» 2-Д, але дещо можна взяти для розробки і з гри OVERDRIFT FESTIVAL, до прикладу механіку та різноманітність трас, та автомобілей.

1.3. Формулювання вимог до гри

Гра "Гонки" є двовимірною аркадною грою, у якій гравець керує автомобілем, уникаючи перешкод та намагаючись протриматися на трасі якомога довше без зіткнень. Основна мета — зберегти контроль над транспортом, долаючи різноманітні труднощі, та досягти максимального результату. Щоб забезпечити якісний ігровий процес, до гри висуваються певні функціональні та нефункціональні вимоги.

До функціональних вимог належить реалізація управління автомобілем за допомогою клавіш на клавіатурі, що дозволяє здійснювати повороти вліво і вправо, а також контролювати прискорення та гальмування. Автомобіль має реагувати на зіткнення з перешкодами — при цьому звучать звукові ефекти удару, а також змінюється візуальний стан гри: з’являється пил або туман, який створює ефект затримки руху. У разі, якщо автомобіль тривалий час їде без аварій, активується режим поступового прискорення, що підвищує динаміку гри. Крім того, при дрифті транспортного засобу на трасі з’являються характерні чорні сліди від шин, що додає візуальної реалістичності.

Музика супроводжує гравця протягом усього ігрового процесу, змінюючись відповідно до ситуацій у грі. Звуки керування, зіткнень та інших дій створюють атмосферу напруги та азарту.

Нефункціональні вимоги передбачають стабільну роботу гри на більшості сучасних пристроїв із базовими технічними характеристиками. Інтерфейс повинен бути простим і зрозумілим для користувача, а також мати мінімалістичний дизайн. Всі візуальні елементи мають бути адаптовані для роботи в 2D-графіці, що сприятиме плавній та безперебійній анімації. Особливу

увагу слід приділити швидкості реакції на керування — вона має бути максимально короткою для збереження чутливості управління.

ВИСНОВКИ ДО РОЗДІЛУ 1

Проведений аналіз підтверджує значущість і динамічний розвиток індустрії відеоігор у сучасному цифровому світі. Зокрема, ігри гоночного жанру демонструють стабільний інтерес з боку користувачів завдяки поєднанню швидкісної динаміки, змагального духу та візуальної привабливості.

У цьому контексті дослідження існуючих ігор, таких як OVERDRIFT FESTIVAL і NitroRally, дозволяє сформувати розуміння актуальних підходів до побудови геймплею, організації ігрового простору та впровадження користувацьких налаштувань. Перша з них, попри належність до 3D-формату, демонструє багатий функціонал, гнучкість кастомізації та глибоку деталізацію, тоді як друга — приклад компактної 2D-гри, орієнтованої на швидкий доступ і простоту взаємодії.

Такий порівняльний аналіз слугує підґрунтям для планування власного проєкту, де ключовими орієнтирами є технічна ефективність, інтуїтивно зрозумілий інтерфейс і захоплюючий ігровий процес.

РОЗДІЛ 2
ПРОЕКТУВАННЯ АРХІТЕКТУРИ ГРИ «ГОНКИ»

2.1. Структурування алгоритму розв’язання задачі

Проєктування гри потребує чіткого планування і поетапної реалізації ключових складових, тому формування загального алгоритму розв’язання задачі є одним із найважливіших етапів. Завдяки структурованому підходу можна не лише забезпечити логічну послідовність у реалізації проєкту, а й уникнути численних технічних труднощів, що виникають під час розробки ігрових додатків. У випадку створення гри жанру «Гонки» важливо визначити основні завдання, засоби їх досягнення та оптимальні способи взаємодії окремих модулів системи.

Початковим кроком виступає аналіз вимог до майбутнього застосунку. На цьому етапі формується загальне уявлення про функціональні можливості гри, її ігрову логіку, правила, спосіб керування та візуальну складову. Для гри «гонки» доцільно передбачити можливість керування автомобілем, присутність перешкод на дорозі, визначення фінішної лінії, а також реалізацію підрахунку часу проходження траси. Крім того, варто врахувати технічні характеристики пристроїв, на яких планується запуск, що допоможе коректно обрати графічну бібліотеку, інструменти для рендерингу, мову програмування та логіку обробки ресурсів.

Наступним логічним етапом є створення концептуальної структури гри. Розробник має визначити набір основних компонентів: графічний рушій, модулі керування, фізичну модель руху, обробку подій, а також головний цикл гри. Цей цикл забезпечує безперервне оновлення стану системи, включаючи отримання введення від користувача, обчислення змін положення об’єктів у просторі, візуалізацію змін на екрані та перевірку умов завершення гри. Важливою складовою є також проєктування взаємодії між модулями, що дозволяє уникнути дублювання коду та підвищити стабільність роботи програми.

У процесі подальшої деталізації алгоритму здійснюється проєктування класів та об’єктів. Виходячи з обраної парадигми об’єктно-орієнтованого програмування, кожен ключовий елемент гри реалізується у вигляді окремого класу. Наприклад, транспортний засіб моделюється через об’єкт, що має властивості позиції, швидкості, напрямку руху та стану (рухається чи стоїть). Під час створення гри трасу визначають як окрему сутність, що зберігає дані про її контури, координати поворотів і розміщення перешкод, які можуть впливати на траєкторію руху гравця. Кожен клас повинен бути максимально ізольованим, але водночас мати можливість взаємодіяти з іншими складовими системи через чітко визначені інтерфейси.

Реалізація механіки гри відбувається на основі визначених раніше принципів. Фізична модель руху транспортного засобу в грі реалізується за допомогою обчислень параметрів таких, як швидкість, прискорення, гальмування, дія сили тертя й інерційні властивості. Одним з ключових елементів є система виявлення зіткнень, що дозволяє фіксувати моменти контакту між різними об’єктами в ігровому середовищі. Наслідком таких зіткнень може бути зменшення швидкості, втрата бонусів або дострокове завершення рівня, що позитивно впливає на занурення у геймплей.

Одночасно з реалізацією основної логіки важливо забезпечити наявність зручного та інтуїтивно зрозумілого інтерфейсу користувача. Інтерфейс може містити панелі зі статистикою, кнопки для переходу між екранами, а також графічні елементи управління. Проєктування інтерфейсу користувача має базуватися на принципах доступності, візуальної привабливості та відповідності дизайнерському оформленню всієї гри. Варто також забезпечити гнучке масштабування графічних елементів під різні екрани й гарантувати коректне відображення незалежно від зміни роздільної здатності пристрою.

Кінцевим етапом структурування алгоритму є організація процесу тестування. Перевірка працездатності гри здійснюється шляхом симуляції різних сценаріїв поведінки користувача: швидкого руху, зіткнень, неправильного керування тощо. Під час тестування виявляються помилки логіки, візуальні

недоліки та можливі збої в роботі програми. За потреби вносяться корективи, а також проводиться оптимізація продуктивності — зменшення кількості обчислень, які виконуються в кожному кадрі, або спрощення рендерингу об'єктів, що не впливають на геймплей.

Ретельно спланована логіка розробки гри дає можливість послідовно реалізовувати її функціональні частини, уникати логічних помилок та зберігати цілісність усієї системи (рис. 2.1).

Рис. 2.1. Алгоритм роботи гри

Опис алгоритму:

1. Аналіз вимог до гри: визначення цілей гри, її жанру, логіки, типу управління, очікуваних візуальних і технічних характеристик. Формування початкових технічних вимог, які стане основою проєктування.

2. Розробка концептуальної структури гри: проєктування базової архітектури: графічного рушія, логіки обробки подій, фізики об'єктів, головного циклу гри, завантаження ресурсів.

3. Проєктування класів і об'єктів: моделювання основних ігрових компонентів (автомобіль, траса, перешкоди, бонуси, об'єкти інтерфейсу) з урахуванням принципів ООП.

4. Реалізація ігрової механіки: кодування функціоналу руху, прискорення, гальмування, зіткнень, системи бонусів. Встановлення взаємодії між об'єктами гри.

5. Проєктування інтерфейсу користувача: розробка елементів GUI (кнопки, меню, індикатори статистики), забезпечення зручності та адаптивності для користувача.

6. Тестування та оптимізація: проведення перевірки роботи гри в різних сценаріях. Виявлення та виправлення помилок, оптимізація продуктивності, покращення зручності користування.

ВИСНОВКИ ДО РОЗДІЛУ 2

Фаза проєктування має критичне значення для розробки гри, оскільки саме тут визначаються ключові модулі, їх структура та способи взаємодії між ними. Завдяки чіткому плануванню та системному підходу вдається сформувати логічну послідовність реалізації, уникнути суперечностей у роботі окремих модулів і забезпечити стабільну роботу всієї системи.

У випадку розробки гри типу «Гонки» важливо заздалегідь визначити набір необхідних компонентів: трасу з її характеристиками, транспортний засіб із відповідною фізикою руху, обробку зіткнень і ігрові події. Реалізація кожного

з елементів у вигляді окремих об’єктів з чітко заданими властивостями дозволяє забезпечити зручність подальшого вдосконалення гри та її підтримки.

Також значну увагу слід приділити зовнішньому вигляду гри — інтерфейс користувача має бути інтуїтивним, гнучким до змін екранних параметрів і відповідати загальному стилю продукту. Завершальний етап передбачає проведення тестування, що дозволяє перевірити правильність роботи всіх складових, виявити технічні недоліки та провести оптимізацію використання системних ресурсів. Таким чином, проєктування — це фундамент, який визначає подальшу якість гри, її зручність для гравця та можливості масштабування.

РОЗДІЛ 3
ОРГАНІЗАЦІЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Структура файлової системи проєкту

Всі необхідні ігрові дані розташовані в кореневій директорії контенту у вигляді логічно структурованої ієрархії папок. Такий підхід до організації ресурсів є розумним вибором, оскільки він дозволяє забезпечити чітке розділення різноманітних типів контенту, таких як графіки, звуки, скрипти, шрифти, карти тощо. Структура виглядає як Y-подібна розгалужена система, з кожною папкою, яка відповідає окремому елементу гри. Це упорядковує проект, зменшує ймовірність помилок у файлах і значно спрощує розробку та підтримку.

Модульність цього підходу є його основною перевагою. Наприклад, оновлення текстур у папці Images не впливає на структуру карт чи логіку скриптів. Це дозволяє розробникам проводити локальні тести змін, не боячись непередбачуваних результатів. Подібна структура полегшує командну роботу, оскільки окремі члени команди можуть працювати незалежно над матеріалами в різних папках. Крім того, це зручно з точки зору автоматизації, оскільки можна пакетно обробляти або зчитувати файли певного типу, просто звертаючись до відповідної директорії.

Основним недоліком цієї схеми може бути невелике початкове ускладнення для початківців, які повинні зрозуміти логіку розподілу ресурсів. Крім того, може статися фрагментація файлів, якщо надто багато дрібних файлів розкидається по великій кількості папок. З іншого боку, ці недоліки можна мінімізувати при правильному плануванні.

Через її універсальність, простоту масштабування та відповідність принципам чистої архітектури було вирішено використовувати саме таку структуру для цього проєкту. Таблиця 3.1 містить опис кожної папки та функціональне призначення для логіки та реалізації гри.

Таблиця 3.1

Опис папки та призначенння для реалізації гри «Гонки»

№	Назва папки	Призначення ресурсу
1	Animations	Зберігання анімацій спрайтів (наприклад, рух авто)
2	Assets	Загальні ресурси, які можуть бути повторно використані у різних сценах
3	Fonts	Шрифти для відображення тексту у грі
4	Images	Зображення, які використовуються як фони, кнопки, текстури
5	Maps	Рівні гри або перешкоди для гоночного середовища
6	Saves	Файли з результатами гри, рекордами, налаштуваннями
7	Scripts	Файли з логікою (якщо використовується скриптова система)
8	Sounds	Аудіофайли: звуки двигуна, зіткнення, музика фону
9	UI	Елементи інтерфейсу: меню, кнопки, панелі
10	Windows	Параметри інтерфейсних вікон, позиції, розміри

Усе добре та конструктивно розписано, дана структура проєкту є логічно розроблено.

3.2. Система збереження даних у грі

У грі використовується проста та надійна система, що базується на зовнішньому текстовому файлі, щоб зберегти прогрес гравця, включаючи його рекорди, досягнення, персональні налаштування та інші важливі параметри. Уся інформація зберігається в одному файлі в директорії контенту/збереження. Це

дозволяє легко копіювати чи переносити дані та відокремлює їх від іншого ігрового контенту.

Для забезпечення запису та зчитування даних у проєкті використовується спеціалізований клас Save, який встановлюється під назвою файлу Save.h. Цей клас може серіювати та десеріювати дані у форматі ключ=значення, який є простим, зручним і гнучким рішенням. Таким чином, файл можна легко редагувати вручну під час тестування або налагодження, а також швидко додати нові параметри, не змінюючи формат.

У цьому файлі кожен рядок є логічною парою, де ключ містить сутність або параметр, а значення містить відповідну інформацію. Наприклад, можна використовувати ключ «BestTime» або «PlayerName», а значення — «01:23.45» або «Ivan». Під час використання цього методу можна легко обробляти записи під час запуску гри або при збереженні нових результатів. Розширення файлу, підтримка кількох ігрових слотів або введення додаткових метаданих є можливостями формату файлу.

Приклад вмісту стандартного файлу record.txt, який зберігається у папці Saves, наведено в таблиці нижче (табл. 3.2).

Таблиця 3.2

Опис вмісту файлу «record.txt»

Функція	**Призначення**
SaveData(...)	Зберігає дані у файл. Підтримує два режими: перезапис та дозапис
LoadData(...)	Зчитує файл построково, перетворює його у вектор пар ключ-значення

Процес збереження та зчитування даних із зовнішніх файлів, які зберігаються в директорії Content/Saves, відповідає класу Збереження. З його допомогою можна працювати з простим текстовим форматом, у якому кожен рядок містить пари ключ=значення. Це забезпечує зручний спосіб зберігання важливих параметрів, таких як імена гравців, рівні та ігрові рекорди. Клас виконує дві основні завдання. По-перше, функція SaveData дозволяє записувати

дані у файл у потрібний спосіб. Вона може виконувати повний перезапис вмісту файлу або дозапис нових рядків до існуючого файлу. Це дозволяє гнучко управляти даними, наприклад, оновлювати існуючі записи або зберігати історію проходження. Друга функція, LoadData, призначена для строкового зчитування файлу.

Вона розділяє кожен рядок на групи ключів і значень, а потім формує вектор із цих пар. Це дозволяє зручно працювати з даними в програмі, легко отримуючи потрібну інформацію для відображення або обробки під час гри. Таким чином, клас Save надає простий, але корисний спосіб працювати з файлами збереження.

Інформаційне забезпечення гри організовано у вигляді структурованої файлової системи, що полегшує управління ресурсами, а механізм збереження гарантує постійний прогрес користувача. Архітектура проекту гнучка, масштабована та зручна для розширення, оскільки клас Save окремо відповідає за взаємодію з файлами у директорії Saves.

Якщо потрібно, можу включити UML-діаграму збереження або розповісти про те, як реалізується безпечне читання та перезапис файлу в умовах багатопотоковості.

ВИСНОВКИ ДО РОЗДІЛУ 3

Було розглянуто внутрішню організацію проєкту та принципи збереження даних у грі. Структура логічна та зручна: кожна папка має чітке призначення, що полегшує розробку, тестування та подальшу підтримку гри. Такий підхід дозволяє не заплутатися навіть у великому проєкті. А система збереження, яка базується на звичайному текстовому файлі, виявилася зручною і надійною — дані легко читати, зберігати і навіть редагувати вручну, якщо потрібно. Загалом, обрана структура проєкту та спосіб роботи з даними виявились вдалими і відповідають потребам гри.

РОЗДІЛ 4
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Загальна характеристика класів

В цьому підпункті буде розглянуто основні класи гри «Гонки». Клас «SlicedImage» використовується для обробки зображень, які складаються з кількох елементів (наприклад, кадрів анімації або частин спрайтів). У цьому класі реалізовано функцію «CalculateVertexes», яка призначена для визначення координат, необхідних для виведення окремих частин зображення на екран.

Цей метод приймає індекс рядка, початкову колонку та, за потреби, максимальну колонку, до якої потрібно виконати побудову. Якщо максимальна колонка не задана, то використовується вся ширина зображення. Початкове значення колонки коригується, якщо воно менше нуля.

Для кожного окремого елементу визначаються координати текстур, зважаючи на розміри кадру, загальний розмір зображення та його позицію в масиві. Ці координати додаються до структури vertexes, яка зберігає інформацію про візуальні межі кожного елемента зображення.

Завдяки цьому методу велике зображення може бути розбите на менші частини, які надалі використовуються окремо під час візуалізації у грі або графічному застосунку.

Фрагмент коду класу «SlicedImage», що відповідають за створення анімацій для вершин (VertexAnimation) на основі розрізаного зображення:

```
std::unique_ptr<VertexAnimation> SlicedImage::CreateVertexAnimation(std::string name, Coord
start_vertex, Coord end_vertex, int duration)
{
	std::vector<std::pair<int, std::vector<Coord>>> frames;
	{
		bool end = false;
		for (int i = start_vertex.Y; i <= end_vertex.Y; i++) {
			for (int j = start_vertex.X; j < imageArraySize.width; j++) {
				if (i == end_vertex.Y && j > end_vertex.X) {
					break;
				}

				const std::pair<Coord, Coord>& vertex = vertexes[i][j];
				frames.push_back(
```

```
                                        std::make_pair(
                                                duration,
                                                std::vector<Coord>({
                                                        vertex.first, vertex.second
                                                        })
                                        )
                                );
                        }
                }
        }
        return std::make_unique<VertexAnimation>(
                name,
                (int)(frames.size() * .7),
                true, false, nullptr, frames
        );
}
std::unique_ptr<VertexAnimation> SlicedImage::CreateVertexAnimation(
        int vertex_row_index,
        std::pair<std::string, int> frame_setting
)
{
        if (vertex_row_index < 0 || vertex_row_index >= vertexes.size()) {
                return nullptr;
        }
        std::vector<std::pair<int, std::vector<Coord>>> frames;
        for (std::pair<Coord, Coord>& vertex : vertexes[vertex_row_index]) {
                frames.push_back(
                        std::make_pair(
                                frame_setting.second,
                                std::vector<Coord>({
                                        vertex.first, vertex.second
                                        })
                        )
                );
        }
        return std::make_unique<VertexAnimation>(
                frame_setting.first,
                (int)(frames.size() * .7),
                true, false, nullptr, frames
        );
}
std::vector<std::unique_ptr<VertexAnimation>> SlicedImage::CreateVertexAnimations()
{
        std::vector<std::string> names;
        std::vector<int> durations;
        for (int i = 0; i < vertexes.size(); i++) {
                std::string name = "slice_" + GenerateRandomString(5);
                names.push_back(name.c_str());
                durations.push_back(300);
        }
        return CreateVertexAnimations(
                std::make_pair(names, durations)
        );
}
std::vector<std::unique_ptr<VertexAnimation>> SlicedImage::CreateVertexAnimations(
        std::pair<std::vector<std::string>, std::vector<int>> frames_settings
)
{
        std::vector<std::unique_ptr<VertexAnimation>> vertex_animations;
        for (int i = 0; i < vertexes.size(); i++) {
                std::vector<std::pair<int, std::vector<Coord>>> frames;
                for (std::pair<Coord, Coord>& vertex : vertexes[i]) {
                        frames.push_back(
                                std::make_pair(
```

```
                                frames_settings.second[i],
                                std::vector<Coord>({
                                        vertex.first, vertex.second
                                })
                        )
                );
        }
        vertex_animations.push_back(
                CreateVertexAnimation(
                        i,
                        std::make_pair(
                                frames_settings.first[i],
                                frames_settings.second[i]
                        )
                )
        );
    }
    return vertex_animations;
}
```

Також розглянемо фрагмент коду функції «CalculateVertexes», даний фрагмент коду відповідає за розрахунок координат текстурних вершин (UV-координат) та розбиття (слайсинг) зображення (текстури) на окремі частини (тайли) для подальшого використання в грі:

```
std::pair<Coord, Coord> SlicedImage::CalculateTextureVertexes(
    Size tileSize,
    Size textureSize,
    int columns,
    int columnIndex)
{
    const int tileWidth = tileSize.width;
    const int tileHeight = tileSize.height;
    const int width = textureSize.width;
    const int height = textureSize.height;
    int atlasX = columnIndex % columns;
    int atlasY = columnIndex / columns;
    float tileU = (float)tileWidth / (float)width;
    float tileV = (float)tileHeight / (float)height;
    return {
        Coord(((float)atlasX + 1.0f) * tileU, 1.0f - ((float)(atlasY + 1) *
(float)tileV)),
        Coord((float)atlasX * (float)tileU, 1.0f - ((float)atlasY * tileV))
    };
}
std::pair<Coord, Coord> SlicedImage::CalculateTextureVertexes(
    Size tileSize,
    Size textureSize,
    Coord vertex_coord
)
{
    const Size imageIndexSize = Size(
        std::roundf(textureSize.width / tileSize.width),
        std::roundf(textureSize.height / tileSize.height)
    );
    return CalculateTextureVertexes(
        tileSize,
        textureSize,
        imageIndexSize.width,
        vertex_coord.Y * imageIndexSize.width + vertex_coord.X
    );
```

```
}
void SlicedImage::Slice(std::vector<int> widths, int height)
{
	for (int i = 0; i < height; i++) {
		CalculateVertexes(i, 0, widths[i]);
	}
}
void SlicedImage::Slice(Size size)
{
	for (int i = 0; i < imageArraySize.height; i++) {
		CalculateVertexes(i, 0, size.width);
	}
}
void SlicedImage::Slice()
{
	for (int i = 0; i < imageArraySize.height; i++) {
		CalculateVertexes(i, 0);
	}
	}
```

Далі, клас «GameMap» виступає центральною ланкою в управлінні ігровим процесом: він створює ігрове середовище, координує взаємодію між об’єктами та забезпечує постійні зміни у грі відповідно до дій користувача. У процесі ініціалізації клас завантажує необхідні графічні ресурси, зокрема зображення перешкод, які надалі будуть з’являтися на ігровому полі. Ці об’єкти зберігаються у спеціальному контейнері, що дозволяє ефективно керувати їх створенням та оновленням.

Окрему увагу приділено створенню об’єкта гравця, для якого встановлюються графічне представлення, фізичні властивості (розміри, швидкість, реакція на зіткнення) та початкова позиція. Кожного кадру фон прокручується вниз, і швидкість цього руху залежить від набраних гравцем балів, що поступово підвищує складність гри. У випадку загибелі гравця, рух фону зупиняється, а на екрані з’являється повідомлення про завершення гри разом з інформацією про поточний і найкращий результати.

Розглянемо фрагмент коду класу, що відповідає цьому опису:

1. Створення гравця:

```
void GameMap::SpawnPlayer()
{
    // Початкова позиція гравця
    Coord playePos = Coord(350, 600);

    std::unique_ptr<Material> playerMaterial = std::make_unique<BaseFigureMaterial>();
```

```
// Шейдер
playerMaterial->SetShader(
    ShadersController::GetShaderID("BaseFigure")
);

// Колір та текстура
playerMaterial->SetDiffuse(Color(1, 1, 1));
playerMaterial->SetDiffuseMap(
    std::make_shared<Image>(
        ImagesController::LoadImg(
            "Content/Assets/Cars/simple/mytire4.png",
            "Car"
        )
    )
);

// Створення Racer (гравець)
racer = std::make_unique<Racer>(
    "Player",
    std::make_unique<BoxCollision>(
        playePos,
        Size(60, 64), // Фізичні розміри
        -1,
        (char*)"Player",
        CollisionTypes::Box
    ),
    std::move(playerMaterial),
    Directions::UP,
    playePos,
    Size(82, 82),  // Візуальні розміри
    20,            // Швидкість?
    500,
    10,
    100,
    100,
    true,
    false,
    false
);
racer->Initialize();

// Камера стежить за гравцем
MainWindow::GetCamera().lock()->SetObservedObj(racer);
```

```
}
```

2. Прокрутка фону:

```
void GameMap::UpdateBackground()
{
    if (racer->IsDead()) {
        scrollSettings.currentScrollSpeed = 0.0f;  // Зупинка при загибелі
    }

    if (!racer->IsDead()) {
        float score = static_cast<float>(racer->GetScore());

        // Залежність швидкості від балів гравця
        float targetSpeed = std::min(scrollSettings.startScrollSpeed + score * scrollSettings.growRate, scrollSettings.maxScrollSpeed);

        scrollSettings.currentScrollSpeed += (targetSpeed - scrollSettings.currentScrollSpeed) * 0.05f;

        float deltaTime = Window::GetTimer()->GetDeltaTime();
        scrollSettings.scrollOffsetY += scrollSettings.currentScrollSpeed * deltaTime;

        scrollSettings.scrollOffsetY = fmod(scrollSettings.scrollOffsetY, 1.0f);
    }

    // Малювання фону
    Window::GetImagesController().lock()->DrawParallaxImage(
        "road",
        Coord(-249, -182),
        Coord(0, scrollSettings.scrollOffsetY),
        Window::GetRenderResolution()
    );
}
```

3. Кінець гри, рахунок та найкращий результат:

```
void GameMap::GameOver()
{
    const int score = racer->GetScore();
    const int bestScore = racer->GetBestScore();

    const int X = 440;

    Fonts::GetFont("NotJamGlasgow")->RenderText(
        L"Game Over!",
```

```
    Coord(X, Window::GetRenderResolution().GetHeight() - 200),
    std::make_unique<FontRenderOptions>(
        1.0f,
        .0f,
        nullptr,
        nullptr,
        Color(0.86f, .7f, .23f)
    )
);

Fonts::GetFont("NotJamGlasgow")->RenderText(
    L"Score: " + std::to_wstring(score),
    Coord(X, Window::GetRenderResolution().GetHeight() - 300),
    std::make_unique<FontRenderOptions>(
        1.0f,
        .0f,
        nullptr,
        nullptr,
        Color(1.0f, .9f, .0f)
    )
);

if (score >= bestScore) {
    Fonts::GetFont("NotJamGlasgow")->RenderText(
        L"New record!",
        Coord(X, Window::GetRenderResolution().GetHeight() - 400),
        std::make_unique<FontRenderOptions>(
            1.0f,
            .0f,
            nullptr,
            nullptr,
            Color(1.0f, .9f, .0f)
        )
    );
}
else {
    Fonts::GetFont("NotJamGlasgow")->RenderText(
        L"Best score: " + std::to_wstring(bestScore),
        Coord(X, Window::GetRenderResolution().GetHeight() - 400),
        std::make_unique<FontRenderOptions>(
            1.0f,
            .0f,
            nullptr,
```

```
                nullptr,
                Color(1.0f, .9f, .0f)
            )
        );
    }

    Fonts::GetFont("NotJamGlasgow")->RenderText(
        L"Press space to restart",
        Coord(X, Window::GetRenderResolution().GetHeight() - 500),
        std::make_unique<FontRenderOptions>(
            1.0f,
            .0f,
            nullptr,
            nullptr,
            Color(1.0f, .9f, .0f)
        )
    );
}
```

Гравець постійно оновлюється: змінюється його положення, перевіряється стан зіткнень та взаємодія з перешкодами. Перешкоди також регулярно оновлюються – вони рухаються вниз або здійснюють горизонтальні коливання. Якщо будь-який з об'єктів досягає нижнього краю екрана, він видаляється, щоб звільнити місце для нових. У разі зіткнення гравця з перешкодою викликається метод загибелі, що призводить до завершення гри.

Рахунок гравця збільшується зі зростанням складності, а відповідна інформація виводиться на екран за допомогою текстових об’єктів. Окрім цього, клас відповідає за створення нових перешкод, контролюючи унікальність їх розташування та перевіряючи, чи є достатньо простору для їхнього додавання.
Тепер розглянемо фрагмнент коду класу «GameMap», що відповідає за рахунок гравця:

1. Збереження та огтримання поточного рахунку:

   ```
   float score = static_cast<float>(racer->GetScore());
   ```

2. Відображення рахунку на екрані:

   В методі `UpdateCounters():`

   ```
   Fonts::GetFont("NotJamGlasgow")->RenderText(
       L"Score: " + std::to_wstring(racer->GetScore()),
   ```

```
    Coord(30, Window::GetRenderResolution().GetHeight() - 100),
    std::make_unique<FontRenderOptions>(...));
```

Та найкращі бали `Best Score:`

```
Fonts::GetFont("NotJamGlasgow")->RenderText(
    L"Best score: " + std::to_wstring(racer->GetBestScore()), ... );
```

В методі `GameOver():`

```
const int score = racer->GetScore();
const int bestScore = racer->GetBestScore();

Fonts::GetFont("NotJamGlasgow")->RenderText(
    L"Score: " + std::to wstring(score), ...);

if (score >= bestScore) {
    Fonts::GetFont("NotJamGlasgow")->RenderText(L"New record!", ...);
} else {
    Fonts::GetFont("NotJamGlasgow")->RenderText(
        L"Best score: " + std::to wstring(bestScore), ...);
}
```

3. Використання рахунку для зміни складності гри в методі `UpdateDifficulty()`:

```
float score = racer->GetScore();
float difficultyFactor = std::min(score / 3000.0f, 1.0f);
adaptiveMinDistance = minSpawnDistance - difficultyFactor * 100.0f;
adaptiveMaxDistance = maxSpawnDistance - difficultyFactor * 150.0f;
currentObstacleSpeed = 10.0f + speedFactor * 7.0f;
```

У підсумку, «GameMap» – це комплексна структура, яка координує рух фону, появу перешкод, керування гравцем та оновлення стану гри.

Клас «Obstacle» відповідає за створення та керування перешкодами в ігровому просторі. Він наслідує клас «Rect» та розширює його функціональність можливістю горизонтального руху (lateral motion) і взаємодією з бонусами (апгрейдами). Клас встановлює початкові значення параметрів руху, створює випадковий апгрейд через фабричний метод «UpgradesFacroty» і забезпечує можливість отримання інформації про рух і анімації.

Фрагмент коду класу «Obstacle», що відповідає за створення, ініціалізацію та управлінням перешкод, що є у грі:

```
void Obstacle::GenerateUpgrade()
{
    upgrade = std::move(UpgradesFacroty::GenerateUpgrade());
}
```

```
void Obstacle::Initialize()
{
    hasLateralMotion = false;

    lateralSpeed = lateralAmplitude = lateralTime = originalX = .0f;
    GenerateUpgrade();
}

Obstacle::Obstacle(std::string title, Coord position, Size size, Color color)
    : Rect(std::move(title), position, size, color)
{
    Initialize();
}

bool Obstacle::HasLateralMotion()
{
    return hasLateralMotion;
}

void Obstacle::SetLateralMotion(bool hasLateralMotion)
{
       this->hasLateralMotion = hasLateralMotion;
}

float Obstacle::GetLateralSpeed()
{
    return lateralSpeed;
}

void Obstacle::SetLateralSpeed(float lateralSpeed)
{
    this->lateralSpeed = lateralSpeed;
}

float Obstacle::GetLateralAmplitude()
{
    return lateralAmplitude;
       }
```

І останній клас в описі «Racer». Клас «Racer» відповідає за логіку поведінки автомобіля у грі. Цей об'єкт може бути під контролем як гравця, так і комп’ютера. У ньому реалізовані основні функції руху, обробки зіткнень, а також керування анімацією, звуками та збереженням ігрового процесу.

При ініціалізації об'єкта встановлюються базові параметри: початкова позиція, швидкість, запас енергії, список анімацій та звукових ефектів. Клас має логіку для перевірки зіткнень із перешкодами та об'єктами, що присутні на ігровій мапі. Під час зіткнення, залежно від обставин, можуть відтворюватися звукові ефекти або змінюватися стан гравця — наприклад, у разі серйозної ситуації це може призвести до його загибелі. Фрагмент коду:

1. Ініціалізація базових параметрів:

```
void Racer::Initialize()
{
```

```
	energy = 100.0f;
	axiosX = axiosY = 0.0f;

	LoadAnimations();  // список анімацій
	LoadEffects();     // ефекти (якщо є)
	LoadAudio();       // звукові ефекти
	LoadSaves();       // завантаження збережень
}
```

2. Перевірка зіткнення з об’єктами:

```
bool Racer::CheckForCollision()
{
	if (isDead) {
		return false;
	}

	if (std::shared_ptr<IGameObject> triggeredObject = GetTriggeredObject(position,
size)) {
		std::cout << "Collision: " << triggeredObject->GetTitle() << std::endl;
		std::cout << "X: " << collision->GetPoints()[0].X << " Y: " << collision-
>GetPoints()[0].Y << std::endl;
		std::cout << "Width: " << collision->GetSize().width << " Height: " <<
collision->GetSize().height << std::endl;

		std::cout << "\nPlayer X: " << position.X << " Y: " << position.Y <<
std::endl;
		std::cout << "Player Width: " << size.width << " Height: " << size.height
<< std::endl;
		return false;
	}

	return true;
}
```

3. Закінчення гри, зіткнення – реакція на смерть:

```
void Racer::Die()
{
	Pawn::Die();
	audioController->Stop("breaking");
	audioController->Stop("ride-1");
	audioController->Stop("ride-2");
	audioController->Stop("start");

	audioController->Play("explose"); // звук вибуху при загибелі
```

```
            if (score > bestScore) {
                  bestScore = score;
                  save->SaveData(
                        "Content/Saves/racer.save",
                        std::vector<std::pair<std::string, std::string>>{
                              {"BestScore", std::to_string(bestScore)}
                        },
                        true
                  );
            }
      }
```

Цей клас відповідає за постійне оновлення позиції автомобіля на екрані, змінюючи її відповідно до натискань клавіш управління (вліво, вправо, вперед, назад). При цьому враховуються швидкість, інерція та бонуси до руху. У грі реалізовано плавні анімаційні переходи та адаптивний звуковий супровід, який змінюється залежно від дій гравця — від запуску мотора до моментів прискорення або гальмування. «Racer» не тільки рухається, а ще й може взаємодіяти з іншими об’єктами, зберігати рекорд у файл та оновлювати візуальне положення на екрані.

«Update» є невід'ємною частиною ігрової логіки: він регулярно викликається під час гри та відповідає за оновлення стану об’єкта відповідно до поточних подій. Він відповідає за всі зміни в об'єкті на кожному кадрі: рух, перевірка енергії, облік очок, перевірка меж карти, оновлення графіки та аудіо. Фрагмент коду методу:

```
void Racer::Update()
{
    float deltaTime = Window::GetTimer()->GetDeltaTime();

    Move();
    //Raycasting();

    if (!isDead) {
        score += (speed + speedBonus) * deltaTime;
        if (speedBonus > 0) {
            speed += speedBonus * deltaTime;
        }
```

```
        float friction = 1.0f * deltaTime;
        speed *= friction;

        if (speed > maxSpeed) speed = maxSpeed;
        if (speed < minSpeed) speed = minSpeed;

        if (energy < 100) {
            energy += 1.0f * deltaTime;
        }

        if (energy > 100) {
            energy = 100.0f;
        }

        Drag(Coord(position.X - axiosX * (speed + speedBonus) * 0.5f, position.Y - axiosY *
(speed + speedBonus) * 0.5f));

        UpdateAudio();
    }
    Draw();
}
```

Клас «GameObjects» створений для того, щоб зібрати в одному місці всі об’єкти, які є в грі. У ньому все просто: якщо з’являється новий об’єкт, його додають у спеціальну колекцію. Додати можна по-різному — окремо, відразу багато, або через розумні вказівники. Є можливість знайти об’єкт за назвою або за назвою з вказаним шаром, якщо таких кілька. Якщо якийсь об’єкт більше не потрібен, його можна просто прибрати, а якщо потрібно повністю оновити гру — є метод, що очищає все. Фрагмент коду, що відповідає за додавання ігрових об’єктів:

```
void GameObjects::Add(IGameObject* gameObject)
{
	std::shared_ptr <IGameObject> obj = std::shared_ptr <IGameObject>(gameObject);
	gameObjects[obj->GetTitleString()] = obj;

	if (std::shared_ptr <class Pawn> pawn = std::dynamic_pointer_cast<class Pawn>(obj)) {
		pawns[pawn->GetTitleString()] = pawn;
	}
}

void GameObjects::Add(class Pawn* pawn)
{
	std::shared_ptr <class Pawn> p = std::shared_ptr <class Pawn>(pawn);
	const std::string& title = p->GetTitleString();
```

```
	pawns[title] = p;
	gameObjects[title] = p;
}

void GameObjects::Add(std::shared_ptr<IGameObject> gameObject)
{
	gameObjects[gameObject->GetTitleString()] = gameObject;

	if (std::shared_ptr<class Pawn> pawn = std::dynamic_pointer_cast<class
Pawn>(gameObject)) {
		pawns[pawn->GetTitleString()] = pawn;
	}
}

void GameObjects::Add(std::shared_ptr<class Pawn> gameObject)
{
	const std::string& title = gameObject->GetTitleString();

	pawns[title] = gameObject;
	gameObjects[title] = gameObject;
}

void GameObjects::Add(std::vector<IGameObject*>* gameObjects)
{
	for (IGameObject*& gameObject : *gameObjects)
	{
		Add(gameObject);
	}
}

void GameObjects::Add(std::vector<std::weak_ptr<IGameObject>>* gameObjects)
{
	for (std::weak_ptr<IGameObject>& gameObject : *gameObjects)
	{
		std::shared_ptr<IGameObject> gameObjectPtr = gameObject.lock();
		if (gameObject.expired() || gameObjectPtr == nullptr) {
			continue;
		}
		Add(gameObjectPtr);
	}
}
```

Усе це дозволяє зробити автомобіль живим і динамічним об’єктом у грі, який реагує на дії користувача або ігрове середовище.

4.2. Опис гри «Гонки»

Почнемо з того, що гру можна інсталювати зі спеціально розробленого сайту, який виступає як сайт-візитка проєкту. Сайт створено не просто для того, щоб користувач міг легко завантажити гру — він також допомагає скласти перше враження про сам проєкт. Темне оформлення сайту відповідає стилю гри й одразу занурює відвідувача в її атмосферу. Користувач може відчути атмосферу

гри ще до моменту її завантаження. Завдяки зручному інтерфейсу, інформативному опису, візуальним матеріалам та чітко поданим системним вимогам, сайт створює позитивний користувацький досвід і підвищує зацікавленість у встановленні гри. (рис. 4.1).

Рис. 4.1. Сайт-візитка гри «Viper»

Верхня частина сайту містить короткий, але інформативний опис гри, який дозволяє користувачу одразу зрозуміти суть ігрового процесу та особливості геймплею. Нижня частина оформлена у вигляді фотопрезентації, що демонструє основні візуальні елементи гри, її графіку та інтерфейс. Фото дають можливість майбутньому гравцю, як гра виглядає в реальності та оцінити її. Окрім того, на сторінці вказані системні вимоги, що допомагає потенційному гравцю переконатися, що гра сумісна з його комп’ютером і буде працювати стабільно. Завершує структуру сайту зручна кнопка для завантаження гри, яка дозволяє швидко перейти до встановлення. (рис. 4.2).

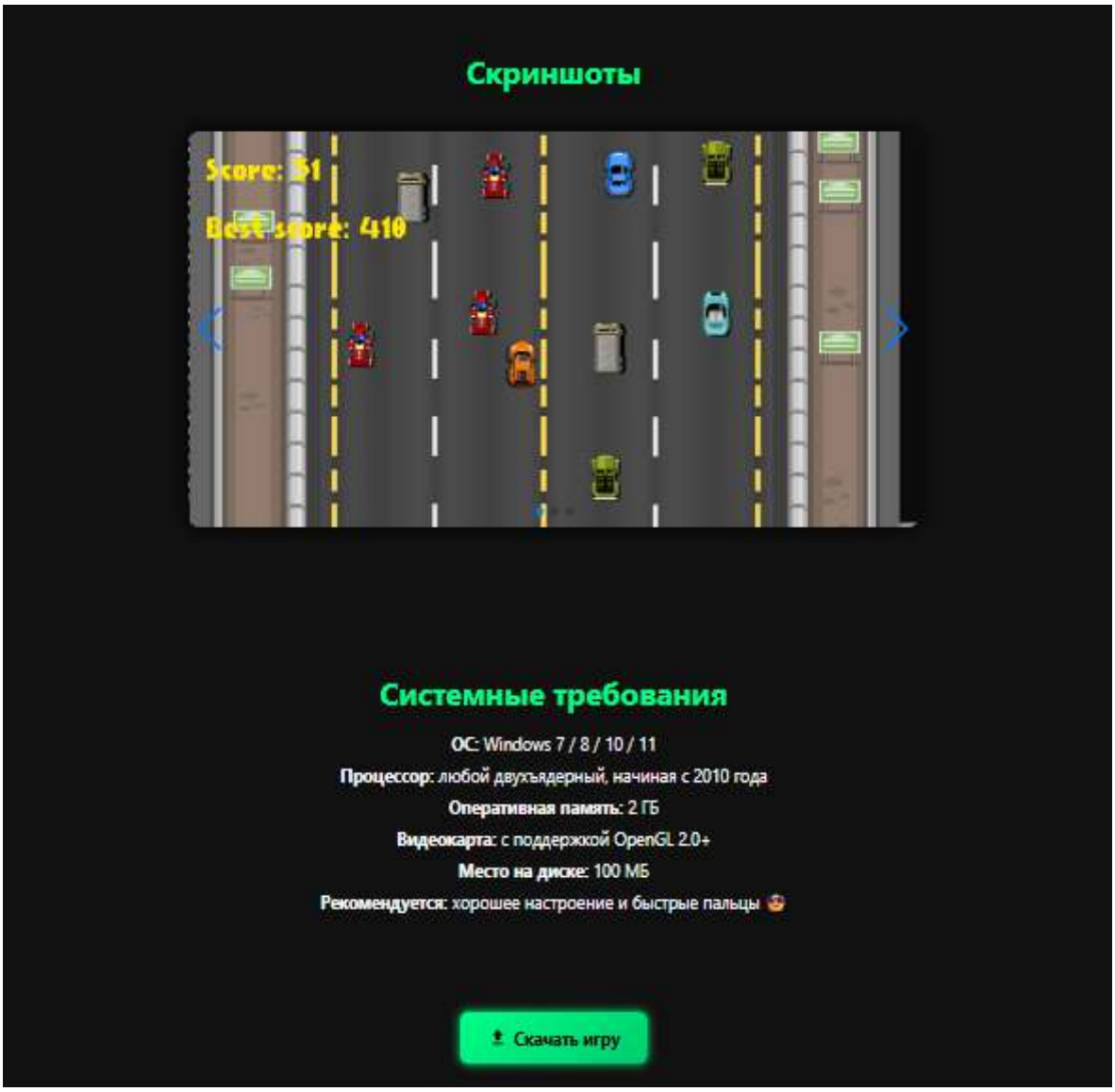

Рис. 4.2. Сайт-візитка гри «Viper»

Відбувається швидке завантаження та встановлення, про це свідчить повідомлення про успішне встановлення гри на ПК (рис. 4.3).

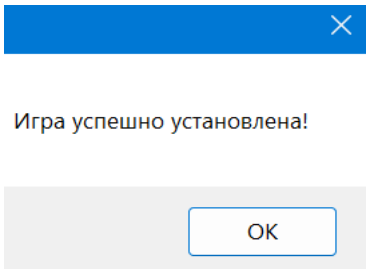

Рис. 4.3. Повідомлення про успішне встановлення гри «Viper»

Запуск гри відбувається швидко, одразу після ввімкнення треба швидко зреагувати та почати грати. Безліч машин, гарна дорога та перешкоди, не врізатися просто неможливо, гри тримає в напруженні та в максимальній

уважності. В лівому кутку видно рахунок та найкращий рахунок гравця, тобто найкращий рахунок за попередні ігри. Звуки двигуну додають адреналін, керування відбувається за допомогою клавіш вправо та вліво. Коли відбувається зіткнення, гра зупиняється і під час зіткнення є відповідний звук (рис. 4.4).

Рис. 4.4. Процес гри «Viper»

Як було вище зазначено, що після зіткнення гра завершується. Після завершення з’являється повідомлення про це з набраними балами, найкращими балами та можливістю почати гру спочатку натиснувши просто на пробіл (рис. 4.5).

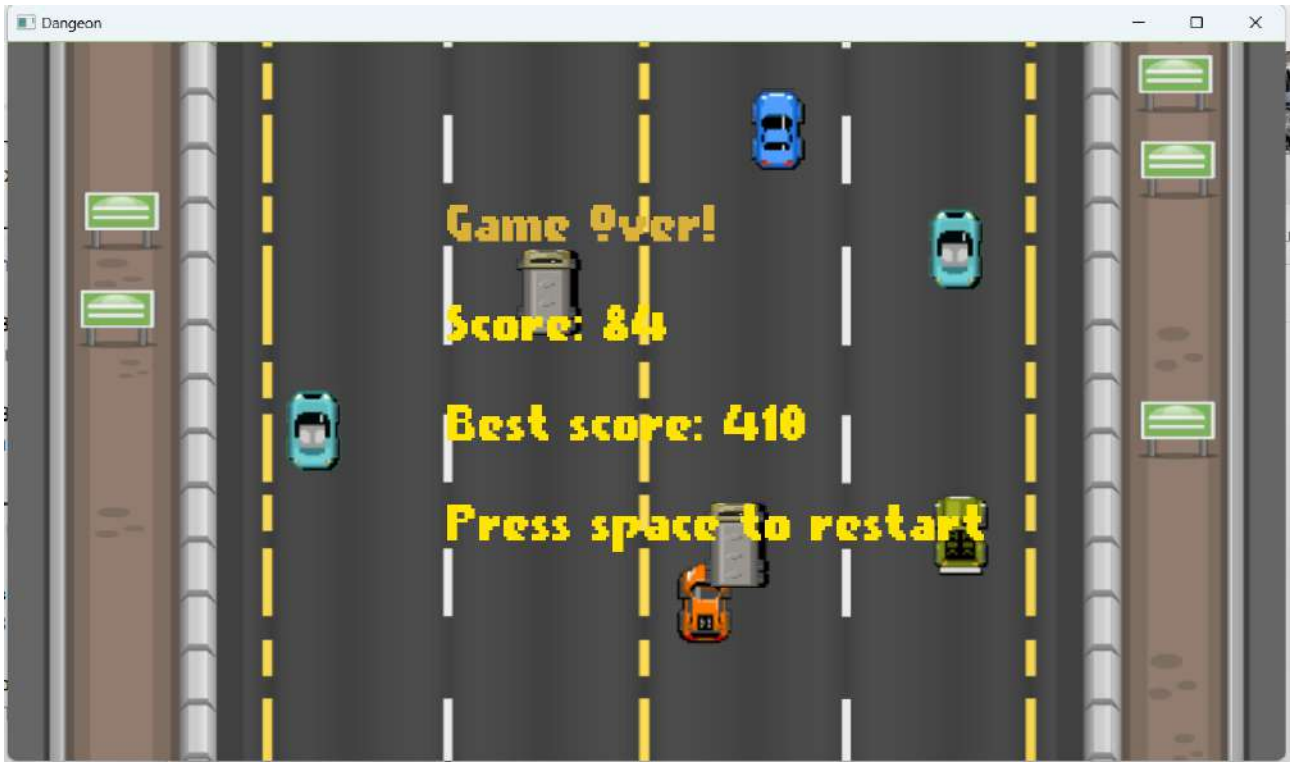

Рис. 4.5. Повідомлення про завершення гри «Viper»

Гра виглядає цікавою, яскравою, з чудовою картою для гонок – це все є важливим для цього жанру гри. Гра відповідає абсолютно всім очікувань без різних багів.

ВИСНОВКИ ДО РОЗДІЛУ 4

Усі ключові частини програмної частини гри були уважно розібрані, а сама гра пройшла перевірку в реальному використанні. Кожен з класів виконує чітко визначену роль у побудові ігрового процесу — від обробки графіки та анімацій до логіки керування та взаємодії з об'єктами. Завдяки чітко побудованій структурі та постійному оновленню станів об’єктів, гра сприймається як жива, динамічна та максимально наближена до реальності. Додаткову глибину проєкту забезпечує сайт-візитка, що дозволяє користувачу ознайомитися з грою ще до встановлення. У підсумку, розроблене програмне забезпечення не тільки відповідає технічним вимогам, а й створює захоплюючий ігровий досвід.

ВИСНОВКИ

У ході виконання дипломної роботи було реалізовано повний цикл створення 2D-аркадної гри в жанрі «гонки» — від аналізу потреб користувачів і формулювання задачі до безпосередньої розробки програмного забезпечення.

Розробка гри на мові C++ із використанням OpenGL дала можливість не лише попрактикуватися у застосуванні алгоритмічного мислення й об’єктно-орієнтованого програмування, а й зануритися у технічні особливості побудови ігрової графіки. Водночас проєктування гри як системи дозволило продумати зручну взаємодію між окремими модулями, забезпечити розширюваність, а також передбачити різні ігрові ситуації, що робить геймплей більш цікавим та динамічним.

У результаті роботи було створено гру, що враховує потреби сучасного користувача: проста у використанні, візуально приваблива, із чіткою логікою поведінки об’єктів, адаптованим інтерфейсом та системою зворотного зв’язку у вигляді анімацій і звуків. Крім того, було створено інсталяційний модуль, який забезпечує можливість встановлення гри на персональні комп’ютери з операційною системою Windows.

Загалом, реалізований проєкт продемонстрував, що навіть порівняно проста 2D-гра вимагає комплексного підходу до проєктування та глибокого розуміння внутрішніх механізмів, починаючи з графіки й закінчуючи взаємодією користувача з ігровим середовищем. Одержані знання та досвід стануть надійним підґрунтям для подальшої роботи в галузі розробки програмного забезпечення та комп’ютерних ігор.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Зеленський А. С., Лисенко В. С., Баран С. В. Методичні вказівки для самостійного вивчення роботи з базами даних на Visual C++ з використанням об’єктів ActiveX Data Object (ADO) / Криворізький економічний інститут ДВНЗ «КНЕУ імені Вадима Гетьмана». — Кривий Ріг: КЕІ, 2008. — 54 с.
2. Яценко О. М. Розробка програмного забезпечення комп’ютерних ігор: теорія та практика / Дніпро: Національний технічний університет, 2020. — 142 с.
3. Шилдт Г. C++: повний курс / Пер. з англ. — К.: Діалектика, 2021. — 976 с.
4. Страуструп Б. Мова програмування C++: принципи та практика використання / Пер. з англ. — К.: Видавництво «Вільямс», 2020. — 1312 с.
5. Стефан Л. Програмування ігор на C++: основи створення графіки та фізики / Пер. з англ. — М.: ДМК Пресс, 2021. — 416 с.
6. Черкашин І. Ю. Основи комп’ютерної графіки та візуалізації на C++ / Київський національний університет ім. Т. Шевченка. — К.: КНУ, 2020. — 88 с.
7. Петренко А. П., Сидоренко І. В. Методичні рекомендації до створення 2D-ігор мовою C++ з використанням бібліотеки SFML / Національний університет «Львівська політехніка». — Львів: ЛП, 2021. — 54 с.
8. Сергеєв І. О. Створення 2D-аркадних ігор мовою C++: навчально-методичний посібник / Дніпровський національний університет. — Дніпро: ДНУ, 2020. — 92
9. Густокашин В. І. Розробка 2D-ігор мовою програмування C++ з використанням бібліотеки SDL / Харківський національний технічний університет. — Харків: ХНТУ, 2022. — 102 с.
10. Фленаган Д. Основи графічного програмування: OpenGL та C++ / М.: Пітер, 2019. — 352 с.
11. Джонсон Д., Мур Б. OpenGL SuperBible: Comprehensive Tutorial and Reference / 7th ed. — Boston: Addison-Wesley, 2015. — 880 p.

ДОДАТКИ

Додаток А

Лістинг програмного коду класу «Racer»

```
#include "Racer.h"

#include "../../../Dodge/utilities/ptrs.h"
#include "../../../Dodge/threads/Thread.h"
#include "../../../Dodge/raycast/RayFactory.h"
#include "../../../Dodge/raycast/Raycast.h"
#include "../../../Dodge/GameObjects.h"

#include "../../../Dodge/collisions/BoxCollision.h"

#include "../Obstacles/Obstacle.h"
#include "../../../Dodge/Save.h"

std::shared_ptr<IGameObject> Racer::GetTriggeredObject(Coord& position, Size& size)
{
	std::unordered_map<std::string, std::shared_ptr<IGameObject>> solidCollisionsObjects = GameObjects::GetAll();

	for (auto& collisionObj : solidCollisionsObjects) {
		if (!collisionObj.second) {
			continue;
		}

		std::shared_ptr<ICollision> collision = collisionObj.second->GetCollision().lock();
		if (!collision || collision->GetType() != CollisionTypes::Obstacle) {
			continue;
		}

		if (collisionObj.second == nullptr || collision == nullptr) {
			continue;
		}

		if (collision->IsCollisionEnter(position, size)) {
			return collisionObj.second;
		}
```

```
	}

	return nullptr;
}

bool Racer::CheckForCollision()
{
	if (isDead) {
		return false;
	}

	if (std::shared_ptr<IGameObject> triggeredObject = GetTriggeredObject(position, size))
{

		//if	(std::shared_ptr<class	Obstacle>	obstacle	=
std::dynamic_pointer_cast<class Obstacle>(triggeredObject)) {
		//	if (obstacle->IsBreakable() && attack) {
		//		obstacle->Break();
		//		return true;
		//	}
		//}

		std::cout << "Collision: " << triggeredObject->GetTitle() << std::endl;
		std::cout << "X: " << collision->GetPoints()[0].X << " Y: " << collision-
>GetPoints()[0].Y << std::endl;
		std::cout << "Width: " << collision->GetSize().width << " Height: " << collision-
>GetSize().height << std::endl;

		std::cout << "\nPlayer X: " << position.X << " Y: " << position.Y << std::endl;
		std::cout << "Player Width: " << size.width << " Height: " << size.height <<
std::endl;
		return false;
	}

	return true;
}

bool Racer::CheckForCollision(Coord position, Size size)
{
	if (isDead) {
		return false;
	}
```

Продовження додатку А

```
	if (std::shared_ptr<IGameObject> triggeredObject = GetTriggeredObject(position, size))
{

		if (std::shared_ptr<class Obstacle> obstacle = std::dynamic_pointer_cast<class
Obstacle>(triggeredObject)) {
			return true;
		}

		std::cout << "Collision: " << triggeredObject->GetTitle() << std::endl;
		std::cout << "X: " << collision->GetPoints()[0].X << " Y: " << collision-
>GetPoints()[0].Y << std::endl;
		std::cout << "Width: " << collision->GetSize().width << " Height: " << collision-
>GetSize().height << std::endl;

		std::cout << "\nPlayer X: " << position.X << " Y: " << position.Y << std::endl;
		std::cout << "Player Width: " << size.width << " Height: " << size.height <<
std::endl;

		return false;
	}

	return true;
}

void Racer::MathSide(double& sideSize, bool isWidth)
{
	const Coord& vertex1 = renderInstance->GetVertex1();
	const Coord& vertex2 = renderInstance->GetVertex2();

	float glDelta = (float)sideSize / (float)Window::GetRenderResolution().GetWidth() *
2.0f;

	if (isWidth) {
		if (sideSize > 0) {
			renderInstance->SetVertex1(
				Coord(vertex1.X + glDelta, vertex1.Y)
			);
		}
		else {
```

```
                renderInstance->SetVertex2(
                        Coord(vertex2.X - glDelta, vertex2.Y)
                );
            }
        }
        else {
            if (sideSize > 0) {
                renderInstance->SetVertex1(
                        Coord(vertex1.X + glDelta, vertex1.Y)
                );
            }
            else {
                renderInstance->SetVertex2(
                        Coord(vertex2.X, vertex2.Y - glDelta)
                );
            }
        }

        size.SetWidth((vertex1.X - vertex2.X) * Window::GetRenderResolution().GetWidth() /
2.0f);
        size.SetHeight((vertex1.Y - vertex2.Y) * Window::GetRenderResolution().GetHeight() /
2.0f);

        position.X = Window::GLXToPixel((vertex1.X + vertex2.X) / 2.0f);
        position.Y = Window::GLYToPixel((vertex1.Y + vertex2.Y) / 2.0f);
}

void Racer::AIMovement()
{
}

void Racer::InitializeRender()
{
        renderInstance->Initialize();
}

void Racer::UpdateAudio()
{
        bool startEnd = false;

        if (std::shared_ptr<Audio> audio = audioController->operator[]("start").lock()) {
```

Продовження додатку А

```
		if (audio->GetState() == AudioStates::STOPPED) {
			audioController->Play("ride-1");
			audio->SetState(AudioStates::FINISHED);
		}
		if (audio->GetState() == AudioStates::FINISHED) {
			startEnd = true;
		}
	}

	if (!startEnd) {
		return;
	}

	if (axiosY == -1.0f) {
		if (std::shared_ptr<Audio> audio = audioController->operator[]("breaking").lock()) {
			if (
				audio->GetState() == AudioStates::INITIALIZED ||
				audio->GetState() == AudioStates::STOPPED ||
				audio->GetState() == AudioStates::FINISHED
			) {
				audioController->Play("breaking");
			}
		}
	}

	if (axiosY == 1.0f) {
		if (std::shared_ptr<Audio> audio = audioController->operator[]("ride-2").lock()) {
			if (
				audio->GetState() == AudioStates::INITIALIZED ||
				audio->GetState() == AudioStates::STOPPED ||
				audio->GetState() == AudioStates::FINISHED
			) {
				audioController->Stop("breaking");
				audioController->Stop("ride-1");
				audioController->Play("ride-2");
			}
		}
	}
```

Продовження додатку А

```
	if (axiosY == .0f) {
		if (std::shared_ptr<Audio> audio = audioController->operator[]("ride-1").lock()) {
			if (
				audio->GetState() == AudioStates::INITIALIZED ||
				audio->GetState() == AudioStates::STOPPED ||
				audio->GetState() == AudioStates::FINISHED
			) {
				audioController->Stop("breaking");
				audioController->Stop("ride-2");
				audioController->Play("ride-1");
			}
		}
	}

	//if (std::shared_ptr<Audio> audio = audioController->operator[]("ride-1").lock()) {
	//	if (audio->GetState() == AudioStates::STOPPED) {
	//		audio->SetState(AudioStates::FINISHED);
	//		audioController->Play(rand() % 2  ? "ride-1" : "ride-2");
	//	}
	//}

	//if (std::shared_ptr<Audio> audio = audioController->operator[]("ride-2").lock()) {
	//	if (audio->GetState() == AudioStates::STOPPED) {
	//		audio->SetState(AudioStates::FINISHED);
	//		audioController->Play(rand() % 2 ? "ride-1" : "ride-2");
	//	}
	//}
}

Racer::Racer(
	std::string title,
	std::shared_ptr<ICollision> collision, std::shared_ptr<Material> material,
	Directions moveDirection, Coord position, Size size,
	float speed, float maxSpeed, float minSpeed, float health,
	float maxHealth, bool isPlayable, bool isKinematic,
	bool isHidden, std::vector<std::shared_ptr<IAnimation>> animations)
	: Pawn(title, collision, material, moveDirection, position, size, speed, maxSpeed, minSpeed, health, maxHealth, isPlayable, isKinematic, isHidden, animations)
{
```

```
}

void Racer::Initialize()
{
	energy = 100.0f;
	axiosX = axiosY = 0.0f;
	LoadAnimations();
	LoadEffects();
	LoadAudio();
	LoadSaves();
}

void Racer::Revive()
{
	Pawn::Revive();

	score = 0.0f;
	audioController->Play("start");
}

void Racer::SetSideSize(Sides sides, bool render)
{
	if (sides.bottom != 0) {
		MathSide(sides.bottom, false);
	}

	if (sides.top != 0) {
		MathSide(sides.top, false);
	}

	if (sides.left != 0) {
		MathSide(sides.left, true);
	}

	if (sides.right != 0) {
		MathSide(sides.right, true);
	}

	if (!render) {
		return;
	}
```

Продовження додатку А

```
	UpdateVertices();
}

const Coord& Racer::GetDistanceTo(IGameObject& gameObject)
{
	return position - gameObject.GetPos();

}

float Racer::GetFloatDistanceTo(IGameObject& gameObject)
{
	return CalculateDistanceWithSize(
		position,
		gameObject.GetPos(),
		gameObject.GetSize()
	);
}

void Racer::LoadAnimations()
{

}

void Racer::LoadEffects()
{

}
void Racer::LoadAudio()
{
	audioController->Load("start", "Content/Sounds/Car/start.wav");
	audioController->Load("ride-1", "Content/Sounds/Car/ride-1.wav");
	audioController->Load("ride-2", "Content/Sounds/Car/ride-2.wav");
	audioController->Load("breaking", "Content/Sounds/Car/breaking.wav");
	audioController->Load("explose", "Content/Sounds/Car/explose.wav");
	audioController->Play("start");
}
void Racer::LoadSaves()
```

Продовження додатку А

```
{
```

```
	save = std::make_unique<Save>();

	std::vector<std::pair<std::string, std::string>> data = save->LoadData("Content/Saves/racer.save");
	for (std::pair<std::string, std::string>& pair : data) {
		if (pair.first == "BestScore") {
			bestScore = std::stof(pair.second);
		}
	}
}

void Racer::Draw()
{
	renderInstance->Render();
}

void Racer::Drag(Coord newPos)
{
	std::cout << "Player X: " << newPos.X << " Y: " << newPos.Y << std::endl;

	if (newPos.X <= 10 || newPos.X >= 750) {
		newPos.X = position.X;
	}

	if (newPos.Y <= 245 || newPos.Y >= 825) {
		newPos.Y = position.Y;
	}

	SetPos(newPos);
	collision->SetPoints({ newPos });

	if (!CheckForCollision(newPos, collision->GetSize())) {
		Die();
		return;
	}
}

void Racer::Move()
{
	std::shared_ptr<Keyboard> keyboard = Window::GetKeyboard().lock();
	headRay.Reset();

	if (!health || !keyboard) {
```

```
		return;
	}

	axiosX = axiosY = .0f;

	if (keyboard->Pressed(KeyboardKeys::A)) {
		axiosX = 1.0f;
	}

	if (keyboard->Pressed(KeyboardKeys::D)) {
		axiosX = -1.0f;
	}

	if (keyboard->Pressed(KeyboardKeys::W)) {
		axiosY = 1.0f;
	}

	if (keyboard->Pressed(KeyboardKeys::S)) {
		axiosY = -1.0f;
	}

	attack = false;
	action = Actions::Run;
}

bool Racer::IsNear(IGameObject& gameObject)
{
	return (std::abs(gameObject.GetPos().X) == std::abs(this->position.X)
		&& std::abs(gameObject.GetPos().Y) == std::abs(this->position.Y)) ||
		(std::abs(gameObject.GetPos().X - this->position.X) <= damageDistance &&
			std::abs(gameObject.GetPos().Y - this->position.Y) <= damageDistance);
}

bool Racer::IsNear(Coord position)
{
	return (std::abs(position.X) == std::abs(this->position.X)
		&& std::abs(position.Y) == std::abs(this->position.Y)) ||
		(std::abs(position.X - this->position.X) <= damageDistance &&
			std::abs(position.Y - this->position.Y) <= damageDistance);
```

Продовження додатку А

```
}
```

```
std::string_view Racer::GetAnimationName()
{
	if (action == Actions::Die) {
		return "Die";
	}

	if (action == Actions::Explose) {
		return "Explose";
	}

	if (action == Actions::Sprint) {
		return "Sprint";
	}

	return "Run";
}

std::string_view Racer::GetAnimationMovementName(Coord& direction)
{
	return std::string_view();
}

std::string_view Racer::GetEffectName()
{
	if (action == Actions::Explose) {
		return "Explose";
	}

	if (action == Actions::Sprint && speedBonus) {
		return "Sprint";
	}

	return "Empty";
}

void Racer::Die()
{
	Pawn::Die();
	audioController->Stop("breaking");
```

Продовження додатку А

```
	audioController->Stop("ride-1");
```

```
	audioController->Stop("ride-2");
	audioController->Stop("start");

	audioController->Play("explose");

	if (score > bestScore) {
		bestScore = score;
		save->SaveData(
			"Content/Saves/racer.save",
			std::vector<std::pair<std::string, std::string>>{
				{"BestScore", std::to_string(bestScore)}
			},
			true
		);
	}
}

void Racer::Update()
{
	float deltaTime = Window::GetTimer()->GetDeltaTime();

	Move();
	//Raycasting();

	if (!isDead) {

		score += (speed + speedBonus) * deltaTime;
		if (speedBonus > 0) {
			speed += speedBonus * deltaTime;
		}

		float friction = 1.0f * deltaTime;
		speed *= friction;

		if (speed > maxSpeed) speed = maxSpeed;
		if (speed < minSpeed) speed = minSpeed;

		if (energy < 100) {
			energy += 1.0f * deltaTime;
		}
```

Продовження додатку А

```
		if (energy > 100) {
			energy = 100.0f;
		}

		Drag(Coord(position.X - axiosX * (speed + speedBonus) * 0.5f, position.Y - 
axiosY * (speed + speedBonus) * 0.5f));

		UpdateAudio();
	}

	Draw();
}

void Racer::UpdateVertices()
{
	renderInstance->UpdateVertices();
}

void Racer::UpdateVertices(std::vector<float>& vertices)
{
	renderInstance->UpdateVertices(vertices);
}

const int& Racer::GetScore()
{
	return score;
}

const int& Racer::GetBestScore()
{
	return bestScore;
}

const int& Racer::GetEnergy()
{
	return energy;
}

void Racer::AddEnergy(int value)
{
```

Продовження додатку А

```
	energy = energy + value > 100
```

```
                ? 100
                : energy + value;
}

const std::type_index& Racer::GetClassTypeId()
{
        return typeid(Racer);
}
```

Додаток Б

Лістинг програмного коду класу «Obstacle»

```
#include "Obstacle.h"
#include "../../../Dodge/animator/AnimationController.h"
#include "../../../Dodge/GameObjects.h"

#include "../Upgrades/UpgradesFacroty.h"

void Obstacle::GenerateUpgrade()
{
    upgrade = std::move(UpgradesFacroty::GenerateUpgrade());
}

void Obstacle::Initialize()
{
    hasLateralMotion = false;

    lateralSpeed = lateralAmplitude = lateralTime = originalX = .0f;
    GenerateUpgrade();
}

Obstacle::Obstacle(std::string title, Coord position, Size size, Color color)
    : Rect(std::move(title), position, size, color)
{
    Initialize();
}

bool Obstacle::HasLateralMotion()
{
    return hasLateralMotion;
}

void Obstacle::SetLateralMotion(bool hasLateralMotion)
{
       this->hasLateralMotion = hasLateralMotion;
}

float Obstacle::GetLateralSpeed()
{
    return lateralSpeed;
}

void Obstacle::SetLateralSpeed(float lateralSpeed)
{
    this->lateralSpeed = lateralSpeed;
}

float Obstacle::GetLateralAmplitude()
{
    return lateralAmplitude;
}

void Obstacle::SetLateralAmplitude(float lateralAmplitude)
{
    this->lateralAmplitude = lateralAmplitude;
}

float Obstacle::GetLateralTime()
{
    return lateralTime;
}
```

Продовження додатку Б

```
void Obstacle::SetLateralTime(float lateralTime)
{
    this->lateralTime = lateralTime;
}

float Obstacle::GetOriginalX()
{
    return originalX;
}

void Obstacle::SetOriginalX(float originalX)
{
    this->originalX = originalX;
}

AnimationController* Obstacle::GetAnimations()
{
    return animations.get();
}

Upgrade* Obstacle::GetUpgrade()
{
    return upgrade.get();
}
```

Додаток В

Лістинг програмного коду класу «GameMap»

```
#include "GameMap.h"

#include "../../Dodge/Color.h"
#include "../../Dodge/materials/figures/BaseFigureMaterial.h"
#include "../../Dodge/shaders/ShadersController.h"
#include "../../Dodge/collisions/BoxCollision.h"
#include "../../Dodge/figures/Rect.h"
#include "../../Dodge/GameObjects.h"
#include "../../Dodge/MainWindow.h"
#include "../../Dodge/font/Fonts.h"

#include "../Scripts/Obstacles/ObstacleFactory.h"
#include <random>

void GameMap::UpdateSpacing()
{
}

void GameMap::InitializeObstacles()
{
    if (obstaclePool.obstacleImages.empty()) {
        obstaclePool.obstacleImages["gentire1"] = ObstacleSetting(
            std::make_shared<Image>(
                ImagesController::LoadImg("Content/Assets/Cars/simple/gentire1.png",
"gentire1")
            ),
            Size(82, 82),
            Size(60, 64)
        );

        obstaclePool.obstacleImages["gentire2"] = ObstacleSetting(
            std::make_shared<Image>(
                ImagesController::LoadImg("Content/Assets/Cars/simple/gentire2.png",
"gentire2")
            ),
            Size(82, 82),
            Size(60, 64)
        );
        obstaclePool.obstacleImages["gentire5"] = ObstacleSetting(
```

Продовження додатку В

```
            std::make_shared<Image>(
                ImagesController::LoadImg("Content/Assets/Cars/simple/gentire5.png",
"gentire5")
            ),
            Size(82, 82),
            Size(60, 64)
        );

        obstaclePool.obstacleImages["mytire14"] = ObstacleSetting(
            std::make_shared<Image>(
                ImagesController::LoadImg("Content/Assets/Cars/simple/mytire14.png",
"mytire14")
            ),
            Size(82, 82),
            Size(60, 64)
        );

        obstaclePool.obstacleImages["mytire15"] = ObstacleSetting(
            std::make_shared<Image>(
                ImagesController::LoadImg("Content/Assets/Cars/simple/mytire15.png",
"mytire15")
            ),
            Size(82, 82),
            Size(60, 64)
        );

        obstaclePool.obstacleImages["mytire16"] = ObstacleSetting(
            std::make_shared<Image>(
                ImagesController::LoadImg("Content/Assets/Cars/simple/mytire16.png",
"mytire16")
            ),
            Size(82, 82),
            Size(60, 64)
        );

        obstaclePool.obstacleImages["trtire1"] = ObstacleSetting(
            std::make_shared<Image>(
                ImagesController::LoadImg("Content/Assets/Cars/simple/trtire1.png",
"trtire1")
            ),
            Size(256, 86),
            Size(64, 64)
```

Продовження додатку В

```
        );
    }

    racer->SetPos(Coord(350, 600));
    float playerY = racer->GetPos().Y;
    for (int i = 0; i < 4; ++i) {
        nextSpawnYPerLane[i] = playerY - startSpawnDistance;
    }

    adaptiveMinDistance = minSpawnDistance;
    adaptiveMaxDistance = maxSpawnDistance;

    currentObstacleSpeed = 5.0f;

    obstaclePool.pool.resize(obstaclePool.maxObstacleCount);
}

void GameMap::SpawnPlayer()
{
    //350, 500
    Coord playePos = Coord(350, 600);

    std::unique_ptr<Material> playerMaterial = std::make_unique<BaseFigureMaterial>();

    playerMaterial->SetShader(
        ShadersController::GetShaderID("BaseFigure")
    );

    playerMaterial->SetDiffuse(Color(1, 1, 1));
    playerMaterial->SetDiffuseMap(
        std::make_shared<Image>(
            ImagesController::LoadImg(
                "Content/Assets/Cars/simple/mytire4.png",
                "Car"
            )
        )
    );

    racer = std::make_unique<Racer>(
        "Player",
        std::make_unique<BoxCollision>(
            playePos,
```

Продовження додатку В

```
            Size(60, 64),
            -1,
            (char*)"Player",
            CollisionTypes::Box
        ),
        std::move(playerMaterial),
        Directions::UP,
        playePos,
        Size(82, 82),
        20,
        500,
        10,
        100,
        100,
        true,
        false,
        false
    );
    racer->Initialize();


    MainWindow::GetCamera().lock()->SetObservedObj(racer);
}

void GameMap::UpdateBackground()
{
    if (racer->IsDead()) {
        scrollSettings.currentScrollSpeed = 0.0f;
    }

    if (!racer->IsDead()) {
        float score = static_cast<float>(racer->GetScore());

        float targetSpeed = std::min(scrollSettings.startScrollSpeed + score *
scrollSettings.growRate, scrollSettings.maxScrollSpeed);

        scrollSettings.currentScrollSpeed += (targetSpeed -
scrollSettings.currentScrollSpeed) * 0.05f;

        float deltaTime = Window::GetTimer()->GetDeltaTime();
        scrollSettings.scrollOffsetY += scrollSettings.currentScrollSpeed * deltaTime;
        scrollSettings.scrollOffsetY = fmod(scrollSettings.scrollOffsetY, 1.0f);
```

```
    }

    Window::GetImagesController().lock()->DrawParallaxImage(
        "road",
        Coord(-249, -182),
        Coord(0, scrollSettings.scrollOffsetY),
        Window::GetRenderResolution()
    );
}

void GameMap::UpdatePlayer()
{
    racer->Update();
    //floor->Update();
}

void GameMap::UpdateCounters()
{
    Fonts::GetFont("NotJamGlasgow")->RenderText(
        L"Score: " + std::to_wstring(racer->GetScore()),
        Coord(30, Window::GetRenderResolution().GetHeight() - 100),
        std::make_unique<FontRenderOptions>(
            1.0f,
            .0f,
            nullptr,
            nullptr,
            Color(1.0f, .9f, .0f)
        )
    );

       Fonts::GetFont("NotJamGlasgow")->RenderText(
              L"Best score: " + std::to_wstring(racer->GetBestScore()),
              Coord(30, Window::GetRenderResolution().GetHeight() - 200),
              std::make_unique<FontRenderOptions>(
                     1.0f,
                     .0f,
                     nullptr,
                     nullptr,
                     Color(1.0f, .9f, .0f)
              )
       );
}
```

Продовження додатку В

```
void GameMap::GameOver()
{
    const int score = racer->GetScore();
    const int bestScore = racer->GetBestScore();

    const int X = 440;

       Fonts::GetFont("NotJamGlasgow")->RenderText(
              L"Game Over!",
              Coord(X, Window::GetRenderResolution().GetHeight() - 200),
              std::make_unique<FontRenderOptions>(
                     1.0f,
                     .0f,
                     nullptr,
                     nullptr,
                     Color(0.86f, .7f, .23f)
              )
       );

    Fonts::GetFont("NotJamGlasgow")->RenderText(
        L"Score: " + std::to_wstring(score),
        Coord(X, Window::GetRenderResolution().GetHeight() - 300),
        std::make_unique<FontRenderOptions>(
            1.0f,
            .0f,
            nullptr,
            nullptr,
            Color(1.0f, .9f, .0f)
        )
    );

    if (score >= bestScore) {
              Fonts::GetFont("NotJamGlasgow")->RenderText(
                     L"New record!",
                     Coord(X, Window::GetRenderResolution().GetHeight() - 400),
                     std::make_unique<FontRenderOptions>(
                            1.0f,
                            .0f,
                            nullptr,
                            nullptr,
                            Color(1.0f, .9f, .0f)
                     )
```

```
            );
        }
    else {
            Fonts::GetFont("NotJamGlasgow")->RenderText(
                L"Best score: " + std::to_wstring(bestScore),
                Coord(X, Window::GetRenderResolution().GetHeight() - 400),
                std::make_unique<FontRenderOptions>(
                    1.0f,
                    .0f,
                    nullptr,
                    nullptr,
                    Color(1.0f, .9f, .0f)
                )
            );
    }

        Fonts::GetFont("NotJamGlasgow")->RenderText(
            L"Press space to restart",
            Coord(X, Window::GetRenderResolution().GetHeight() - 500),
            std::make_unique<FontRenderOptions>(
                1.0f,
                .0f,
                nullptr,
                nullptr,
                Color(1.0f, .9f, .0f)
            )
        );
}

void GameMap::UpdateObstacles()
{
    UpdateDifficulty();

    for (auto& obstacle : obstaclePool.pool)
    {
        if (obstacle && obstacle->GetPos().Y > 940.0f)
        {
            obstacle.reset();
            continue;
        }

        if (obstacle)
```

Продовження додатку В

```
        {
            if (!racer->IsDead()) {
                if (obstacle->HasLateralMotion()) {
                    obstacle->SetLateralTime(obstacle->GetLateralTime()    +    0.05f);    //
deltaTime или фикс

                    float phase = obstacle->GetLateralTime() * obstacle->GetLateralSpeed();
                    float offset = std::sin(phase) * obstacle->GetLateralAmplitude();
                    float newX = obstacle->GetOriginalX() + offset;

                    obstacle->SetPos(Coord(newX,            obstacle->GetPos().Y            +
currentObstacleSpeed));
                }
                else {
                    obstacle->SetPos(Coord(obstacle->GetPos().X,    obstacle->GetPos().Y    +
currentObstacleSpeed));
                }

                if (obstacle->GetPos().Y >= 250.0f &&
                    obstacle->CollisionEnter(*racer)
                )
                {
                    racer->Die();
                }
            }

            if (obstacle->GetPos().Y >= 250.0f)
            {
                obstacle->Update();
            }

            std::cout  <<  "Obstacle  X:  "  <<  obstacle->GetPos().X  <<  "  Y:  "  <<  obstacle-
>GetPos().Y << "\n";
        }
    }

    // Генерация новых вперёд
    UpdateObstacleBuffer();
}

void GameMap::UpdateObstacleBuffer()
{
```

Продовження додатку В

```
    float playerY = racer->GetPos().Y;

    std::vector<int> laneOrder = { 0, 1, 2, 3 };
    std::shuffle(laneOrder.begin(), laneOrder.end(),
std::default_random_engine(std::random_device{}()));

    for (int lane : laneOrder)
    {
        bool shouldSpawn = nextSpawnYPerLane[lane] > playerY - spawnBufferDistance;

        if (!shouldSpawn)
        {
            for (const auto& obstacle : obstaclePool.pool)
            {
                if (!obstacle)
                {
                    shouldSpawn = true;
                    break;
                }
            }
        }

        while (shouldSpawn)
        {
            if (lane == lastUsedLaneIndex)
            {
                lane = GetRandomLaneExcluding(lastUsedLaneIndex);
            }

            float spawnY = nextSpawnYPerLane[lane];
            nextSpawnYPerLane[lane] -= RandomFloat(adaptiveMinDistance, adaptiveMaxDistance);

            GenerateObstacleAtLane(lane, spawnY);

            lastUsedLaneIndex = lane;

            shouldSpawn = false;
            for (const auto& obstacle : obstaclePool.pool)
            {
                if (!obstacle)
                {
                    shouldSpawn = nextSpawnYPerLane[lane] > playerY - spawnBufferDistance;
```

Продовження додатку В

```
                    break;
                }
            }
        }
    }
}

void GameMap::UpdateDifficulty()
{
    float score = racer->GetScore();

    float difficultyFactor = std::min(score / 3000.0f, 1.0f);

    adaptiveMinDistance = minSpawnDistance - difficultyFactor * 100.0f;
    adaptiveMaxDistance = maxSpawnDistance - difficultyFactor * 150.0f;

    adaptiveMinDistance = std::max(adaptiveMinDistance, 80.0f);
    adaptiveMaxDistance = std::max(adaptiveMaxDistance, adaptiveMinDistance + 20.0f);

    float speedFactor = std::min(score / 3000.0f, 1.0f);
    currentObstacleSpeed = 10.0f + speedFactor * 7.0f;
}

void GameMap::GenerateObstacleAtLane(int laneIndex, float spawnY)
{
    int x = obstaclePool.roadPos[laneIndex];

    auto it = obstaclePool.obstacleImages.begin();
    std::advance(it, rand() % obstaclePool.obstacleImages.size());

    std::unique_ptr<class Obstacle> obstacle = ObstacleFactory::Create(
        it->first,
        Coord(x, spawnY),
        it->second.size,
        it->second.collisionSize,
        it->second.texture
    );

    if (RandomFloat(0.0f, 1.0f) < 0.4f)
    {
        obstacle->SetLateralMotion(true);
        obstacle->SetOriginalX(x);
```

```
        obstacle->SetLateralSpeed(RandomFloat(1.0f, 2.5f));
        obstacle->SetLateralAmplitude(RandomFloat(30.0f, 80.0f));
        obstacle->SetLateralTime(RandomFloat(0.0f, 3.14f)); // Случайная стартовая фаза
    }

    // Вставляем в пул
    for (int i = 0; i < obstaclePool.maxObstacleCount; ++i)
    {
        if (!obstaclePool.pool[i])
        {
            obstaclePool.pool[i] = std::move(obstacle);
            return;
        }
    }

    // Опционально: если пул заполнен, заменяем самый дальний
    int farIndex = -1;
    float maxY = -10000.0f;
    for (int i = 0; i < obstaclePool.maxObstacleCount; ++i)
    {
        if (obstaclePool.pool[i] && obstaclePool.pool[i]->GetPos().Y > maxY)
        {
            maxY = obstaclePool.pool[i]->GetPos().Y;
            farIndex = i;
        }
    }

    if (farIndex != -1) {
        obstaclePool.pool[farIndex] = std::move(obstacle);
    }
}

int GameMap::GetRandomLaneExcluding(int excludeIndex)
{
    std::vector<int> lanes;

    for (int i = 0; i < 4; ++i)
    {
        if (i != excludeIndex)
            lanes.push_back(i);
    }
    if (lanes.empty()) return rand() % 4;
```

```
    return lanes[rand() % lanes.size()];
}

float GameMap::RandomFloat(float min, float max)
{
    static std::default_random_engine engine(std::random_device{}());
    std::uniform_real_distribution<float> dist(min, max);
    return dist(engine);
}

void GameMap::Update()
{
    UpdateBackground();

    UpdatePlayer();
    UpdateObstacles();

    if (racer->IsDead()) {
        return GameOver();
    }

    UpdateCounters();
    UpdateSpacing();
}

void GameMap::Initialize()
{
    SpawnPlayer();
    InitializeObstacles();
}

void GameMap::Restart()
{
    if (!racer) {
        return;
    }
    if (!racer->IsDead()) {
        return;
    }
    InitializeObstacles();
    racer->Revive();
}
```

Додаток Г

Лістинг програмного коду класу «SlicedImage»

```
#include "SlicedImage.h"

#include "../pawn/Pawn.h"

void SlicedImage::CalculateVertexes(int row_index, int start_column, int max_column)
{
	bool haveMaxColumn = max_column > 0;

	for (
		int i = (start_column >= 0 ? start_column : 0);
		i < (haveMaxColumn ? max_column : imageArraySize.width);
		i++
	) {
		if (!i) {
			vertexes.push_back(
				std::vector<std::pair<Coord, Coord>>()
			);
		}

		vertexes[row_index].push_back(
			CalculateTextureVertexes(
				frameSize,
				image->GetSize(),
				imageArraySize.width,
				i + (row_index * imageArraySize.width)
			)
		);
	}
}

void SlicedImage::Initilize()
{
	imageArraySize = Size(
		image->GetSize().width / frameSize.GetWidth(),
		image->GetSize().height / frameSize.GetHeight()
	);
}

bool SlicedImage::ValidateIndex(Coord& vertex_coord)
{
	if (vertex_coord.Y < 0 || vertex_coord.Y > vertexes.size() ||
		vertex_coord.X < 0 || vertex_coord.X > vertexes[vertex_coord.Y].size()) {
		return false;
	}
	return true;
}

SlicedImage::SlicedImage(std::shared_ptr<Image> image,
std::vector<std::vector<std::pair<Coord, Coord>>> vertexes, Size frameSize)
{
	this->image = image;
	this->vertexes = vertexes;
	this->frameSize = frameSize;
	this->Initilize();
}

SlicedImage::SlicedImage(std::shared_ptr<Image> image, std::vector<int> widths, int height,
Size frameSize)
{
```

```
	this->image = image;
	this->frameSize = frameSize;
	this->Initilize();
	Slice(widths, height);
}

SlicedImage::SlicedImage(std::shared_ptr<Image> image, Size frameSize)
{
	this->image = image;
	this->frameSize = frameSize;
	this->Initilize();
	Slice();
}

//SlicedImage::~SlicedImage()
//{
//	if (image != nullptr) {
//		delete image;
//	}
//
//	if (!vertexes.empty()) {
//		vertexes.clear();
//	}
//}

std::pair<Coord, Coord> SlicedImage::CalculateTextureVertexes(
	Size tileSize,
	Size textureSize,
	int columns,
	int columnIndex)
{
	const int tileWidth = tileSize.width;
	const int tileHeight = tileSize.height;

	const int width = textureSize.width;
	const int height = textureSize.height;

	int atlasX = columnIndex % columns;
	int atlasY = columnIndex / columns;

	float tileU = (float)tileWidth / (float)width;
	float tileV = (float)tileHeight / (float)height;

	return {
		Coord(((float)atlasX + 1.0f) * tileU, 1.0f - ((float)(atlasY + 1) *
(float)tileV)),
		Coord((float)atlasX * (float)tileU, 1.0f - ((float)atlasY * tileV))
	};
}

std::pair<Coord, Coord> SlicedImage::CalculateTextureVertexes(
	Size tileSize,
	Size textureSize,
	Coord vertex_coord
)
{
	const Size imageIndexSize = Size(
		std::roundf(textureSize.width / tileSize.width),
		std::roundf(textureSize.height / tileSize.height)
	);

	return CalculateTextureVertexes(
		tileSize,
```

```
			textureSize,
			imageIndexSize.width,
			vertex_coord.Y * imageIndexSize.width + vertex_coord.X
		);
}

void SlicedImage::Slice(std::vector<int> widths, int height)
{
		for (int i = 0; i < height; i++) {
			CalculateVertexes(i, 0, widths[i]);
		}
}

void SlicedImage::Slice(Size size)
{
		for (int i = 0; i < imageArraySize.height; i++) {
			CalculateVertexes(i, 0, size.width);
		}
}

void SlicedImage::Slice()
{
		for (int i = 0; i < imageArraySize.height; i++) {
			CalculateVertexes(i, 0);
		}
}

int SlicedImage::GetHeight()
{
		return vertexes.size();
}

std::weak_ptr<Image> SlicedImage::GetImage()
{
		return image;
}

void SlicedImage::SetImage(std::shared_ptr<Image> image, bool copy)
{
		if (copy) {
			*this->image = *image;
			return;
		}

		this->image = image;
}

void SlicedImage::UseDiffuseMapVertexes(Material* material, Coord vertex_coord)
{
		if (!ValidateIndex(vertex_coord)) {
			return;
		}

		material->SetDiffuseMapVerticies(vertexes[vertex_coord.Y][vertex_coord.X]);
}

void SlicedImage::UseNormalMapVertexes(Material* material, Coord vertex_coord)
{
		if (!ValidateIndex(vertex_coord)) {
			return;
		}

		material->SetNormalMapVerticies(vertexes[vertex_coord.Y][vertex_coord.X]);
```

```
}

void SlicedImage::UseSpecularMapVertexes(Material* material, Coord vertex_coord)
{
	if (!ValidateIndex(vertex_coord)) {
		return;
	}

	material->SetSpecularMapVerticies(vertexes[vertex_coord.Y][vertex_coord.X]);
}

void SlicedImage::UseEmissiveMapVertexes(Material* material, Coord vertex_coord)
{
	if (!ValidateIndex(vertex_coord)) {
		return;
	}

	material->SetEmissiveMapVerticies(vertexes[vertex_coord.Y][vertex_coord.X]);
}

std::unique_ptr<VertexAnimation> SlicedImage::CreateVertexAnimation(int vertex_row_index)
{
	std::string name = "slice_" + GenerateRandomString(5);
	return CreateVertexAnimation(
		vertex_row_index,
		std::make_pair(name.c_str(), 1)
	);
}

std::unique_ptr<VertexAnimation> SlicedImage::CreateVertexAnimation(std::string name, Coord
start_vertex, Coord end_vertex, int duration)
{
	std::vector<std::pair<int, std::vector<Coord>>> frames;

	{
		bool end = false;
		for (int i = start_vertex.Y; i <= end_vertex.Y; i++) {
			for (int j = start_vertex.X; j < imageArraySize.width; j++) {
				if (i == end_vertex.Y && j > end_vertex.X) {
					break;
				}

				const std::pair<Coord, Coord>& vertex = vertexes[i][j];

				frames.push_back(
					std::make_pair(
						duration,
						std::vector<Coord>({
							vertex.first, vertex.second
							})
					)
				);
			}
		}
	}

	return std::make_unique<VertexAnimation>(
		name,
		(int)(frames.size() * .7),
		true, false, nullptr, frames
	);
}
std::unique_ptr<VertexAnimation> SlicedImage::CreateVertexAnimation(
```

```
	int vertex_row_index,
	std::pair<std::string, int> frame_setting
)
{
	if (vertex_row_index < 0 || vertex_row_index >= vertexes.size()) {
		return nullptr;
	}

	std::vector<std::pair<int, std::vector<Coord>>> frames;

	for (std::pair<Coord, Coord>& vertex : vertexes[vertex_row_index]) {
		frames.push_back(
			std::make_pair(
				frame_setting.second,
				std::vector<Coord>({
					vertex.first, vertex.second
					})
			)
		);
	}

	return std::make_unique<VertexAnimation>(
		frame_setting.first,
		(int)(frames.size() * .7),
		true, false, nullptr, frames
	);
}

std::vector<std::unique_ptr<VertexAnimation>> SlicedImage::CreateVertexAnimations()
{
	std::vector<std::string> names;
	std::vector<int> durations;

	for (int i = 0; i < vertexes.size(); i++) {
		std::string name = "slice_" + GenerateRandomString(5);
		names.push_back(name.c_str());
		durations.push_back(300);
	}

	return CreateVertexAnimations(
		std::make_pair(names, durations)
	);
}

std::vector<std::unique_ptr<VertexAnimation>> SlicedImage::CreateVertexAnimations(
	std::pair<std::vector<std::string>, std::vector<int>> frames_settings
)
{
	std::vector<std::unique_ptr<VertexAnimation>> vertex_animations;

	for (int i = 0; i < vertexes.size(); i++) {
		std::vector<std::pair<int, std::vector<Coord>>> frames;

		for (std::pair<Coord, Coord>& vertex : vertexes[i]) {
			frames.push_back(
				std::make_pair(
					frames_settings.second[i],
					std::vector<Coord>({
						vertex.first, vertex.second
					})
				)
			);
		}
```

Продовження додатку Г

```
			vertex_animations.push_back(
				CreateVertexAnimation(
					i,
					std::make_pair(
						frames_settings.first[i],
						frames_settings.second[i]
					)
				)
			);
		}

		return vertex_animations;
}

std::vector<std::vector<std::pair<Coord, Coord>>>::iterator SlicedImage::begin()
{
		return vertexes.begin();
}

std::vector<std::vector<std::pair<Coord, Coord>>>::iterator SlicedImage::end()
{
		return vertexes.end();
}

std::vector<std::pair<Coord, Coord>>* SlicedImage::operator[](int index)
{
		if (index >= 0 && index < vertexes.size()) {
			return &vertexes[index];
		}
		return nullptr;
}
```

Додаток Д

Лістинг програмного коду класу «GameObjects»

```
#include "GameObjects.h"

#include "Layers.h"
#include "pawn/Pawn.h"

std::unordered_map<std::string, std::shared_ptr<IGameObject>> GameObjects::gameObjects;
std::unordered_map<std::string, std::shared_ptr<class Pawn>> GameObjects::pawns;

void GameObjects::Add(IGameObject* gameObject)
{
	std::shared_ptr <IGameObject> obj = std::shared_ptr <IGameObject>(gameObject);
	gameObjects[obj->GetTitleString()] = obj;

	if (std::shared_ptr <class Pawn> pawn = std::dynamic_pointer_cast<class Pawn>(obj)) {
		pawns[pawn->GetTitleString()] = pawn;
	}
}

void GameObjects::Add(class Pawn* pawn)
{
	std::shared_ptr <class Pawn> p = std::shared_ptr <class Pawn>(pawn);
	const std::string& title = p->GetTitleString();

	pawns[title] = p;
	gameObjects[title] = p;
}

void GameObjects::Add(std::shared_ptr<IGameObject> gameObject)
{
	gameObjects[gameObject->GetTitleString()] = gameObject;

	if (std::shared_ptr<class Pawn> pawn = std::dynamic_pointer_cast<class
Pawn>(gameObject)) {
		pawns[pawn->GetTitleString()] = pawn;
	}
}

void GameObjects::Add(std::shared_ptr<class Pawn> gameObject)
{
	const std::string& title = gameObject->GetTitleString();

	pawns[title] = gameObject;
	gameObjects[title] = gameObject;
}

void GameObjects::Add(std::vector<IGameObject*>* gameObjects)
{
	for (IGameObject*& gameObject : *gameObjects)
	{
		Add(gameObject);
	}
}

void GameObjects::Add(std::vector<std::weak_ptr<IGameObject>>* gameObjects)
{
	for (std::weak_ptr<IGameObject>& gameObject : *gameObjects)
	{
		std::shared_ptr<IGameObject> gameObjectPtr = gameObject.lock();
		if (gameObject.expired() || gameObjectPtr == nullptr) {
```

Продовження додатку Д

```
                        continue;
                }
                Add(gameObjectPtr);
        }
}

void GameObjects::Erase(std::string title)
{
        gameObjects.erase(title);
        pawns.erase(title);
}

std::weak_ptr<IGameObject> GameObjects::GetByTitle(std::string_view title)
{
        auto it = gameObjects.find(std::string(title));
        if (it == gameObjects.end()) {
                return std::shared_ptr<IGameObject>(nullptr);
        }

        return it->second;
}

std::weak_ptr<IGameObject> GameObjects::GetByTitle(std::string_view title, Layer layer)
{
        for (const auto& pair : gameObjects)
        {
                const auto& obj = pair.second;
                if (obj && obj->GetTitle() == title && obj->GetLayer() == layer)
                {
                        return obj;
                }
        }
        return {};
}

std::unordered_map<std::string, std::shared_ptr<IGameObject>>& GameObjects::GetAll()
{
        return gameObjects;
}

std::unordered_map<std::string, std::shared_ptr<IGameObject>>& GameObjects::GetAll(Layer
layer)
{
        std::unordered_map<std::string, std::shared_ptr<IGameObject>> result;

        for (auto& pair : gameObjects)
        {
                if (pair.second->GetLayer() == layer)
                {
                        result[pair.second->GetTitleString()] = pair.second;
                }
        }
        return result;
}

std::unordered_map<std::string, std::shared_ptr<class Pawn>>& GameObjects::GetAllPawns()
{
        return pawns;
}

std::unordered_map<std::string, std::shared_ptr<class Pawn>>& GameObjects::GetAllPawns(Layer
layer)
```

```
{
	std::unordered_map<std::string, std::shared_ptr<class Pawn>> result;

	for (auto& pair : pawns)
	{
		if (pair.second->GetLayer() == layer)
		{
			result[pair.second->GetTitleString()] = pair.second;
		}
	}
	return result;
}

void GameObjects::Clear()
{
	gameObjects.clear();
	pawns.clear();
}
```