Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Антіпова Ігоря Дмитровича
Прізвище, ім'я, по батькові студента(ки)

на здобуття ступеня вищої освіти Бакалавра

«Програмне забезпечення симулятора автоперегонів на базі Unity»
Назва теми

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.2101068.01.01.ПЗ

Виконав студент IV курсу, група ІПЗ-21-1 _____ Ігор АНТІПОВ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник канд. пед. наук, доцент _____ Оксана ЯШИНА
Науковий ступінь, звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер канд. пед. наук, доцент _____ Юрій ФОРКУН
Науковий ступінь, звання Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

5 червня 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
2.01 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Антіпову Ігорю Дмитровичу
Прізвище, ім'я, по батькові студента

1. Тема кваліфікаційної роботи Програмне забезпечення симулятору автоперегонів на базі Unity

Керівник кваліфікаційної роботи Яшина Оксана Миколаївна, канд. тех. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 07.02.2025 р. № 23
2. Строк подання студентом роботи на кафедру 01.06.2025 р.
3. Вихідні дані до роботи Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області, постановка задачі, проєктування програмного забезпечення симулятора автоперегонів на базі Unity, реалізація та тестування
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень) Презентаційні матеріали (слайди, 18 шт.), діаграма варіантів використання (Use Case), багаторівнева діаграма архітектури ігрової системи (за принципами MVC), спрощена UML-діаграма модулів, повна діаграма класів системи генерації, загальна діаграма логіки гри, діаграма процесів фізичної симуляції, діаграма логіки проходження заїзду, діаграма компонентів ігрового застосунку, діаграма потоків даних (DFD), діаграма послідовності для взаємодії з трасою, серія з 4 скріншотів з інтерфейсом HUD та процедурно згенерованими трасами, макети GUI (головне меню та ігровий HUD), таблиця порівняння технологій розробки, таблиця системних вимог до гри.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Форкун Ю.В	12.05.2025	28.05.2025
Антиплагіат	Форкун Ю.В	12.05.2025	28.05.2025

7. Дата видачі завдання «07 » лютого 2025р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Ознайомлення з тематикою дипломного проєктування, визначення та узгодження індивідуальних тем кваліфікаційних робіт (КвР)	01.12 – 31.12.2024	
2 Збір матеріалу за темою КвР; дослідження предметної області, в якій планується використання програмного забезпечення (ПЗ), визначення задач та вимог, розробка технічного завдання	01.01 – 20.02.2025	
3 Проєктування програмного забезпечення	21.02 – 20.03.2025	
4 Програмна реалізація з використанням відповідних засобів розробки. Тестування ПЗ	21.03 – 30.04.2025	
5 Написання вступу, загальних висновків, оформлення переліку джерел посилання та додатків. Оформлення пояснювальної записки КвР згідно вимог	01.05 – 25.05.2025	
6 Попередній захист КвР	Травень 2025	Згідно графіка
7 Перевірка КвР на плагіат, нормоконтроль, отримання відгуків, рецензій та інших супровідних документів. Брошурування (зшиття) пояснювальної записки.	26.05 – 30.05.2025	
8 Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії ХНУ; підготовка до захисту та захист КвР	з 01.06.2025	

Студент _____ Антіпов І. Д.
Підпис Ініціали, прізвище

Керівник роботи _____ Яшина О. М.
Підпис Ініціали, прізвище

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Програмне забезпечення симулятора автоперегонів на базі Unity».

Автор роботи: Антіпов Ігор Дмитрович.

Керівник роботи: Яшина Оксана Миколаївна.

Пояснювальна записка: 68 с., 11 рис., 3 табл., 2 дод., 42 джерела.

Графічна частина: 17 слайдів.

У дипломній роботі розглянуто процес проєктування та розробки програмної системи процедурної генерації трас для гоночної гри в середовищі Unity. Темою кваліфікаційної роботи є проєктування та розробка інструменту, що автоматично формує унікальні ігрові маршрути з урахуванням складності, фізики руху, параметрів середовища та навичок користувача.

В межах роботи проведено аналіз предметної області, класифікацію типів гоночних ігор, досліджено існуючі рішення у сфері генерації контенту. Визначено вимоги до системи. На основі отриманих даних побудовано архітектуру застосунку, розроблено модульну структуру, що включає генератор геометрії треку, аналітику складності, генератор оточення, модуль перевірки прохідності, фізичну модель руху та інтерфейс користувача.

Програмна реалізація виконана мовою C# із використанням рушія Unity, що дозволило забезпечити гнучкість у проєктуванні, кросплатформеність, а також ефективну візуалізацію та інтеграцію графічного інтерфейсу. Проведено всебічне тестування працездатності системи: модульне, інтеграційне, навантажувальне.

Результатом роботи є працездатний прототип гри з процедурною генерацією трас, що забезпечує високу здатність до реітерації, стабільність та можливість масштабування. Отримані результати можуть бути застосовані в подальших дослідженнях, інді-розробках або як основа для продуктів у жанрі «racing».

05.06.2025
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ.2101068.01.01.ПЗ	Пояснювальна записка	90		
2	А4		Завдання на кваліфікаційну роботу	1		
3	А4		Анотація	1		
			Графічні документи			
4	А4	КвРІПЗ.2101068.01.01.Е8	Діаграма процесів гри	1		
5	А4	КвРІПЗ.2101068.01.01.Е8	Діаграма класів	1		
6	А4	КвРІПЗ.2101068.01.01.Е8	DFD-діаграма	1		
7	А4	КвРІПЗ.2101068.01.01.Е8	Презентаційні матеріали	17		

Змн.	Арк.	№ докум.	Підпис	Дата	КвРІПЗ.2101068.01.01.ПЗ			
Виконав		Антіпов І.Д		04.06	Програмне забезпечення симулятора автоперегонів на базі Unity. Пояснювальна записка	Літ.	Арк.	Аркушів
Керівник		Яшина О.М.		04.06			1	1
						ХНУ, ІПЗ-21-1		
Н. Контр.		Форкун Ю.В		04.06				
Зав. Каф.		Бедратюк Л.П.		05.06				

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

AI – Artificial intelligence

API – Application Programming Interface

CPU – Central Processing Unit

DFD – Data Flow Diagram

FPS – Frames Per Second

GPU – Graphical Processing Unit

GUI – Graphical User Interface

IDE – Integrated Development Environment

LOD – Level of Detail

ВСТУП

В наш час відеоігри давно вже зайняли поважне місце серед як серйозних видів спорту, так і опцій для гарного проведення вільного часу. Ця індустрія розвивається кожного року, і щоразу генерує все кращі і кращі результати. Безперервно продовжують випускатися нові проекти тому розробники змушені придумувати кожного разу все більш і більш креативні рішення для привертання і утримання уваги гравців. Одним з варіантів виходу для розробників стала процедурна генерація [1]. Це метод за допомогою якого розробники створюють унікальний контент на основі певних існуючих алгоритмів а не самостійно. Це доволі актуально для ігор жанру «перегони», де заранння створений розробниками набір трас може швидко стати не цікавим для гравця, також повторюваність мап в такому випадку буде тільки ще більше погіршувати ситуацію. Але якщо кожна сесія пропонує новий, до цього не бачений виклик гравці будуть проводити набагато більше часу в грі і повертатися знову і знову.

Гоночні ігри є дуже популярною галуззю через те, що вони вміло можуть поєднувати адреналін, відчуття швидкості та суперництва, також на це має величезний вплив правильно спроєктована гоночна траса, це є навіть більш важливим аспектом ніж автомобілі чи їх механіка керування. Майже всі ігри, які були створені за весь час мають заранння підготовані розробниками треки для заїздів, через що, з часом вони набридають гравцям, через шо вони шукають додаткового контенту або в редакторі мап, або в інших іграх. Використання процедурної генерації треку дозволить гравцеві кожного разу отримувати новий досвід гри [9].

Основною метою роботи є проєктування та розробка інструменту, що автоматично формує унікальні ігрові маршрути з урахуванням складності, фізики руху, параметрів середовища та навичок користувача Тому для цього потрібно буде дізнатися як створити правильні алгоритмі на прикладі вже

існуючих рішень, вибрати які алгоритми будуть найкраще виконувати цю задачу, та як збалансувати ігрову логіку та варіативність дороги.

В процесі створення гри ставляться такі задачі:

– проаналізувати вже існуючи сучасні рішення процедурної генерації що вже використовуються в іграх;

– переглянути за якою логікою створюється дизайн трас у сучасних та класичних іграх такого характеру;

– зробити вибір алгоритму, шо дозволить створити збалансовані та цікаві маршрути;

– розробити прототип гри, де можна буде продемонструвати та переглянути використані ідеї;

– виконати тестування та зробити висновок, наскільки правильним було обране рішення;

З структурного боку, робота має в собі декілька ключових розділів. Для старту буде розказано про теоретичну частину, пояснено що таке процедурна генерація, де її використовують, для чого, які плюси і мінуси вона має. Після чого буде проведено аналіз варіантів проєктування треків, в кінці будуть показані результати, оцінено ефективність взятого варіанту генерації та розглянуто напрямки, в яких в подальшому зможе розвиватися проєкт.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ.

1.1 Аналіз предметної області, функціональних і структурних особливостей.

Такий жанр ігор як Racing game займають значну частину ігрового ринку так як вони маю в собі чудове поєднання високої динамічності та доволі доступні для аудиторії гравців. Ця віха індустрії заманює до себе як фанатів реалістичних симуляторів, з від сканованими реальними треками у вигляді мап, та фізичної моделі автівки де кожен градус температури резини міняє сотні факторів під час руху. Так і любителів Mario Cart та інших аркадних перегонів. Саме дуже обширна варіація підходів до створення таких ігор дозволяє їм як зберігати популярність серед старих гравців, так і отримувати увагу від нових користувачів, які раніше не були дуже зацікавленими у цій галузі індустрії. Також за допомогою цієї різноманітності розробники наділяють свої проєкти унікальним ігровим досвідом. Гравці звикли ділити такі ігри на ти фундаментальні типи.

Першим з цих типів є симулятори. Симуляторні гоночні ігри, це ігри які роблять головний акцент усього геймплею на схожості того що діється з реальністю. Вони намагають максимально точно симулювати рух автомобілю, створюють дуже складні фізичні моделі автівок. Підписують контракти з реальними брендами виробників авто. До таких ігор відносять такі проєкти як IRacing, Assetto Corsa, Серія DIRT Rally, Forza Motorsport, Gran Turismo.

Другим типом є аркадні ігри. Ці ігри менш реалістичні, але розробники таких ігор ставлять акцент на максимальній динамічності, простоті і видовищності геймплею, також велика увага приділяється доступності гри для великої аудиторії. До цього типу відносять такі ігри як серія Forza Horizon, Need for Speed, GRID. Фізичні моделі цих ігор використовують більш спрощену фізику автомобілів та часто роблять акцент на відкритому ігровому світі.

Ще одним типом є картингові ігри. Це ігри які розбавляють звичайну ідею перегонів своїми додатковими механіками типу різних бонусів чи зброї які гравець може підбирати під час заїзду.

Яким би не був підтип цього жанру, але головним елементом постійно буде виступати саме трек на якому буде проходити заїзд. Саме траси, їхня варіативність, дизайн та якість найбільше впливають на інтерес гравця.

Основною ключовою функціональною вимогою до процедурної системи генерації треків є створення мапи з високою динамікою, яка буде змінюватися кожного разу, коли минула сесія буде завершуватися. Така схема дозволить позбутися традиційної повторюваності рівнів , що на разі є стандартом для сучасних і більш класичних ігор з заранння створеними мапами. Алгоритм який буде генерувати карту має певну кількість параметрів які він повинен враховувати. Для прикладу такими параметрами є довжина траси, її складність.

Для того щоб алгоритм міг генерувати достатньо варіативні та одночасно можливі для проходження дуже важливо є досягти балансу між випадковістю та логічною структурою дороги. Якщо цей баланс буде порушено, в процесі гри гравець часто буде стикатися з занадто різкими поворотами які неможливо подолати належним чином, незрозумілими перепадами висот, об’єктами які будуть закрити дорогу в місцях, де вони не повинні цього робити. До того ж система повинна вміти правильно підбирати та розставляти різні об’єкти по типу пішоходів, знаків, та інших декоративних елементів які будуть підсилювати відчуття швидкості.

До того ж дуже важливим є можливість трас адаптуватися під рівень навичок гравця, що дасть можливість змінювати складність маршруту та робити геймплей цікавішим. У підсумку процедурна генерація повинна створювати мапи які будуть відповідати заданим потребам вище, також потрібно буде звернути достатньо уваги для оптимізації процесу генерації тому що він може бути достатньо ресурсовитратним для комп’ютера гравця [7].Отже основними функціональними можливостями є:

– формування маршруту за допомогою алгоритму – система має генерувати унікальні треки за заданими умовами.

– балансування між хаосом і структурою – треки які буде будувати алгоритм повинні зберігати логічну структуру маршруту, без різких поворотів або відрізків які неможливо пройти, але залишатися унікальними.

– спавн об’єктів – алгоритм має створювати на трасі перешкоди в реальному часі.

– адаптивність – генератор повинен підлаштовуватися під майстерність гравця змінюючи характеристики дороги.

– оптимізація – гра має видавати стабільну картинку і працювати достатньо добре для комфортної гри

Процес процедурної генерації трас є доволі складним, що включає в себе ключові компоненти, які виконують свої ролі в створенні локації. Першим кроком генератор генерує каркас конфігурацію треку за допомогою певних алгоритмів і підходів. Прикладом таких підходів є сплайн, шумові алгоритми чи точки з інтерполяцією. Потім система, отримавши певну інформацію, починає аналізувати її та регулювати різні фактори, такі як кількість поворотів, схилів, підйомів, та тому подібних факторів для того, щоб забезпечити нормальний баланс у геймплеї.

Після цього генератор оточення розпочинає свою роботу: він додає візуальні елементи, такі як ландшафт, різноманітні об’єкти та будівлі, щоб забезпечити глибше занурення гравця у процес. Паралельно фізична модель прораховує взаємодію автомобіля з трасою, зважаючи на особливості рельєфу та покриття.

Для запобігання створенню непрохідних треків (тобто таких, які неможливо завершити), буде реалізована система перевірки на прохідність. Вона аналізуватиме згенеровану трасу та визначатиме, чи здатен транспортний засіб подолати її повністю без порушення основних фізичних ігрових принципів та правил.

1.2 Аналіз існуючих програмних рішень

Використання процедурної генерації трас в іграх такого типу допомогло дуже збільшити межі жанру, представило гравцям велику кількість нових можливостей. Але ця ідея не є новітньою чи революційною, раніше вже була створена доволі велика кількість перегонів які тим чи іншим чином використовували процедурну генерацію. У цій частині роботи буде переглянуто популярні рішення для використання процедурної генерації, які плюси вони принесли в гру та з якими мінусами і труднощами зустрілися розробники, які вирішили використовувати цей метод.

Прикладом гри з генерацією локацій такого типу приходить відома гра TruckMania. В цьому проєкті було використано підхід, при якому для створення трас використовуються модульні частинки, заранння створені розробниками або гравцями в редакторі. Або також є режим повністю автоматичної генерації треку з наявних у грі ассетів. Плюсом цього підхожу є велика варіативність доріг. Це зображено на рисунку 1.1

Рисунок 1.1 – Приклад згенерованої траси в TruckMania

Також варто згадати Dirt Rally, ця грає є повною протилежністю TruckMania, але також вміло використовує процедурну генерацію спец. ділянок. У цьому випадку розробники заранння власними силами створили декілька коротких ділянок які віддали під керування алгоритму, який вирішує де будуть ці ділянки і як їх поєднувати між собою. Розробники гарно впоралися з балансом хаосу, так як в цій грі фізична модель є дуже реалістичною, невдалі поєднання ділянок за допомогою алгоритму швидко б виводили авто гравця з ладу, що не дуже позитивно б сприяло на якість геймплею. Прикладом є фото на рисунку 1.2

Рисунок 1.2 – Приклад траси в DIRT Rally.

1.3 Визначення функціональних та нефункціональних вимог

Визначення вимог до системи процедурної генерації трас для гоночної гри є одним із ключових етапів розробки, оскільки саме від правильного формулювання очікувань залежить успішність реалізації проєкту та рівень задоволення кінцевого користувача. Система має відповідати як функціональним, так і не функціональним вимогам, що охоплюють основні технічні та якісні характеристики проєкту.

Основним функціональним завданням системи є створення унікальних трас у напівавтоматичному режимі без залучення розробника до кожного окремого варіанту. Генератор має забезпечити формування траси з урахуванням встановлених параметрів, таких як довжина, рівень складності, тип місцевості та бажана кількість перешкод або особливих ділянок. При цьому особливу увагу необхідно приділити коректності побудови маршруту – траса має бути прохідною, без само перетинів або фізично неможливих ділянок.

Крім основної побудови маршруту, система повинна автоматично інтегрувати об’єкти навколишнього середовища, що підвищують реалістичність ігрового простору. Йдеться про ландшафт, декоративні елементи, дорожні знаки, глядацькі трибуни, рекламні щити та інші об'єкти, які роблять трасу живою і правдоподібною. Оточення має не лише естетичне значення, але і впливати на ігровий процес, наприклад, обмежуючи видимість або змінюючи поведінку автомобіля на різних поверхнях.

Іншим важливим функціональним аспектом є адаптивність. Система повинна враховувати рівень майстерності гравця та надавати йому траси відповідної складності. Це можна реалізувати шляхом аналізу ігрової статистики, середньої швидкості проходження попередніх заїздів, кількості помилок на маршруті та інших параметрів. Завдяки цьому новачки отримуватимуть менш складні треки з плавними поворотами і широкими проїздами, тоді як досвідчені гравці матимуть змогу випробувати себе на

складних маршрутах з численними перепадами висот, крутими поворотами та вузькими ділянками.

Ще однією обов'язковою функціональною вимогою є забезпечення реалістичної фізики руху. Це означає, що кожна траса має відповідати фізичним законам взаємодії транспортного засобу з дорожнім покриттям: коефіцієнт тертя, ухил дороги, наявність нерівностей і погодні умови повинні правильно враховуватися в обчисленнях, забезпечуючи передбачуване та реалістичне поводження автомобіля.

Нарешті, система повинна мати механізм верифікації створених трас. Перед початком заїзду необхідно автоматично перевіряти, чи можливе проходження траси без застрягання або неможливих для проходження сегментів. У разі виявлення критичних помилок траса повинна перегенеровуватися або коригуватися відповідним чином.

Що стосується не функціональних вимог, перш за все потрібно забезпечити високу продуктивність системи. Генерація траси має відбуватися за короткий час, щоб не змушувати гравця чекати. Бажано, щоб час генерації не перевищував кількох секунд навіть на середньостатистичних пристроях. Для цього необхідно оптимізувати алгоритми побудови траси та мінімізувати кількість обчислювальних операцій без втрати якості результату.

Дуже важливим є також рівень стабільності роботи системи. Програма не повинна аварійно завершувати роботу або створювати некоректні дані, що можуть призвести до неможливості запуску гонки або її некоректного проходження. Регулярне тестування генератора трас і обробка виняткових ситуацій повинні стати невід’ємною частиною процесу розробки, іншою критично важливою не функціональною вимогою є портативність.

Генератор трас має працювати на широкому спектрі пристроїв: від потужних ПК до мобільних телефонів і консолей. Це означає, що рішення має бути максимально оптимізованим за використанням пам’яті, процесорних ресурсів та інших обчислювальних потужностей. Особливо важливо зменшувати графічне навантаження без шкоди для візуальної привабливості

гри, застосовуючи технології рівнів деталізації (LOD), процедурного створення текстур та інші методи оптимізації.

Важливу роль також відіграє масштабованість системи. Архітектура генератора має бути спроєктована так, щоб її можна було розширювати без значних змін базового коду. Це дасть змогу у майбутньому легко додавати нові типи місцевості, погодні ефекти, додаткові елементи оточення або навіть принципово нові режими генерації.

Крім цього, варто враховувати і безпеку. Навіть у випадку офлайн-ігор захист від пошкодження даних або помилкових конфігурацій допомагає підвищити якість користувацького досвіду та зменшити ризики появи помилок.

Отже, визначення функціональних і не функціональних вимог дозволяє чітко окреслити основні цілі розробки генератора трас: створення унікального, цікавого й доступного геймплею із високим рівнем реалістичності, ефективною продуктивністю, адаптивністю та надійністю системи. Саме ці критерії будуть визначати напрям подальшого проєктування та реалізації програмного рішення.

Для кращого розуміння ситуації було створено таблицю 1. На ній продемонстровано основні вимоги і їх опис.

Таблиця 1 – Таблиця функціональних і не функціональних вимог

Категорія	**Вимога**	**Опис**
Функціональні вимоги	Генерація трас	Автоматичне створення унікальних маршрутів на основі заданих параметрів (довжина, складність, місцевість).
	Інтеграція об'єктів середовища	Додавання ландшафту, декоративних елементів, знаків, перешкод.

Продовження таблиці 1

	Адаптація до рівня гравця	Формування трас відповідної складності залежно від майстерності користувача.
	Реалістична фізика руху	Коректне відображення взаємодії транспортного засобу з покриттям і середовищем.
	Перевірка прохідності трас	Автоматичний аналіз маршруту для виявлення помилок і неможливих ділянок.
Нефункціональні вимоги	Продуктивність	Генерація траси має тривати кілька секунд навіть на середніх пристроях.
	Стабільність	Робота без збоїв, аварійних завершень або пошкоджених даних.
	Масштабованість	Можливість легко додавати нові типи трас, оточення або режими генерації
	Безпека	Захист від пошкодження даних, обробка виняткових ситуацій.

1.4 Постановка задачі.

У сучасних умовах розробки відеоігор особливої актуальності набувають методи автоматизації створення ігрового контенту. Це зумовлено як економічними факторами, так і прагненням забезпечити гравцям динамічний, невичерпний досвід взаємодії з грою. Одним із найбільш перспективних напрямів у цьому контексті є застосування процедурної генерації контенту, зокрема – процедурної генерації гоночних трас. Саме тому перед даною роботою постає комплексна задача створення ефективної системи, яка б автоматично генерувала унікальні маршрути для гоночної гри, дотримуючись вимог і обмежень, необхідних для збереження ігрового балансу та забезпечення позитивного досвіду користувача [4].

Постановка задачі включає кілька ключових аспектів, що потребують як теоретичного обґрунтування, так і практичної реалізації. В першу чергу слід визначити методику генерації трас, яка б дозволяла досягти оптимального співвідношення між випадковістю та логічною цілісністю створюваних маршрутів. Повна випадковість у формуванні треку може призвести до появи нелогічних, незручних для проходження або навіть фізично неможливих ділянок, що негативно вплине на ігровий процес. Водночас надмірна детермінованість позбавить трасу оригінальності й ефекту новизни. Отже, необхідно розробити алгоритм, який на основі заданих параметрів буде створювати варіативні, але водночас придатні до проходження маршрути.

Ще одним важливим аспектом є адаптивність системи генерації. Оскільки цільова аудиторія гоночних ігор є різноманітною за рівнем навичок і вподобаннями, система повинна враховувати можливість налаштування складності створюваних трас. Логічним буде інтегрувати механізми, що дозволяють змінювати кількість поворотів, ухилів, довжину треку, розмір і розташування перешкод, а також рівень деталізації оточення залежно від обраного режиму гри або попередніх результатів гравця. Таким чином, гравець

отримуватиме індивідуально підібраний досвід, що підвищить його залученість і задоволеність продуктом [3].

Важливо також врахувати обмеження технічного характеру. Оскільки сучасні ігри часто виходять на різних платформах, від потужних персональних комп'ютерів до мобільних пристроїв із обмеженими ресурсами, необхідно забезпечити оптимізацію створюваних трас як за обсягом даних, так і за складністю обробки графічних і фізичних моделей. Надмірна складність генерованої траси може призвести до падіння продуктивності гри, що, в свою чергу, негативно позначиться на ігровому процесі та враженнях користувача. Отже, одним із завдань є розробка системи контролю складності об'єктів на трасі, що забезпечуватиме баланс між якістю зображення, реалістичністю фізики та технічною доступністю гри для широкої аудиторії.

Необхідно також розглянути питання інтеграції в систему додаткових елементів ігрового середовища, таких як рекламні щити, будівлі, дерева, бар'єри тощо. Їх розташування має не лише підвищувати візуальну привабливість трас, а й відігравати роль у геймплейних механіках, наприклад, формуючи вузькі проїзди чи зони ризику. Генерація цих об'єктів повинна здійснюватися так, щоб забезпечувати логічність розміщення, не перешкоджати проїзду та не створювати несправедливих умов для гравця. Тому система повинна передбачати перевірку коректності розташування кожного елемента середовища щодо основної траси.

Ще однією задачею є розробка і реалізація механізму тестування генерованих трас. Незважаючи на використання алгоритмів контролю на етапі створення, можливі ситуації, коли трасу буде складно або неможливо пройти через специфіку випадкових комбінацій. Тому доцільно передбачити попереднє автоматичне тестування треків за допомогою віртуального «бота», який проїжджає маршрут і перевіряє його на прохідність. Таким чином, основна задача дипломної роботи полягає у розробці системи процедурної генерації трас для гоночної гри, що повинна відповідати наступним вимогам: забезпечувати варіативність і унікальність маршрутів, зберігати логічність і придатність до

проходження, враховувати рівень майстерності користувача, оптимізувати ресурсоспоживання залежно від платформи, інтегрувати об'єкти середовища з логічним розташуванням, а також передбачати механізми контролю якості та тестування створених маршрутів [5].

1.5 Висновок розділу 1

У результаті проведеного аналізу було з’ясовано, що жанр гоночних ігор залишається стабільно популярним завдяки динамічному геймплею, широкій варіативності підходів і технічній доступності для гравців. Визначено ключові структурні компоненти системи процедурної генерації трас, серед яких - генератор геометрії, аналітика складності, модуль оточення та фізична симуляція. Сформовано вимоги до програмного забезпечення як з функціонального, так і нефункціонального боку, що забезпечують адаптивність, реалістичність і ефективність гри. Також проведено огляд існуючих рішень у цій галузі, що дозволило врахувати їхні сильні сторони й уникнути поширених недоліків при проєктуванні власної системи. З цього можна зробити висновки, що цей пункт був дуже важливим для подальшої роботи з проєктом. Він надає базові розуміння і точку бачення з якої потрібно починати.

2. ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

2.1 Вибір типу архітектури та шаблонів проєктування.

На етапі розробки системи процедурної генерації трас особливе значення має правильний вибір архітектурної моделі проєкту та відповідних шаблонів проєктування. Від якості архітектурного рішення залежить не лише ефективність і стабільність роботи програми, але й можливість її подальшого розширення, супроводження та адаптації до нових вимог. Саме тому на цьому етапі необхідно провести детальний аналіз можливих архітектурних підходів і обґрунтувати вибір найбільш доцільного варіанту.

З огляду на характер системи процедурної генерації, яка передбачає взаємодію між кількома логічно незалежними модулями (генератор маршруту, система контролю складності, фізична модель, генератор оточення, модуль тестування тощо), найбільш відповідною видається багаторівнева (багатошарова) архітектура. Цей тип архітектури передбачає розподіл системи на окремі рівні відповідальності, що дозволяє досягти високого рівня модульності, зрозумілості коду та спрощення процесів тестування і масштабування.

У даному випадку можна умовно виділити такі основні рівні:

– рівень бізнес-логіки, який буде відповідати за реалізацію алгоритмів процедурної генерації трас, перевірку їх коректності, розрахунок параметрів складності та взаємодію між різними компонентами системи;

– рівень даних, що міститиме структури даних, необхідні для опису траси, елементів оточення та налаштувань параметрів генерації;

– рівень представлення, який опікуватиметься візуалізацією результатів генерації, інтеграцією трас у графічне середовище гри або симулятора.

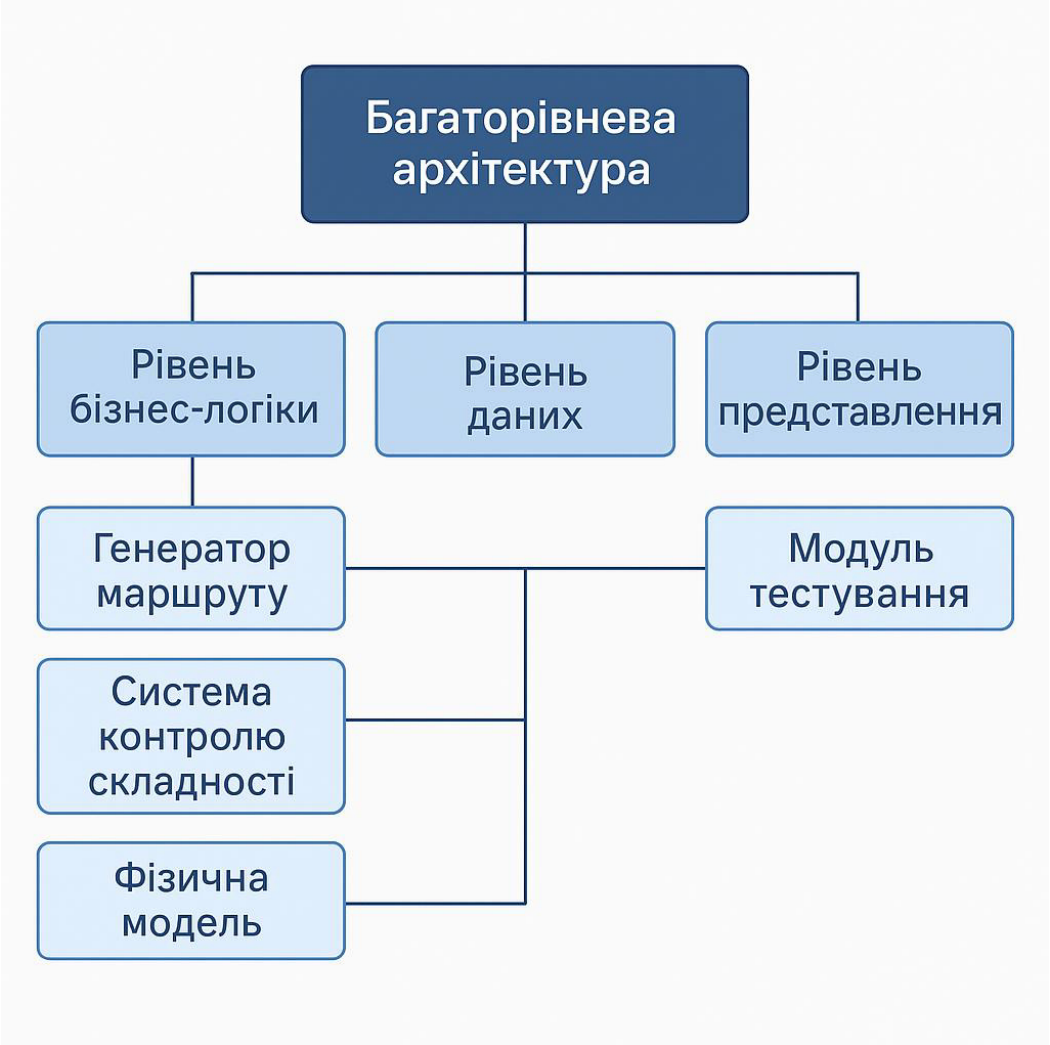


Рисунок 2.1 – Багаторівнева архітектура

Вибір багаторівневої архітектури, яку можна побачити на рисунку 2.1, пояснюється також тим, що вона забезпечує чітке розділення обов'язків: модулі, що відповідають за логіку генерації, не залежать від конкретної реалізації графічного рушія або зовнішніх інтерфейсів. Це, у свою чергу, відкриває можливості для багаторазового використання основних алгоритмів у різних проєктах або адаптації системи до різних платформ без необхідності кардинальної переробки коду.

Щодо вибору шаблонів проєктування, слід зазначити, що для даної системи доцільно застосувати кілька класичних патернів, які сприятимуть підвищенню гнучкості та масштабованості проєкту.

Першим і найважливішим є шаблон «Фабричний метод» (Factory Method), який буде використовуватися для створення різних типів елементів треку – наприклад, окремих сегментів дороги, перешкод, елементів оточення. Застосування цього патерну дозволяє централізувати процес створення об'єктів і спростити додавання нових типів елементів у майбутньому без необхідності змінювати існуючу бізнес-логіку [20].

Другим важливим шаблоном є «Стратегія» (Strategy). Він дозволить організувати вибір різних алгоритмів генерації маршруту залежно від заданих параметрів. Наприклад, у залежності від обраного режиму гри може використовуватися сплайн-генерація для м'яких трас або шумові алгоритми для більш хаотичних та природних маршрутів. Стратегія дозволить легко переключати логіку без переписування основного коду генератора.

Окрім того, доцільним є використання патерну «Будівельник» (Builder) для поетапного формування складних об'єктів, зокрема повної конфігурації треку із сегментів та об'єктів середовища. Це дозволить розділити процес побудови траси на послідовні кроки і забезпечить більшу керованість у випадках, коли потрібно створити різні варіанти трас із різною кількістю елементів і особливостями розташування.

Для управління загальним процесом генерації можна розглянути застосування шаблону «Фасад» (Facade), який об'єднає підсистеми маршруту, оточення, фізики та перевірки трас в єдиний інтерфейс. Це значно спростить керування процесом як із точки зору внутрішніх механізмів гри, так і для можливого зовнішнього використання бібліотеки генерації.

У підсумку можна зазначити, що обраний тип архітектури і набір шаблонів проєктування не лише забезпечать високу якість програмного рішення, але й зроблять систему більш зрозумілою, гнучкою і такою, що легко адаптується до змін. Це дозволить у майбутньому доповнювати або модифікувати окремі частини без ризику порушення цілісності всієї системи, що є надзвичайно важливим як для підтримки продукту, і для його розвитку.

2.2 Проєктування алгоритмів роботи ігрових механік

Проєктування алгоритмів, які відповідають за роботу основних ігрових механік, є одним із найважливіших етапів у розробці гоночної гри з процедурною генерацією трас [26]. Саме алгоритмічна основа визначає не лише працездатність гри, але й рівень задоволення гравця від взаємодії із системою. Для забезпечення високої якості ігрового досвіду необхідно не просто розробити окремі функціональні блоки, а й інтегрувати їх у цілісну, узгоджену систему, де всі механізми гармонійно взаємодіють між собою.

Основною задачею алгоритмів є створення динамічного і при цьому логічно зв’язаного середовища для гравця. Оскільки гра базується на процедурній генерації трас, основна увага приділяється тому, щоб побудова траси була не випадковим набором сегментів, а обґрунтованою послідовністю елементів, які створюють відчуття цілісного шляху. Враховуючи це, першим критичним механізмом стає алгоритм генерації маршруту.

Цей алгоритм працює за принципом поетапної побудови траси: спочатку створюється загальна крива маршруту із заданими параметрами (довжина, кількість поворотів, загальний ухил), після чого відбувається деталізація сегментів із врахуванням фізичних обмежень і особливостей місцевості. Базовими технологіями для цього є інтерполяція сплайнами або застосування процедурного шуму (наприклад, Perlin Noise), що забезпечує плавні та природні переходи між ділянками дороги. Щоб уникнути утворення непридатних для заїзду сегментів, застосовується допоміжний алгоритм перевірки, який тестує кожен згенерований елемент на відповідність базовим правилам прохідності (відсутність надто гострих кутів, неможливих підйомів або «зламаних» геометрій) [30].

Після побудови основного маршруту настає етап роботи алгоритму контролю складності. Він адаптує трасу відповідно до обраного рівня гри або навичок гравця. Якщо обрано легкий режим, то алгоритм автоматично зменшує

кількість крутих поворотів, робить ухили плавнішими та додає ширші ділянки дороги. У режимі високої складності навпаки – збільшується кількість вузьких відрізків, додаються технічні елементи на кшталт «серпантинів» або «S»-подібних поворотів, що вимагають від гравця точного керування.[18]

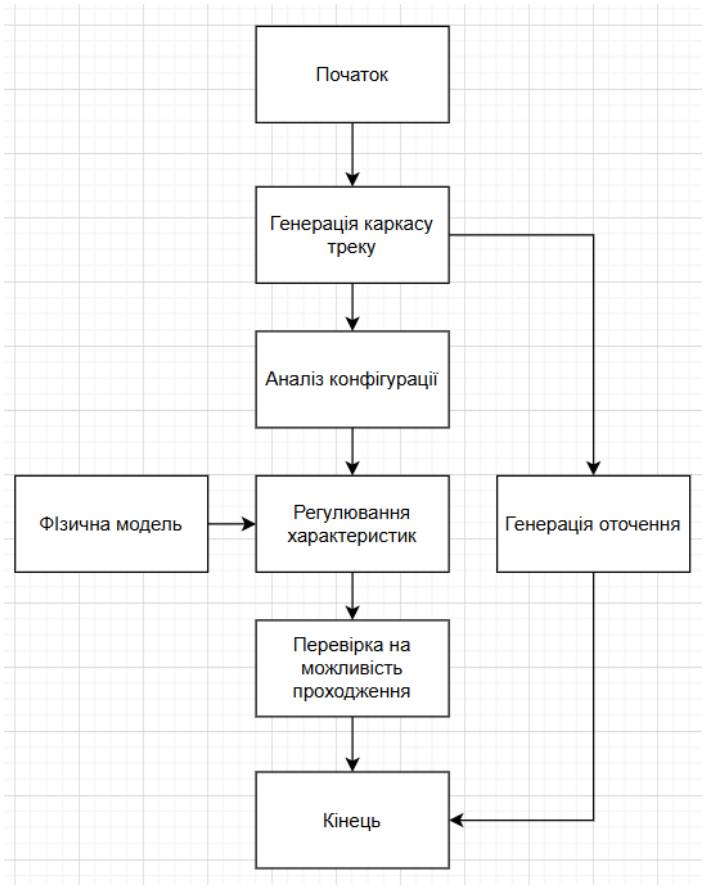


Рисунок 2.2 – Логіка генерування треку

Ще одним важливим блоком є алгоритм генерації оточення, який зображений на рисунку 2.2, і відповідає за візуальне наповнення простору навколо траси. Він не лише розміщує статичні об'єкти, такі як дерева, будівлі чи трибуни, а й створює динамічні елементи: рухливі об'єкти, що реагують на присутність гравця (наприклад, глядачі, що махають прапорами, або тварини, які перетинають трасу). Для досягнення ефекту живого світу використовується система тригерів, які активують ті чи інші скрипти залежно від позиції гравця на карті.

Фізична модель взаємодії автомобіля з трасою реалізується через комплексний алгоритм моделювання фізики руху. Він має забезпечити достовірну симуляцію зчеплення шин із покриттям дороги, інерційних сил, впливу погодних умов тощо [13]. Тут важливо не тільки створити фізику, близьку до реальної, але й адаптувати її під ігрові реалії, забезпечивши баланс між складністю управління і задоволенням від процесу. До прикладу, занадто реалістична фізика може зробити гру важкодоступною для новачків, тоді як надто спрощена – відлякати шанувальників симуляторів.

Особливу увагу слід приділити алгоритму тестування і корекції. Оскільки траси створюються динамічно і не проходять ручної верифікації на етапі розробки, необхідно автоматизувати процес перевірки: аналіз геометрії траси на непрохідні ділянки, тестування фізики проходження поворотів, виявлення помилок у розміщенні об'єктів. Ідеальним підходом тут є використання віртуального агента-водія, який автоматично «проїжджає» трасу і оцінює її на основі заданих критеріїв.

Архітектура алгоритмів базується на принципах модульності та інкапсуляції, що дозволяє легко вносити зміни в окремі частини системи без ризику порушення її цілісності. Кожен механізм реалізується як окремий сервіс або клас, що має чітко визначений інтерфейс взаємодії. Для зменшення рівня зв'язності між компонентами активно застосовуються шаблони проєктування, такі як Фабричний метод для створення об'єктів трас і оточення, Стратегія для вибору алгоритму генерації залежно від рівня складності, а також Спостерігач для оновлення динамічних елементів під час руху гравця.

Узагальнюючи, проєктування алгоритмів роботи ігрових механік у цьому проєкті передбачає створення самодостатніх, гнучких і взаємопов'язаних систем, що забезпечують:

– генерацію різноманітних трас зі збереженням балансу між випадковістю і ігровою логікою;

– моделювання реалістичної, але доступної фізики руху;

– динамічне і правдоподібне оточення;

– постійний автоматичний контроль якості створених рівнів;

– адаптацію складності гри під навички конкретного користувача.

Це дозволяє не тільки реалізувати основну ідею процедурної генерації, а й створити продукт, що відповідає сучасним вимогам якості у сфері ігрової індустрії, підвищуючи реіграбельність і залучення користувачів до тривалого використання гри [22].

Рисунок 2.2 – Візуалізація основних алгоритмів.

2.3 Проєктування модулів.

Одним із ключових завдань у процесі створення системи процедурної генерації трас є формування чіткої, структуровано впорядкованої модульної архітектури. Модулі в контексті даного проєкту виступають не просто як частини коду з певним функціоналом, а як окремі автономні одиниці з визначеним набором обов’язків, інтерфейсами взаємодії та життєвим циклом.

Завдяки цьому забезпечується як масштабованість, так і підтримуваність усього програмного рішення [20].

Система умовно поділяється на кілька великих підсистем, кожна з яких реалізує один із ключових етапів генерації: формування геометрії треку, обробку складності, наповнення сцени об’єктами оточення, симуляцію фізичних параметрів і фінальну верифікацію. На рівні реалізації у Unity кожен модуль представлено окремим класом або набором скриптів, які ізольовані за логікою, але взаємодіють через чітко визначені інтерфейси.

Центральною одиницею всієї системи виступає менеджер генерації треку – координатор усіх модулів, який ініціалізує послідовність дій, передає параметри, а також відповідає за синхронізацію стадій генерації. Саме через нього відбувається передача даних між модулями, запуск кожного компонента та фінальне складання повноцінної сцени. Менеджер працює з декількома підлеглими компонентами, кожен з яких виконує спеціалізовану функцію.

На першому етапі активується модуль генерації геометрії, який формує маршрут майбутньої траси у вигляді послідовності координатних точок або кривих. На даному етапі застосовуються сплайнові структури або інтерпольовані шумові карти. Модуль розраховує загальну траєкторію з урахуванням початкових параметрів: довжини, щільності поворотів, висотного профілю. Важливо, що геометрія створюється не як візуальна сцена, а як внутрішнє представлення даних, які потім використовуються іншими модулями.

Після побудови маршруту запускається модуль аналізу складності, який регулює характеристики траси відповідно до рівня майстерності гравця. Він працює на базі збережених метрик або змінних, пов’язаних із поведінкою користувача в попередніх сесіях. Визначаються допустимі значення для параметрів: кутів повороту, допустимої довжини прямолінійних відрізків, кількості потенційно небезпечних зон. Цей модуль не створює нову трасу, а коригує вже сформовану геометрію [20].

Далі в систему включається модуль генерації оточення, який отримує доступ до структури маршруту та розміщує вздовж нього об’єкти середовища.

Розміщення відбувається не випадково, а з урахуванням логіки ландшафту, зон швидкості, зон перегляду та ширини дороги. На цьому етапі модуль інтегрується з Unity-ресурсами (префабами), використовуючи механізми об’єктного пулінгу для оптимізації продуктивності при великій кількості елементів сцени.

Паралельно або після заповнення сцени об’єктами працює модуль фізичних властивостей, який зв’язує побудовану трасу з фізичним середовищем рушія. У цьому модулі встановлюються фізичні матеріали, що визначають коефіцієнти тертя, ковзання, поведінку при зіткненнях, а також типи поверхонь, що впливають на поведінку коліс автомобіля. Передбачена можливість динамічного коригування параметрів у разі змін погодних умов, освітлення або навантаження на трасу.

Для забезпечення валідності згенерованого рівня в систему включено модуль перевірки прохідності, який за допомогою умовного «агента-гравця» проходить трасу в симульованому режимі. Алгоритм визначає, чи можна проїхати маршрут без зіткнень, чи не виникає тупикових сегментів або нерозв’язних ситуацій. У випадку виявлення помилок генератор повертається на попередній етап або використовує запасний варіант конфігурації.

Уся взаємодія модулів базується на принципі інверсії залежностей: низькорівневі компоненти не знають про існування інших, а вся координація здійснюється через абстрактні інтерфейси, що дозволяє легко змінювати реалізації, тестувати компоненти ізольовано, а також масштабувати функціональність без серйозного втручання в існуючий код. Усі дані, які передаються між модулями, представлені у вигляді структурованих об'єктів або серіалізованих контейнерів, що дозволяє уніфікувати обробку та зберігання інформації [36].

У результаті, модульна архітектура системи процедурної генерації трас формує гнучку й розширювану структуру, що повністю відповідає принципам сучасної розробки ігор Її приклад можна побачити на рисунку 2.4. Вона дозволяє не лише підтримувати стабільну продуктивність і логіку, але й у

перспективі легко розширювати функціонал, наприклад, шляхом додавання нових погодних сценаріїв, генерації декількох треків одночасно або використання мережевої синхронізації для мультиплеєрних режимів.

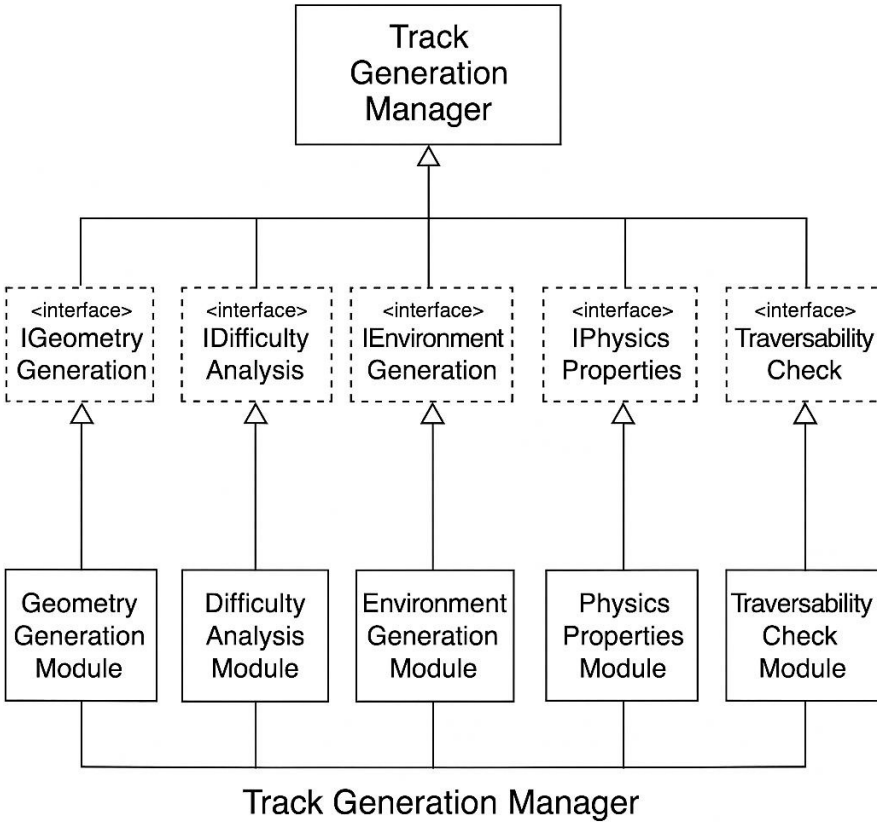


Рисунок 2.4 – Архітектурна діаграма взаємодії модулів і менеджера

2.4 Проєктування GUI (Graphical User Interface).

Важливою складовою будь-якого програмного продукту, зокрема ігрового, є система графічного інтерфейсу користувача (Graphical User Interface, або скорочено GUI). Саме через неї гравець взаємодіє з програмою, отримує візуальний зворотний зв'язок, керує налаштуваннями, запускає гру, змінює параметри ігрового процесу та отримує повідомлення про свій прогрес. У випадку реалізації гоночної гри з процедурною генерацією трас, GUI виступає

не просто допоміжним засобом, а повноцінним інструментом керування логікою гри, її поведінкою і зовнішнім виглядом.

Проєктування інтерфейсу має базуватися не лише на естетичних вподобаннях, а й на глибокому розумінні користувацького досвіду (User Experience – UX). Саме ефективне поєднання візуальної простоти, логічної послідовності і функціональної завершеності робить GUI не просто зручним, а й непомітним – у позитивному сенсі. Інакше кажучи, ідеальний інтерфейс не відволікає від гри, а підсилює її, стає частиною геймплею.

У нашому проєкті основна роль інтерфейсу полягає у наданні гравцеві зручного доступу до налаштувань генерації треку, запуску заїзду, перегляду статистики, управління профілем та виходу з гри. Також передбачено виведення системних повідомлень, таймерів, індикаторів швидкості, карти траси та підказок на екрані під час самої гонки. Важливо, що інтерфейс має бути адаптивним і не перевантаженим, оскільки перенасичення елементами негативно впливає на зосередженість гравця, особливо в жанрі, де реакція і увага мають критичне значення [39].

Процес проєктування GUI традиційно розпочинався з побудови макетів майбутніх екранів – це дозволяє на ранньому етапі протестувати логіку навігації, порядок розміщення елементів і загальний стиль подачі інформації. Основні екрани інтерфейсу включають головне меню, екран налаштувань, екран вибору параметрів генерації, ігрову сцену з елементами HUD (head-up display), а також завершення заїзду з результатами. Усі вони мають бути стилістично узгодженими, використовувати єдину палітру кольорів, типографіку та розміщення елементів, що відповідає сучасним принципам дизайну інтерфейсів.

Варто зазначити, що графічний інтерфейс реалізовано з використанням внутрішніх засобів рушія Unity – зокрема UI Canvas, панелей, текстових блоків, кнопок, слайдерів, іконок та анімованих елементів. Система використовує адаптивну сітку, яка дозволяє зберігати правильні пропорції навіть при зміні роздільної здатності екрана або при запуску гри на пристроях із різними

характеристиками. Такий підхід забезпечує масштабованість GUI і дозволяє уникнути графічних дефектів при відображенні [33].

Особливої уваги заслуговує реалізація меню генерації треку. Цей екран дозволяє користувачеві самостійно визначати ключові параметри майбутнього маршруту: загальну довжину, тип ландшафту, рівень складності, присутність перешкод і кількість візуальних елементів. Кожен із параметрів пов'язаний із відповідним слайдером або випадаючим списком, що дозволяє оперативно змінювати значення без потреби вводити числові дані вручну. Зміни параметрів відображаються в режимі реального часу, і гравець бачить попередній перегляд карти, яка буде створена. Така інтерактивність надає користувачеві більше контролю та створює враження залученості у процес генерації.[21]

Рисунок 2.5 – Модель головного меню гри

Під час самої гри активується HUD – частина інтерфейсу, яку ви можете бачити на рисунку 2.5, і на рисунку 2.6, відображає поточну швидкість автомобіля, пройдений відсоток маршруту, таймер, позицію гравця (у випадку багатокористувацького режиму), карту траси, а також індикатори зіткнень або штрафів. Важливо, що HUD розміщується на екрані таким чином, щоб не заважати огляду та не перекривати критично важливу інформацію. Елементи HUD використовують напівпрозорі текстури та динамічну зміну кольору залежно від ситуації (наприклад, червоний при небезпечному прискоренні або зіткненні), що дозволяє швидко зчитувати інформацію без зайвого аналізу.

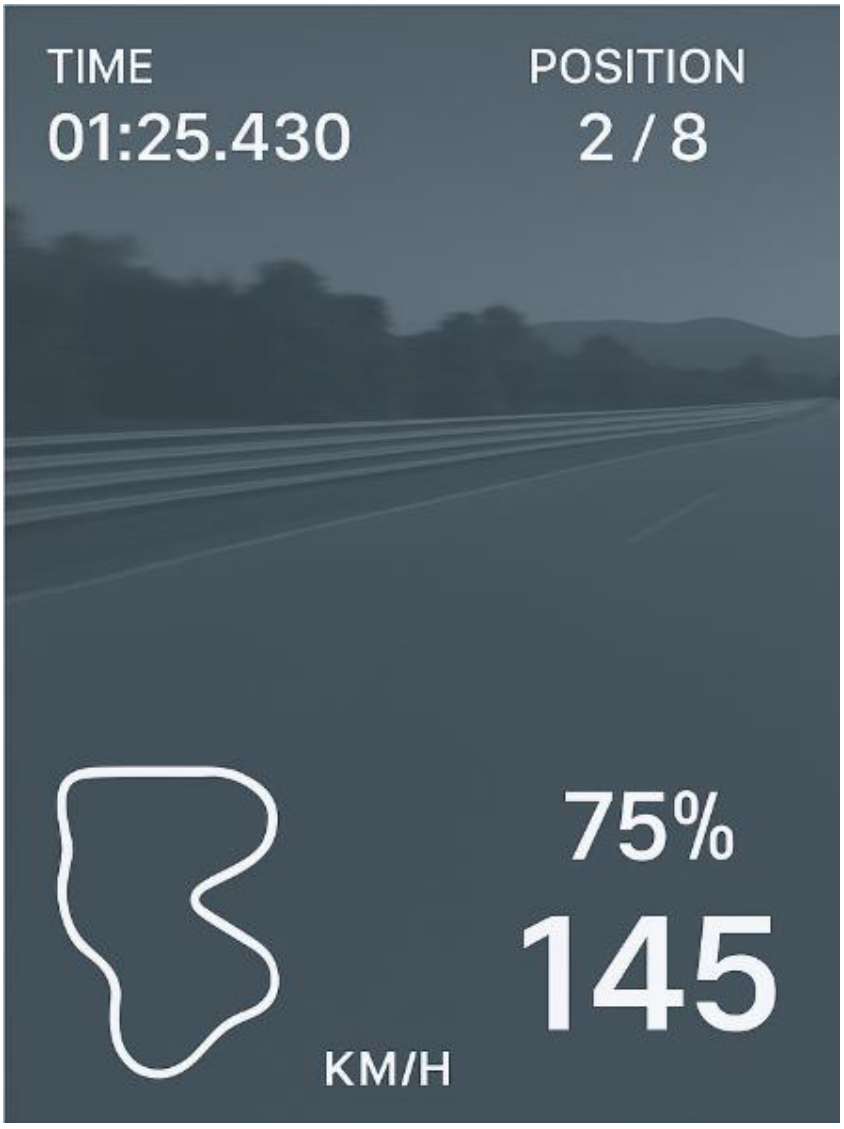


Рисунок 2.6 – Основні елементи HUD гравця.

На окрему увагу заслуговує логіка адаптації GUI до геймпаду або сенсорного керування (у випадку розширення гри на мобільні платформи). Інтерфейс проєктувався таким чином, щоб усі ключові дії могли бути виконані без миші – це забезпечує гнучкість і готує основу для кросплатформенності. Зокрема, використовуються великі зони натискання, інтуїтивна навігація за допомогою кнопок «вліво/вправо», а також можливість змінювати порядок елементів без потреби вказівника [20].

Таким чином, розробка графічного інтерфейсу користувача не обмежується візуальною частиною, а є комплексним завданням, що охоплює питання зручності, технічної реалізації, естетики та функціональності. У нашому проєкті GUI виконує роль не просто інструменту взаємодії, а повноцінного каналу між гравцем і системою процедурної генерації, який забезпечує контроль, гнучкість, інформативність та глибше занурення у гру. Такий підхід дозволяє підвищити загальну якість користувацького досвіду і зробити проєкт більш привабливим для широкої аудиторії.[35]

2.5 Аналіз методів і технологій реалізації ПЗ

Розробка програмного забезпечення у форматі повноцінної гри, навіть якщо вона не має великого комерційного масштабу, вимагає комплексного підходу до вибору інструментів, технологій та методів реалізації. Головним викликом під час створення інтерактивних систем, таких як генератор трас у гоночній грі, є необхідність знайти баланс між швидкістю реалізації, продуктивністю, гнучкістю та підтримуваністю коду. Саме тому критичну роль у проєктному процесі відіграє детальний аналіз і усвідомлений вибір технологічного стеку, який дозволяє максимально ефективно реалізувати поставлені вимоги.

Основним рушієм, обраним для реалізації проєкту, став Unity. Це одне з найпопулярніших ігрових середовищ, що забезпечує потужний інструментарій

для створення тривимірних і двовимірних ігор, підтримку різних платформ (ПК, мобільні пристрої, WebGL), наявність великої кількості готових компонентів і високу швидкість розробки прототипів. Однією з головних переваг Unity є його компонентно-орієнтована архітектура, що дозволяє гнучко комбінувати поведінки різних об’єктів без надмірної складності в логіці коду.

Для реалізації логіки проєкту використовувалася мова програмування C#, яка є основною для Unity. Вона поєднує у собі виразну об’єктно-орієнтовану модель із широкими можливостями управління ресурсами, що дозволяє створювати ігрову логіку з чіткою структурою, а також застосовувати шаблони проєктування, які значно підвищують підтримуваність та гнучкість архітектури. Під час реалізації модулів генерації, фізики, інтерфейсу та перевірки прохідності широко використовувались такі принципи, як інкапсуляція, спадкування, композиція, також патерни «Фасад», «Будівельник», «Стратегія».

Що стосується роботи з тривимірним середовищем, Unity надає багатий набір інструментів для побудови сцен, взаємодії з об’єктами, роботи з фізикою та оптимізації продуктивності. Для створення трас використовувалися сплайни (Bezier curves), процедурне розміщення Mesh-компонентів та динамічне оновлення сцен через скриптові методи Instantiate і Destroy, з попередньою оптимізацією через пулінг об'єктів (Object Pooling). Такий підхід дозволяє зменшити навантаження на систему при постійному створенні і знищенні об'єктів, що є критично важливим у динамічному середовищі.[18]

З метою покращення продуктивності при рендерингу сцен використовувалися механізми LOD (Level of Detail), які дозволяють відображати об’єкти з різним ступенем деталізації залежно від відстані до камери. Це суттєво знижує навантаження на GPU без втрати якості зображення для користувача. Також використовувались базові можливості Unity з обмеженням глибини тіней, спрощенням фізичних колайдерів і розділенням логіки апдейтів (через FixedUpdate, LateUpdate, Coroutine), що дозволило уникнути «затиків» при великій кількості активних об’єктів.

Для розробки інтерфейсу GUI був використаний вбудований UI Canvas системи Unity. Елементи інтерфейсу були реалізовані із застосуванням стандартних компонентів: TextMeshPro, Button, Slider, Toggle та інших. Застосування адаптивного дизайну дозволило створити інтерфейс, що коректно масштабується на різних екранах і підтримує зміну роздільної здатності. GUI проєкту також реалізовувався з урахуванням принципів UX – зручність користування, інтуїтивне розміщення елементів, мінімальна кількість кліків до досягнення цілі.

Окрему увагу було приділено внутрішній організації коду. Для зберігання конфігураційних параметрів використовувалися ScriptableObject – спеціальний формат Unity, що дозволяє зберігати і повторно використовувати дані в межах проєкту без необхідності створювати зайві копії в пам’яті. Також у проєкті впроваджена базова система логування помилок та налагодження, що дозволяє відстежувати аномальні ситуації в генерації трас або помилки в логіці взаємодії об’єктів [31].

Із метою забезпечення стабільності генерації використовувалися методи перевірки на колізії, трасування за допомогою Raycasting, а також автоматичні симуляції проїзду маршруту умовним агентом. Для контролю якості було реалізовано тестування за допомогою Unity Test Framework - зокрема , модульні тести для окремих компонентів генератора, що перевіряють правильність побудови траси та логіку її проходження.

Таким чином, реалізація програмного забезпечення була побудована на сучасному технологічному фундаменті, який поєднує високу гнучкість і масштабованість Unity, зручність C#, ефективність компонентної архітектури та багатство вбудованих інструментів оптимізації і візуалізації. Застосовані методи дозволяють не лише реалізувати поставлені вимоги, але й забезпечити розширення функціоналу, підтримку кросплатформенності, а також подальше вдосконалення системи без значної переробки архітектури. Краще можна побачити всі ці переваги на поданій нижче таблиці 2.

Таблиця 2 – Порівняння Unity і Unreal Engine

Критерій	**Unity (обрано)**	**Unreal Engine**
Мова програмування	C# (простий, популярний, читабельний)	C++ (потужний, але складніший)
Поріг входу	Низький – швидке освоєння	Вищий – складніший інтерфейс
Оптимізація для мобільних	Добра, велика кількість туторіалів	Більш ресурсомісткий, складніший порт
Вбудовані інструменти	UI-система, анімація, фізика, генерація	Більше підтримки AAA-графіки
Гнучкість прототипування	Висока, ідеально для інді-проєктів	Краще підходить для масштабних ігор

За допомогою цієї таблиці можна добре побачити плюсі і мінуси цих двох найпопулярніших ігрових рушіїв. Вони обидва є хорошими варіантами для створення гри, але все ж таки вибір впав на рушій Unity [40].

2.6 Висновки до розділу 2

У другому розділі було здійснено комплексне проєктування системи програмного забезпечення, що лежить в основі гоночної гри з процедурною генерацією трас. На основі попереднього аналізу визначено доцільну архітектуру, орієнтовану на модульність, розділення логіки і візуалізації, а також гнучке масштабування компонентів. Для забезпечення стабільної

взаємодії між частинами застосовано шаблони проєктування, що сприяють зрозумілості й повторному використанню коду.

Особливу увагу приділено алгоритмам генерації треків, де закладено баланс між випадковістю та структурною логікою маршруту. Проєктування основних модулів охоплювало створення генератора геометрії треку, модуля аналізу складності, генератора оточення, а також підсистему взаємодії з користувачем через HUD. Для кожного з компонентів були побудовані блок-схеми, UML-діаграми й загальна модель процесу роботи гри.

Також було розроблено інтерфейс користувача, орієнтований на простоту взаємодії, з урахуванням основних вимог до зручності та візуального сприйняття. Крім цього, виконано аналіз методів реалізації програмного продукту, що дало змогу обґрунтувати вибір рушія Unity та мови програмування C# як основної технологічної бази.

Таким чином, у рамках цього розділу сформовано повну технічну основу майбутньої реалізації гри, яка забезпечує гнучкість, адаптивність та ефективну інтеграцію ключових механік у межах обраної платформи. В подальшому просто потрібно втілити головні ідеї в життя та створити їх в програмі. Наступний пункт буде саме про це. Тому створення логічного ланцюжку , по якому буде рухатися розробка є дуже зручним і важливим моментом. Це збереже багато часу і кратно зменшить кількість витраченого часу і помилок, які зможуть в майбутньому проявити себе.

3.ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ.

3.1 Програмна реалізація модулів.

Після завершення етапів аналізу предметної області, проєктування архітектури та визначення ключових функціональних і нефункціональних вимог, логічним продовженням стало безпосереднє програмне втілення спроєктованих модулів. Саме реалізація є тією фазою, де абстрактні концепції, схеми та моделі набувають конкретної форми у вигляді коду, структур даних і взаємодіючих компонентів. Розробка кожного модуля супроводжувалася не лише технічною імплементацією, а й глибоким розумінням його місця в загальній архітектурі системи, а також постійним врахуванням особливостей ігрової логіки, продуктивності та зручності подальшої підтримки.

На цьому етапі в центрі уваги перебуває практична реалізація тих підсистем, що були раніше визначені як основні для функціонування генератора трас: модуль побудови маршруту, конфігуратор складності, генератор оточення, фізичний блок, система перевірки прохідності та менеджер взаємодії. Кожен із цих компонентів було реалізовано з урахуванням принципів модульності, інкапсуляції та розділення відповідальностей, що дозволяє підтримувати код у чистому, структурованому вигляді, легкому для тестування та масштабування.

Особливу увагу приділялося не лише забезпеченню функціональності окремих частин, але й логіці їх взаємодії в рамках єдиного життєвого циклу генерації. При цьому велике значення мали засоби Unity, що дозволяють керувати процесом у режимі реального часу, з використанням подій, корутин та адаптивних оновлень кадру. Програмні рішення були реалізовані таким чином, щоб користувач мав можливість без зайвого втручання ініціювати процес генерації, отримати результат, налаштувати параметри і візуально взаємодіяти з отриманими елементами сцени [19].

Варто підкреслити, що реалізація не зводилася до механічного написання коду. Кожен крок вимагав аналізу, порівняння можливих варіантів, експериментів із поведінкою об’єктів та оптимізації за умов змінного

навантаження. Зокрема, при створенні механізмів побудови маршруту постало питання точності інтерполяції, при реалізації генерації об’єктів – розподілу навантаження на рендеринг, а при налаштуванні фізики – реалістичності поведінки автомобіля на різних типах поверхонь. Усе це потребувало не лише технічних знань, але й розуміння ігрової логіки, зручності користувача та практичного тестування.

Отже, даний підпункт відкриває собою розділ, присвячений прикладній реалізації розроблених рішень. У подальших частинах буде представлено огляд конкретних модулів системи, з поясненням їх призначення, архітектури, принципів роботи та особливостей коду. Такий підхід дозволяє продемонструвати не лише результати проєктування, а й підтвердити їхню практичну реалізованість, ефективність і відповідність поставленим вимогам.

Основою будь-якої гоночної гри є траса – шлях , по якому рухається транспортний засіб. У межах цього проєкту формування маршруту здійснюється за допомогою процедурної генерації, що дозволяє отримати унікальні конфігурації треків кожного запуску гри. Відповідальний за це компонент – модуль генерації геометрії треку, - виконує побудову маршруту як просторової кривої, що проходить через низку контрольних точок, які обчислюються відповідно до заданих параметрів: довжина, складність, стиль траси та інше [34].

У технічному плані процес генерації починається зі створення послідовності точок (Vector3), які формують основу маршруту. Кожна нова точка обчислюється на основі попередньої та випадково згенерованого вектора напряму. Щоб уникнути хаотичних вигинів, зміна напряму обмежується в межах заданого кута повороту:

```
for (int i = 1; i < pointCount; i++)
{
    float angle = Random.Range(-turnVariation, turnVariation);
    direction = Quaternion.Euler(0, angle, 0) * direction;
    currentPosition += direction.normalized * segmentLength;
    controlPoints.Add(currentPosition);
}
```

Цей підхід дозволяє досягти балансу між передбачуваністю й варіативністю, створюючи траси, які цікаві для проходження, але не викликають дезорієнтації в гравця. Результатом роботи цього алгоритму є масив контрольних точок, через які пізніше буде побудовано саму трасу.

Далі ці точки використовуються для створення візуального представлення маршруту. У реалізованій системі це відбувається шляхом екструзії окремих префабів дороги вздовж векторів між точками. Кожен сегмент будується в середині між парою точок та орієнтується в напрямку руху:

```
Vector3 position = (start + end) / 2f;
Quaternion rotation = Quaternion.LookRotation(end - start);

GameObject segment = Instantiate(roadSegmentPrefab, position, rotation);
segment.transform.localScale = new Vector3(1, 1, Vector3.Distance(start, end));
```

Ключовою вимогою під час побудови траси є забезпечення її безперервності. Для цього сегменти підлаштовуються під довжину відповідного вектора, що дозволяє уникнути розривів або перекривань між ними. При потребі – наприклад , при створенні більш складних форм – система може використовувати інтерполяцію по кривих типу Bezier або Catmull-Rom.

У контексті підтримки продуктивності генератор реалізовано так, щоб мати можливість очищення попередньої траси перед створенням нової. Це дозволяє уникнути нагромадження об’єктів у сцені:

```
foreach (var segment in generatedSegments)
{
    Destroy(segment);
}
generatedSegments.Clear();
controlPoints.Clear();
```

З технічного погляду модуль оформлено як клас у Unity з публічними параметрами, які можуть змінюватися як через інтерфейс, так і програмно.

Завдяки цьому систему легко інтегрувати в загальну логіку гри, надаючи гравцеві або штучному інтелекту можливість формувати треки під конкретні умови.

В результаті, реалізований модуль виконує побудову унікальної траси при кожному запуску, враховуючи параметри складності та типу місцевості. Він створює послідовність точок маршруту, автоматично формує геометрію дороги між ними та гарантує безперервність, стабільність і гнучкість генерації. Такий підхід не лише пришвидшує створення ігрового контенту, а й забезпечує гравцям унікальний досвід кожного разу, коли вони виходять на нову трасу.

Оскільки маршрут у реалізованій грі створюється динамічно, виникає потреба в автоматичному аналізі його характеристик для оцінки складності. Завданням модуля аналітики складності є обробка згенерованого набору контрольних точок, виявлення параметрів, що визначають геймплейну навантаженість (наприклад, частота поворотів чи перепади висоти), і класифікація маршруту за рівнем складності. Це дозволяє надалі адаптувати трасу до рівня майстерності або відфільтрувати непридатні конфігурації [19].

Базою для аналізу служить масив Vector3, що представляє точки треку. Для кожної трійки послідовних точок визначається кут повороту – тобто зміна напрямку руху. Для цього розраховуються нормалізовані вектори між сусідніми точками та обчислюється кут між ними:

```
Vector3 dir1 = (points[i] - points[i - 1]).normalized;
Vector3 dir2 = (points[i + 1] - points[i]).normalized;
float angle = Vector3.Angle(dir1, dir2);
```

Аналогічно оцінюються перепади висоти. Для кожного відрізка береться модуль різниці координат Y, що дає змогу виявити ділянки з підйомами або спусками. Максимальні значення таких перепадів зберігаються для подальшого порівняння:

```
float heightDelta = Mathf.Abs(points[i + 1].y - points[i].y);
```

```
if (heightDelta > maxHeightDelta)
    maxHeightDelta = heightDelta;
```

Усі зібрані показники – середній кут повороту, максимальний перепад висоти, середня довжина відрізків – нормалізуються та порівнюються із заздалегідь визначеними порогами. На їх основі маршрут класифікується за трьома рівнями складності: легкий, середній або складний. Наприклад, кут повороту до 20 градусів та незначні ухили формують легкий трек, а наявність частих і різких змін напрямку – складний :

```
if (avgTurnAngle < 20f && maxHeightDelta < 2f)
    difficulty = DifficultyLevel.Easy;
else if (avgTurnAngle < 40f && maxHeightDelta < 4f)
    difficulty = DifficultyLevel.Medium;
else
    difficulty = DifficultyLevel.Hard;
```

Окремо реалізовано можливість адаптації складності під навички гравця. Для цього використовується змінна playerSkillLevel, що задається вручну або розраховується динамічно. Якщо маршрут виявляється надто складним порівняно з поточним рівнем, система може сигналізувати про потребу його спрощення або перегенерації:

```
if (playerSkillLevel <= 3 && difficulty == DifficultyLevel.Hard)
{
    Debug.Log("Траса занадто складна. Можлива перегенерація.");
}
```

У результаті реалізований модуль дозволяє програмно оцінити ігрову складність маршруту до моменту запуску заїзду. Він працює автономно та не потребує втручання користувача, що є особливо цінним для процедурно згенерованих треків, де ручна перевірка неможлива. Крім того, аналітика може застосовуватися в інтерфейсі гри для відображення гравцеві рівня складності траси або у внутрішній логіці балансу, адаптації складності та геймдизайну.

Таким чином, модуль аналітики складності виступає своєрідним фільтром якості: він допомагає виявити невдалі конфігурації маршруту, автоматично оцінити потенційну складність заїзду, а також забезпечує інтелектуальну підтримку адаптації під гравця. У комплексі з генератором геометрії та оточення він формує базу для побудови цікавого, динамічного й керованого ігрового простору [36].

У процесі створення процедурно згенерованої траси важливо не лише побудувати її геометрію, а й створити навколишнє середовище, яке посилює ефект занурення, формує атмосферу заїзду та візуально структурує простір. Саме для цього було реалізовано модуль генерації оточення – компонент , що автоматично розміщує декоративні й функціональні об'єкти вздовж траси. Його головна задача – забезпечити візуальне наповнення сцени без втручання розробника або дизайнера рівнів.

Основна ідея реалізації полягає в тому, щоб уздовж траси через рівні інтервали розміщувати об'єкти оточення з обох боків дороги. Для цього використовується масив префабів – дерев , дорожніх знаків, трибун, бар’єрів тощо. Розміщення відбувається з відступом від центральної лінії маршруту, який обчислюється на основі вектора напрямку між контрольними точками. Щоб визначити позиції для розміщення, обчислюється вектор, перпендикулярний до напряму руху, - він задає ліву й праву сторони треку:

```
Vector3 direction = (end - start).normalized;
Vector3 normal = Vector3.Cross(direction, Vector3.up);
Vector3 leftPos = start + normal * distanceFromTrack;
Vector3 rightPos = start - normal * distanceFromTrack;
```

Для збереження продуктивності об'єкти розміщуються не на кожному сегменті, а через певний інтервал – наприклад , кожні 10 метрів. Це дозволяє уникнути перенасичення сцени та підтримує стабільну частоту кадрів навіть на довгих трасах.

Саме розміщення об'єктів виконується методом Instantiate(), який створює об’єкт з вибраного випадковим чином префаба. Щоб уникнути візуальної монотонності, кожен елемент отримує випадкове масштабування, обертання й (за потреби) варіації текстур [28].

```
GameObject prefab = environmentPrefabs[Random.Range(0, environmentPrefabs.Length)];
GameObject obj = Instantiate(prefab, position, Quaternion.Euler(0, Random.Range(-30f, 30f), 0));
float scale = Random.Range(0.8f, 1.2f);
obj.transform.localScale = Vector3.one * scale;
```

У реалізації також передбачено пулінг об’єктів – тобто повторне використання раніше створених елементів без постійного створення й знищення. Це особливо актуально при процедурній генерації, де середовище може оновлюватися в реальному часі або під час кожного заїзду. Об'єкти створюються на початку гри й лише переміщуються в нові позиції за потреби:

```
if (useObjectPooling) {
    obj = pooledObjects[currentIndex];
    currentIndex = (currentIndex + 1) % poolSize;
    obj.transform.position = position;
    obj.SetActive(true);
}
```

Важливо, що модуль не просто заповнює простір декоративними елементами, а також підтримує контекстну адаптацію до типу середовища. Наприклад, якщо було вибрано «пустельну» тему, використовуються кактуси, каміння, пісок. Якщо ж «лісову» - дерева , кущі, мох. Це забезпечується через окремі набори префабів, які активуються залежно від глобальних параметрів генерації.

У деяких випадках модуль оточення також може впливати на геймплей. Наприклад, у складніших режимах можуть з’являтися перешкоди, шлагбауми або динамічні об’єкти, які впливають на прохідність траси. У таких випадках

логіка їхнього розміщення базується не лише на геометрії, а й на рівні складності, розрахованому модулем аналітики:

```
if (difficulty == DifficultyLevel.Hard && Random.value > 0.7f) {
    PlaceObstacleAt(position);
}
```

У підсумку, модуль генерації оточення забезпечує динамічне й контекстне формування сцени навколо треку, враховуючи як візуальну складову, так і технічні обмеження. Завдяки варіативності, підтримці пулінгу, автоматичному позиціюванню та взаємодії з іншими модулями, він забезпечує повноцінне наповнення ігрового простору без втрати продуктивності й без потреби в ручному редагуванні рівнів. Це дозволяє кожній згенерованій трасі мати свій неповторний стиль і підвищує загальну реіграбельність гри.

Фізична модель керування автомобілем є одним із ключових компонентів ігрового процесу, оскільки саме вона формує у гравця відчуття динаміки, контролю та реалізму. У грі, де траси створюються процедурно, важливо, щоб модель руху транспортного засобу не лише адекватно реагувала на змінні параметри середовища, а й залишалася стабільною незалежно від конфігурації маршруту. Для цього реалізовано модуль фізичної взаємодії на основі компонентів Rigidbody, WheelCollider та механізмів внутрішньої фізики Unity.

Корпус автомобіля змодельовано як фізичне тіло з використанням компонента Rigidbody, що дозволяє симулювати гравітацію, імпульси та взаємодію з іншими об’єктами сцени. Для відображення коліс використано чотири WheelCollider – по одному для кожного колеса. Їхнє положення у просторі синхронізується з візуальними моделями, щоб зберігати відповідність між фізичною симуляцією й зображенням на екрані. Це здійснюється через метод GetWorldPose, який кожного кадру оновлює положення й обертання 3D-моделей коліс згідно з фактичною позицією колайдера:

```
private void UpdateWheelPose(WheelCollider collider, Transform mesh) {
```

```
    Vector3 pos;
    Quaternion rot;
    collider.GetWorldPose(out pos, out rot);
    mesh.position = pos;
    mesh.rotation = rot;
}
```

Що стосується керування, то основні дії гравця – натискання на педалі газу, гальмування й повороти – оброблюються у методі FixedUpdate. Кермо обчислюється пропорційно до значення осі Horizontal і передається лише на передні колеса, які забезпечують поворот:

```
float horizontal = Input.GetAxis("Horizontal");
currentSteerAngle = maxSteeringAngle * horizontal;
frontLeft.steerAngle = currentSteerAngle;
frontRight.steerAngle = currentSteerAngle;
```

Сила тяги генерується на задніх колесах. Вона розраховується на основі значення осі Vertical, що відповідає натисканню на газ, і множиться на максимальний крутний момент. Окремо перевіряється натискання на клавішу гальмування (наприклад, пробіл), після чого на всі чотири колеса подається гальмівна сила:

```
float vertical = Input.GetAxis("Vertical");
currentMotorTorque = motorTorque * vertical;

bool isBraking = Input.GetKey(KeyCode.Space);
currentBrakeForce = isBraking ? brakeForce : 0f;

rearLeft.motorTorque = currentMotorTorque;
rearRight.motorTorque = currentMotorTorque;

frontLeft.brakeTorque = currentBrakeForce;
frontRight.brakeTorque = currentBrakeForce;
rearLeft.brakeTorque = currentBrakeForce;
rearRight.brakeTorque = currentBrakeForce;
```

Важливим аспектом є забезпечення адекватної реакції транспортного засобу на зміну поверхні. Під час генерації траси кожному сегменту можуть бути призначені різні PhysicMaterial, які змінюють коефіцієнти тертя. Наприклад, рух по гравію або льоду відчувається зовсім інакше, ніж на сухому асфальті. Це досягається автоматичним призначенням фізичних матеріалів колайдерам ділянок треку, залежно від заданого типу середовища або погодних умов.

Додатково в реалізації враховується кут нахилу поверхні, що особливо важливо для процедурно згенерованих маршрутів зі змінною топографією. При русі вгору потужність автомобіля змінюється залежно від кута, а при спуску може виникнути ефект неконтрольованого прискорення. В окремих ситуаціях - наприклад, при надмірному нахилі кузова – вмикається стабілізаційна сила або обмежується подача тяги, щоб уникнути перевертання автомобіля.

Для покращення стійкості на поворотах реалізовано просту систему антизаносу, яка слідкує за перевищенням критичних значень зчеплення і за потреби зменшує потужність або додає протидіючі сили. У поєднанні з правильно налаштованими колесами, вагою кузова та налаштуванням підвіски це дозволяє досягти ефекту реалістичного, але контрольованого заносу на великій швидкості.

Важливо також, що всі фізичні обчислення винесено у метод FixedUpdate, який працює з постійним кроком часу і дозволяє уникати нестабільності при високому навантаженні. Тим часом метод Update відповідає лише за зчитування введення користувача, що дозволяє чітко розділити логіку фізики та контролю.

У результаті реалізована система забезпечує повноцінну фізичну модель для гоночного автомобіля, яка реагує на зміну середовища, адаптується до різних конфігурацій маршруту і залишає простір для розширення – наприклад , шляхом впровадження передач, пошкоджень, системи штучного інтелекту супротивників або різних типів автомобілів. Вона органічно вписується в загальну архітектуру гри й виконує критичну функцію у формуванні користувацького досвіду, зберігаючи баланс реалістичності та комфорту [11].

У системі процедурної генерації трас критично важливим є забезпечення не лише варіативності, а й ігрової коректності. Створений маршрут повинен бути не просто візуально цікавим, а обов’язково прохідним – тобто таким, який гравець може реально подолати без застрягання, неможливих поворотів, перешкод у критичних місцях або розривів у геометрії. Для цього реалізовано окремий модуль перевірки прохідності, який аналізує згенеровану трасу до її фінального затвердження у грі.

Основна ідея реалізації модуля полягає в віртуальному проходженні треку від початку до кінця. При цьому система не запускає повноцінного фізичного симулятора, а використовує спрощену логіку проходження контрольних точок, щоб виявити потенційні проблеми з геометрією маршруту або розміщенням об'єктів.

Насамперед аналізується відстань між точками. Якщо вона перевищує допустиму межу (тобто створюється великий розрив або «дірка» в трасі), система автоматично позначає маршрут як некоректний:

```
float segmentLength = Vector3.Distance(points[i], points[i + 1]);
if (segmentLength > maxAllowedSegmentLength)
{
    Debug.LogWarning("Занадто великий розрив між точками.");
    return false;
}
```

Наступний крок – перевірка радіусів поворотів. Якщо між векторами напрямку двох суміжних сегментів кут зміни є надто гострим, траса вважається потенційно непридатною для проходження, особливо на високій швидкості. Це особливо важливо у зв’язку з тим, що керування автомобілем базується на фізиці WheelCollider, яка не прощає занадто різких поворотів:

```
Vector3 dir1 = (points[i] - points[i - 1]).normalized;
Vector3 dir2 = (points[i + 1] - points[i]).normalized;
float turnAngle = Vector3.Angle(dir1, dir2);
```

```
if (turnAngle > maxTurnThreshold)
{
    Debug.LogWarning("Поворот надто крутий.");
    return false;}
```

Окрім геометричних параметрів, модуль також перевіряє колізії з об'єктами оточення. Наприклад, якщо після генерації ландшафту або розміщення декорацій деякі з них опинилися безпосередньо на дорозі, це може зробити трасу непрохідною. Для цього використовується простий Physics.CheckBox, який перевіряє наявність колізій уздовж уявного об’єму.

```
if (Physics.CheckBox(position, boxSize, Quaternion.identity, obstacleMask))
{
    Debug.LogWarning("Виявлено перешкоду на трасі.");
    return false;
}
```

Залежно від складності та тривалості маршруту, ця перевірка може бути оптимізована – наприклад, аналізувати лише кожну другу або третю точку. При виявленні проблем модуль або повністю скасовує маршрут, або передає сигнал генератору для повторного формування сегмента, не знищуючи всю трасу цілком. Це дозволяє зберігати продуктивність і мінімізувати повторне використання ресурсів.

Ще однією важливою функцією модуля є оцінка висоти між точками. Якщо спостерігається різкий перепад у вертикальній координаті - наприклад, траса «стрибає» вгору або вниз більш ніж на дозволений ліміт - маршрут вважається помилковим. Це дозволяє уникнути ситуацій, коли транспорт не може подолати підйом або неконтрольовано падає в ущелину:

```
float deltaY = Mathf.Abs(points[i + 1].y - points[i].y);
if (deltaY > maxSlopeHeight)
{
    Debug.LogWarning("Перепад висоти надто великий.");
    return false;}
```

Модуль також виконує остаточну симуляцію умовного проходження: від першої точки до останньої система «рухається» уявним курсором, оцінюючи можливість досягти фінішу без блокувань. Це не є фізична симуляція, але достатньо точна, щоби виявити критичні помилки до початку реального заїзду.

У фіналі, якщо всі перевірки проходять успішно, модуль повертає позитивний результат і підтверджує трасу як прохідну. В іншому випадку виводиться повідомлення про помилку, і трасу або перегенеровують автоматично, або виводять повідомлення для розробника [17].

Таким чином, модуль перевірки прохідності забезпечує важливу ланку між генерацією маршруту та його використанням у геймплеї. Він працює як гарантія якості, дозволяючи уникати технічних проблем, які могли б порушити логіку гри, зіпсувати досвід користувача або спричинити аварійні ситуації в ігровому процесі. Його використання дозволяє процедурній генерації залишатися не лише гнучкою, а й надійною з точки зору проєктування та тестування.

У результаті реалізації всіх основних модулів генерації геометрії трас, аналітики складності, побудови оточення, фізичної симуляції та перевірки прохідності – було створено повноцінну ігрову сцену, яка динамічно змінюється при кожному запуску. Система повністю автоматизована та не вимагає втручання користувача або розробника для побудови нового маршруту.

Нижче наведено серію скріншотів, що демонструють результати роботи розробленої системи в реальному часі. Зображення ілюструють функціонування інтерфейсу HUD (із відображенням швидкості, таймера та прогресу), зовнішній вигляд автомобіля у перспективі від третьої особи, а також різноманітні варіації згенерованих трас – включно з асфальтованими прямими, нічними заїздами та сценами з елементами оточення.

Зображення підтверджують коректність і цілісність роботи всієї системи, а також демонструють візуальну частину проєкту як повноцінного грального продукту, що реагує на параметри генерації та підтримує стабільний рівень продуктивності.

Рисунок 3.1 – Демонстрація частити треку з рослинністю і оточенням

На першому рисунку демонструється відрізок треку з грунтовою дорогою та скелею, алгоритм відпрацював гарно, створив досить привабливу дорогу, без якихось проблем.

Рисунок 3.2 – Демонстрація асфальтованої частини

Як видно з рисунку 3.2 в грі також реалізована різні «біоми» в яких можуть проходити за їзди і різний час, зміну якого видно на рисунку 3.3

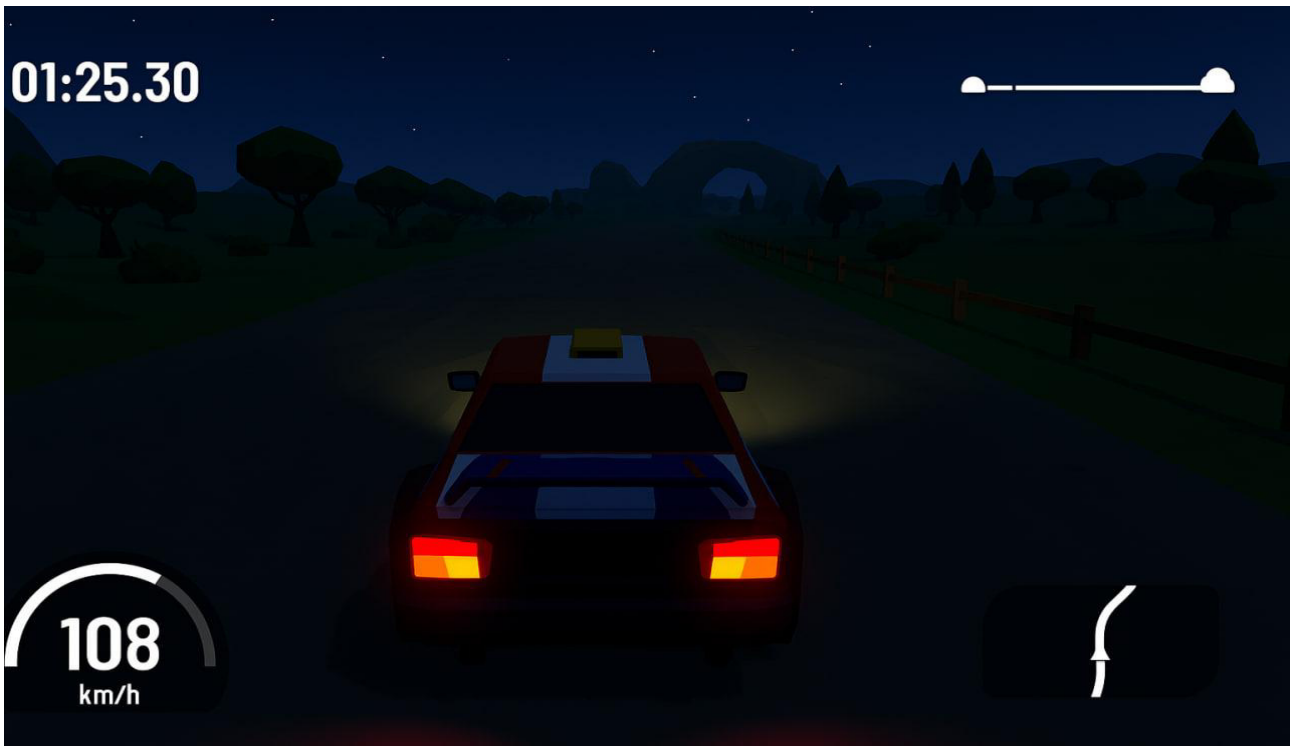


Рисунок 3.3 – Демонстрація зміни часу в грі.

3.2 Системні вимоги

Для забезпечення стабільної роботи проєкту, створеного в середовищі Unity з використанням процедурної генерації, фізики та 3D-графіки, було визначено базові системні вимоги до пристроїв, на яких запускається додаток. Ураховуючи використання low poly-стилістики, оптимізацію ресурсів і відсутність складних візуальних ефектів, гра може функціонувати навіть на пристроях середнього рівня продуктивності.

Мінімальні системні вимоги:

- операційна система: Windows 7 / 8 / 10 (64-bit);
- процесор: Intel Core i3 4-го покоління або AMD FX-6300;
- оперативна пам’ять: 4 ГБ
- відеокарта: NVIDIA GTX 750 Ti / AMD R7 260X (1 ГБ відеопам’яті)

– вільне місце на диску: 1.5 ГБ

– DirectX: версія 11

Рекомендовані системні вимоги:

– операційна система: Windows 10 / 11 (64-bit)

– процесор: Intel Core i5 8-го покоління або AMD Ryzen 5

– оперативна пам’ять: 8 ГБ

– відеокарта: NVIDIA GTX 1050 Ti або новіша / AMD RX 560 або новіша

– вільне місце на диску: 2 ГБ

– DirectX: версія 12

Завдяки спрощеній графіці, використанню процедурних методів і динамічному завантаженню об'єктів, програму можна запускати на ноутбуках та комп'ютерах із базовою конфігурацією без значної втрати якості геймплею. Однак для повноцінного ігрового досвіду з високою частотою кадрів і стабільною продуктивністю рекомендується використовувати систему, що відповідає рекомендованим параметрам [29].

3.3 Тестування програмного забезпечення

Якість програмного забезпечення, зокрема в галузі розробки ігор, безпосередньо залежить від ефективності проведеного тестування. У межах даного проєкту тестування виконувалося на всіх основних рівнях: від юніт-тестування окремих модулів до перевірки цілісної інтегрованої системи в умовах максимально наближених до реального використання.[11]

Першочергову увагу було приділено юніт-тестуванню, яке дозволяє перевірити правильність роботи окремих функціональних одиниць. Зокрема, було протестовано:

– генерацію контрольних точок маршруту з урахуванням параметрів складності;

– розрахунок кутів поворотів для аналізу складності;

– адаптацію значень до рівня гравця;

– коректність математичних обчислень фізичної моделі;

– розміщення об'єктів оточення по обидва боки дороги.

Для цього використовувалися власні тестові сцени в Unity, а також скрипти перевірки за допомогою методу Assert, які дозволяли автоматично виявляти логічні помилки при побудові маршруту чи розрахунках.

Водночас було проведено інтеграційне тестування, що передбачає перевірку взаємодії кількох модулів у єдиній системі. Наприклад, тестувалася коректність взаємодії генератора треку з модулем аналітики складності: чи передаються всі параметри вірно, чи відповідає оцінка складності реальним характеристикам маршруту. Подібним чином перевірялося злагоджене функціонування модуля генерації оточення з фізичною симуляцією - чи не перекривають декоративні об’єкти маршрут, чи не виникають колізії.

Тестування графічного інтерфейсу охоплювало перевірку усіх функціональних елементів HUD: динамічне оновлення швидкості, таймера, прогресу, а також реакцію кнопок меню. Було перевірено, що на різних роздільностях і співвідношеннях сторін інтерфейс залишається коректно відображеним і не виходить за межі екрану.

У рамках навантажувального тестування перевірялася швидкість генерації трас при різних параметрах. Було встановлено, що навіть на середньому за потужністю обладнанні повна генерація (геометрія + оточення + фізика) займає не більше 2,5 секунд, що є прийнятним для реального ігрового процесу. Також оцінювалося споживання пам’яті та стабільність FPS при збільшенні довжини треку. Результати свідчать про достатній рівень оптимізації: при трасах довжиною понад 1000 метрів середній FPS не опускався нижче 50 кадрів на секунду на тестовому ПК з процесором Intel Core i5 та відеокартою GTX 1050.

Не менш важливим етапом було функціональне тестування на прохідність трас, яке проводилося як вручну (з використанням ігрового керування), так і автоматично - за допомогою умовного віртуального проходження через масив контрольних точок. Це дозволяло виявити непрохідні сегменти, надто різкі

повороти, розриви геометрії або конфлікти з об’єктами оточення. При виявленні помилок трасу перегенеровували з іншим набором параметрів.

Загалом, комплекс проведених тестів дозволив переконатися в стабільності, логічній завершеності та узгодженості всіх модулів гри. Завдяки модульній архітектурі, тестування можна було здійснювати поетапно, ізоляційно, що дало змогу швидко виявляти та виправляти потенційні проблеми на ранніх етапах. Це особливо важливо для систем із процедурною генерацією, де кожна нова конфігурація є унікальною та не може бути перевірена вручну заздалегідь.

3.4 Висновки до розділу 3

У межах третього розділу було безпосередньо реалізовано основні функціональні модулі програмної системи, що забезпечують повноцінну роботу гоночної гри з процедурною генерацією трас. Впроваджено генератор геометрії маршруту, який створює унікальні конфігурації треків з урахуванням параметрів складності, довжини, кута поворотів та інших змінних. Завдяки застосуванню алгоритмів побудови маршруту з використанням сплайнів, випадкових точок та шумових функцій вдалося досягти природності траєкторії та уникнути одноманітності.

Крім базового генератора, було реалізовано модуль аналітики, який оцінює складність траси на основі геометричних показників і дозволяє адаптувати маршрут під рівень гравця. Важливим доповненням стала система генерації оточення, що автоматично наповнює локацію об’єктами довкілля, підвищуючи рівень занурення. Реалізовано фізичну модель взаємодії автомобіля з дорогою, з урахуванням тертя, прискорення, типу покриття та рельєфу. Окрему увагу приділено HUD - елементам інтерфейсу, що забезпечують інформування користувача про швидкість, прогрес проходження та таймеру.

Кожен модуль був спроєктований із дотриманням принципів зрозумілої структури коду, з поясненням логіки реалізації й відповідними прикладами коду. Програмна реалізація супроводжувалась демонстраційними скріншотами, які наочно ілюструють результати роботи кожного компонента системи. Усі елементи інтегровані в єдину ігрову логіку, що забезпечує цілісний і стабільний користувацький досвід.

Таким чином, у результаті реалізації третього розділу було створено повноцінне програмне середовище з широким функціоналом, яке відповідає технічному завданню і готове до тестування та подальшого розширення.

ВИСНОВКИ

У ході виконання дипломного проєкту було розроблено програмну систему, що реалізує функціонал процедурної генерації трас для гоночної гри в середовищі Unity. Проєкт охоплює повний цикл розробки - від постановки задачі та моделювання архітектури до програмної реалізації основних модулів, тестування системи та візуальної демонстрації її роботи. У результаті виконано глибокий аналіз предметної області, опрацьовано сучасні методи генерації ігрового контенту, а також створено повністю працездатний прототип гри, здатної формувати унікальні маршрути під час кожного запуску.

Розробка такого типу проєкту є актуальною у зв'язку зі зростаючим попитом на ігри з високою реіграбельністю, динамічним контентом і мінімальними затратами на ручне проєктування рівнів. Традиційні підходи до створення трас у гоночних іграх мають суттєві обмеження: вони потребують значних витрат часу для побудови кожної траси вручну, мають обмежену варіативність і швидко втрачають привабливість для користувача. Запропонована система процедурної генерації дозволяє розв'язати ці проблеми за допомогою автоматизованого створення маршрутів, які відрізняються за довжиною, конфігурацією, складністю та візуальним оформленням.

Система побудована на модульному принципі, що забезпечує її масштабованість та легкість подальшої модернізації. У рамках роботи було розроблено низку основних модулів: генератор геометрії треку, модуль аналітики складності, генератор ігрового оточення, модуль перевірки прохідності маршруту, а також фізичну модель взаємодії транспортного засобу з трасою. Кожен з них було реалізовано окремо, із забезпеченням внутрішньої цілісності та чітко визначеними зонами відповідальності. Завдяки такій архітектурі стало можливим гнучко налаштовувати параметри генерації, здійснювати підбір рівня складності в залежності від ігрового досвіду

користувача, а також динамічно адаптувати зовнішній вигляд трас до обраної тематики (лісова, пустельна, нічна тощо).

Окрему увагу в рамках проєкту було приділено розробці графічного інтерфейсу користувача. Реалізовано головне меню з базовими функціями запуску гри, налаштувань та виходу, а також HUD - інформаційний дисплей, що відображає швидкість, час, прогрес проходження маршруту та інші важливі елементи. Усі компоненти інтерфейсу адаптовані під різні роздільності екранів і відповідають загальній стилістиці low poly, обраній для цього проєкту.

Важливим етапом роботи стало тестування програмного продукту. Здійснено модульне тестування кожного з компонентів, зокрема перевірку генерації контрольних точок, обчислення кутів повороту, визначення перепадів висоти та виявлення потенційно непридатних ділянок. Проведено інтеграційне тестування взаємодії між модулями, а також оцінку загальної продуктивності системи при генерації трас різної складності. Результати підтверджують стабільну роботу гри навіть на пристроях середнього рівня продуктивності. Також перевірено правильність відображення інтерфейсу, обробку подій користувача та реакцію на змінні параметри генерації. Жодних критичних помилок у роботі системи не зафіксовано.

Під час тестових запусків гри виявлено, що середній час генерації повноцінної траси з оточенням і перевіркою прохідності становить менше 3 секунд. Це дозволяє використовувати систему не лише в тестових цілях, а й як реальну ігрову основу для майбутнього масштабування. Варіативність маршрутів, підтримка адаптивності під гравця, стабільна фізика та зручний інтерфейс створюють цілісну ігрову екосистему, яка здатна забезпечити позитивний користувацький досвід.

Таким чином, у межах кваліфікаційної роботи вдалося реалізувати повноцінну ігрову систему, що поєднує сучасні підходи до генерації контенту, модульну архітектуру, ефективну реалізацію й практичну придатність. Робота доводить, що за допомогою доступних інструментів, таких як Unity та мова C#, можна створити ефективну та розширювану ігрову інфраструктуру з

процедурною генерацією, придатну як для навчальних, так і комерційних проєктів.

Отримані результати можуть бути покладені в основу подальших досліджень та вдосконалення гри, зокрема:

– реалізація багатокористувацького режиму;

– впровадження більш складної системи штучного інтелекту.

– підтримка збереження улюблених трас;

– генерація ландшафту з погодними ефектами;

– застосування систем навчання або досягнень.

У перспективі проєкт може бути доопрацьований до повноцінного комерційного демо-продукту або слугувати базовим шаблоном для індивідуальних чи командних проєктів з розробки ігор у жанрі «racing».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Shaker N., Togelius J., Nelson M. Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Chapter 1: Introduction. URL: https://www.pcgbook.com/chapter01.pdf (дата звернення: 14.03.2025).

2. Togelius J., Yannakakis G., Stanley K., Browne C. Search-Based Procedural Content Generation: A Taxonomy and Survey. URL: https://stars.library.ucf.edu/scopus2010/2825/ (дата звернення: 27.04.2025).

3. Yannakakis G., Togelius J. Artificial Intelligence and Games. Повний текст книги (PDF): https://gameaibook.org/book.pdf (дата звернення: 13.03.2025).

4. Hendrikx M., Meijer S., Van Der Velden J., Iosup A. Procedural Content Generation for Games: A Survey. URL: https://dl.acm.org/doi/fullHtml/10.1145/2505057 (дата звернення: 23.04.2025).

5. Parish Y. I. H., Müller P. Procedural Modeling of Cities. SIGGRAPH 2001. URL: https://history.siggraph.org/learning/procedural-modeling-of-cities-by-parish-and-muller/ (дата звернення: 12.04.2025).

6. Nystrom R. Game Programming Patterns. Повний текст книги: https://gameprogrammingpatterns.com/contents.html (дата звернення: 05.04.2025).

7. Whitehead J. Procedural Level Generation for Platform Games. AIIDE 2010. URL: https://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/view/2153 (дата звернення: 26.04.2025).

8. Eberly D. H. Game Physics. URL (PDF): https://www.geometrictools.com/Documentation/GamePhysics.pdf (дата звернення: 23.03.2025).

9. Lengyel E. Mathematics for 3D Game Programming and Computer Graphics. URL: https://www.delmarlearning.com/companions/content/1435458869/ (дата звернення: 18.03.2025).

10. Unity Technologies. Unity Manual. URL: https://docs.unity3d.com/Manual/index.html (дата звернення: 17.04.2025).

11. Unity Technologies. Unity Scripting API. URL: https://docs.unity3d.com/ScriptReference/ (дата звернення: 22.03.2025).

12. Unity Technologies. Unity Learn – Онлайн-курси. URL: https://learn.unity.com/ (дата звернення: 20.03.2025).

13. Unity Asset Store. URL: https://assetstore.unity.com/ (дата звернення: 29.04.2025).

14. Unity Blog – Developers & Technology. URL: https://blog.unity.com/ (дата звернення: 15.04.2025).

15. Unity Technologies. Optimization Tips and Best Practices. URL: https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity.html (дата звернення: 04.03.2025).

16. Stack Overflow. Теми: Unity3D, procedural generation. URL: https://stackoverflow.com/questions/tagged/unity3d (дата звернення: 29.04.2025).

17. Unity Forum – Unity Community. URL: https://forum.unity.com/ (дата звернення: 06.04.2025).

18. Reddit – r/Unity3D. URL: https://www.reddit.com/r/Unity3D/ (дата звернення: 17.04.2025).

19. Sebastian Lague. Procedural Generation Series. URL: https://www.youtube.com/watch?v=4u6BbsG7qkQ (дата звернення: 01.04.2025).

20. Brackeys. Unity Tutorials URL: https://www.youtube.com/watch?v=IlKaB1etrik (дата звернення: 11.04.2025).

21. Holmer F. Game Dev Math. URL: https://www.youtube.com/watch?v=6z7GQewK-Ks (дата звернення: 30.03.2025).

22. Quilez I. Математика для генерації та шейдерів. URL: https://iquilezles.org/articles/palettes/ (дата звернення: 29.04.2025).

23. Catmull-Rom Splines – Математичні основи. URL: https://www.mvps.org/directx/articles/catmull/ (дата звернення: 15.03.2025).

24. Perlin Noise Algorithm. URL: https://thebookofshaders.com/11/ (дата звернення: 26.03.2025).

25. Simplex Noise – Опис і реалізація. URL: https://weber.itn.liu.se/~stegu/simplexnoise/ (дата звернення: 17.04.2025).

26. RedBlobGames – A Pathfinding & Geometry Algorithms. URL: https://www.redblobgames.com/ (дата звернення: 20.03.2025).

27. ShaderToy – Демонстрації генерації ландшафтів. Приклад шейдера: https://www.shadertoy.com/view/4dS3Wd (дата звернення: 13.04.2025).

28. The Nature of Code – Шум, фізика, генерація. URL: https://natureofcode.com/ (дата звернення: 01.04.2025).

29. Kelly T. Level Design Processes and Pipeline. URL: https://www.gamasutra.com/view/feature/134949/level_design_processes_and_pipeline.php (дата звернення: 20.04.2025).

30. Togelius J., Shaker N., Nelson M. Procedural Content Generation: Goals, Challenges and Actionable Steps. URL: https://dl.acm.org/doi/fullHtml/10.1145/3386252 (дата звернення: 25.04.2025).

31. Freeman B. Terrain Generation Techniques in Unity. URL: https://gamedevacademy.org/how-to-generate-procedural-terrain-using-unity/ (дата звернення: 30.03.2025).

32. GDC Vault. Procedural Tricks in AAA Racing Games [Відео]. URL: https://www.gdcvault.com/play/1025864/Procedural-Tricks-in-AAA-Racing (дата звернення: 26.03.2025).

33. Thompson J. Game Design: Principles, Practice, and Techniques – The Ultimate Guide for the Aspiring Game Designer. URL: https://surl.li/kxgahx (дата звернення: 25.04.2025).

34. Millington I., Funge J. Artificial Intelligence for Games. URL: https://www.crcpress.com/Artificial-Intelligence-for-Games/Millington-Funge/p/book/9780123747310 (дата звернення: 06.04.2025).

35. Sweetser P. Emergence in Games. URL: https://www.cengage.com/c/emergence-in-games-1e-sweetser/9781401881235/ (дата звернення: 13.04.2025).

36. Ashmore C., Nitsche M. The Quest in a Generated World: Writing Dynamic Stories for Games. URL: https://www.tandfonline.com/doi/abs/10.1080/14626260701531974 (дата звернення: 04.04.2025).

37. Bidarra R., Smelik R. Procedural Content Generation for Unity Game Development. URL: https://homepages.cwi.nl/~bidarra/ (дата звернення: 23.04.2025).

38. GameDev.net – Форум розробників ігор. URL: https://www.gamedev.net (дата звернення: 24.04.2025).

39. Akenine-Möller T., Haines E., Hoffman N. Real-Time Rendering. URL: https://www.crcpress.com/Real-Time-Rendering-Fourth-Edition/Akenine-Moller-Haines-Hoffman/p/book/9781138627000 (дата звернення: 21.03.2025).

40. Knuth D. E. The Art of Computer Programming. Vol. 2: Seminumerical Algorithms. URL: https://www-cs-faculty.stanford.edu/~knuth/taocp.html (дата звернення: 14.03.2025).

41. Tang C. An Introduction to Game Physics with Box2D. URL: https://www.crcpress.com/An-Introduction-to-Game-Physics-with-Box2D/Tang/p/book/9781482220134 (дата звернення: 27.04.2025).

42. Koster R. A Theory of Fun for Game Design. URL: https://www.oreilly.com/library/view/a-theory-of/9781449363215/ (дата звернення: 14.03.2025).

ДОДАТОК А

(обов'язковий)

КОД ПРОГРАМИ

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

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
```

```
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
```

```
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
```

```
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
```

```
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3  offset  =  new  Vector3(Random.Range(-10f,  -5f),  0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
```

```
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3   offset   =   new   Vector3(Random.Range(-10f,   -5f),   0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
            }
```

```
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
```

```
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}
```

```
// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
```

```
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
```

```
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
```

```
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3   offset   =   new   Vector3(Random.Range(-10f,   -5f),   0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

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

```
public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3   offset   =   new   Vector3(Random.Range(-10f,   -5f),   0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
```

```
                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;
```

```
    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
```

```
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}
```

```
// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
```

```
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3   offset   =   new   Vector3(Random.Range(-10f,   -5f),   0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;
```

```
    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3   offset   =   new   Vector3(Random.Range(-10f,   -5f),   0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
```

```
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
```

```
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
```

```
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
```

```
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
```

```
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
```

```
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3   offset   =   new   Vector3(Random.Range(-10f,   -5f),   0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
```

```
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3   offset   =   new   Vector3(Random.Range(-10f,   -5f),   0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
```

```
                Instantiate(decorations[Random.Range(0,     decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition    +=    currentRotation    *    Vector3.forward    *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
```

```
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
```

```
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}

using UnityEngine;
using System.Collections.Generic;

public class TrackGenerator : MonoBehaviour
{
    public GameObject roadSegmentPrefab;
    public int segmentCount = 100;
    public float segmentLength = 10f;
    public float maxTurnAngle = 30f;

    private List<GameObject> segments = new List<GameObject>();
    private Vector3 currentPosition = Vector3.zero;
    private Quaternion currentRotation = Quaternion.identity;

    void Start()
    {
        GenerateTrack();
    }

    void GenerateTrack()
    {
        for (int i = 0; i < segmentCount; i++)
        {
            float turn = Random.Range(-maxTurnAngle, maxTurnAngle);
            currentRotation *= Quaternion.Euler(0, turn, 0);
            currentPosition += currentRotation * Vector3.forward *
segmentLength;
            GameObject segment = Instantiate(roadSegmentPrefab, currentPosition,
currentRotation);
            segments.Add(segment);
        }
    }
}

// Клас аналітики складності треку
public class TrackDifficultyAnalyzer
```

```
{
    public float AnalyzeDifficulty(List<Vector3> points)
    {
        float totalTurn = 0f;
        for (int i = 2; i < points.Count; i++)
        {
            Vector3 a = points[i - 2];
            Vector3 b = points[i - 1];
            Vector3 c = points[i];
            float angle = Vector3.Angle(b - a, c - b);
            totalTurn += angle;
        }
        return totalTurn / points.Count;
    }
}

// Модуль генерації оточення
public class EnvironmentSpawner : MonoBehaviour
{
    public GameObject[] decorations;
    public float spacing = 15f;
    public int countPerSide = 50;

    public void SpawnAlongTrack(List<Vector3> trackPoints)
    {
        foreach (var point in trackPoints)
        {
            for (int i = 0; i < countPerSide; i++)
            {
                Vector3 offset = new Vector3(Random.Range(-10f, -5f), 0,
Random.Range(-5f, 5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);

                offset = new Vector3(Random.Range(5f, 10f), 0, Random.Range(-5f,
5f));
                Instantiate(decorations[Random.Range(0, decorations.Length)],
point + offset, Quaternion.identity);
            }
        }
    }
}
```

ДОДАТОК Б

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ СЛАЙДИ

Рисунок Б.1 – Титульний слайд.

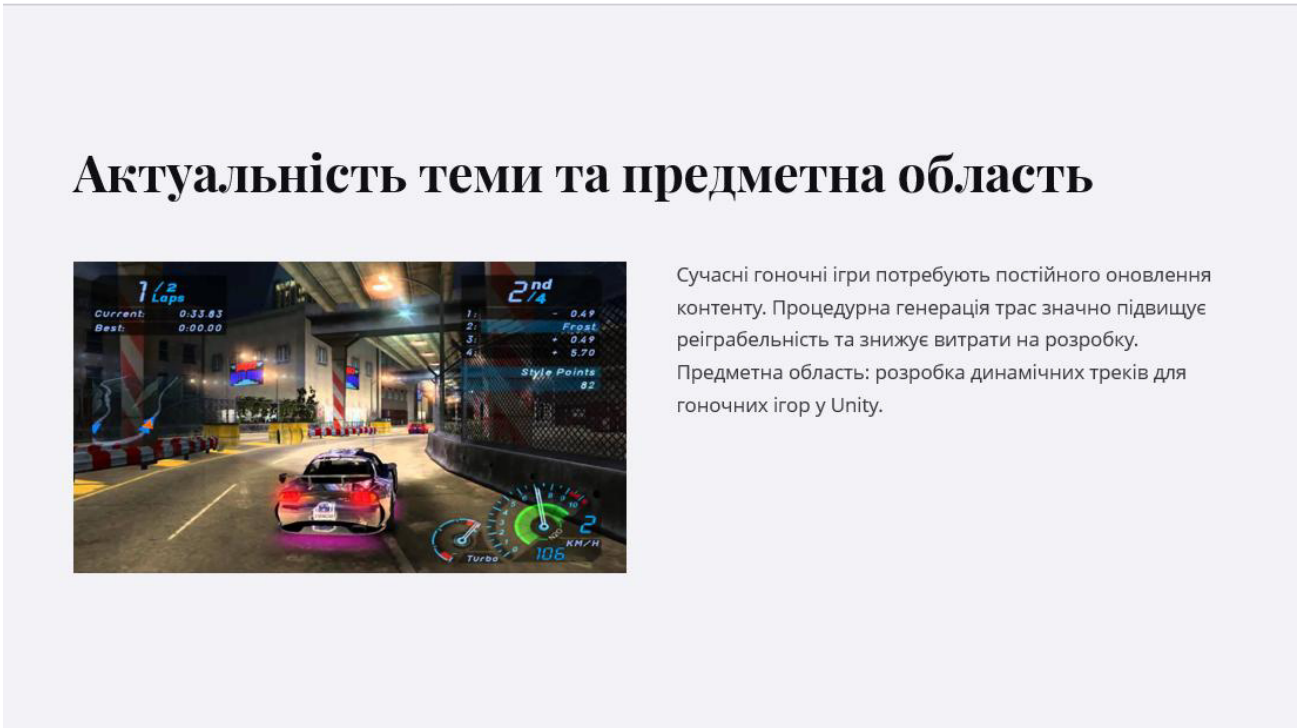


Рисунок Б.2 – Слайд «Актуальність теми та предметна область»

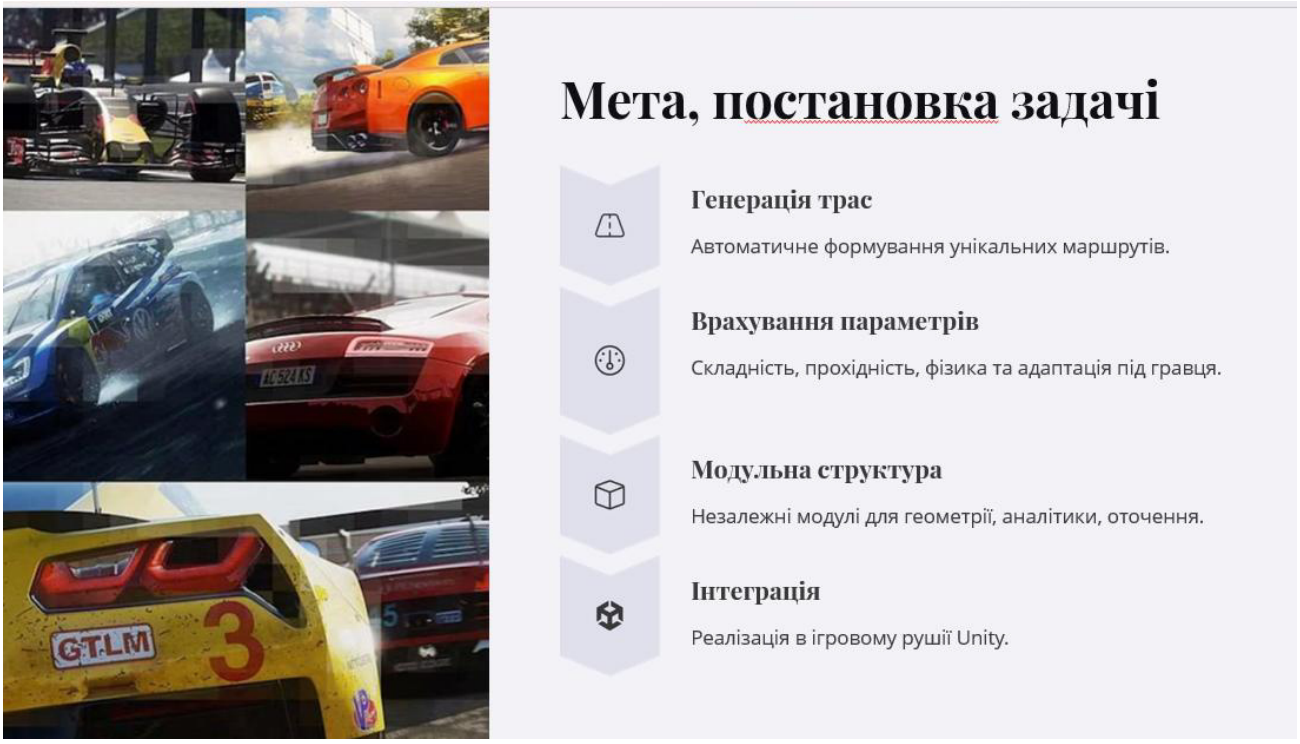


Рисунок Б.3 – Слайд «Мета, постановка задачі»

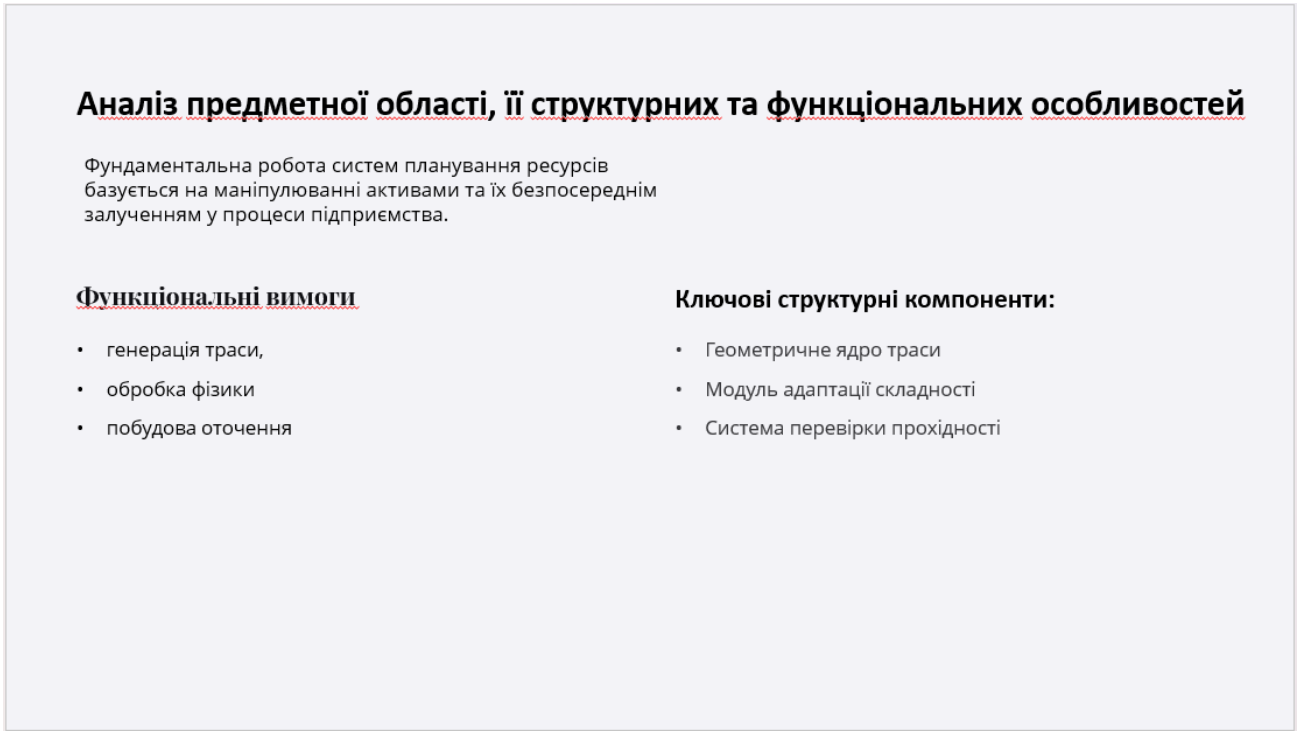


Рисунок Б.4 – Слайд «Аналіз предметної області»

Існуючі рішення

Існуючі рішення часто мають обмеження у динамічності або адаптивності. Вони не завжди враховують повний спектр взаємодій, що робить їх менш гнучкими для створення справді унікальних і захопливих трас.

Рисунок Б.5 – Слайд «Існуючі рішення»

Аналіз вимог до застосунку

Функціональні вимоги

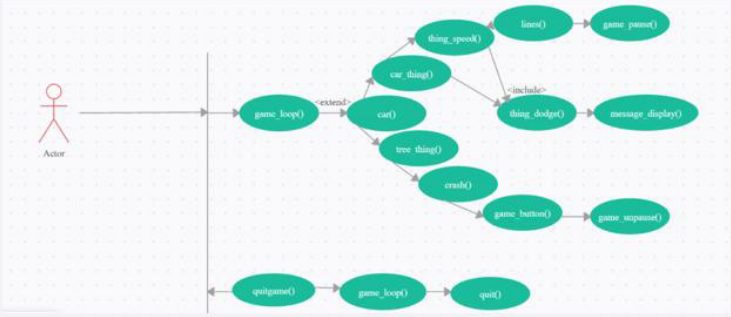
- Генерація унікальних трас.
- Автоматична оцінка складності.
- Додавання оточення (знаки, дерева).
- Перевірка прохідності траси.
- Приємна фізика руху авто.
- Адаптація складності.

Нефункціональні вимоги

- Швидка генерація (до 2–3 секунд).
- Стабільна робота.
- Масштабованість.
- Можливість розширення функціоналу.

Рисунок Б.6 – Слайд «Аналіз вимог до застосунку»

Діаграма використання



Використання системи

Ця діаграма показує взаємодію користувача із системою генерації трас. Користувач може ініціювати генерацію, налаштовувати параметри та отримувати готові траси.

Рисунок Б.7 – Слайд «Діаграма використання»

Загальний алгоритм гри

Старт гри
Ініціалізація та завантаження ресурсів.

Генерація траси
Процедурне створення маршруту.

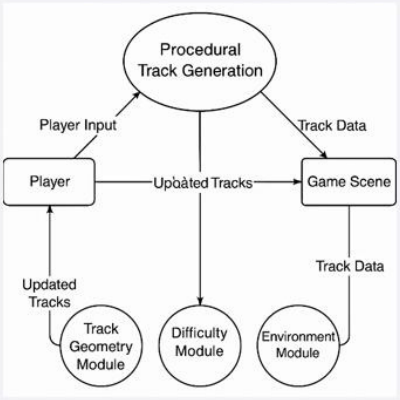
Симуляція руху
Фізика автомобіля та взаємодія з трасою.

Завершення перегонів
Оцінка результатів та виведення фіналу.

Рисунок Б.8 – Слайд «Загальний алгоритм гри»

DFD-діаграма

Діаграма потоків даних (DFD) ілюструє, як інформація переміщується в системі генерації трас. Вона показує зовнішні сутності, процеси, сховища даних та інформаційні потоки.



Вхідні дані

Параметри генерації, профіль гравця.

Процеси

Генерація, аналіз, адаптація, візуалізація.

Вихідні дані

Згенерована траса, звіт про складність.

Рисунок Б.9 – Слайд «DFD-діаграма»

Діаграма процесів гри

Ця діаграма деталізує послідовність виконання процесів під час ігрового сеансу. Вона демонструє взаємодію між основними модулями гри та їхній життєвий цикл.

Ввід гравця

Керування автомобілем.

Оновлення фізики

Розрахунок руху автомобіля.

Рендеринг

Візуалізація сцени.

Аналіз адаптації

Коригування складності.

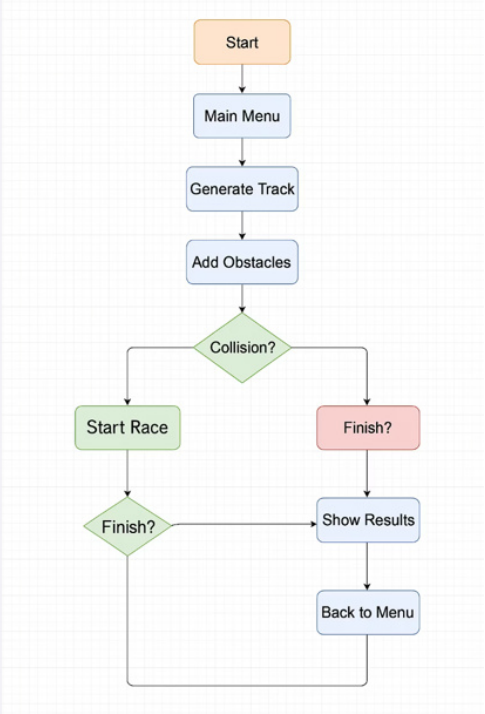


Рисунок Б.10 – Слайд «Діаграма процесів гри»

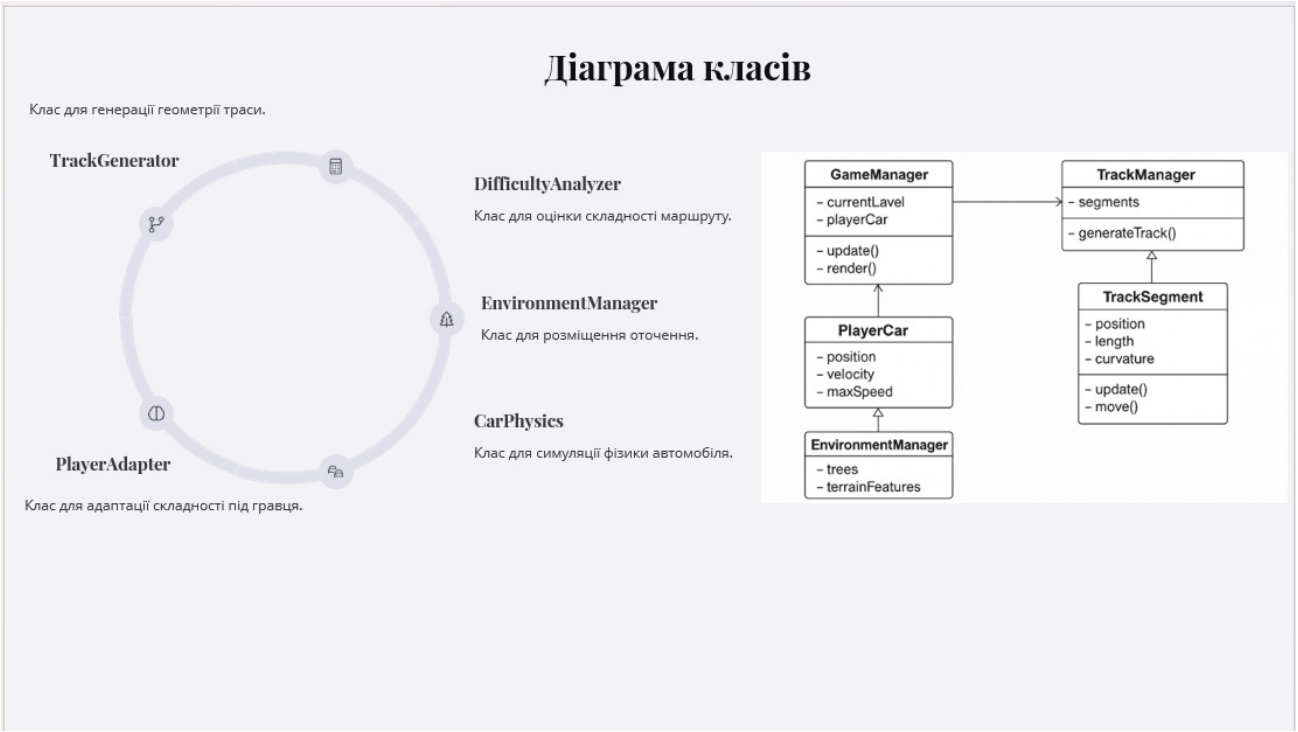


Рисунок Б.11 – Слайд «Діаграма класів»

Рисунок Б.12 – Слайд «Вибір технологій»

Рисунок Б.13 – Слайд «Системні вимоги»

Рисунок Б.14 – Слайд «Реалізація»

Тестування

У процесі розробки було проведено модульне та інтеграційне тестування основних компонентів гри: генерації траси, адаптації складності, фізики автомобіля та управління геймплеєм. Результати тестів підтвердили коректність роботи логіки, стабільність і відповідність функціональних вимог. Тестування проводилось у середовищі Unity з використанням вбудованих засобів перевірки.

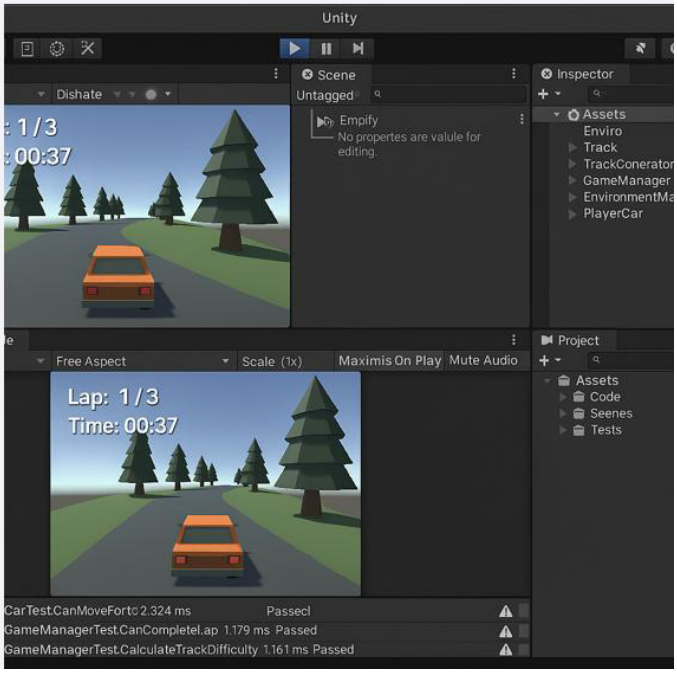


Рисунок Б.15 – Слайд «Тестування»

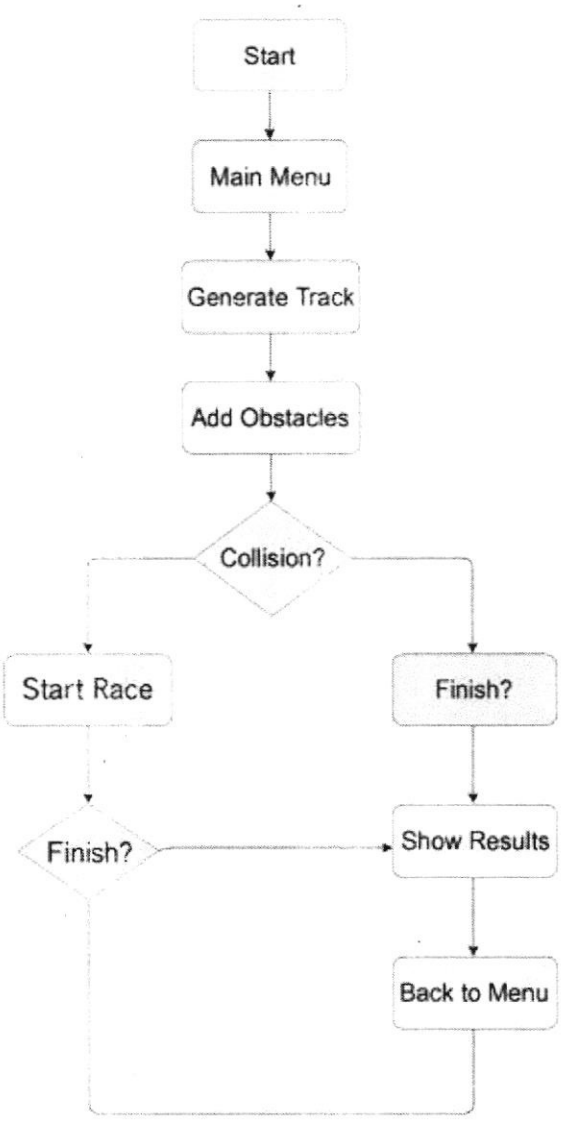
Висновки

Аспект	Опис
Результат розробкки	Створено повноцінну гоночну гру з процедурною генерацією трас.
Архітектура	Модульна, масштабована, гнучка до розширення.
Реалізовані функції	Генерація трас, фізика, адаптація складності, оточення, інтерфейс HUD.
Використані технології	Unity, C#, Perlin Noise, Catmull-Rom Splines.
Перевірка якості	Проведено модульне та інтеграційне тестування.
Потенціал розвитку	Мережевий режим, покращення візуалізації, нові механики

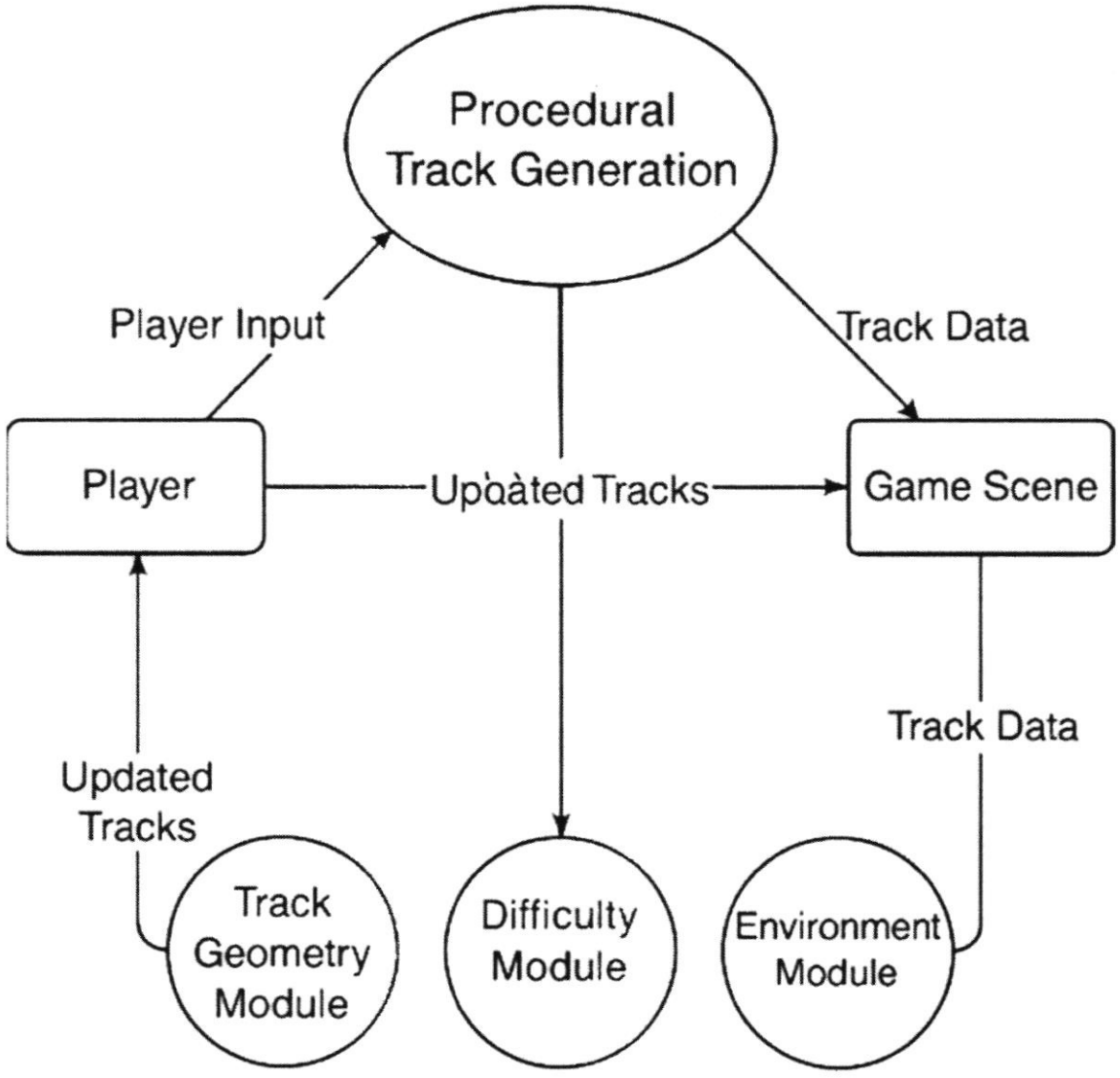
Рисунок Б.16 – Слайд «Висновки»

Рисунок Б.17 – Слайд «Дякую за увагу»

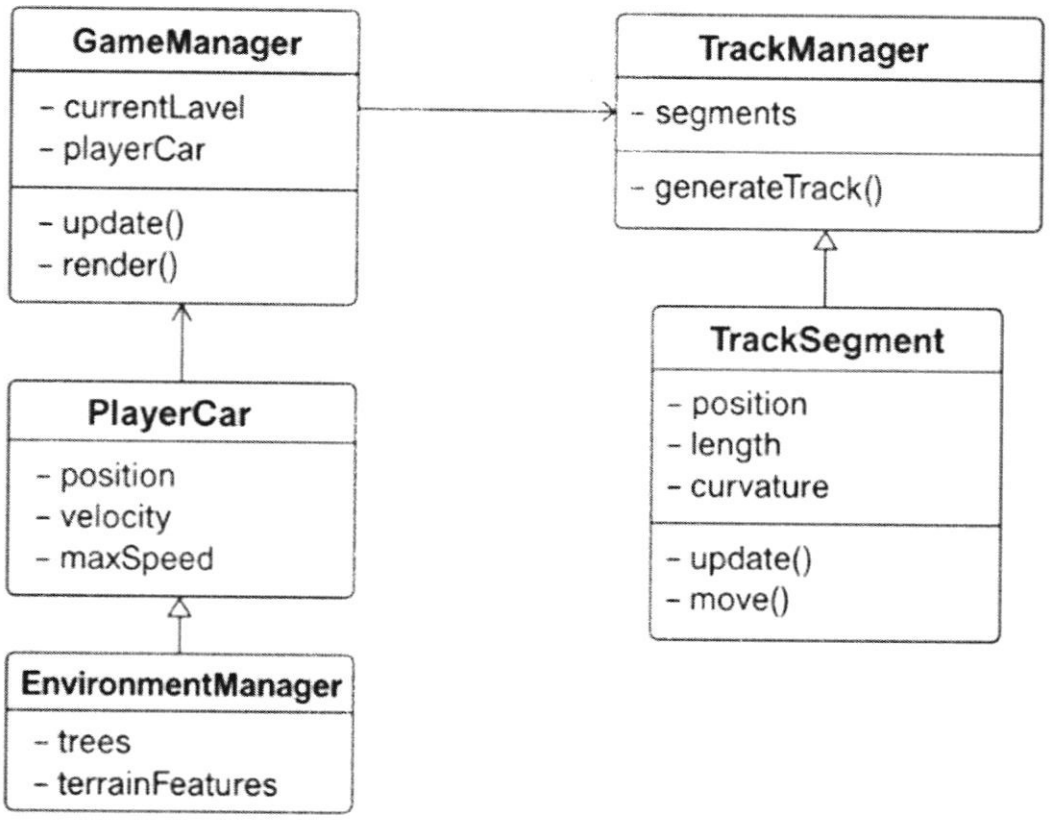
ГРАФІЧНІ МАТЕРІАЛИ



					КвРІПЗ.2101068.01.01.Е8		
					Програмне забезпечення симулятора автоперегонів на базі Unity	Літ.	Маса.	Масштаб
Змн.	Арк.	№ докум.	Підпис	Дата				
Розробив		Антіпов І. Д.		04.06				
Керівник		Яшина О. М.		04.06	Діаграма процесів гри	Аркуш 1	Аркушів 2	
Н. Контр.		Форкун Ю. В.		04.06		ХНУ, ІПЗ-21-1		
Зав. каф.		Бедратюк Л. П.		05.06				



					КвРІПЗ.2101068.01.01.Е8	
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*	Програмне забезпечення симулятору автоперегонів на базі Unity	*Літ.* / *Маса.* / *Масштаб*
Розробив		Антіпов І. Д.		04.06		
Керівник		Яшина О. М.		04.06	DFD-діаграма	Аркуш 2 / Аркушів 3
Н. Контр.		Форкун Ю. В.		05.06		ХНУ, ІПЗ-21-1
Зав. каф.		Бедратюк Л. П.		05.06		



					КвРІПЗ.2101068.01.01.Е8		
					Програмне забезпечення симулятору автоперегонів на базі Unity	Літ.	Маса.	Масштаб
Змн.	Арк.	№ докум.	Підпис	Дата				
Розробив		Антіпов І. Д.		04.06				
Керівник		Яшина О. М.		04.06	Діаграма класів	Аркуш 3	Аркушів 3	
Н. Контр.		Форкун Ю. В.		05.06		ХНУ, ІПЗ-21-1		
Зав. каф.		Бедратюк Л. П.		05.06				

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Антіпова Ігоря Дмитровича
факультет ІТ, IV курс, група ІПЗ-21-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

04.06.2025
дата

підпис

СУПРОВІДНІ ДОКУМЕНТИ

Mon Jun 02 01:12:08 EEST 2025, Форкун Юрій Вікторович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (UA) v-15.281 Educational

The maximum coincidence with one document 1.0%

Dictionaries check: en_US, ru_RU, ua_UA. **Errors in the documents: 9%**

	Document		Sum coincidence on the DB	
ID: 242711 Title: БКР_Програмне забезпечення симулятора автоперегонів на базі Unity Added in a DB: 2025-06-02 Authors: АНТІПОВ ІГОР Heads: ЯШИНА ОКСАНА канд. техн. наук, доцент Consultants: Opponents:	Symbols	Lexemes	Symbols	Lexemes
	70533	1065	1264 (2%)	17 (2%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Антіпов Ігор

Співавтор:

Назва: БКР_Програмне забезпечення симулятора автоперегонів на базі Unity

Науковий керівник:

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:1.3%

Коефіцієнт подібності 2:0.2%

Мікропробіли: 0

Заміна букв: 0

Інтервали: 0

Білі знаки: 96

Дата створення звіту: 2025-06-01 16:03:07.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата експерт

29.05.2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

освітнього ступеня «Бакалавр»

Дипломник Антіпов Ігор Дмитрович

Тема Програмне забезпечення симулятора автоперегонів на базі Unity

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3 кількість сторінок записки 68

1.Короткий зміст пояснювальної записки та прийнятих рішень У кваліфікаційній роботі розглянуто розробку гоночної гри з процедурною генерацією трас у середовищі Unity. Проведено змістовний аналіз предметної області, вивчено особливості сучасних підходів до створення ігор відповідного жанру, сформульовано функціональні й нефункціональні вимоги до системи. Обґрунтовано вибір архітектурних шаблонів, модульної структури проєкту та ключових технологій реалізації, включаючи використання C#, шумових алгоритмів і шаблонів проєктування.У процесі реалізації створено систему динамічного формування траси з адаптивною складністю, модуль аналітики, генератор оточення, фізичну модель поведінки транспорту, а також графічний інтерфейс з HUD. Проведено тестування функціональності, що підтвердило стабільність і коректну роботу реалізованих компонентів. Результатом є повноцінне програмне рішення, здатне генерувати унікальні заїзди, підтримуючи реіграбельність і візуальну привабливість гри

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана відповідно до поставленого завдання та з дотриманням всіх вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі обґрунтовано актуальність теми, визначено мету, завдання, об’єкт і предмет дослідження.У першому розділі проаналізовано предметну область, охарактеризовано жанри гоночних ігор, досліджено існуючі рішення з процедурною генерацією трас, сформульовано функціональні та нефункціональні вимоги. У другому розділі спроєктовано архітектуру гри, побудовано діаграми класів, алгоритмів і взаємодії, створено макети інтерфейсу, визначено структуру модулів та взаємозв’язки між ними. У третьому розділі реалізовано основні компоненти гри в середовищі Unity з використанням C#, впроваджено модулі генерації траси, аналітики, оточення, фізики й HUD. Проведено тестування та ілюстрацію роботи. У роботі використано сучасні алгоритми генерації контенту (сплайни, Perlin Noise), шаблони проєктування та підходи модульного програмування, що забезпечило гнучкість і масштабованість реалізації

4. Позитивні сторони роботи Тематика кваліфікаційної роботи є актуальною, оскільки процедурна генерація контенту набуває широкого застосування у сучасних ігрових проєктах. Реалізована система дозволяє автоматично створювати унікальні траси, що значно підвищує реіграбельність гри та скорочує час на створення контенту вручну. Робота вирізняється ґрунтовним аналізом предметної області, продуманим архітектурним проєктуванням, використанням сучасних технологій і підходів, зокрема Unity, C#, Perlin Noise, Catmull-Rom сплайнів. Окрему увагу приділено зручності гравця, візуальній складовій та адаптації складності, що свідчить про орієнтацію на якісний користувацький досвід.

5. Негативні сторони роботи Алгоритм генерації трас ще потребує оптимізації для складніших конфігурацій

6. Оцінка графічного оформлення та пояснювальної записки Візуальні матеріали розроблено у відповідності до змісту кваліфікаційної роботи та подано у формі графіків і зображень. Текстова частина проєкту оформлена згідно з установленими нормативами.

7. Відгук про кваліфікаційну роботу в цілому Кваліфікаційна робота демонструє належний рівень виконання та заслуговує схвальної оцінки. Пояснювальна записка відзначається логічною побудовою, доступністю викладу та послідовністю подачі, що сприяє легкому сприйняттю змісту в контексті обраної теми проектування. Графічні матеріали ефективно ілюструють основні елементи та рішення, прийняті в процесі розробки.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана у повному обсязі, відповідає поставленій задачі та заслуговує на оцінку «відмінно».

РЕЦЕНЗЕНТ Лисенко Сергій Миколайович, доктор технічних наук, професор, доцент кафедри факультету інформаційних технологій та комп'ютерної техніки

"4" червня 2025 р. _____
(підпис)

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Програмне забезпечення симулятора автоперегонів на базі Unity
Автор Антіпов Ігор Дмитрович
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»
Рівень вищої освіти Перший (бакалаврський)
Спеціальність 121 «Інженерія програмного забезпечення»
Науковий керівник: Яшина Оксана Миколаївна, канд. техн. наук, доцент

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	**відповідає**
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті кваліфікаційної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, завдання, анотація, відомість документів), у структурі змісту, назвах розділів/підрозділів тощо, у назвах публікацій та у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посиланання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає 1,3 %, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 5.06.2025

Завідувач кафедри _____ Підпис Леонід БЕДРАТЮК Ім'я, ПРІЗВИЩЕ

Гарант освітньої програми _____ Підпис Леонід БЕДРАТЮК Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи _____ Підпис Оксана ЯШИНА Ім'я, ПРІЗВИЩЕ