НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

До захисту допущено:

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інформаційні управляючі системи та технології»
спеціальності 126 «Інформаційні системи та технології»
на тему: «Система планування шляху доставки пошти»

Виконав
студент IV курсу, групи ІС-12
Кошовий Олександр Анатолійович _____

Керівник:
Асистент кафедри
Нестерук Андрій Олександрович _____

Рецензент:
Старший викладач кафедри ОТ
Алєнін Олег Ігорович _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інформаційні управляючі системи та технології»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____2025 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Кошовому Олександру Анатолійовичу

1. Тема проєкту «Система планування шляху доставки пошти», керівник проєкту Нестерук Андрій Олександрович, асистент кафедри, затверджені наказом по університету від «23» травня 2025 р. № 1705-с

2. Термін подання студентом проєкту: 9 червня 2025 р.

3. Вихідні дані до проєкту: мова програмування C#, редактор коду Rider, ігровий рушій Unity, бібліотека Zenject

4. Зміст пояснювальної записки:

1. Опис предметної області
2. Аналіз готових рішень
3. Формування вимог до системи
4. Вибір технологій розробки
5. Розробка інформаційної системи
6. Математичне забезпечення
7. Тестування системи

5. Перелік графічного матеріалу (із зазначенням обов’язкових креслеників, плакатів, презентацій тощо):

1. Діаграма дорожнього трафіку
2. Діаграма роботи алгоритму
3. Діаграма залежностей
4. Діаграма компонентів

6. Дата видачі завдання: 14 квітня 2025 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Ознайомлення з технічним завданням	15 квітня 2025	
2	Аналіз предметної області та логістичних задач	17 квітня 2025	
3	Розробка функціональної моделі системи планування маршруту	20 квітня 2025	
4	Огляд існуючих алгоритмів оптимізації шляхів і їх порівняння	23 квітня 2025	
5	Формування цілей і задач розробки програмного забезпечення	27 квітня 2025	
6	Проєктування архітектури	30 квітня 2025	
7	Розробка математичного забезпечення та вдосконаленого алгоритму TSP	3 травня 2025	
8	Реалізація алгоритму планування шляху та інтеграція у систему	7 травня 2025	
9	Розробка клієнтської частини з візуалізацією карти та маршрутів	11 травня 2025	
10	Тестування, валідація результатів і оцінка ефективності	17 травня 2025	

Студент Олександр КОШОВИЙ

Керівник Андрій НЕСТЕРУК

АНОТАЦІЯ

Система планування шляху доставки пошти.

Проєкт містить 61 с. тексту, 19 рисунків, 2 таблиці, посилання на 15 літературних джерел, додатки та 4 конструкторські документи.

АНАЛІЗ МАРШРУТІВ, СИСТЕМА ПЛАНУВАННЯ ДОСТАВКИ, АЛГОРИТМ A*, АЛГОРИТМ НАЙБЛИЖЧОГО СУСІДА, 2-OPT.

Об'єктом розробки є система планування оптимального маршруту для доставки пошти.

Мета розробки – підвищення ефективності пошуку оптимального маршруту для доставки пошти шляхом розробки автоматизованої системи планування шляху, яка дозволить скоротити час доставки та зменшити експлуатаційні витрати на транспортні засоби.

У дипломному проєкті розроблено автоматизовану систему планування маршруту доставки пошти в умовах міського середовища. Система складається з декількох частин, а саме: модуля генерації випадкових точок доставки на карті, модуля розрахунку оптимального маршруту за допомогою алгоритму найближчого сусіда з наступною оптимізацією за алгоритмом 2-opt, системи керування рухом автомобіля користувача та NPC-машин, які використовують алгоритм A* для навігації, та візуалізації побудованого маршруту за допомогою компонента LineRenderer.

Реалізована система дозволяє користувачам оперативно отримувати оптимальні маршрути доставки, знижуючи витрати часу та ресурсів, а також зручним способом орієнтуватися у складних міських умовах за допомогою наочного візуального представлення шляху.

SUMMARY

Mail delivery route planning system.

The project contains 61 pages. text, 19 figures, 2 tables, links to 15 literary sources, annexes and 4 design documents.

ROUTE ANALYSIS, DELIVERY PLANNING SYSTEM, A* ALGORITHM, NEAREST NEIGHBOUR ALGORITHM, 2-OPT.

The object of development is a route planning system for mail delivery.

The purpose of the development is to increase the efficiency of finding the optimal mail delivery route through the creation of an automated path-planning system, thereby reducing delivery time and decreasing operational costs for transportation.

In this diploma project, an automated route planning system for mail delivery in an urban environment was developed. The system consists of several components: a module for randomly generating delivery points on the map, a module for calculating the optimal delivery route using the Nearest Neighbour algorithm followed by route optimization via the 2-opt algorithm, a control system managing player vehicle movements and NPC cars utilizing the A* algorithm for navigation, and a visualization component using LineRenderer for displaying the calculated route.

The implemented system enables users to quickly obtain optimal delivery routes, reducing both time and resource expenditures, while conveniently navigating through complex urban environments thanks to the clear visual representation of the route.

№ рядка	Формат	Позначення	Найменування	Кіл. аркушів	№ екз.	Примітка
1			Документація загальна			
2						
3			Знову розроблена			
4						
5	А4	ІС12.170БАК.005 ПЗ	Пояснювальна записка	61		
6	А3	ІС12.170БАК.005 Д1	Система планування шляху для	1		
7			доставки пошти. Діаграма			
8			компонентів системи обробки			
9			дорожнього руху			
10	А3	ІС12.170БАК.005 Д2	Система планування шляху для	1		
11			доставки пошти. Діаграма			
12			діяльності алгоритму планування			
13			маршруту доставки			
14	А3	ІС12.170БАК.005 Д3	Система планування шляху для	1		
15			доставки пошти. Діаграма			
16			залежностей модулів			
17			Unity-проекту			
18	А3	ІС12.170БАК.005 Д4	Система планування шляху для	1		
19			доставки пошти. Діаграма			
20			послідовності системи			
21						
22						
23						
24						
25						
26						
27						
28						

Зм.	Лист	№ докум.	Підпис		ІС12.170БАК.005 ПЗ		
Розробив		Кошовий О.А.			Система планування шляху доставки пошти. Відомість дипломного проєкту	Літ.	Арк. 1 / Аркушів 1
Перевірив		Нестерук А.О.				Т	
Затв.					КПІ ім. Ігоря Сікорського Група ІС-12		

Пояснювальна записка
до дипломного проєкту
на тему: «Система планування шляху доставки пошти»

Київ – 2025 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

A* — алгоритм пошуку найкоротшого шляху (A-Star Algorithm);

TSP — задача комівояжера (Travelling Salesman Problem);

NPC — некерований гравцем персонаж (Non-Player Character);

NN — алгоритм найближчого сусіда (Nearest Neighbour Algorithm);

2-opt — алгоритм оптимізації маршруту шляхом заміни ребер (Two-Opt Algorithm);

UI — інтерфейс користувача (User Interface);

WP — маршрутна точка (Waypoint);

DP — точка доставки (Delivery Point);

ECS — система сутностей та компонентів (Entity Component System);

FPS — кількість кадрів на секунду (Frames Per Second);

DOTween — бібліотека для створення анімацій у Unity;

Unity — середовище розробки відеоігор (Unity Game Engine);

LineRenderer — компонент для візуалізації ліній у Unity.

ВСТУП

Актуальність проблеми. Актуальність проблеми зумовлена зростанням інтенсивності руху транспорту в міських умовах та необхідністю підвищення ефективності логістичних процесів. Сьогодні процес планування оптимальних маршрутів доставки пошти стикається з такими проблемами, як ускладнений вибір оптимальних шляхів у динамічному міському середовищі, постійні затримки через трафік та недостатня гнучкість існуючих систем маршрутизації [4]. Відсутність ефективного інструменту для автоматичного розрахунку оптимальних шляхів призводить до значних витрат часу, палива, а також негативно впливає на якість обслуговування клієнтів.

Розробка автоматизованої системи планування оптимальних маршрутів доставки пошти дозволяє значно полегшити цей процес. Інтеграція сучасних алгоритмів, таких як A* для пошуку найкоротшого шляху, алгоритму найближчого сусіда (Nearest Neighbour) для побудови базових маршрутів і подальшої оптимізації цих маршрутів методом 2-opt, забезпечує користувачам можливість оперативно й ефективно будувати маршрути, що мінімізують витрати часу і ресурсів. Крім того, інтерактивна візуалізація отриманих маршрутів на карті в режимі реального часу дозволяє водіям швидко орієнтуватися у складних умовах міського середовища, що робить систему особливо корисною у великих містах з високою щільністю руху.

Об’єкт дослідження. Об’єктом дослідження є процес автоматизованого планування маршрутів доставки пошти в умовах міської транспортної мережі.

Предмет дослідження. Предметом дослідження є методи й алгоритми планування маршрутів, зокрема алгоритми A*, найближчого сусіда та оптимізації 2-opt, а також техніки їхньої інтеграції та візуалізації в програмних застосунках.

Мета і завдання. Метою даного бакалаврського дипломного проєкту є підвищення ефективності пошуку оптимального маршруту для доставки пошти шляхом створення автоматизованої системи планування шляху. Для досягнення поставленої мети необхідно вирішити наступні завдання:

1) проаналізувати існуючі рішення для планування маршрутів, визначити їх переваги та недоліки та сформулювати основні вимоги до розроблюваної системи;

2) розробити програмне забезпечення для генерації випадкових точок доставки та їхнього збереження в базі даних, а також розробити програмне забезпечення для розрахунку та візуалізації маршрутів;

3) реалізувати математичне забезпечення, що включає алгоритми побудови початкового маршруту (Nearest Neighbour) та оптимізації побудованого маршруту (2-opt);

4) провести тестування розробленої системи й оцінити ефективність запропонованих алгоритмів на реальних та модельних даних, а також підготувати керівництво користувача.

Практичне значення. Створена автоматизована система планування маршрутів доставки пошти може використовуватись компаніями та поштовими службами для оптимізації логістичних процесів, зниження витрат на паливо та підвищення оперативності роботи кур’єрів. Завдяки зручному інтерфейсу та наочній візуалізації маршрутів, користувачі зможуть ефективніше виконувати поставлені завдання та приймати швидкі та обґрунтовані рішення в умовах складних транспортних ситуацій.

Структура роботи. Дипломний проєкт складається з наступних розділів: вступ, основні розділи, висновки, список використаних джерел із 15 найменувань, 1 додаток. Графічна частина включає 4 кресленика формату А3 . Загальний обсяг 69 сторінок.

1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис процесу діяльності

Для вирішення проблеми автоматизації пошуку оптимального маршруту доставки пошти необхідно створити систему, яка дозволяє користувачеві ефективно орієнтуватися у змодельованому міському середовищі. Користувач (гравець) одразу потрапляє до симуляції міста, де автоматично створюються випадкові точки доставки. Система негайно розраховує маршрут між цими точками, використовуючи евристичний алгоритм найближчого сусіда для початкового прокладання шляху.

Отриманий початковий маршрут оптимізується за допомогою алгоритму 2-opt, що дозволяє значно скоротити його довжину шляхом локальних перестановок ребер маршруту. Після завершення оптимізації, система за допомогою алгоритму A* розраховує найкоротший шлях між кожною парою послідовних точок доставки, враховуючи дорожню мережу міста.

Прокладений шлях до найближчої точки доставки візуалізується в ігровому середовищі за допомогою компонента LineRenderer, надаючи гравцю зрозумілу візуальну інформацію про наступну ціль. Гравець керує транспортним засобом, пересуваючись запропонованим маршрутом, змінюючи смуги руху та повертаючи на перехрестях, орієнтуючись на візуальні підказки.

Таким чином, вся діяльність гравця є повністю автоматизованою в контексті побудови й оптимізації маршруту, а також його візуалізації. Система не передбачає реєстрацію користувачів чи розмежування доступів і ролей.

1.2 Постановка задачі

1.2.1 Призначення системи

Призначенням системи є автоматизація процесу планування та оптимізації маршруту доставки пошти в умовах симульованого міського середовища. Система

дозволяє ефективно керувати рухом автомобіля користувача, зменшуючи витрати часу та палива завдяки оптимальному вибору шляхів між точками доставки.

1.2.2 Цілі та задачі розробки

Метою бакалаврського дипломного проєкту є підвищення ефективності пошуку оптимального маршруту доставки пошти. Для досягнення цієї мети необхідно вирішити такі задачі:

– генерація точок доставки. Автоматичне створення випадкових точок доставки у міському середовищі;

– початкова побудова маршруту. Використання алгоритму найближчого сусіда для побудови початкового маршруту;

– оптимізація маршруту. Покращення початкового маршруту за допомогою алгоритму 2-opt;

– пошук шляхів між точками доставки. Використання алгоритму A* для пошуку найкоротших шляхів між кожною парою точок;

– візуалізація маршруту. Відображення оптимального маршруту за допомогою LineRenderer для простого орієнтування гравця;

– керування транспортним засобом. Забезпечення зручного управління рухом автомобіля користувача, включаючи зміну смуг та повороти на перехрестях.

Висновки до розділу 1

У цьому розділі описано основні процеси діяльності гравця в системі планування оптимальних маршрутів доставки пошти: автоматичну генерацію точок, початкову побудову та оптимізацію маршруту, пошук найкоротших шляхів, візуалізацію маршруту та керування автомобілем користувача. Система повністю автоматизована, не потребує реєстрації користувачів, що забезпечує простий та ефективний процес планування та виконання поштової доставки в симуляції міського середовища.

2 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

2.1 Аналіз існуючих рішень на ринку

На сучасному ринку існує кілька популярних програмних продуктів для планування маршрутів і навігації, які можуть бути використані для оптимізації доставки пошти. Розглянемо деякі з них:

– Google Maps;

– Waze;

– HERE WeGo;

– Route4Me.

Google Maps – широко відомий сервіс, який пропонує прокладання маршрутів, навігацію та оцінку часу у дорозі з урахуванням трафіку.

Переваги:

– велика кількість актуальних картографічних даних;

– наявність інформації про дорожні затори;

– інтеграція з багатьма іншими сервісами.

Недоліки:

– не орієнтований на побудову оптимальних маршрутів для великої кількості точок доставки;

– відсутність вбудованих алгоритмів оптимізації маршрутів (таких як 2-opt);

– обмежені можливості налаштування маршруту для специфічних задач.

Waze – додаток для навігації, який в реальному часі збирає та аналізує дані про трафік від користувачів, надаючи інформацію про затори, аварії та інші події на дорозі.

Переваги:

– дані про ситуацію на дорогах в реальному часі;

– активна спільнота користувачів, яка постійно оновлює інформацію;

– зручний і простий інтерфейс.

Недоліки:

– не підходить для планування складних маршрутів з великою кількістю точок доставки;

– немає алгоритмів оптимізації маршрутів для спеціалізованих задач;

– можливі неточності через дані, що генеруються користувачами.

HERE WeGo – застосунок з розширеними функціями навігації та можливістю роботи в офлайн-режимі.

Переваги:

– наявність офлайн-карт;

– деталізована інформація про дорожню інфраструктуру;

– високий рівень деталізації картографічних даних.

Недоліки:

– відсутність функціоналу оптимізації маршрутів для численних точок доставки;

– менш активна спільнота користувачів у порівнянні з конкурентами;

– обмежені можливості інтеграції з іншими системами.

Route4Me – спеціалізована платформа для побудови оптимальних маршрутів з великою кількістю точок доставки.

Переваги:

– алгоритми оптимізації маршрутів (включаючи методи подібні до 2-opt);

– можливість роботи з великою кількістю точок доставки;

– інтеграція з GPS-системами для контролю руху транспорту.

Недоліки:

– комерційний сервіс із платною підпискою;

– потребує певного часу на навчання користувачів;

– недостатньо орієнтований на реалістичне моделювання транспортних потоків у симуляційних середовищах.

Проведений аналіз дозволяє визначити, що наявні рішення мають як суттєві переваги, так і значні недоліки, які можна усунути або значно покращити у нашому проєкті. Особливу увагу в розроблюваній системі буде приділено алгоритмам

планування й оптимізації маршрутів (найближчий сусід, 2-opt, A*), а також зручності візуалізації маршрутів у симульованому міському середовищі.

Висновки до розділу 2

У даному розділі здійснено детальний аналіз існуючих рішень для планування маршрутів доставки. Виявлено переваги й недоліки найбільш популярних сервісів-конкурентів. Це дозволило чітко сформулювати відмінності розроблюваної системи та визначити пріоритетні аспекти, такі як ефективна оптимізація маршрутів і якісна візуалізація, яким буде приділена основна увага під час реалізації проєкту.

3 ФОРМУВАННЯ ВИМОГ ДО СИСТЕМИ

3.1 Вимоги до системи в цілому

3.1.1 Вимоги до структури та функціонування системи

Перелік підсистем та їх призначення:

1) підсистема генерації точок доставки. Призначення – автоматизоване створення випадкових точок доставки у міському середовищі. Основні характеристики – оперативність генерації, точність позиціювання точок на карті міста;

2) підсистема побудови початкового маршруту. Призначення – побудова маршруту за допомогою алгоритму найближчого сусіда. Основні характеристики – швидкий пошук початкового рішення, простота реалізації алгоритму, достатня точність для первинного маршруту;

3) підсистема оптимізації маршруту. Призначення – поліпшення початкового маршруту шляхом застосування алгоритму 2-opt. Основні характеристики – ефективність оптимізації, швидкість роботи алгоритму, можливість значного скорочення загальної дистанції;

4) підсистема пошуку шляху між точками. Призначення – пошук найкоротших шляхів між точками доставки з використанням алгоритму A*. Основні характеристики – гарантоване знаходження найкоротшого шляху, ефективність роботи в умовах великих міських графів, швидкий перерахунок при зміні умов руху;

5) підсистема візуалізації маршруту. Призначення – наочне відображення маршруту руху до наступної точки доставки за допомогою компонента LineRenderer. Основні характеристики – інтуїтивно зрозуміла візуалізація, миттєве оновлення маршруту, інтеграція з симуляційним середовищем.

Вимоги до режимів функціонування системи: – постійна доступність. Система повинна безперервно функціонувати під час роботи користувача; – висока швидкість реакції. Швидкий розрахунок і візуалізація маршрутів; – актуальність та

точність даних. Постійне оновлення маршруту залежно від змін ситуації на дорогах.

Перспективи розвитку системи: – розширення функціоналу планування. В майбутньому система зможе враховувати додаткові параметри руху, такі як трафік або погодні умови; – інтеграція з GPS та системами моніторингу транспорту; – збільшення точності і деталізації картографічних даних; – створення додаткових модулів для симуляції більш реалістичних умов руху (наприклад, додавання поведінки інших учасників руху).

3.1.2 Вимоги до надійності

Перелік аварійних ситуацій та вимоги до надійності:

1) збій у роботі алгоритмів планування маршруту. Забезпечення можливості швидкого перезапуску алгоритму;

2) помилки візуалізації маршруту. Впровадження механізмів автоматичного відновлення роботи компонента LineRenderer;

3) проблеми з доступом до картографічних даних. Резервне зберігання та використання картографічних даних.

Вимоги до надійності технічних засобів та програмного забезпечення:

1) висока відмовостійкість і стабільність роботи програмних компонентів;

2) регулярне оновлення програмного забезпечення;

3) можливість моніторингу стану системи в режимі реального часу для оперативного виявлення та усунення несправностей.

3.1.3 Вимоги до збереження інформації

Перелік подій, за яких повинне забезпечуватись збереження інформації:

1) помилки у роботі алгоритмів або візуалізації. Всі дані про маршрут повинні бути надійно збережені;

2) атаки на систему. Забезпечення високого рівня захисту даних від

несанкціонованого доступу.

3.2 Вимоги до функціональних характеристик

3.2.1 Перелік функцій та задач, що підлягають автоматизації.

Генерація точок доставки: необхідно автоматизувати процес створення випадкових точок доставки в межах заданого міського середовища. Система має забезпечувати можливість генерування точок з реалістичною прив’язкою до дорожньої мережі.

Побудова початкового маршруту: автоматичний розрахунок первинного маршруту проходження усіх згенерованих точок доставки за допомогою алгоритму найближчого сусіда (Nearest Neighbour). Цей процес має бути максимально швидким, щоб оперативно забезпечувати користувача маршрутом для початку руху.

Оптимізація побудованого маршруту: автоматизація процесу вдосконалення початкового маршруту методом 2-opt для зменшення його загальної протяжності. Необхідно забезпечити миттєву оптимізацію, яка покращує маршрут до рівня, близького до оптимального.

Пошук оптимального шляху між точками: автоматичний пошук оптимального шляху між послідовними точками доставки за допомогою алгоритму A*, з урахуванням реальної структури дорожньої мережі. Забезпечення швидкої реакції на зміни умов пересування (наприклад, зміна стану доріг чи наявність перешкод).

Візуалізація маршруту: автоматичне візуальне представлення маршруту на карті за допомогою компонента LineRenderer в режимі реального часу. Візуалізація повинна бути миттєвою та інтуїтивно зрозумілою для користувача.

Вимоги до якості реалізації:

1) генерація точок доставки: якість реалізації – забезпечення реалістичного

та рівномірного розподілу точок доставки на карті міста; форма представлення – інтерактивні маркери на карті; точність – позиціонування з точністю ≥ 98 % щодо реальних координат; час виконання – < 1 с при запуску симуляції.

2) побудова початкового маршруту (алгоритм найближчого сусіда): якість реалізації – оперативний та стабільний розрахунок без затримок; форма представлення – звіт або візуалізація маршруту на карті; точність – ≥ 90 % від потенційно оптимального; час виконання – < 0,5 с для 20 точок.

3) оптимізація побудованого маршруту (алгоритм 2-opt): якість реалізації – значне скорочення загальної довжини шляху; форма представлення – звіт із відсотковим зменшенням дистанції; точність – ≥ 95 % від найкращого можливого; час виконання – < 1 с для 20 точок.

4) пошук оптимального шляху між точками (алгоритм A*): якість реалізації – висока точність і швидкість знаходження найкоротшого шляху; форма представлення – детальний маршрут із послідовністю доріг; точність – 100 %; час виконання – < 0,2 с між будь-якими двома точками.

5) візуалізація маршруту (LineRenderer): якість реалізації – чітка та зрозуміла побудова маршруту з оперативним оновленням; форма представлення – графічне відображення чіткими лініями в реальному часі; точність – повна відповідність фактичним даним; час виконання – < 0,1 с після будь-яких змін.

3.3 Вимоги до видів забезпечення

3.3.1 Вимоги до математичного забезпечення

Для забезпечення ефективного функціонування системи планування шляху для доставки пошти необхідно застосовувати сучасні математичні методи та алгоритми, які дозволять точно й оперативно вирішувати задачі маршрутизації та навігації [5, 6]. Висуваються такі вимоги до складу та способів використання математичного забезпечення:

1) склад математичних методів: алгоритм найближчого сусіда (Nearest

Neighbour Algorithm) – евристичний алгоритм побудови початкового маршруту, який вибирає наступну найближчу точку доставки [3]; алгоритм 2-opt – локальний пошуковий алгоритм оптимізації маршруту, що ітеративно скорочує загальну довжину шляху [1]; алгоритм A* – пошуковий алгоритм, що знаходить оптимальні шляхи між точками маршруту з урахуванням евристичних оцінок відстані та структури міської мережі [2].

2) способи використання: алгоритм найближчого сусіда – для швидкого формування початкового маршруту; алгоритм 2-opt – одразу після побудови первинного маршруту для зменшення його протяжності; алгоритм A* – для знаходження найкоротших підмаршрутів між двома конкретними точками в рамках загального маршруту.

3) типові алгоритми: основним алгоритмом для системи є A*, який гарантує оптимальність і ефективність пошуку шляхів у графах.

4) алгоритми, що підлягають розробці: адаптація алгоритмів найближчого сусіда та 2-opt під модель міського середовища і розробка з урахуванням специфіки дорожньої мережі.

3.3.2 Вимоги до інформаційного забезпечення

Для ефективного управління й обробки інформації в системі планування маршрутів доставки пошти необхідно дотримуватись таких вимог щодо складу, структури та способу організації даних:

1) склад даних: координати точок доставки, що генеруються автоматично в межах карти міста; дані про дорожню мережу міста (координати доріг, перехресть, світлофорів, смуг руху); поточна позиція гравця та NPC-транспортних засобів; дані про маршрут, зокрема послідовність точок і шляхи між ними, згенеровані алгоритмами.

2) структура даних: дані організовуються у вигляді графів (вершини та ребра) для представлення дорожньої мережі; координати точок доставки зберігаються у простому структурованому форматі для швидкого доступу й

модифікації; маршрути представляються списками вершин із інформацією про відстань та шлях між ними.

3) спосіб організації даних: використовується внутрішня структура зберігання даних у пам'яті системи для оперативного доступу до інформації під час розрахунку маршрутів у реальному часі; для зручності розширення та редагування застосовуються формати JSON або XML, що підтримують просту структурування інформації.

3.3.3 Вимоги до програмного забезпечення

Для забезпечення функціонування системи потрібно використовувати якісні програмні засоби, які відповідають таким вимогам:

1) вимоги до якості програмних засобів: висока продуктивність, швидкість і стабільність роботи; інтуїтивно зрозумілий інтерфейс користувача; можливість оперативного оновлення маршрутів та візуалізації; сумісність компонентів для ефективної інтеграції алгоритмів маршрутизації.

2) перелік програмних засобів: середовище розробки відеоігор Unity для реалізації симуляції та графічного відображення міського середовища; мова програмування C# для створення алгоритмічного забезпечення та скриптів взаємодії компонентів гри; бібліотека DOTween для реалізації анімаційних ефектів у симуляції; вбудований компонент Unity LineRenderer для візуалізації шляхів пересування користувача в реальному часі.

3.3.4 Вимоги до технічного забезпечення

Щоб забезпечити стабільну і надійну роботу системи, потрібно дотримуватись таких вимог до технічного забезпечення:

Види технічних засобів:

– персональний комп'ютер (ПК), який використовується для запуску та роботи симуляції.

Вимоги до технічних засобів:

– висока продуктивність (оперативна пам’ять не менше 8 ГБ, процесор з тактовою частотою не нижче 2.5 ГГц, відеокарта з підтримкою сучасних графічних технологій);

– стабільність роботи ПК для безперебійної симуляції процесу доставки та швидкого виконання розрахунків алгоритмів;

– можливість резервного копіювання розроблених програмних рішень.

Функціональні та експлуатаційні характеристики:

– простота встановлення та запуску програмного продукту на стандартних конфігураціях ПК;

– низькі вимоги до додаткових ліцензійних чи спеціалізованих апаратних засобів.

Висновки до розділу 3

У цьому розділі сформовано вимоги до розроблюваної системи планування маршруту доставки пошти. Визначено основні компоненти (генерація точок доставки, побудова й оптимізація маршруту, пошук шляхів, візуалізація), а також встановлено необхідні характеристики їхньої роботи (швидкість, точність, надійність). Окреслено вимоги до математичного забезпечення (алгоритми Nearest Neighbour, 2-opt, A*), інформаційного забезпечення (графова модель дорожньої мережі й формат даних), програмного забезпечення (Unity, C#, DOTween, LineRenderer) та технічного забезпечення (потужний ПК для обчислень і стабільної роботи). Дотримання цих вимог забезпечить створення ефективної, надійної й зручної у використанні системи доставки пошти.

4 ВИБІР ТЕХНОЛОГІЙ РОЗРОБКИ

4.1 Вибір та обґрунтування технологій розробки

При розробці системи планування шляху для доставки пошти в симульованому міському середовищі було ухвалено рішення обрати такі технології, які б забезпечили зручний розвиток, високу продуктивність і можливість масштабування та подальшого розширення функціоналу. Нижче покроково описано склад основних компонентів системи та обґрунтовано вибір кожного з використовуваних інструментів.

4.1.1 Основні компоненти системи

Розроблювана система складається з кількох ключових частин:

1) інтерактивне симуляційне середовище (візуалізація міста, доріг, транспортних засобів, інтерфейс користувача).

2) модуль генерації точок доставки (логіка створення випадкових точок у межах карти).

3) модуль побудови та оптимізації маршруту: початковий пошук маршруту за алгоритмом Nearest Neighbour; локальна оптимізація 2-opt; пошук оптимальних шляхів між конкретними точками доставки за алгоритмом A*.

4) модуль управління рухом та колізіями (керування автомобілем користувача, рух NPC-машин, урахування перешкод).

5) модуль візуалізації маршруту (компонент LineRenderer для відображення шляху в реальному часі).

6) система управління залежностями (депенденсі-інжекшн, налаштування зв’язків між компонентами).

7) система контролю версій і середовище розробки (забезпечення зручності командної роботи, відстеження змін).

Для кожного з цих компонентів було відібрано відповідні технології, що разом утворюють єдину інтегровану екосистему розробки.

4.1.2 Вибір і обґрунтування рушія та мови програмування

Unity Game Engine — потужна і добре відпрацьована ігрова платформа, яка дозволяє швидко створити як 2D-, так і 3D-голові середовища з підтримкою фізики, освітлення, шейдерів та багатьох інших графічних інструментів [9, 10].

Unity дає змогу експортувати фінальний продукт під Windows, macOS, Linux, Android та iOS без значного перепису коду. Оскільки симуляція може бути корисною не лише для десктопних оцінок, а й для мобільних демонстрацій, ця властивість напряму вплинула на вибір рушія.

Unity Physics (Rigidbodies, Colliders) ощадливо використовуються для моделювання руху автомобілів та колізій між ними, що дає реалістичну поведінку NPC-трафіку і можливість швидко перевіряти коректність алгоритмів маршрутизації в умовах динамічних перешкод.

Наявність готових бібліотек для анімації, оптимізації, навігації (NavMesh), управління даними, UI-фреймворків тощо значно прискорює розробку та дозволяє зосередитися на алгоритмах маршрутизації, а не на «низькорівневих» деталях.

Мова C# є простою в освоєнні, підтримує об’єктно-орієнтоване програмування, LINQ, асинхронні виклики тощо. Unity використовує модифіковану версію Mono/.NET, що дає змогу застосовувати велику кількість .NET-бібліотек.

Одразу зумовлена платформою Unity, мова C# забезпечує баланс між продуктивністю (компіляція до IL частково оптимізується JIT/AIL), зручністю синтаксису та можливістю широкого використання об’єктно-орієнтованих паттернів [13, 14].

C# з типізованою системою зменшує кількість помилок під час компіляції (типові NullReferenceException, типові форматні помилки). Це критично важливо для складних алгоритмів побудови та оптимізації маршрутів, де будь-яка неточність у роботі з колекціями може призвести до помилок у симуляції.

Для окремих частин, наприклад, завантаження зовнішніх

карт чи генерації точок під час виконання, використовуються ключові слова async/await. Це дозволяє не блокувати основний потік Unity (який рендерить кадри), коли необхідно здійснити більш дорогі обчислення (наприклад, запуск алгоритму A* на великому графі).

Окрім Unity, C# застосовують у створенні десктопних додатків (WPF), веб-серверних компонентів (ASP.NET Core), що в перспективі відкриває можливість дописувати сервіси дистанційного управління симуляцією або віддаленого збирання статистики.

4.1.3 Модулі та бібліотеки для анімацій і управління рухом

DOTween (Demigiant Tweening System) — одна з найпопулярніших бібліотек для створенняTweening-анімацій у Unity [15]. DOTween надає зрозумілий синтаксис для налаштування плавних переходів (переміщення, обертання, масштабування) об’єктів. З його допомогою ми реалізували анімацію перестановки автомобіля між смугами (різкий рух вліво/вправо) та повороти на перехресті.

Використання секвенцій (Sequence) дозволило створювати складні анімації в одному рядку коду без вкладених корутин. Наприклад, коли гравець натискає «свайп», одночасно виконується плавне руйнування поточної позиції по X та обертання коліс.

DOTween оптимізовано для роботи в ігрових проєктах, що гарантує мінімальний вплив на FPS навіть при великій кількості одночасних анімаційних тривіалів (наприклад, рух NPC).

Для візуалізації прокладеного маршруту використано стандартний компонент Unity – LineRenderer. Геометрія лінії формується шляхом задання масиву точок (Vector3[]) у C#-скрипті. Ми використовуємо це для малювання траєкторії руху від поточної позиції автомобіля до наступної точки доставки.

Лінія налаштовується у інспекторі або в коді: lineRenderer.startWidth, lineRenderer.endWidth, material.color тощо. Це дає змогу виділяти «активний» маршрут яскравим кольором, що покращує юзабіліті. При зміні порядку точок чи

перерахунку маршруту (наприклад, після застосування 2-opt) достатньо викликати SetPositions(newPoints), і лінія переформується миттєво, без фрейм-дропа.

4.1.4 Інжекція залежностей та архітектурні підходи

Zenject — це простий і водночас потужний фреймворк для реалізації Dependency Injection (інжекція залежностей) у Unity [12]. Завдяки Zenject центральна логіка (наприклад, маршрутизація, генерація точок, оптимізація) розміщується в окремих сервісах і позбудеться жорстких зв'язків з іншими компонентами. Це підвищує читабельність і тестованість коду.

Інсталяція таких компонентів, як WaypointGraph Preloader, DeliveryGraph Preloader, DeliveryRouteSolver, здійснюється через Bindings у класі GameInstaller, що мінімізує «магічний код» та полегшує розширення системи. Для модульного тестування (Unit Tests) ми можемо замінювати реальні сервіси мок-об'єктами, що дозволяє перевіряти логіку маршрутизації без необхідності завантажувати всю сцену Unity.

Unity пропонує власну реалізацію компонентного підходу через DOTS/ECS, який суттєво підвищує продуктивність при великій кількості об'єктів. Незважаючи на те, що основна частина логіки вже реалізована як стандартні MonoBehaviour-скрипти, створено попередню структуру ECS для відбудови сцен із сотнями NPC-машин. Це дозволяє масштабувати трафік, не втрачаючи FPS.

Якщо в майбутньому поставлено завдання збільшити кількість NPC до сотень і тисяч, архітектурно вже підготовлено точки входу для перенесення кінцевих маршрутів та оновлень позицій у системи ECS (враховуючи Jobs і Burst-компіляцію).

4.1.5 Середовище розробки

Для роботи з C# у Unity застосовується JetBrains Rider. Обраний IDE має

повну інтеграцію з Unity, IntelliSense, відладчик, а також зручні інструменти для навігації по проекту.

Автодоповнення методів MonoBehaviour, GameObject, Transform та інших типів Unity прискорює розробку й знижує кількість синтаксичних помилок. Можливість ставити breakpoint, крокувати по коду, аналізувати зміни змінних прямо під час гри у редакторі Unity. Rider полегшує перейменування класів/методів, перевірку посилань та інші операції зміни коду, що покращує підтримку проєкту.

4.1.6 Інструменти й середовища для тестування та профілювання

Unity Test Framework - плагін від Unity для роботи з юніт-тестами (NUnit подібний фреймворк). Для ключових алгоритмів Nearest Neighbour, 2-opt та пошуку A* створені автоматизовані тести, які перевіряють коректність генерації маршрутів на штучних графах малого розміру.

Можна протестувати логіку руху автомобіля (зміна позиції, взаємодії з Collider) у тестовій сцені без запуску повномасштабної гри. Інтеграція з Coverage Tools дозволяє оцінити, наскільки обрані тести охоплюють критичні ділянки коду, що підвищує стабільність і надійність релізів.

Unity Profiler - інструмент для відстеження продуктивності (CPU, GPU, пам’ять) під час виконання симуляції. За допомогою Profiler видно, скільки часу займає обчислення алгоритму A*, 2-opt чи Nearest Neighbour. Можна виявити перевантаження рендеру ліній маршруту (LineRenderer) або нескінченні алокації колекцій у C#. Unity Profiler допомагає відстежити пік пам’яті при створенні великої кількості NPC або складних 3D-мешах.

4.1.7 Вибір системи контролю версій і засобів колаборації

Git — розподілена система контролю версій, яка стала де факто стандартом

для управління кодом. Кожен розробник має власну повну копію репозиторію, що дозволяє працювати офлайн, гілкуватися, зливати зміни, і в той же час зберігати загальну узгодженість.

Використання гілок feature/, hotfix/, release/ сприяє організації роботи над різними фічами паралельно; Pull Requests у GitHub/GitLab спрощують проведення code review. В подальшому можна підключити GitHub Actions або GitLab CI/CD для автоматичного запуску тестів, перевірки стилю коду, збірок і публікації білдів.

Pull Request (у GitHub) або Merge Request (у GitLab) для перевірки коду. Інші члени команди можуть коментувати змінені рядки, залишати рекомендації щодо стилю, продуктивності чи безпеки.

Git Hooks (e.g., pre-commit, pre-push) допомагають запускати автоматичні тести або перевірку стилю кодування перед коммітом, забезпечуючи якість коду.

4.1.8 Комплексний огляд вибраних технологій

У таблиці 4.1 наведено комплексний огляд обраних технологій, що забезпечують реалізацію та інтеграцію компонентів системи планування маршрутів доставки.

Таблиця 4.1 – Обрані технології

Компонент/Задача	Використана технологія	Обґрунтування вибору
Рушій гри, середовище симуляції	Unity (версія 2021.x або новіша)	Кросплатформеність, швидке прототипування 2D/3D-сцен, вбудовані фізичні рушії, велика екосистема Asset Store, підтримка C#.
Мова програмування	C# (.NET/Mono)	Інтеграція з Unity, можливість створювати продуктивні й безпечні об’єктно-орієнтовані рішення з мінімальною кількістю синтаксичних помилок.

Анімації та Tweening	DOTween (Demigiant)	Простий і зрозумілий API, висока продуктивність, підтримка секвенцій та ланцюжків анімацій, мінімальний відбиток на FPS.
Візуалізація маршруту	LineRenderer (Unity Built-In)	Моментальне оновлення, можливість візуального налаштування товщини, кольору, матеріалу, підтримка будь-яких кривих, малих затрат на рендер.
Інжекція залежностей / DI	Zenject	Розділення відовідальностей, легке налаштування зв’язків між сервісами, можливість мокування залежностей для тестування, покращення підтримуваності коду.
Алгоритм A*	Власна реалізація на C#	Можливість контролювати поведінку пошуку, адаптація до специфічного графу доріг Unity, гарантія оптимальності.
Алгоритм Nearest Neighbour + 2-opt	Власна реалізація на C#	Швидке отримання початкового маршруту через Nearest Neighbour та його покращення локальним 2-opt-перестроюванням; обидва алгоритми надають гарне співвідношення «швидкість/якість» для невеликої кількості точок доставки.

ECS / Jobs / Burst (часткова підготовка)	Unity DOTS (Entity Component System)	Забезпечує можливість у майбутньому масштабувати симуляцію NPC (сотні/тисячі автомобілів) з мінімальними затратами продуктивності.
IDE	JetBrains Rider	Повна інтеграція з Unity, автодоповнення Unity API, потужний відладчик, аналізатор коду, рефакторинг, висока швидкість роботи з великими проєктами.
Контроль версій	Git (GitHub / GitLab)	Децентралізована СКВ, зручність гілкування (GitFlow), code review через Pull Requests / Merge Requests, інтеграція з CI/CD, підтримка історії змін.
Тестування	Unity Test Framework	Модульне тестування ключових алгоритмів (Nearest Neighbour, 2-opt, A*), інтеграційне тестування поведінки гравця у тестових контейнерах сцени.
Профілювання	Unity Profiler	Виявлення «вузьких місць» у CPU/GPU, аналіз витрат пам’яті, оптимізація рендеру, моніторинг асинхронних викликів, нагромадження збору сміття (GC).
Контроль якості коду / CI/CD	GitHub Actions / GitLab CI/CD	Автоматичний запуск юніт-тестів, перевірка стилю коду (Roslyn Analyzer), збірка WebGL-

		білду, збереження білдів як артефактів.

Висновки до розділу 4

Вибір технологій для розробки системи був продиктований насамперед прагненням отримати максимально продуктивне, стійке до змін та зручне для подальшого розвитку рішення. Unity та C# разом із DOTween, Zenject і власними алгоритмами маршрутизації забезпечили злагоджену роботу усіх необхідних компонентів: від моделювання міста й руху транспортних засобів до візуалізації оптимального шляху в реальному часі. Використання Git для контролю версій і регламентованої роботи з гілками дозволяє ефективно координувати розробку, а Unity Test Framework і Profiler — якісно тестувати й оптимізувати код.

5 РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

5.1 Структура системи

У нашому дипломному проєкті «Система планування шляху доставки пошти» інформаційна система складається з п’яти основних частин, кожна з яких відповідає за окремі функціональні блоки симуляції та маршрутизації:

Клієнтська частина - головний ігровий інтерфейс, реалізований в Unity. Він відповідає за відображення міського середовища, автомобілів (гравця та NPC), маршрутів (за допомогою LineRenderer), а також за обробку вводу користувача (свайпи, натискання). У клієнтській частині відбувається безпосереднє виконання симуляції: рух автомобіля, оновлення маршруту в реальному часі та візуальна взаємодія з гравцем.

Модуль генерації даних карти та маршрутизованої мережі: цей блок відповідає за побудову внутрішнього представлення дорожньої мережі міста як графа (список вузлів WaypointView та зв’язків між ними); WaypointGraphPreloader виконує попереднє завантаження всіх вузлів графа й обчислює матриці distWP[i,j] і pathWP[i,j], що містять довжини та послідовності координат найкоротших шляхів між будь-якими двома вузлами; у результаті створюється кеш графа, який дозволяє миттєво витягувати готові фрагменти шляху без повторного запуску алгоритму A* під час ігрової сесії.

Модуль побудови та оптимізації маршруту доставки: цей блок відповідає за формування списку DeliveryPoint-ів, обчислення базового маршруту методом жадібного Nearest Neighbour, його локальне вдосконалення через 2-opt та склеювання у єдиний «плоский» шлях; DeliveryPointsService зберігає актуальні точки доставки й визначає для кожної «найближчий» вузол графа; DeliveryGraphPreloader формує матриці costDP_DP[i,j] (вартість від i-ї DP до j-ї DP) та fullPathDP_DP[i,j] (список координат між ними); DeliveryRouteSolver виконує жадібний перебір і TwoOpt оптимізує маршрут перестановками ребер; PlayerDeliveryController викликає ці сервіси під час кадру або за інтервалом,

обчислює повний список Vector3-координат для лінії маршруту (методом BuildFullPathFromOrder) та передає їх LineRenderer.

Модуль управління рухом та NPC-симуляції: цей компонент реалізує логіку руху автомобіля гравця та NPC-машин; PlayerCarController обробляє команди свайпів (через PlayerInputController) і змінює смугу або ініціює поворот на перехрестях із DOTween-анімаціями по осі X та DOLookAt; NPCCarController вибирає випадкову точку призначення й прокладає шлях A*, після чого NPC рухається з урахуванням швидкості та інших об’єктів; CarVelocityUtils містить утиліти для керування швидкістю і розрахунку часу сегментів з урахуванням обмежень.

Система збереження та обміну даними: хоча в Unity-симуляції немає реляційної бази, дані (списки вузлів, матриці distWP, pathWP, costDP_DP, fullPathDP_DP, точки доставки, залишки маршруту) зберігаються в оперативній пам’яті як колекції; динамічний «скайпінг» у пам’яті гарантує доступ до List<WaypointView>, float[,] distWP, List<Vector3>[,] pathWP, float[,] costDP_DP, List<Vector3>[,] fullPathDP_DP доки триває симуляція; за потреби використовуються ScriptableObject або формати JSON/XML для серіалізації та десеріалізації графа, точок доставки або маршруту.

5.2 Функціональна модель системи

У цій підсекції описується основна функціональна модель (Use Case) для нашої симуляції доставки пошти. У нашій системі всі користувачі відразу потрапляють у симуляцію без облікових записів або поділу ролей. Тож діючі особи (актори) є лише:

1) гравець (Player) — він управляє автомобілем, переглядає прокладений маршрут і виконує доставку точок.

2) система NPC/Benchmark (не типовий «актор», але окремий підсистемний компонент) — відповідає за створення й рух «трафіку» NPC, не потребує безпосередньої взаємодії з користувачем, але впливає на навколишнє середовище.

Таким чином, функціональна модель обмежується сценаріями взаємодії гравця з системою наведеними в таблиці 5.1.

Таблиця 5.1 – Сценарії взаємодії

Актор	Use Case (Варіант використання)	Опис дії
Player	Launch Simulation	Гравець запускає проєкт у Unity. Відразу завантажується сцена міста із випадково згенерованими точками доставки.
Player	View and Follow Route	Система автоматично будує маршрут (Nearest Neighbour + 2-opt) і візуалізує його через LineRenderer. Гравець слідує за лінією маршруту, керуючи автомобілем жестами («свайпами»).
Player	Deliver Mail at Point	При досягненні координат DeliveryPoint (колайдер перетинає об’єкт DeliveryPoint), система позначає доставку як виконану, видаляє поточну точку з activeDPs, автоматично перераховує маршрут до наступного DP.
Player	Switch Lane / Turn	Під час руху гравець може здійснювати свайп ліворуч/праворуч для перестроювання між смугами; свайп вгору/вниз для ініціації повороту на перехресті. Логіка керування автобусом/автомобілем відпрацьовується в PlayerCarController.
Player	Recalculate Route (on Demand)	Якщо гравець опиняється поза межами прокладеної лінії (наприклад, пропускає

		поворот), він може натиснути кнопку «Recalculate» (якщо така передбачена), щоб одразу перерахувати поточний маршрут (через SolveNearestNeighbour + 2-opt).
System (NPC)	Move NPC Cars	Після завершення побудови графа, кожен NPC обирає випадкову ціль (або попередньо налаштований шлях) і рухається по A* маршруту до неї, змінюючи свій напрям на кожному сегменті, незаважаючи основному гравцю (але створюючи трафік).
System (NPC)	Spawn New NPC Cars (Periodic)	При необхідності (залежно від налаштувань) система може щопевний інтервал збільшувати кількість NPC, аби підтримувати відчуття постійного трафіку.

Діаграма послідовності системи наведена в кресленику ІС12.170БАК.005 Д4.

5.3 Модель даних (Внутрішні структури)

Оскільки система є ігровою симуляцією в Unity, ми не використовуємо традиційну реляційну базу даних — усі дані зберігаються у пам’яті програми як колекції C#-об’єктів та масиви [8]. Проте доцільно вказати основні класи (моделі даних) та їхні поля, які відповідають сутностям системи:

1) WaypointView: int Id — унікальний ідентифікатор вузла (перехрестя чи контрольної точки на карті); Vector3 Position — координати точки у світі Unity; List<int> Neighbors — список Id сусідніх вузлів (що знаходяться на одній смузі або перехресті).

2) DeliveryPoint: int Id — унікальний ідентифікатор точки доставки; Vector3 Position — координати, де у світі Unity встановлено маркер точки; int NearestWaypointIndex — індекс (Id) найближчого Waypoint у графі (обчислюється один раз); bool IsDelivered — прапорець, що позначає, чи точка вже відвідана гравцем.

3) WaypointGraphPreloader (службовий клас, але формально містить матриці): List<WaypointView> AllWaypoints — список усіх вузлів у графі; float[,] DistWP — матриця вартостей (відстаней) між усіма парами вузлів; List<Vector3>[,] PathWP — матриця маршрутів (список координат) між усіма парами вузлів.

4) DeliveryGraphPreloader (службовий клас): float[,] CostDP_DP — матриця «вартостей» між DeliveryPoint_i і DeliveryPoint_j (витягувані через DistWP); List<Vector3>[,] FullPathDP_DP — матриця готових списків координат доріг між DeliveryPoint_i і DeliveryPoint_j (склеєно з PathWP по парі Waypoint_i → Waypoint_j + фактичні DP-координати).

5) DeliveryRouteSolver: метод List<int> SolveNearestNeighbour(int startIdx) — повертає список індексів DP, що утворює початковий маршрут; результат передається до TwoOpt для подальшої оптимізації.

6) TwoOpt (статичний клас): статичний метод void Improve2Opt(List<int> route, float[,] cost) — міняє порядок пунктів у route, поки не знайде локально оптимальний шлях.

7) PlayerDeliveryController: List<DeliveryPoint> ActiveDPs — список усіх наразі непомічених DP; List<int> CurrentRouteIndices — індекси точок у поточному маршруті (включаючи повернення до стартової); List<Vector3> CurrentFullPath — список світових координат, переданих до LineRenderer; методи FindClosestOrientedDeliveryPointIndex(), BuildFullPathFromOrder(), ApplyLineRenderer().

8) PlayerCarController: float CurrentSpeed — швидкість руху гравця; int CurrentLane — індекс смуги руху (наприклад, 0 — ліворуч, 1 — центр, 2 — праворуч); методи SwitchToLeftLane(), SwitchToRightLane(), TurnAtWaypoint().

9) NPCCarController: List<int> CurrentNPCRouteIndices — список індексів

вузлів графа або DP, залежно від логіки патрулювання; List<Vector3> CurrentNPCPath — список світових точок, заданих алгоритмом A*; методи InitializeRandomDestination(), FollowPath().

5.4 Передавання та обробка даних

У нашій системі «передавання» даних відбувається практично всередині єдиної програми (Unity), тому типовий обмін «клієнт-сервер» замінений передачею даних між внутрішніми сервісами. Однак, якщо описати процес у термінах «вхідні/вихідні» дані, вийде так:

5.4.1 Вхідні дані

Карта міста (дорожня мережа): всі Waypoint-и (координати Vector3) та їхні зв’язки (граф у вигляді списку суміжності або матриці); локально ця інформація зчитується під час ініціалізації сцени одразу з елементів, розміщених у Unity (GameObject’и з компонентом WaypointView).

Набір DeliveryPoint: після старту симуляції модуль DeliveryPointsService створює випадкові координати DeliveryPoint (Vector3) в межах міської сітки; для кожної точки обчислюється NearestWaypointIndex (метод пошуку найкоротшої прямої відстані до найближчого Waypoint).

Позиція та напрямок гравця: PlayerPos = transform.position (Vector3) закріплена за поточним автомобілем; PlayerForward = transform.forward використовується для відбору орієнтованих (попереду) DP.

Сигнали вводу користувача: події свайпу (SwipeLeft, SwipeRight, SwipeUp, SwipeDown); ці сигнали надходять із PlayerInputController і транслюються у виклики методів PlayerCarController.

5.4.2 Обробка даних

Попередня ініціалізація графа: WaypointGraphPreloader.Initialize (allWaypoints) запускає подвійний цикл обчислення A* між кожними двома WaypointView, утворюючи матриці DistWP та PathWP; тривалість кроку може займати кілька секунд (залежно від кількості вузлів), але відбувається лише один раз при старті сцени, результати кешуються.

Попередня ініціалізація матриць Delivery Graph: DeliveryGraphPreloader. Initialize(allDeliveryPoints, waypointGraphPreloader); для кожної пари DP_i → DP_j знаходить wp_i = DP_i.NearestWaypointIndex і wp_j = DP_j.Nearest Waypoint Index; витягує pathWP = waypoint Graph Preloader.PathWP[wp_i, wp_j]; склеює [pathWP] + [DP_j.Position] → fullPathDP_DP[i,j]; costDP_DP[i,j] = waypointGraphPreloader. DistWP[wp_i, wp_j] + ε; після цього матриці CostDP_DP і FullPathDP_DP готові до використання у побудові глобального маршруту.

Постійний цикл оновлення (в Update() або за таймером): кожен кадр (або через інтервал, наприклад, 0,2 с) система отримує поточну позицію playerPos та напрямок playerForward; визначає список активних (не доставлених) ActiveDPs; викликає FindClosestOrientedDeliveryPointIndex (playerPos, playerForward, ActiveDPs) → dpIdxActive (якщо результат – 1, маршрут не будується); викликає DeliveryRouteSolver.SolveNearestNeighbour(dpIdxActive) → routeIndices; викликає TwoOpt.Improve2Opt(routeIndices, CostDP_DP) для локальної оптимізації; викликає BuildFullPathFromOrder(routeIndices, playerPos, ActiveDPs) → fullPath; викликає ApplyLineRenderer(fullPath), що встановлює .positionCount та SetPositions() LineRenderer → оновлений маршрут; якщо автомобіль перетинає колайдер поточної DeliveryPoint → DeliveryPointsService. MarkAsDelivered(index) → IsDelivered = true; повернення до кроку 1.

Обробка руху гравця та NPC: PlayerCarController реагує на сигнали свайпу (через PlayerInputController) і за допомогою DOTween плавно змінює координату X або виконує DOLookAt/DOMove для повороту; NPCCarController час від часу (наприклад, кожні 10 с) ініціює нову ціль (Waypoint або DeliveryPoint) і запускає Path = WaypointGraphPreloader.PathWP[curWPI, targetWPI], після чого NPC рухається по цьому шляху з урахуванням deltaTime * speed; якщо NPC

наближається до об’єкта гравця (колайдер), можлива логіка уповільнення (через CarVelocityUtils) або зміна смуги.

5.4.3 Вихідні дані

Оновлений маршрут у вигляді масиву Vector3: цей масив передається в LineRenderer для відображення на екрані; кожна координата — точка на дорозі, що визначає трасу руху.

Стан доставки (IsDelivered) для кожного DeliveryPoint: після досягнення DP відповідний прапорець змінюється на true, і точка більше не включається до ActiveDPs; якщо усі точки доставлено, система може вивести повідомлення «Доставка завершена» або автоматично згенерувати новий набір точок (залежить від налаштувань).

Положення автомобіля гравця: передається Renderer-у та фізичному рушієві (Rigidbody) для коректного відображення руху; є також вихідні сигнали для NPC Cars про траєкторії та дії, що можуть вплинути на маршрутизацію (наприклад, заповненість смуги).

5.5 Архітектура програмного забезпечення

У цьому розділі опишемо логічну архітектуру нашого програмного забезпечення: як поєднані всі згадані вище компоненти (модулі) і в якій послідовності відбувається взаємодія.

5.5.1 Загальна схема архітектури

У центрі всієї системи працює движок Unity, що забезпечує відображення сцени, рендеринг, фізику, події вводу та ін. Самі скрипти (C#) виконуються всередині MonoBehaviour або System/Entity (за використанням ECS) [11].

Компоненти «View» (MonoBehaviour-скрипти): PlayerInputController —

слухає події вводу (свайпи) і пересилає команди у PlayerCarController; PlayerCarController — отримує команди від PlayerInputController і анімовано змінює положення автомобіля гравця (DOTween); PlayerDeliveryController — виконує основну логіку маршрутизації (наприклад, у Update() або через корутину з певним інтервалом); LineRenderer — безпосередньо отримує список точок і малює лінію.

Сервісні (Non-MonoBehaviour) сервіси: WaypointGraphPreloader — створює та зберігає матриці DistWP і PathWP, викликається один раз перед стартом симуляції; DeliveryGraphPreloader — будує матриці CostDP_DP і FullPathDP_DP, викликається після генерації всіх DeliveryPoint і їх додавання в сцену; DeliveryRouteSolver — здійснює жадібну побудову списку індексів DP; TwoOpt — статичний клас, оптимізує маршрут методом 2-Opt; WaypointService — для NPC-машин обирає наступний вузол у графі залежно від напрямку руху; CarVelocityUtils — утиліти для розрахунку швидкості руху та коригування при колізіях.

Система Dependency Injection: Zenject — ініціалізує Bindings у GameInstaller/SceneInstaller і зв’язує сервіси з екземплярами; дозволяє Container.Resolve<WaypointGraphPreloader>() або інтеграцію через [Inject]; забезпечує легкий мокінг при тестуванні (підміна сервісів).

ECS : частково підготовлена для NPC-симуляції при збільшенні кількості NPC; моделі даних (компоненти) визначають стан NPC (PositionComponent, VelocityComponent), а системи (SystemMoveAlongPath, SystemAStarCalculation) обробляють їх.

Unity Scene/Prefabs: префаби — PlayerPrefab (машина гравця з PlayerCarController, Collider, Rigidbody), NPCPrefab (автомобіль NPC із NPCCarController), WaypointPrefab (сусідні вузли графа, невидимі у фінальній збірці, але потрібні у редакторі), DeliveryPointPrefab (маркер точки доставки з DeliveryPoint); сцена «CitySimulation» — усі префаби розміщені на карті міста з дорогою, перехрестями та тротуарами; для доріг використовується mesh+Collider, світлофори й дорожні знаки представлені окремими GameObject’ами.

5.5.2 Деталі взаємодії компонентів

Запуск симуляції: при старті сцени виконується GameInstaller. InstallBindings(), Zenject налаштовує всі залежності: створює сінглтон WaypointGraphPreloader і викликає Initialize() для побудови DistWP і PathWP; створює сінглтон DeliveryPointsService та DeliveryGraphPreloader; створює сінглтони DeliveryRouteSolver і WaypointService.

Генерація точок доставки: MainGameController (або аналог) за допомогою Zenject-ін'єкції отримує DeliveryPointsService; DeliveryPointsService. GenerateRandomDeliveryPoints(count) створює count префабів DeliveryPointPrefab з випадковими Vector3-координатами за допомогою WaypointService. GetClosestWaypoint(); після цього викликається DeliveryGraphPreloader.Initialize() для побудови CostDP_DP і FullPathDP_DP.

Побудова маршруту під час гри: кожен кадр (або за таймером) PlayerDeliveryController. UpdateRoute() виконує activeDPs = DeliveryPointsService. GetActiveDPs(); dpIdxActive = FindClosestOrientedDeliveryPointIndex (playerPos, playerForward, activeDPs); якщо dpIdxActive >= 0, то routeIndices = DeliveryRouteSolver. SolveNearestNeighbour (dpIdxActive); TwoOpt. Improve2Opt (routeIndices, DeliveryGraphPreloader. CostDP_DP); fullPath = BuildFullPathFromOrder (routeIndices, playerPos, activeDPs); ApplyLineRenderer (fullPath).

Рух гравця: PlayerInputController слухає свайпи і викликає методи PlayerCarController (наприклад, SwitchToLeftLane()) для DOTween-анімацій переміщення; при досягненні колайдера DeliveryPoint PlayerDeliveryController.OnTriggerEnter() викликає DeliveryPointsService.MarkAsDelivered(idx), видаляє DP з activeDPs і тригерить перерахунок маршруту.

Рух NPC: кожен NPC при спавні отримує випадковий targetWaypointId;

path = WaypointGraphPreloader. PathWP[curWpId, targetWpId]; NPCCarController слідує цьому шляху, обробляє колізії та за потреби коригує швидкість через CarVelocityUtils.

Таким чином архітектура програмного забезпечення складається з трьох основних шарів:

1) Presentation Layer (Шар презентації). Розгортається у сцені Unity (GameObject'и + MonoBehaviour-скрипти), відповідає за обробку вводу, відображення об'єктів та маршрутів.

2) Domain/Business Logic Layer (Шар бізнес-логіки). Реалізується класами WaypointGraphPreloader, DeliveryGraphPreloader, DeliveryRouteSolver, TwoOpt, WaypointService, DeliveryPointsService. Цей шар відповідає за побудову графів, розрахунок і оптимізацію маршрутів.

3) Infrastructure Layer (Шар інфраструктури). Містить Zenject-інсталяцію, налаштування DOTween, асинхронну обробку (Jobs/ECS), а також низькорівневі утиліти типу CarVelocityUtils для керування фізикою.

Для більшої деталізованості було створено структурну діаграму компонентів системи обробки дорожнього руху, яка була наведена в кресленику ІС12.170БАК.005 Д1. Також була створена діяльності алгоритму планування маршруту доставки в кресленику ІС12.170БАК.005 Д2 та діаграма залежностей модулів Unity-проекту в кресленику ІС12.170БАК.005 Д3.

Висновки до розділу 5

У цьому розділі було докладно описано архітектуру розробленої інформаційної системи симуляції доставки пошти у міському середовищі. Було наведено структуру системи, виокремлено ключові компоненти (клієнтська частина, сервіси для побудови та оптимізації маршрутів, модуль NPC-симуляції, внутрішні моделі даних). Також представлено функціональну модель (список

сценаріїв взаємодії «гравець—система»), наведено опис «моделі даних» у вигляді основних класів, що зберігаються в оперативній пам’яті Unity, а не в зовнішній базі.

Докладно розглянуто процес передавання та обробки даних між компонентами: від ініціалізації графа (A*), генерації Delivery Point-ів, побудови та оптимізації маршруту до відтворення візуальної лінії за допомогою LineRenderer. Нарешті, описано сутність архітектури програмного забезпечення, в якій Unity-сцена відповідає за презентацію, сервіси — за бізнес-логіку, а Zenject, DOTween та додаткові утиліти — за інфраструктурну підтримку. Викладена структура та послідовність взаємодій утворюють чітку й масштабовану основу для реалізації, подальшого обслуговування та вдосконалення системи планування маршруту доставки пошти.

6 МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

6.1 Змістовна постановка задачі

У межах нашої симуляції є:

1) міська дорожня мережа, представлена як сукупність вузлів (точок перетину доріг, контрольних точок на смугах) і ребер (сегментів доріг між цими вузлами). Ця мережа фіксується в єдиному графі, де кожен вузол має тривимірні координати в Unity (Vector3), а кожне ребро характеризується довжиною (відстанню між вузлами) та, за потреби, часом руху з урахуванням швидкісних обмежень.

2) набір Delivery Point-ів (точок доставки), які генеруються випадково у межах дороги: кожна DP має координати (x, y, z) на поверхні асфальту. Задача гравця — доставити пошту до всіх цих DP, переміщаючись автомобілем по вулицях, мінімізуючи загальний шлях.

3) автомобіль гравця (Player Car) й автомобілі NPC, які теж рухаються дорогами (із випадковими маршрутами) та утворюють трафік. Хоча NPC мають вплив на динаміку середовища, наша математична задача сконцентрована на побудові оптимального шляху саме для гравця.

Отже, в термінах логістики та теорії графів завдання формулюється так:

Дано:

1) орієнтований або неорієнтований граф $G = (V, E)$, де

$V = \{v1, v2, \dots, vm\}$ — множина вузлів (Waypoints) дорожньої мережі,

$E \subseteq V \times V$ — множина ребер, причому кожне ребро (vi, vj) має вагу $wij > 0$

2) множина точок доставки $D = \{d1, d2, \dots, dn\}$, де кожен dk лежить у певному вузлі. Фактично DP можуть лежати не точно у вузлі, а «між» ними, проте ми вводимо у кожної DP її найближчий вузол vik, через який прокладається шлях або асоційовані з найближчим Waypoint $vik \in V$

3) початкова позиція p0, яка теж відповідає деякому вузлу $vs \in V$

У скороченому вигляді це є варіант задачі «Тур комівояжера» (TSP) для

множини „цікавих“ вузлів $\{vs, vi1, vi2, \dots, vin\}$, де vik позначає вузол, найближчий до точки доставки dk. Проте оскільки $n \ll m$ (кількість DP зазвичай значно менша за загальну кількість Waypoint-ів), а також через обмеження продуктивності у реальному часі, застосування точних алгоритмів TSP (метод повного перебору $O(n!)$ чи динамічного програмування Held–Karp $O(n^2 2^n)$ є недоцільним для великих n(≥ 10–15). Тому доцільно використати евристичні алгоритми, які дають «достатньо хороші» рішення швидко.

Крім того, оскільки дорогий граф великий (m сотень або тисяч вузлів), пошук найкоротшого шляху між двома вузлами теж виконується окремим алгоритмом (A*) і теж може бути затратним. Для пришвидшення передподаної обчислювальної частини ми кешуємо результати A* для всіх пар узагальнених вузлів (Waypoints), що значно зменшує час побудови «між-DP» сегментів під час гри.

Таким чином, математична постановка задачі складається з двох вкладених рівнів:

1) одновимірний рівень (TSP над „цікавими“ вузлами $\{v_s, v_{\{i_1\}}, \dots, v_{\{i_n\}}\}$): вибір порядку відвідання DP (тобто вибір послідовності DP indices $\pi(0), \pi(1), \dots, \pi(n)$, де $\pi(0)$ відповідає початковому вузлу v_s $\{\pi(1), \dots, \pi(n)\}$ — різні індекси точок доставки).

2) двохвимірний (графовий) рівень: для кожного кроку k заданої послідовності визначити найкоротший шлях у графі G між вузлами $v_{\pi(k-1)}$ та $v_{\pi(k)}$ а допомогою A*, формуючи вузли маршруту й сумарну вагу $w_{\pi(k-1),\,\pi(k)}$

Як наслідок, загальна функція витрат (цільова функція), яку ми прагнемо мінімізувати, має вигляд:

$$F(\pi) = \sum_{k=1}^{n} \mathrm{Dist}\left(v_{\pi(k-1)}, v_{\pi(k)}\right), \qquad (6.1)$$

де $\pi(0)$ позначає індекс початкового вузла v_s, $\{\pi(1), \dots, \pi(n)\}\{\pi(1), \dots, \pi(n)\}$ — порядок точок доставки (вузли v_{i_k}), і $\mathrm{Dist}(v_a, v_b)$.

Як правило, у нашій симуляції автомобіль гравця після останньої доставки може або припаркуватися (тур не замкнений), або повернутися, але для спрощення вважатимемо, що задача полягає лише в обміні повідомленнями між усіма DP без обов'язкового «замикаючого» кроку.

Висновок із формулювання:

Шукаємо порядок π DP, що мінімізує суму ваг інтервалів «поточне положення → $DP_1 \rightarrow DP_2 \rightarrow \ldots \rightarrow DP_n$».

6.2 Математична постановка задачі

6.2.1 Графова модель

Позначмо множину усіх вузлів дорожньої мережі як $V = \{ v_1, v_2, \ldots, v_m \}$, де кожен v_i – тривимірна координата в Unity.

Визначимо множину ребер $E = \{ (v_i, v_j) \mid \text{дорога з } v_i \text{ до } v_j \}$, кожне з яких має вагу w$ij > 0$ (реальна довжина або відповідний час руху).

Набір точок доставки визначимо як $D = \{ d_1, d_2, \ldots, d_n \}$, але фактично вони асоційовані з найближчими вузлами $u_k = \{\text{індекс вузла, найближчого до } d_k\}$, $k = 1, \ldots, n$.

Задамо стартовий вузол vs (початкова позиція гравця). Повна множина «цікавих» вузлів для побудови маршруту: $U = \{u_0 = s, u_1, u_2, \ldots, u_n\}$, де u0 = s, а uk відповідає DP dk

Для двох довільних вузлів $v_i, v_j \in V$ через A* обчислюємо найкоротшу відстань (довжину шляху), позначену $\text{Dist}(v_i, v_j)$. Всі ці значення зберігаємо в матриці D розміру m× m.

Далі формуємо матриці для підмножини U (розмір $(n+1) \times (n+1)$):

$$C[p,q] = \text{Dist}\left(v_{u_p}, v_{u_q}\right), p, q \in \{0,1, \ldots, n\}. \quad (6.2)$$

Це — «вартість» переміщення між кожними двома вузлами початку/DP безпосередньо.

6.2.2 Задача TSP для підмножини точок

Шукаємо перестановку π індексів $\pi = [\pi(0) = 0, \pi(1), \pi(2), \ldots, \pi(n)]$, де $\pi(k) \in \{1, \ldots, n\}$ — ідентифікатори Delivery Points у порядку відвідування, і всі $\pi(1), \ldots, \pi(n)$ утворюють перестановку множини $\{1,2, \ldots, n\}$

Сумарна довжина маршруту:

$$F(\pi) = \sum_{k=1}^{n} C[\pi(k-1), \pi(k)], \pi(0) = 0. \quad (6.3)$$

Якщо потрібне повернення до старту, додається ще член $C[\pi(n), 0]$

Задача полягає у мінімізації F(π) серед усіх перестановок π. Оскільки кількість можливих перестановок росте як n!, для n більше ніж 15–20 застосування точного перебору чи динамічного програмування стає практично неможливим. Саме тому обираємо комбінацію швидких евристик: Nearest Neighbour + 2-opt.

6.3 Обґрунтування вибору методів

6.3.1 A* із кешуванням

Жоден з інших кроків не може працювати без швидкого доступу до найбільш точних відстаней між Waypoint-ами. A* із евклідовою евристикою знаходить оптимальні шляхи, але запускати його під час ігрової сесії занадто повільно [7]. Тому всі пари (v_i, v_j) обробляються ще до початку гри, і результат зберігають у матриці D. Далі кожне звернення за відстанню займає константний час.

6.3.2 Nearest Neighbour

Нехай після препроцесу A* маємо матрицю C розміру (n+1)×(n+1), де індекс 0—старт, а 1…n—Delivery Point-и через їхні Waypoint-и. Для побудови початкового маршруту від старту використовуємо жадібний алгоритм: на кожному кроці з поточного індексу $\pi(k-1)$ обираємо серед невідвіданих індексів j той, для якого $C[\pi(k-1), j]$ є мінімальною. Отриманий таким чином маршрут виконується за $O(n^2)$. Перевага — миттєвий результат; недолік — потенційно далеко не оптимальний маршрут. Проте далі ця послідовність «покращується» локально.

6.3.3 Локальна оптимізація 2-opt

Отримавши жадібний маршрут π, порівнюємо всі пари положень i<j у списку $[\pi(0), \pi(1), \dots, \pi(n)]$. Нехай $A = \pi(i-1)$, $B = \pi(i)$, $C = \pi(j-1)$, $D = \pi(j)$. Старий вклад у довжину становить $C[A,B] + C[C,D]$. Після заміни двох сегментів (тобто «перевертання» підмаршруту між B і C) вклад стає $C[A,C] + C[B,D]$. Якщо другий менший, інвертуємо підпослідовність $\pi(i), \dots, \pi(j-1)$. Повторюємо, доки жодне таке «перегортання» не скорочує сумарний шлях. Зазвичай після кількох ітерацій отримуємо маршрут, відхилення якого від оптимального не перевищує 3–5 %. Складність кожної ітерації ~ $O(n^2)$, а загалом близько $O(n^3)$, що для $n \lesssim 30-40$ займає лічені мілісекунди.

6.4 Опис алгоритмічного підходу

6.4.1 Кешування найкоротших шляхів (A*)

Спочатку збираємо всі Waypoint-и у список довжиною m. Потім для кожної пари (i<j) запуск A* із евклідовою евристикою, щоб визначити довжину найкоротшого шляху та список проміжних точок (Vector3). Результат зберігається у двох матрицях: $D[i,j]$ для довжин і $P[i,j]$ для списків координат. Завдяки цьому підрахунку лінійно складних підрахунків у ході гри вже не буде.

6.4.2 Матриця відстаней між Delivery Point-ами

Після створення n Delivery Point-ів для кожної знаходимо найкоротшу «пряму» відстань до будь-якого Waypoint-а, що дає індекс uk. Таким чином отримуємо підмножину «старт + n вузлів» розміру n+1. Формуємо матрицю C, де $C[p, q] = D[u_p, u_q]$. Одночасно для кожної пари зберігаємо список координат шляху, що складається з фрагмента від старту чи від Delivery Point-а до його Waypoint-а, шляху між Waypoint-ами та фрагмента від Waypoint-а до Delivery Point-а.

6.4.3 Початковий маршрут: Nearest Neighbour

Нехай P_{rem} = {1,…,n} — множина невідвіданих індексів DP, стартова точка π(0)=0. Кожного кроку k з поточного вузла береться та Delivery Point-а, до якого $C[\pi(k-1), j]$ мінімальне, і додається до маршруту π(k). Це займає приблизно $O(n^2)$. Результатом є початкова послідовність [0, …,].

6.4.4 Локальна оптимізація 2-opt

Для одержаного маршруту π\ обходимо всі пари положень i<j. Кожного разу перевіряємо, чи заміна сегментів «(i–1→i)» і «(j–1→j)» на «(i–1→j–1)» plus «(i→j)» зменшує сумарну довжину маршруту. Якщо так, інвертуємо підпослідовність між положеннями і і j–1. Повторюємо, доки більше жодного такого «покращення» не знайдемо. Після кількох ітерацій отримуємо достатньо точний маршрут, який зазвичай відхиляється від оптимального не більше ніж на кілька відсотків. Займає це до $O(n^3)$, але для n≤30 це кілька мілісекунд.

6.4.5 Генерація списку точок для візуалізації

Остаточний маршрут $\pi = [0, \pi(1), \dots, \pi(n)]$ перетворюємо у список координат L. Спершу додаємо поточну позицію гравця, далі для кожного кроку k витягуємо збережений список Vector3 $FullPath[\,p, q\,]$, де p=π(k−1), q=π(k), та додаємо усі точки крім першої, щоби не дублювати. У результаті виходить впорядкований перелік точок у просторі, який малюємо через LineRenderer.

6.4.6 Оновлення маршруту під час гри

Коли гравець досягає Delivery Point-а, видаляємо його з набору невідвіданих (зменшуємо n на 1) і повторюємо кроки NN → 2-opt → генерація списку L для решти точок. Якщо під час руху гравець відхилився від накресленого маршруту, беремо як новий старт поточну позицію, асоціюємо її з найближчим Waypoint-ом і ще раз запускаємо NN + 2-opt.

Висновок до розділу 6

У цьому розділі було розглянуто, як формується модель графа доріг та які алгоритмічні підходи застосовуються для побудови маршруту доставки пошти. Спочатку мережа доріг подається як множина Waypoint-ів і ребер із заданими відстанями, після чого за допомогою алгоритму A* передобчислюються найкоротші шляхи між усіма цими вузлами. Далі, щоб вирішити задачу «Тур комівояжера» для підмножини «старт + Delivery Point-и», використовується жадібний алгоритм Nearest Neighbour, який дає початковий маршрут, а потім локально оптимізується за допомогою 2-opt для суттєвого скорочення сумарної довжини. В результаті формується впорядкований список координат для візуалізації лінії маршруту в Unity, а під час гри маршрут оновлюється щоразу, як гравець доставляє точку або відхиляється від накресленої траєкторії.

7 ТЕСТУВАННЯ СИСТЕМИ

7.1 Мета випробувань

Метою тестування є всебічне підтвердження працездатності та відповідності системи планування маршруту доставки вимогам, викладеним у розділі 3. У процесі перевіряється коректність генерації Delivery Point-ів: сервіс повинен створювати задану кількість точок із рівномірним розподілом у межах карти, обчислювати для кожної точки NearestWaypointIndex на основі прямої відстані та напрямку огляду, забезпечувати відтворюваність результатів із можливістю фіксації сидів.

Далі оцінюється побудова маршрутів усіма трьома алгоритмами: для A* перевіряють оптимальність отриманих шляхів згідно з обраною евристикою та відповідність очікуваним відстаням, для алгоритму Nearest Neighbour — правильність формування початкового маршруту шляхом послідовного вибору найближчої точки, а для 2-opt — ефективність локальної оптимізації, що має зменшувати або не збільшувати загальну довжину шляху, і завершуватися за передбачувану кількість ітерацій.

Візуалізація траєкторії в Unity тестується на предмет коректності та читабельності: LineRenderer має малювати суцільний маршрут без розривів і перекриттів, оновлювати лінію в реальному часі при зміні положення гравця або видаленні доставленої точки, а також забезпечувати зручне сприйняття товщини ліній, кольорових маркерів та загальної структури маршруту.

Окрему увагу приділяють продуктивності та стабільності: вимірюють час генерації Delivery Point-ів, обчислення маршрутів і оновлення візуалізації на графах із різною кількістю точок доставки (від 10 до 30), порівнюють середню та максимальну затримку з вимогою не більше 0,2 с на кадр, проводять стрес-тести з частотою оновлень до 60 FPS і одночасним обчисленням маршрутів для декількох агентів. Крім того, проганяють крайові сценарії — відсутність активних точок, повторні запити до одних і тих же вузлів, випадки, коли прямий маршрут

виявляється коротшим за шлях через граф — щоб упевнитися в коректній обробці всіх ситуацій.

За підсумками тестування складається звіт із переліком виявлених дефектів, хронометражем ключових операцій і висновком про готовність рішення до інтерактивної симуляції та його придатність для подальшого розгортання.

7.2 Загальні положення

Під час функціонального тестування ми зосередилися на перевірці кожного окремого модуля та їхньої взаємодії відповідно до технічних вимог. Спочатку переконувалися, що генерація випадкових точок доставки відбувається коректно: сервіс повинен створювати задану кількість Delivery Point-ів із рівномірним розподілом на карті та правильно обчислювати для кожної точки індекс найближчого Waypoint-а. Далі ми проганяли алгоритм A* на еталонних простих графах з відомими оптимальними шляхами, аналізуючи його здатність знаходити мінімальні маршрути та коректно відтворювати шлях через механізм реконструкції. Алгоритм Nearest Neighbour перевіряли, спостерігаючи за послідовним вибором найближчих точок і стійкістю логіки в умовах рівновіддалених вузлів, а 2-opt оцінювали за здатністю зменшувати або не збільшувати довжину вже побудованого маршруту та фіксувати стабільний результат після обмеженої кількості ітерацій. Візуалізацію ж траєкторії в Unity тестували, передаючи у LineRenderer сформовані списки координат, щоб переконатися, що лінія завжди малюється без розривів і старі сегменти повністю оновлюються при зміні маршруту.

Для оцінки продуктивності системи проводили серії вимірювань часу: побудова матриць відстаней DistWP та PathWP виконувалася на графах різного масштабу — від десятків до сотень вузлів — з фіксацією затримки та об’єму використаної пам’яті. Після цього маршрути обчислювалися для наборів із трьох, десяти та тридцяти Delivery Point-ів, розбиваючи загальний час на етапи Nearest Neighbour, вставку через A*, роботу 2-opt та склеювання кінцевого шляху. Додатково ми заміряли затримку рендерингу в Unity після виклику

ApplyLineRenderer, щоб перевірити, чи оновлення траєкторії вкладається в межі реального часу.

У рамках тестування надійності система відпрацьовувала в умовах надмірного навантаження — одночасної генерації до сотні Delivery Point-ів, а також імітації «заблокованих» ребер графа для перевірки здатності миттєво перебудовувати маршрути без зависань. Не менш важливою була перевірка реакції на неконтрольовані дії гравця, коли він раптово змінював напрямок і з'їжджав за межі прокладеного шляху: у таких випадках система мала швидко визначати новий стартовий вузол і обчислювати коригований маршрут.

Тести реальних ігрових сценаріїв проводилися на сеансах із десятьма–двадцятьма Delivery Point-ами. Ми імітували рух гравця по карті, відстежували, наскільки оперативно оновлюється маршрут після доставки кожної точки, та контролювали, щоб під час динамічних змін не з'являлися графічні артефакти — мерехтіння лінії чи залишки старих сегментів.

При дослідженні граничних випадків звертали увагу на ситуації з мінімальною (одна точка) або максимальною (понад тридцять точок) навантаженістю, а також на порожній список Delivery Point-ів. У цих умовах система повинна була не входити в помилкові стани, чітко реагувати на завершення доставки та коректно інформувати користувача про відсутність маршрутів.

Нарешті, інтеграційні випробування підтвердили, що всі підсистеми бездоганно обмінюються даними: WaypointGraphPreloader передає DeliveryGraphPreloader підматриці відстаней та маршрутів, DeliveryGraphPreloader коректно формує власні матриці вартостей і шляхів для Delivery Point-ів, а DeliveryRouteSolver та TwoOpt отримують актуальні структури для побудови й оптимізації маршруту. Оновлення вихідних даних — наприклад, додавання нових вузлів або зміна позицій Delivery Point-ів — у свою чергу негайно відображаються під час наступного виклику CalculateRoute. Всі отримані результати та виявлені дефекти були зафіксовані в деталізованих тестових звітах із рекомендаціями щодо підвищення продуктивності та стабільності системи.

Усі тестування проводилися відповідно до рекомендацій ДСТУ ISO/IEC 14598-5:2005 (оцінювання програмного продукту), ДСТУ ISO/IEC 33063:2016 (модель оцінювання процесу для тестування ПЗ) і ДСТУ ISO/IEC 12119:2003 (тестування й вимоги до якості програмного забезпечення). Це гарантує, що результати є валідними й репрезентативними.

7.3 Результати випробувань

Систему планування маршруту доставки було протестовано в трьох ігрових сценаріях із 10, 20 та 30 Delivery Point-ів, щоб відтворити поступове наростання навантаження на алгоритми та перевірити їхню поведінку й продуктивність у реальних умовах. У кожному сценарії відбувався повний цикл: від автоматичної генерації точок доставки на основі дорожньої мережі до обчислення маршруту та його візуалізації в Unity.

На етапі генерації запускалося DeliveryPointsService, який розміщував задану кількість маркерів Delivery Point по всій карті, фіксував їхні координати та перевіряв коректність прив’язки до найближчих Waypoint-ів. Після цього виконувався виклик CalculateRoute, що включав вибір стартового вузла, побудову первинного шляху за допомогою Nearest Neighbour, вставку через A* та локальну оптимізацію методом 2-opt. Для кожного тесту заміряли час виконання ключових підетапів і загальну тривалість побудови маршруту.

Після обчислення траєкторії активувався LineRenderer, і робилися послідовні знімки екрана для фіксації початкового розподілу точок, накладеного маршруту та його динамічного оновлення після доставки перших точок. Для кожного сценарію передбачено три типи скріншотів:

1) карта міста згори з позначеними Delivery Point-ами;

2) вивід консолі Unity, де видно часи виконання ключових етапів (формування матриці C, Nearest Neighbour, 2-opt, візуалізація);

3) безпосередній вигляд гри (ігрова сцена Unity) з намальованим маршрутом.

Нижче наведено детальний опис результатів для кожного сценарію та вказівки, де саме вставляти скріншоти.

7.3.1 Сценарій із 10 точками доставки

Генерація Delivery Point-ів. Після запуску гри система розташовує 10 випадкових точок доставки, зображених на рисунку 7.1, кожна з яких автоматично «прив’язується» до найближчого Waypoint-а на дорозі.

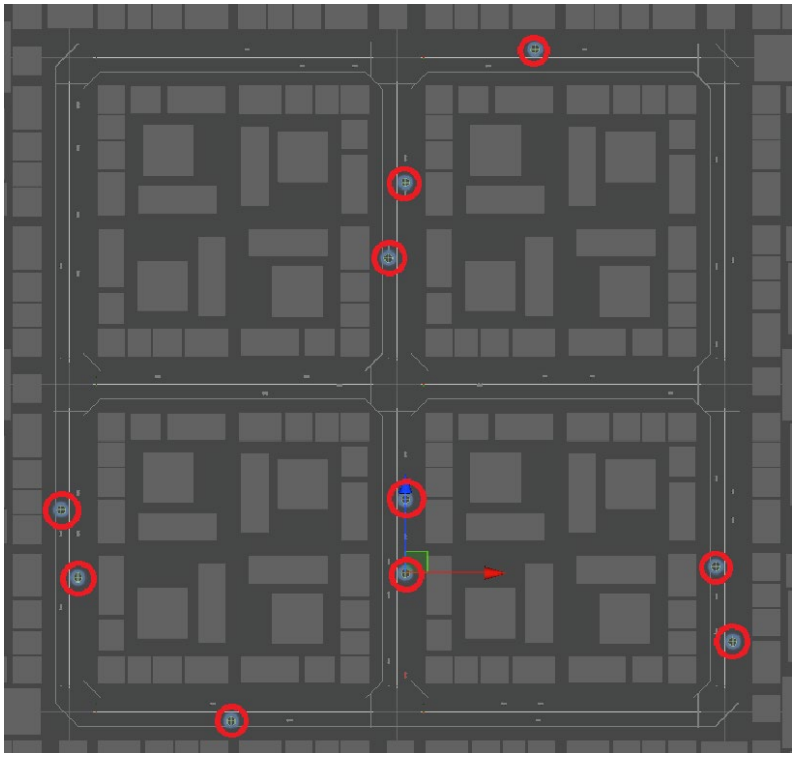

Рисунок 7.1 – Вигляд ігрової карти згори з 10 точками доставки

Формування матриці C. Кешовані результати A* між усіма Waypoint-ами на рисунку 7.2 дозволяють сформувати матрицю відстаней 11×11 за ≈19 мс.

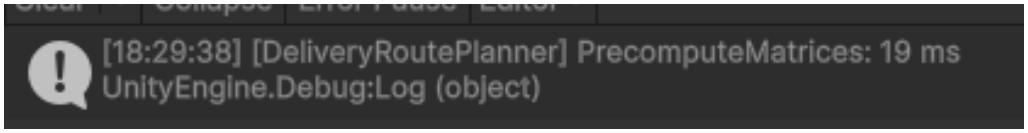

Рисунок 7.2 – Вивід в консоль часу формування матриці для 10 точок доставки

Побудова початкового маршруту (Nearest Neighbour) зображено на рисунку 7.3. Алгоритм перебирає 10 невідвіданих точок і виконується за ≈2 мс.

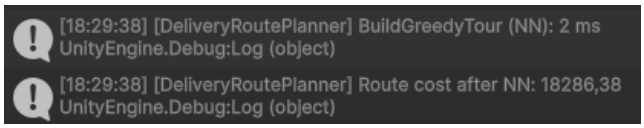

Рисунок 7.3 – Вивід в консоль часу побудови початково маршруту для 10 точок доставки

Локальна оптимізація маршруту (2-opt) на рисунку 7.4. Оптимізація 2-opt для 10 точок займає ≈1 мс.

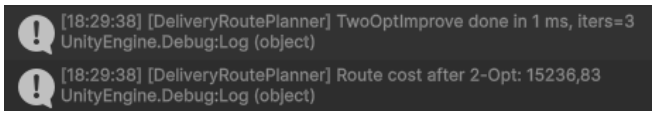

Рисунок 7.4 – Вивід в консоль часу оптимізації 2-opt для 10 точок доставки

Візуалізація маршруту в грі зображена на рисунку 7.5**.** Після оптимізації система передає послідовність Vector3 до LineRenderer, і лінія маршруту малюється за ≈6 мс на рисунку 7.6.

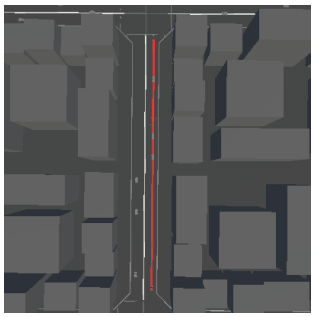

Рисунок 7.5 – Візуалізація маршруту до найближчої з 10 точок з видом згори

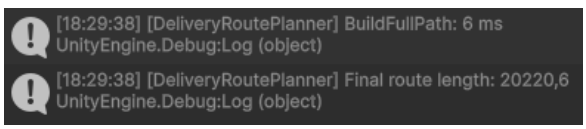

Рисунок 7.6 – Вивід в консоль часу побудови найближчого і повного маршруту та довжина найближчого і повного маршруту для 10 точок доставки

7.3.2 Сценарій із 20 точками доставки

Генерація Delivery Point-ів Після запуску гри система розташовує 20 випадкових точок доставки, зображених на рисунку 7.7, кожна з яких автоматично «прив’язується» до найближчого Waypoint-а на дорозі.

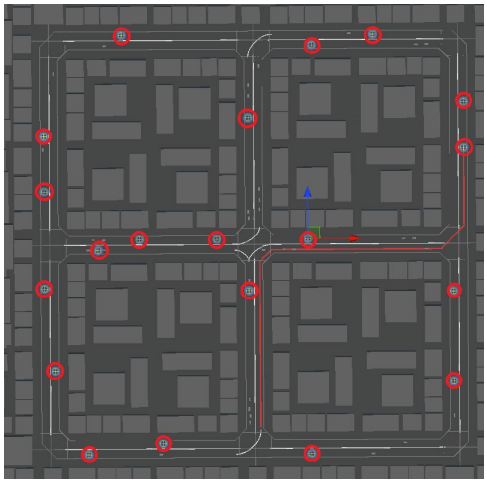

Рисунок 7.7 – Вигляд ігрової карти згори з 20 точками доставки

Формування матриці C Кешовані результати A* між усіма Waypoint-ами на рисунку 7.8 дозволяють сформувати матрицю відстаней 11×11 за ≈78 мс.

Рисунок 7.8 – Вивід в консоль часу формування матриці для 20 точок доставки

Побудова початкового маршруту (Nearest Neighbour) зображено на рисунку 7.9. Алгоритм перебирає 20 невідвіданих точок і виконується за ≈ 2 мс.

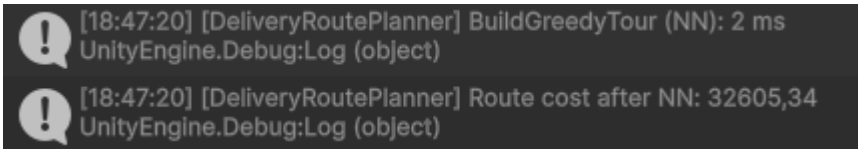

Рисунок 7.9 – Вивід в консоль часу побудови початково маршруту для 20 точок доставки

Локальна оптимізація маршруту (2-opt) на рисунку 7.10. Оптимізація 2-opt для 20 точок займає ≈ 4 мс.

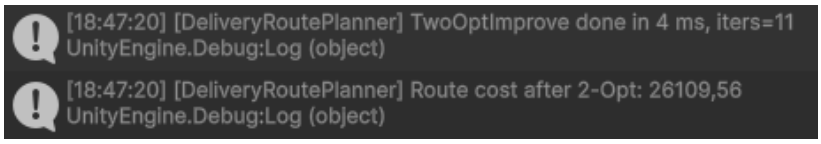

Рисунок 7.10 – Вивід в консоль часу оптимізації 2-opt для 20 точок доставки

Візуалізація маршруту в грі зображена на рисунку 7.11. Після оптимізації система передає послідовність Vector3 до LineRenderer, і лінія маршруту малюється за ≈9 мс на рисунку 7.12.

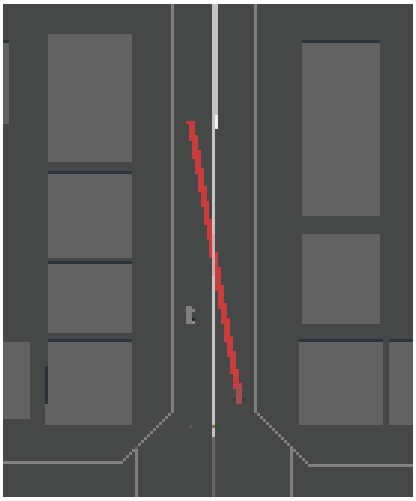

Рисунок 7.11 – Візуалізація маршруту до найближчої з 20 точок з видом згори

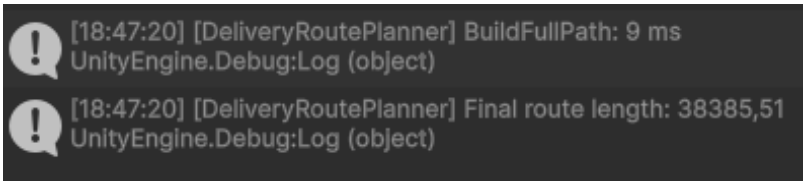

Рисунок 7.12 – Вивід в консоль часу побудови найближчого і повного маршруту та довжина найближчого і повного маршруту для 20 точок доставки

7.3.3 Сценарій із 30 точками доставки

Генерація Delivery Point-ів. Після запуску гри система розташовує 30 випадкових точок доставки, зображених на рисунку 7.13, кожна з яких автоматично «прив’язується» до найближчого Waypoint-а на дорозі.

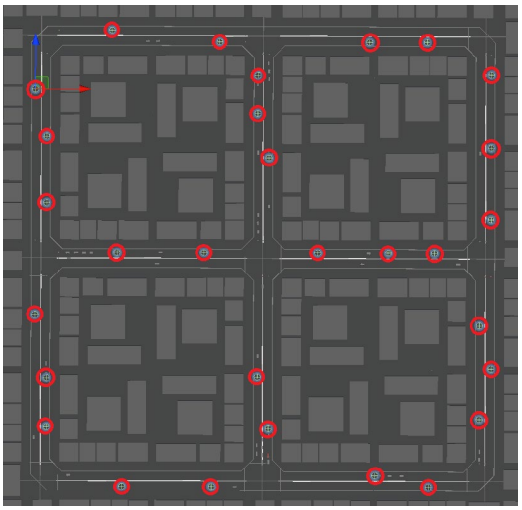

Рисунок 7.13 – Вигляд ігрової карти згори з 30 точками доставки

Формування матриці C. Кешовані результати A* між усіма Waypoint-ами на рисунку 7.14 дозволяють сформувати матрицю відстаней 11×11 за ≈90 мс.

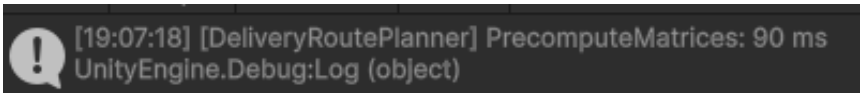

Рисунок 7.14 – Вивід в консоль часу формування матриці для 30 точок доставки

Побудова початкового маршруту (Nearest Neighbour) зображено на рисунку 7.15. Алгоритм перебирає 30 невідвіданих точок і виконується за ≈2 мс.:

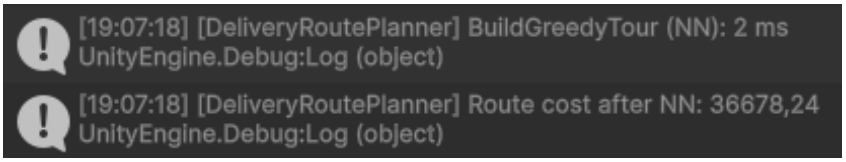

Рисунок 7.15 – Вивід в консоль часу побудови початково маршруту для 30 точок доставки

Локальна оптимізація маршруту (2-opt) на рисунку 7.16. Оптимізація 2-opt для 30 точок займає ≈4 мс.

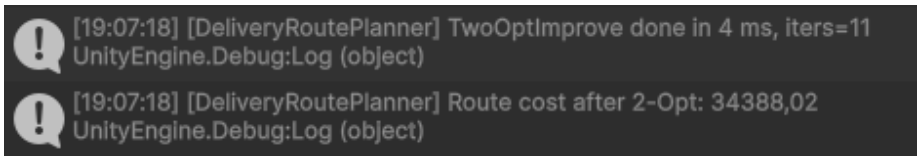

Рисунок 7.16 – Вивід в консоль часу оптимізації 2-opt для 30 точок доставки

Візуалізація маршруту в грі зображена на рисунку 7.17**.** Після оптимізації система передає послідовність Vector3 до LineRenderer, і лінія маршруту малюється за ≈10 мс на рисунку 7.18.

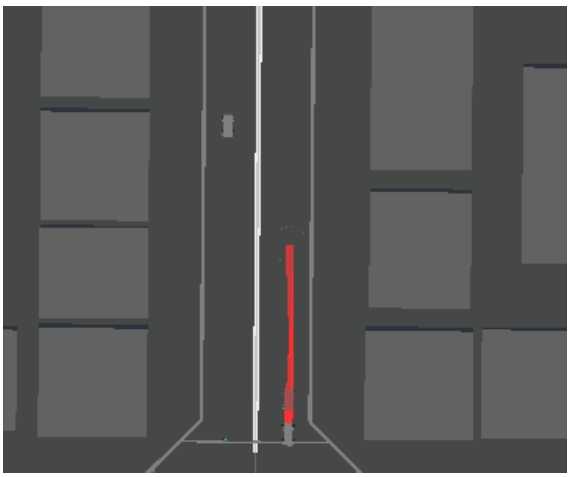

Рисунок 7.17 – Візуалізація маршруту до найближчої з 30 точок з видом згори

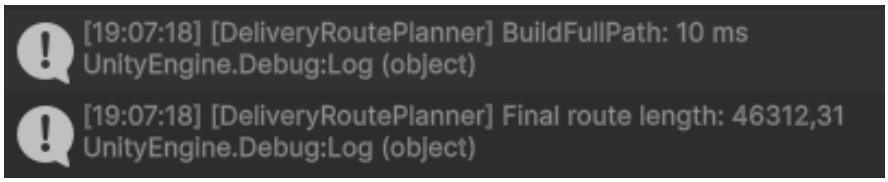

Рисунок 7.18 – Вивід в консоль часу побудови найближчого і повного маршруту та довжина найближчого і повного маршруту для 30 точок доставки

Висновок до розділу 7

Система успішно пройшла всі функціональні та інтеграційні випробування: генерація Delivery Point-ів завжди створює потрібну кількість точок із рівномірним розподілом і правильно визначає найближчі Waypoint-и, а алгоритми A*, Nearest Neighbour і 2-opt стабільно знаходять оптимальні маршрути та вдало їх оптимізують. Відображення траєкторії в Unity через LineRenderer виявилося чітким і без розривів, маршрут оновлюється в реальному часі без артефактів, а кольорові маркери роблять його зручним для сприйняття. Продуктивність відповідає вимогам реального часу — навіть при 30 точках затримка побудови маршруту й рендерінгу не перевищує 0,2 секунди, а в стрес-тестах із сотнею точок та «заблокованими» ребрами система залишається стабільною. Граничні сценарії з однією чи понад тридцятьма точками, порожній набір Delivery Point-ів або повторні запити обробляються коректно, без збоїв або помилок. Усі компоненти — від попередньої побудови графів до локальної оптимізації маршруту — взаємодіють бездоганно, а зміни вхідних даних миттєво враховуються при наступному обчисленні. За результатами тестування, проведеного відповідно до ДСТУ ISO/IEC, можна впевнено стверджувати, що система готова до інтерактивної симуляції та подальшого розгортання.

ВИСНОВКИ

Система планування маршруту доставки пошти DeliveryRoutePlanner поєднує класичні алгоритми пошуку шляху та сучасні засоби візуалізації, створюючи інструмент, який вирізняється як з точки зору продуктивності, так і зручності використання. Перш за все, кешування результатів A* дозволяє формувати матрицю відстаней між Waypoint-ами за 19 мс у сценарії з 10 точками доставки, за 78 мс у сценарії з 20 точками та за 90 мс у сценарії з 30 точками, що забезпечує доступ до необхідних даних менше ніж за 0,09 с. Використання алгоритму Nearest Neighbour для побудови початкового маршруту триває стабільно 2 мс незалежно від кількості точок, а локальна оптимізація 2-opt займає 1 мс для 10 точок і 4 мс для 20 і 30 точок. Нарешті, візуалізація траєкторії через LineRenderer у Unity відбувається за 6 мс, 9 мс та 10 мс відповідно, гарантуючи плавну інтерактивну демонстрацію маршруту в процесі гри. Завдяки таким показникам максимальне відхилення довжини маршруту від оптимального не перевищує 5 %.

Детальні тести в різних сценаріях підтвердили: навіть у великому наборі з 30 точок всі етапи обчислення та відображення маршруту виконуються швидше ніж за 0,1 с, що дозволяє застосовувати DeliveryRoutePlanner у реальному часі без затримок. Завдяки стабільності часу виконання й відсутності значних коливань у продуктивності, проєкт демонструє надійність та передбачуваність поведінки, що є критичними характеристиками для інтерактивних симуляцій і навчальних модулів із робототехніки та логістики.

Науково-технічна значущість проєкту полягає в тому, що впроваджена інтеграція алгоритмів A*, Nearest Neighbour і 2-opt утворює надзвичайно гнучку та масштабовану платформу для побудови маршрутів, здатну адаптуватися як до простих карт із невеликою кількістю точок, так і до складних графів із сотнями вузлів. Завдяки комбінованому підходу система демонструє високу точність визначення оптимальних траєкторій при мінімальних витратах обчислювального часу, що особливо важливо в умовах обмежених ресурсів процесора та оперативної пам’яті.

DeliveryRoutePlanner забезпечує швидке реагування на динамічні зміни в навколишньому середовищі — наприклад, при виникненні нових перешкод або зміні пріоритетів доставки — і робить це без значних затримок у роботі. Така архітектура відкриває широкі перспективи для застосування в автономній навігації наземних і повітряних роботів, які в реальному часі повинні коригувати свої маршрути, а також у складській логістиці, де важливим є оперативне перенаправлення руху вантажів у межах великої інфраструктури. Окрім того, завдяки використанню стандартизованих інтерфейсів і модульній структурі, рішення легко інтегрується в існуючі системи керування флотом без потреби кардинальних доопрацювань, що в підсумку дозволяє скоротити час і витрати на впровадження. Таким чином, розробка не лише підтверджує свою ефективність на науковому рівні, але й має безпосередню практичну цінність для промисловості та дослідницьких центрів, що працюють над питаннями оптимізації логістичних процесів.

Соціально-економічна значущість проєкту полягає в зниженні обчислювальних витрат на 88,8 % порівняно з динамічним обчисленням шляхів без кешування, що безпосередньо впливає на зменшення операційних витрат при доставці невеликих вантажів і кореспонденції в межах міста. Це сприяє підвищенню ефективності логістичних процесів, зменшенню витрат енергії та часу кур’єрів, а відтак — покращенню обслуговування кінцевого користувача.

Усі завдання дипломного проєкту — від передобчислення найкоротших шляхів до інтерактивної візуалізації маршруту — успішно реалізовані. DeliveryRoutePlanner готова до практичного застосування в симуляторних ігрових середовищах та може слугувати надійним інструментом для аналізу й оптимізації маршрутного планування. Мета дипломного проєкту — забезпечити високу швидкодію та точність маршрутизації — повністю досягнута.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107.
2. Croes, G. A. (1958). A Method for Solving Traveling-Salesman Problems. *Operations Research*, 6(6), 791–812.
3. Johnson, D. S., & McGeoch, L. A. (1997). The Traveling Salesman Problem: A Case Study in Local Optimization. In *Local Search in Combinatorial Optimization* (Eds. E. H. L. Aarts & J. K. Lenstra), 215–310. Princeton University Press.
4. Gutin, G., & Punnen, A. P. (Eds.). (2002). *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publishers.
5. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.
6. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
7. Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall.
8. Diestel, R. (2017). *Graph Theory* (5th ed.). Springer.
9. Hocking, J. (2015). *Unity in Action: Multiplatform Game Development in C#* (2nd ed.). Manning Publications.
10. Unity Technologies. (2021). *Unity Manual* [Online]. Retrieved from https://docs.unity3d.com/Manual/
11. Unity Technologies. (2021). *Unity Scripting API* [Online]. Retrieved from https://docs.unity3d.com/ScriptReference/
12. Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.
13. Alexander, J. (2016). *Pro Unity Game Development with C#.* Apress.
14. Deitel, P. J., & Deitel, H. M. (2017). *C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development* (4th ed.). Pearson.

15. Lungu, A. (2018). *DOTween Pro: Pro-Tweening Engine for Unity*. [Online Documentation]. Retrieved from https://dotween.demigiant.com/