МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ СИМУЛЯЦІЇ КЕРУВАННЯ ВІДДАЛЕНИМИ ПРИСТРОЯМИ»**

Виконав: студент 4 курсу, групи 6.1211-2пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
І.В. Костюк
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Кривохата А.Г.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри комп'ютерних наук, професор, д.т.н. Шило Г.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Костюку Іллі Владиславовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка інформаційної системи симуляції керування віддаленими пристроями

керівник роботи Кривохата Анастасія Григорівна, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз наявних рішень та програмних засобів розробки.
2. Аналіз вимог та проєктування.
3. Програмна реалізація та кросплатформове компілювання.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	15.01.2025	
2.	Збір вихідних даних.	10.02.2025	
3.	Обробка методичних та теоретичних	10.03.2025	
	джерел.		
4.	Розробка першого та другого розділу.	07.04.2025	
5.	Розробка третього розділу.	19.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	18.06.2025	

Студент _____ І.В. Костюк
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.Г. Кривохата
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка інформаційної системи симуляції керування віддаленими пристроями»: 54 с., 6 рис., 26 джерел.

ВІДДАЛЕНЕ КЕРУВАННЯ, КРОСПЛАТФОРМНІСТЬ, СИМУЛЯЦІЯ ПОЛЬОТІВ, FPV-ДРОН, UNITY.

Об’єкт дослідження – процес симуляції польоту FPV-квадрокоптера та розробка програмного забезпечення для тренування навичок керування.

Предмет дослідження – симуляція віртуальних середовищ, рушій Unity, фізична модель польоту FPV-дронів.

Мета роботи: розробка доступної інформаційної системи симуляції керування FPV-дроном, що забезпечує реалістичну симуляцію базової фізичної моделі польоту, оптимізована для роботи на персональних комп’ютерах з невисокими технічними характеристиками та на мобільних пристроях Android, з метою зниження бар’єрів для навчання пілотуванню.

Методи дослідження – аналіз існуючих рішень, об’єктно-орієнтоване проєктування для розробки архітектури системи, методи програмної інженерії для реалізації симулятора, експериментальне тестування розробленого програмного продукту на цільових платформах.

У кваліфікаційній роботі розроблено рішення проблеми обмеженої доступності FPV-симуляторів. На основі аналізу, визначених вимог та проєктних рішень в середовищі Unity розроблено програмний продукт, що симулює фізичну модель польоту FPV-дрона та забезпечує кросплатформну роботу для поширених операційних систем персональних комп’ютерів (Linux, Windows, macOS) та мобільних пристроїв (операційна система Android). Створений симулятор є функціональним та доступним інструментом для початкового навчання пілотуванню.

SUMMARY

Bachelor's qualifying paper "Development of an Information System for Simulating the Remote Devices Control": 54 pages, 6 figures, 26 references.

REMOTE CONTROL, CROSS-PLATFORM, FLIGHT SIMULATION, FPV-DRONE, UNITY.

The object of study is the process of FPV quadcopter flight simulation and the development of software for training piloting skills.

The subject of study is the simulation of virtual environments, the Unity engine, and the physical flight model of FPV drones.

The aim of the work is the development of an accessible information system for FPV drone control simulation, which provides a realistic simulation of basic flight physics model, optimized for operation on personal computers with low technical characteristics and on Android mobile devices, in order to lower the barriers to piloting training.

The method of research includes the analysis of existing solutions, object-oriented design for system architecture development, software engineering methods for simulator implementation, and experimental testing of the developed software product on target platforms.

In the qualifying paper, a solution to the problem of limited accessibility of FPV simulators was developed. Based on analysis, defined requirements, and design solutions within the Unity environment, a software product was developed that simulates the physical flight model of an FPV drone and ensures cross-platform operation for common personal computer operating systems (Linux, Windows, macOS) and mobile devices (Android operating system). The created simulator serves as a functional and accessible tool for initial piloting training.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 7
1 Аналіз наявних рішень та програмних засобів 8
1.1 Аналіз наявних рішень симуляції безпілотних польотів 8
1.2 Аналіз програмних засобів..... 12
1.3 Висновки першого розділу..... 15
2 Проєктування..... 17
2.1 Аналіз вимог до розроблюваного продукту 17
2.1.1 Аналіз функціональних вимог 17
2.1.2 Аналіз нефункціональних вимог 19
2.2 Концептуальне проєктування 21
2.3 Логічне проєктування 24
2.4 Фізичне проєктування 32
2.5 Висновки другого розділу 37
3 Програмна реалізація 38
3.1 Програмна реалізація процесу керування дроном..... 38
3.2 Кросплатформне компілювання застосунку 45
3.3 Висновки третього розділу 49
Висновки 51
Перелік посилань..... 52

ВСТУП

Стрімке поширення віддалено керованих апаратів у численних галузях, від розваг та спорту до професійної діяльності та спеціалізованих завдань, підкреслює їх унікальні можливості. Однією з найпоширеніших та перспективних сфер застосування технологій віддаленого керування є використання безпілотних літальних апаратів (БПЛА) з видом від першої особи – FPV-дронів. Водночас, ефективне та безпечне керування FPV-дронами вимагає значних навичок, розвиток яких у реальних умовах є витратним та ризикованим. У цьому контексті програмні симулятори відіграють критичну роль, дозволяючи пілотам відпрацьовувати маневри та освоювати динаміку польоту без ризику для обладнання чи оточуючих.

Аналіз ринку FPV-симуляторів показує, що провідні рішення часто є недоступними для широкого кола користувачів через високі вимоги до апаратного забезпечення, вартість ліцензій та іноді прив’язку до специфічного обладнання. Це створює суттєві перешкоди для початківців, освітніх установ та невеликих організацій. Таким чином, виникає актуальна проблема розробки доступного FPV-симулятора, що поєднує достатній реалізм фізичної моделі польоту з низькими системними вимогами.

Метою даної роботи є розробка інформаційної системи симуляції керування FPV-дроном, яка б забезпечувала реалістичну симуляцію базової фізичної моделі польоту та була оптимізована для роботи на персональних комп’ютерах з невисокими технічними характеристиками, а також на мобільних пристроях під управлінням Android.

Новизна та практична значущість роботи полягає у створенні FPV-симулятора, оптимізованого для роботи на менш потужному обладнанні. Це робить його доступним для широкого кола користувачів, що сприятиме ефективному початковому навчанню пілотуванню, використанню в освітніх цілях та підготовці ентузіастів без значних фінансових витрат.

1 АНАЛІЗ НАЯВНИХ РІШЕНЬ ТА ПРОГРАМНИХ ЗАСОБІВ

1.1 Аналіз наявних рішень симуляції безпілотних польотів

Зростаюча розповсюдженість безпілотних літальних апаратів у різних сферах застосування вимагає розробки та впровадження ефективних методологій симуляції. В даному контексті симуляційне програмне забезпечення традиційно є фундаментальною тренувальною платформою. Віртуальні середовища надають контрольований простір набуття навичок та їх вдосконалення, тим самим зменшуючи операційні ризики та фінансові витрати, пов’язані з фізичними льотними тренуваннями. Незважаючи на численні пропозиції рішень FPV-симуляції на комерційному ринку, детальний аналіз вказує на суттєві недоліки, зокрема щодо доступності, вартості та порогу необхідних специфікацій обладнання. Разом ці фактори перешкоджають впровадженню серед значної бази потенційних користувачів, включаючи індивідуальних користувачів, навчальні заклади та організації з обмеженими ресурсами [1, 2].

Liftoff: FPV Drone Racing – поширений симулятор, відомий своїми можливостями налаштування та різноманітністю контенту, створеного спільнотою на платформі Steam Workshop. Хоча ці функції є привабливими для користувачів-ентузіастів, їх головним обмеженням є потреба у значних обчислювальних ресурсах. Досягнення плавної та візуально узгодженої симуляції вимагає характеристик, зазвичай притаманних спеціалізованим ігровим пристроям. Користувачі, що намагаються експлуатувати Liftoff на стандартних ноутбуках або бюджетних настільних системах, часто стикаються з обмеженнями продуктивності, такими як мала частота кадрів або помітне погіршення користувацької взаємодії, що суттєво знижує тренувальну ефективність симулятора. Витрати на придбання програмного забезпечення значно зростають, якщо врахувати часто необхідну передумову інвестицій у

потужне, а отже дороге, комп’ютерне обладнання, що створює значну перешкоду впровадження [3, 4].

Velocidrone часто згадується серед спільноти змагальників з FPV-перегонів, головним чином завдяки фізичному двигуну, розробленому для високоточної симуляції. Хоча така точність може надавати переваги досвідченим пілотам, які вдосконалюють високоспеціалізовані маневри, вона безпосередньо сприяє обчислювальній складності симулятора. Розрахунок складних фізичних взаємодій, одночасно з візуалізацією достатньо детальних моделей навколишнього середовища для гоночної практики, вимагає значної обсислювальної потужності як від центрального, так і від графічного процесора. Отже, Velocidrone демонструє фундаментальне обмеження, аналогічне Liftoff: він, як правило, не працює на звичайних, неігрових апаратних конфігураціях. Ця сувора залежність від апаратного забезпечення систематично перешкоджає використанню новачками, студентами та організаціями, що не мають доступу до спеціалізованого обчислювального обладнання. Крім того, його зосередженість на змаганнях та відповідна фізична модель польоту можуть створити стримуючу траєкторію навчання для осіб, що прагнуть виключно здобути базові навички польоту FPV-дронів, а не займатися високорівневими змаганнями [5, 6].

Drone Racing League (DRL) Simulator, використовуючи усталену ідентичність бренду професійної гоночної ліги, має вдосконалене графічне відображення, що імітує естетику трансляцій DRL. Втім, такий рівень візуальної досконалості є обчислювально важким. Експлуатаційна ефективність симулятора нерозривно пов’язана з фундаментальними апаратними можливостями. Ефективне використання симулятора вимагає високопродуктивної обчислювальної конфігурації, достатньої для вимогливого процесу графічної візуалізації. Користувачі, оснащені стандартними обчислювальними системами, ймовірно, зіткнуться з проблемами продуктивності, яку суттєво погіршують досвід використання та навчальний потенціал. Його специфічна тематична орієнтація на гоночні траси

DRL, хоча й приваблива для шанувальників ліги, також обмежує його корисність для пілотів ширшої парадигми вільних польотів або різноманітних контекстах застосування, зменшуючи його універсальність як навчальної платформи загального призначення [7, 8].

Відображаючи прогрес технологічних можливостей, FPV SkyDive прагне до фотореалістичної візуалізації та складної симуляції фізичної моделі. Прагнучи максимальної точності, ця мета безпосередньо перетворюється на надзвичайно суворі вимоги до апаратного забезпечення. Симулятор вимагає високорівневих центрального та графічного процесорів, що міцно позіціонує його в преміальному сегменті ринку для ентузіастів. Обчислювальні витрати, пов’язані з передовими методами візуалізації та фізичним моделюванням, роблять його непридатним для розгортання в освітньому середовищі або для використання особами, які не мають значних інвестицій у сучасне ігрове обладнання. Прагнення до надзвичайного реалізму також може призвести до створення симуляції, надзвичайно складної для пілотів-початківців, потенційно перешкоджаючи, а не сприяючи набуттю початкових навичок без значної зовнішньої підтримки. Це є вершиною точності симуляції, але одночасно й вершиною недоступності для пересічного користувача [9, 10].

Альтернативну категорію обмежень ілюструє DJI Virtual Flight (що на сьогодні замінюється їх новішим продуктом SRIZFLY, який має ті ж обмеження), що є програмним забезпеченням, розробленим виробником фізичного обладнання. Програмне забезпечення цієї категорії часто розповсюджується безкоштовно та працює на персональних комп’ютерах, а іноді й на мобільних пристроях, але його основна корисність значно обмежена навмисною залежністю від власного обладнання виробника. Розроблене спеціально для роботи з контролерами DJI, що є інтегральною частиною їхньої лінійки FPV-дронів, воно демонструє несумісність з переважною більшістю стандартних FPV-радіопередавачів, поширених у аматорському та освітньому секторах (наприклад, Radiomaster, FrSky, TBS). Це обмеження перешкоджає використанню DJI Virtual Flight як загального навчального рішення для

користувачів поза межами обмеженої екосистеми продуктів DJI. Хоча він зручний для існуючих клієнтів DJI, він не відповідає належним чином вимогам до широко сумісного симулятора, що використовує поширене, економічно вигідне обладнання для керування FPV. Крім того, його фізичне моделювання загалом вважається порівняно рудиментарним порівняно зі спеціалізованими альтернативними платформами [11, 12].

Ця сукупність обмежень зберігається й для інших доступних симуляторів. Такі платформи, як Rotor Rush, ставлять пріоритетом гоночне використання і, аналогічно Velocidrone, вимагають значних апаратних можливостей. Нішеві або менш зрілі симулятори можуть мати нижчі вимоги до обладнання, але часто демонструють такі недоліки, як непослідовне моделювання фізичної поведінки, неадекватний набір функцій, обмежену доступність контенту або недостатню технічну підтримку. Таким чином, ринковий ландшафт ставить користувачів перед складним рішенням: або виділити значні фінансові ресурси на обладнання, здатне запускати високоточні симулятори, складність яких може бути зайвою для базових навчальних цілей, або терпіти суттєві обмеження в реалізмі та функціональності з більш доступними альтернативами, що потенційно знижує ефективність здобуття та розвитку навичок [13, 14].

Підсумовуючи, існуючий ринок симуляторів FPV, незважаючи на його очевидну різноманітність, демонструє значні спільні недоліки при оцінці на основі критеріїв широкої доступності та ефективності базового навчання. Найбільш потужні симуляційні платформи (включно з Liftoff, Velocidrone, DRL, FPV SkyDive тощо) вимагають високовартістних апаратних специфікацій, тим самим обмежуючи їх практичне застування для користувачів зі стандартними комп’ютерними системами, інституційних комп’ютерних лабораторій або організацій з обмеженими бюджетами. Пов’язані з цим витрати включають не лише ліцензійні збори за програмне забезпечення, але й значні, часто неозвучені, витрати, необхідні для придбання або оновлення обладнання. Неоднорідність сумісності обладнання, прикладом

якої є обмеження екосистеми DJI, ще більше перешкоджає впровадженню стандартизованих протоколів навчання з використанням загального обладнання [15, 16].

Отже, існує очевидна потреба в альтернативному рішенні для навчання FPV-пілотуванню. Таке рішення має бути розроблене з самого початку з урахуванням доступності як фундаментального критерію проєктування, спеціально оптимізоване для виконання на переважній більшості існуючих неспеціалізованих обчислювальних систем та ноутбуків, а також охоплювати мобільні обчислювальні платформи. Воно повинно забезпечувати механізм фізичного моделювання, який, хоча потенційно не прагне максимальної досяжної точності, що спостерігається в провідних гоночних симуляторах, має достатню фізичну точність для адекватного моделювання основної динаміки польоту квадрокоптера FPV, необхідної для набуття фундаментальних компетенцій керування, орієнтації та навичок маневрування. Надзвичайно важливо, щоб воно використовувало стандартизовані протоколи інтерфейсу для забезпечення сумісності із поширеними, доступними контролерами та уникнення залежності від специфічного обладнання. Досягаючи розумного балансу між основним реалізмом фізичної моделі, обчислювальною ефективністю та доступністю, такий симулятор міг би вирішити виявлений недолік, сприяючи ширшому доступу до навчання FPV-польотам та роблячи його дійсно доступним для студентів, бюджетних любителів, освітніх програм та малих організацій, яким досі перешкоджали суворі вимоги та значні пов’язані з ними витрати на сучасні провідні рішення.

1.2 Аналіз програмних засобів

Вибір середовища розробки програмного забезпечення для симулятора, зокрема того, який спрямований на відтворення складних фізичних процесів, таких як політ FPV-дрона, вимагає методичної оцінки багатьох факторів.

Вибір рушія впливає на ефективність розробки, можливості реалізації необхідних функцій, продуктивність кінцевого продукту, сумісність з платформами та вимоги до апаратного забезпечення. Після порівняльного аналізу на основі потреб проєкту, рушій Unity був визначений оптимальним рішенням для тренувального симулятора FPV-дронів, який надає пріоритет точності фізичної поведінки в межах заданих обчислювальних обмежень, у порівнянні з альтернативами, що включають Unreal Engine, Godot Engine та розробку власного рушія.

Фундаментальною вимогою до FPV-симулятора є реалізація достатньо точної фізичної моделі, що фіксує основну динаміку польоту квадрокоптера, включаючи тягу, опір, гіроскопічні ефекти, реакцію двигунів та обробку зіткнень. Обчислювальні витрати, пов’язані з просунутою симуляцією, були визнані несумісними з цілями проєкту.

Інтеграція Unity з NVIDIA PhysX забезпечує документоване, регульоване фізичне рішення з великою історією впровадження в різноманітних сферах застосування. Рушій пропонує інтуітивний доступ до фізичних параметрів через C#-скрипти, що дозволяє точно маніпулювати силами, крутними моментами, властивостями матеріалів та аеродинамічними наближеннями. Ця функціональність сприяє ітеративному вдосконаленню моделі польоту для досягнення належного балансу між реалізмом та продуктивністю [17, 18].

Хоча альтернативні рушії включають порівнянні фізичні системи – Unreal Engine з його фізичним двигуном Chaos та Godot Engine з його фізичним рішенням – усталена інтеграція Unity з PhysX надає суттєві переваги. Деталізована документація та супровідні ресурси, у тому числі специфічні для реалізації динаміки транспортних засобів та польотів, зменшують перешкоди для розробки, що особливо корисно для проєктів з обмеженими спеціалізованими знаннями команди у галузі низькорівневого програмування фізичних рушіїв. Система Chaos в Unreal Engine, хоча й технічно спроможна, вводить потенційно непотрібні складність та обчислювальні вимоги, зайві для

симулятора, який надає пріоритет функціональній механіці над візуальною точністю. Фізична реалізація Godot Engine, незважаючи на постійне вдосконалення, має менш широку історію впровадження для 3D-симуляції польоту порівняно з усталеною інтеграцією PhysX в Unity [19, 20].

Специфікація проєкту вимагала широкої сумісності з обладнанням, включаючи системи без спеціалізованих високопродуктивних графічних процесорів. Це вимагало рушія, що підтримує кросплатформне розгортання з надійними можливостями оптимізації. Unity забезпечує відповідну функціональність у цій галузі із зрозумілими процесами збірки для численних операційних систем. Крім того, Universal Render Pipeline (URP) пропонує детальний контроль над функціями візуалізації, дозволяючи вибірково вимикати обчислювально ресурсоємні ефекти для покращення продуктивності на обладнанні з нижчими специфікаціями. Це узгоджується зі стратегією проєкту щодо впровадження спрощеної графіки для забезпечення ширшої сумісності з обладнанням [21, 22].

У порівняльному аналізі Unreal Engine, хоча й пропонуючи варіанти масштабованості, принципово розроблений для пріоритетності візуальної точності, що потенційно вимагає суттєвих зусиль з оптимізації для досягнення прийнятної продуктивності на апаратному забезпеченні початкового рівня. Godot Engine демонструє чудові характеристики продуктивності на системах з нижчими специфікаціями, що робить його життєздатною альтернативою. Однак, при оцінці зрілості реалізації фізичного рушія Unity та наявної екосистеми ресурсів, що стосуються розробки симуляцій, Unity виявився більш доцільним рішенням для цього конкретного проєкту [23, 24].

Варіант розробки власного движка з використанням C++, графічних бібліотек та окремої реалізації пропрієтарного фізичного рушія був оцінений та визнаний непрактичним. Пропонуючи максимальний контроль над оптимізацією продуктивності та впровадженням функцій, вимоги до розробки систем відображення, керування об’єктами, обробки вхідних даних, інтеграції фізичної моделі, кросплатформної сумісності та інтерфейсу користувача

вимагатимуть великих ресурсів і спеціалізованих знань. Для проєкту, зосередженого на ефективній розробці функціонального симулятора, складність, пов’язана з розробкою власного рушія, є неоптимальним підходом порівняно з використанням усталених середовищ розробки [25, 26].

Вибір рушія Unity є стратегічним рішенням, заснованим на його відповідності вимогам проєкту. Поєднання гнучкої реалізації фізичної моделі, ефективних кросплатформних можливостей з інструментами оптимізації продуктивності та розвинутого середовища розробки забезпечує найефективніший шлях реалізації проєкту. Хоча альтернативні рушії пропонують певні переваги – Unreal Engine з чудовими візуальними можливостями та Godot з ефективними показниками продуктивності – Unity забезпечує найбільш стійкий баланс функцій для розробки симулятора з реалістичною фізичною поведінкою, оптимізованою графікою для обчислювальної доступності та ефективними процесами розробки, тим самим задовольняючи вимоги для розробки тренувального симулятора FPV-польотів.

1.3 Висновки першого розділу

Аналіз наявних симуляторів FPV-польотів та інструментів розробки виявив ключові аспекти, що виправдовують необхідність та дають напрям подальшої роботи.

Аналіз ринку виявив розрив між потребами користувачів та можливостями наявного програмного забезпечення. Провідні симулятори (Liftoff, Velocidrone, DRL Simulator, FPV SkyDive) забезпечують реалістичну фізичну поведінку та графіку, але вимагають надмірних обчислювальних ресурсів, що робить їх недоступними для пересічних користувачів, навчальних закладів та організацій з обмеженими бюджетами. Висока вартість ліцензій та вимоги до обладнання створюють бар’єри для впровадження. Більш доступні

або безкоштовні рішення від виробників дронів (DJI Virtual Flight/SRIZFLY) страждають від обмеженої апаратної сумісності. Менш поширені симулятори мають нижчі вимоги, але гіршу функціональність, реалістичність, якість або технічну підтримку. Існує потреба в симуляторі FPV, який забезпечує адекватну фізичну модель польоту для базового навчання, водночас оптимізований для стандартного обладнання, включаючи бюджетні ноутбуки, та підтримує стандартизовані контролери. Ключовий критерій – доступність, як фінансова, так і технічна.

Аналіз інструментів розробки визначив Unity як найбільш оптимальний рушій для реалізації поставлених завдань. Ключовими перевагами Unity стали збалансована інтеграція з фізичним двигуном NVIDIA PhysX, гнучкі інструменти оптимізації графіки (зокрема URP), що дозволяють досягти прийнятної продуктивності на стандартному обладнанні, та широкі можливості для кросплатформної розробки.

Висновки підтверджують доцільність розробки FPV-симулятора, орієнтованого на доступність, для ефективного базового навчання. Визначена ринкова ніша та вибір рушія Unity забезпечують основу для подальшого розвитку проєкту, спрямованого на користувачів, які прагнуть отримати навички керування FPV-дроном без значних фінансових та технічних бар’єрів.

2 ПРОЄКТУВАННЯ

Після визначення платформи розробки проєкт переходить до фази проєктування. Цей попередній етап активної розробки програмного забезпечення має вирішальне значення для визначення вимог та архітектури. Він забезпечує структурований підхід до перетворення концептуальних вимог у конкретні специфікації програмного забезпечення, тим самим зменшуючи ризики розробки та узгоджуючи кінцевий продукт із запланованими цілями.

2.1 Аналіз вимог до розроблюваного продукту

Ключовим компонентом етапу планування є аналіз вимог, класифікованих на функціональні та нефункціональні. Функціональні вимоги визначають конкретні моделі поведінки та можливості, які повинен демонструвати симулятор, детально описуючи, що робитиме система. І навпаки, нефункціональні вимоги формулюють якості та обмеження, за яких повинен працювати симулятор, такі як контрольні показники продуктивності, стандарти зручності використання та сумісність обладнання, визначаючи, наскільки добре система виконує свої функції.

2.1.1 Аналіз функціональних вимог

Аналіз функціональних вимог визначає конкретні операційні можливості, необхідні для навчального симулятора FPV. Ці вимоги описують основні характеристики системи, що охоплюють симуляцію польоту, взаємодію з користувачем, візуальний вивід та підтримку різних платформ, гарантуючи, що симулятор виконує своє цільове завдання:

- система повинна імітувати реалістичну динаміку польоту квадрокоптера, враховуючи вплив тяги, сили тяжіння, опору повітря та інерції;
- дрон повинен керуватися за допомогою отримуваних від користувача значень для тяги двигунів, тангажу, крену та рискання;
- система повинна підтримувати Acro-режим, де користувач безпосередньо контролює значення обертання дрона без «допомоги» PID-контролерів;
- система повинна виконувати розрахунки тяги окремих двигунів на основі загальної тяги та команд обертання;
- пропелери повинні візуально обертатися, причому їхня швидкість обертання повинна відповідати прикладеній тязі, включаючи мінімальну швидкість обертання та візуальне представлення інерції;
- дрон повинен падати, якщо тяги недостатньо для протидії силі тяжіння або якщо він нахилений під екстремальними кутами;
- система повинна імітувати зіткнення з землею, змушуючи дрон зупинятися або відбиватися належним чином при ударі;
- система повинна підтримувати керування дроном за допомогою стандартних ігрових контролерів (у стилі Xbox, PlayStation) та введення з клавіатури на платформах ПК;
- для мобільних платформ Android система повинна підтримувати керування дроном за допомогою віртуальних джойстиків на екрані, один з яких призначений для тяги/рискання, а інший для тангажу/кренування;
- введення тяги повинно бути накопичувальним, зберігаючи свій рівень, коли введення не здійснюється, та регулюватися на основі позитивного/негативного введення користувача, що точніше імітує поведінку реального контролера;
- система повинна відображати FPV (вид від першої особи) з перспективи камери, встановленої на дроні;

- система повинна реалізувати систему маршрутів на основі контрольних точок для структурованих навчальних вправ, що включає:
  - візуально чіткі контрольні точки з чітким зазначенням активної та наступної контрольних точок, тоді як інші контрольні точки на маршруті залишаються прихованими;
  - точне виявлення проходження дрона через активну контрольну точку та подальшу активацію наступної контрольної точки в послідовності;
  - повідомлення «МАРШРУТ ЗАВЕРШЕНО» відображається після успішної навігації через усі контрольні точки на визначеному маршруті;
- система повинна пропонувати функцію інтерфейсу «Перевернути дрон», яка з’являється, коли дрон падає та встановлюється в стабільному перевернутому положенні, дозволяючи користувачеві повернути дрон у вертикальне положення в його поточному місці;
- система повинна забезпечувати принаймні одне готове до використання 3D-середовище, розроблене для льотної практики;
- система повинна бути розробленою для запуску на платформах ПК (Windows, macOS та Linux);
- система повинна бути розробленою для запуску на мобільних пристроях Android.

Ці функціональні вимоги разом окреслюють основні характеристики, які дозволять симулятору служити ефективним та доступним навчальним інструментом для пілотування FPV-дронів.

2.1.2 Аналіз нефункціональних вимог

Нефункціональні вимоги визначають показники якості та експлуатаційні обмеження симулятора, вказуючи, наскільки добре система

має працювати, а не що вона робитиме. Ці вимоги є критично важливими для забезпечення позитивного досвіду користувача, стабільності системи та майбутньої придатності для технічного супроводження програмного забезпечення:

- симулятор повинен підтримувати мінімальну частоту кадрів 60 кадрів на секунду (FPS) під час типових сценаріїв польоту на цільовому ПК;
- симулятор повинен підтримувати мінімальну частоту кадрів 30 FPS під час типових сценаріїв польоту на цільових пристроях Android середнього класу;
- затримка введення, від фізичного введення користувача до реакції симульованого дрона, повинна бути мінімізована, щоб забезпечити відчуття чуйного керування;
- час завантаження для польотних середовищ повинен бути прийнятним, наприклад, менше 10 секунд для типових сцен;
- сенсорне керування на екрані для версії Android має бути адаптивним та розробленим таким чином, щоб мінімізувати перешкоди для огляду польоту;
- усі елементи інтерфейсу користувача мають бути чіткими, легко читабельними та відповідно масштабованими для читабельності на цільових ПК та мобільних платформах;
- огляд камери FPV має забезпечувати чітке поле зору, ефективно передаючи відчуття швидкості та просторової орієнтації;
- візуальні елементи, що представляють контрольні точки, повинні легко відрізнятися від навколишнього середовища, щоб забезпечити чіткість під час навігації по маршруту;
- модель польоту повинна забезпечувати переконливе та якісно точне наближення поведінки квадрокоптера FPV у реальному світі;
- реакція дрона на вхідні дані керування повинна бути реалістичною щодо прискорення, уповільнення та обертальної інерції;

- архітектура системи має бути розроблена таким чином, щоб розробники могли легко додавати нові моделі дронів з різними характеристиками польоту в майбутніх оновленнях;
- архітектура системи має підтримувати легку інтеграцію нових середовищ польоту або карт розробниками;
- система контрольних точок та маршрутів має бути розроблена таким чином, щоб розробники могли створювати та впроваджувати нові маршрути польоту або навчальні курси;
- структура програмного проєкту має бути логічно організована для сприяння розумінню, модифікації та майбутній розробці;
- збірка Android-додатку повинна підтримувати архітектури процесорів ARMv7 та ARM64 для забезпечення широкої сумісності з пристроями;
- програма не повинна аварійно завершувати роботу або стикатися з невиправними помилками під час нормального робочого використання;
- фізичні розрахунки, що лежать в основі симуляції польоту, повинні бути стабільними, послідовними та передбачуваними за всіх передбачуваних робочих умов.

Разом ці нефункціональні вимоги гарантують, що навчальний симулятор FPV не лише здатний виконувати свої передбачувані функції, але й забезпечує плавний, інтуїтивно зрозумілий, надійний та розширюваний досвід як для кінцевих користувачів, так і для розробників. Дотримання цих якостей є важливим для того, щоб симулятор був ефективним та добре сприйнятим навчальним інструментом.

2.2 Концептуальне проєктування

Діаграма прецедентів (Use Case) служить для визначення основних функціональних можливостей системи з точки зору її користувачів та

ілюструє, як користувачі взаємодіятимуть із симулятором для досягнення своїх цілей. Вона допомагає визначити обсяг системи та забезпечує глибоке розуміння її передбачуваної поведінки. Діаграма, показана на рисунку 2.1, зображує взаємодію між основним учасником, користувачем, та системою симулятора FPV-дрона.

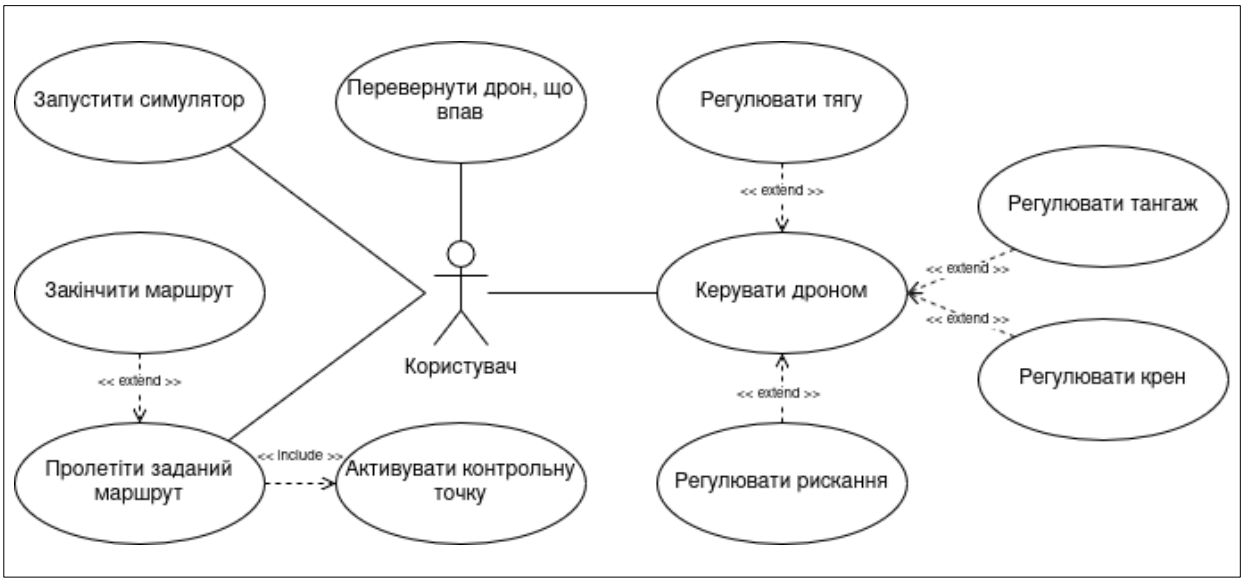


Рисунок 2.1 – Діаграма прецедентів

Діаграма ілюструє ключові функціональні можливості (варіанти використання), такі як керування дроном, навігацію маршрутом та відновлення положення дрону. Такі зв’язки, як “include” та “extend”, використовуються для відображення залежностей та додаткових розширень між варіантами використання.

Користувач є основним учасником, який взаємодіє із симулятором FPV-дрона. Користувач – це особа, яка запускатиме симулятор, керуватиме дроном, намагатиметься виконати маршрути польоту та використовуватиме інші інтерактивні функції системи.

Діаграма виділяє кілька ключових функціональних можливостей, доступних Користувачу.

Запустити симулятор. Це початкова взаємодія, під час якої Користувач запускає програму симуляції FPV-дрона. Вона являє собою точку входу в

систему. Користувач безпосередньо ініціює цей варіант використання, щоб розпочати свій досвід симуляції.

Керувати дроном. Це центральний та фундаментальний варіант використання, що представляє здатність Користувача керувати FPV-дроном у симульованому середовищі. Це включає пряме маніпулювання характеристиками польоту дрона на основі реалізованої реалістичної фізичної моделі. Користувач безпосередньо бере участь у цьому варіанті використання, надаючи дані через підтримувані пристрої (клавіатуру, геймпад або екранні джойстики для мобільних пристроїв). Цей варіант використання розширюється:

- регулювати тягу: представляє дію користувача щодо збільшення або зменшення миттєвої потужності двигунів дрона;
- регулювати тангаж: представляє дію користувача щодо нахилу дрона вперед або назад;
- регулювати крен: представляє дію користувача щодо нахилу дрона вліво або вправо;
- регулювати рискання: представляє дію користувача щодо обертання дрона по горизонталі;
- співвідношення “extend” тут означає, що ці конкретні варіанти керування є необов’язковими вхідними даними, які вносять свій вклад у загальний варіант використання «Керувати дроном» (користувач може не налаштовувати всі осі постійно, але саме ці дії є способом досягнення контролю).

Пролетіти заданий маршрут: цей варіант використання представляє собою конкретну ігрову мету, де Користувач намагається провести дрона через послідовність заздалегідь визначених контрольних точок. Користувач активно пілотує дрон з метою завершення маршруту:

- цей варіант використання включає “Активувати контрольну точку”: зв’язок “include” вказує на те, що успішне проходження контрольних точок є обов’язковою та невід’ємною частиною польоту за заданим

маршрутом; щоразу, коли дрон проходить через активну контрольну точку, виконується прецедент «Активувати контрольну точку»;

– цей варіант використання розширено за допомогою «Закінчити маршрут»: зв’язок “extend” тут означає, що успішне проходження всіх контрольних точок і, таким чином, завершення маршруту є можливим результатом або продовженням спроби польоту за маршрутом – це призведе до появи повідомлення інтерфейсу користувача «МАРШРУТ ЗАВЕРШЕНО».

Перевернути дрон, що впав. Цей варіант використання дозволяє Користувачеві відновити дрон після його приземлення догори дном. Користувач ініціює цю дію, натискаючи кнопку «Перевернути дрон», яка з’являється в інтерфейсі користувача, коли виконуються умови аварії: дрон має бути перевернутим і відносно стабільним на землі, щоб система могла запропонувати цей варіант відновлення.

Діаграма варіантів використання забезпечує чітку, високорівневу візуалізацію передбачуваних функцій симулятора FPV-дрона та того, як гравець взаємодіятиме з ними. Вона служить важливим інструментом на етапі концептуального проєктування, забезпечуючи спільне розуміння обсягу системи та основних функцій, перш ніж переходити до більш детальних етапів проєктування та впровадження. Ця діаграма відображає основні цілі користувача, від базового керування польотом до виконання структурованих маршрутів та обробки ігрових подій, таких як аварії.

2.3 Логічне проєктування

Діаграма класів ілюструє основні класи програмного забезпечення, їхні атрибути, операції та зв’язки між ними. Це забезпечує план внутрішньої архітектури програмного забезпечення, способу управління даними та того, як різні компоненти співпрацюють для забезпечення функціональності симулятора.

Діаграма, представлена на рисунку 2.2, зосереджена на спеціально розроблених класах та їхній взаємодії з основними компонентами двигуна Unity, які відображено як окремі сутності.

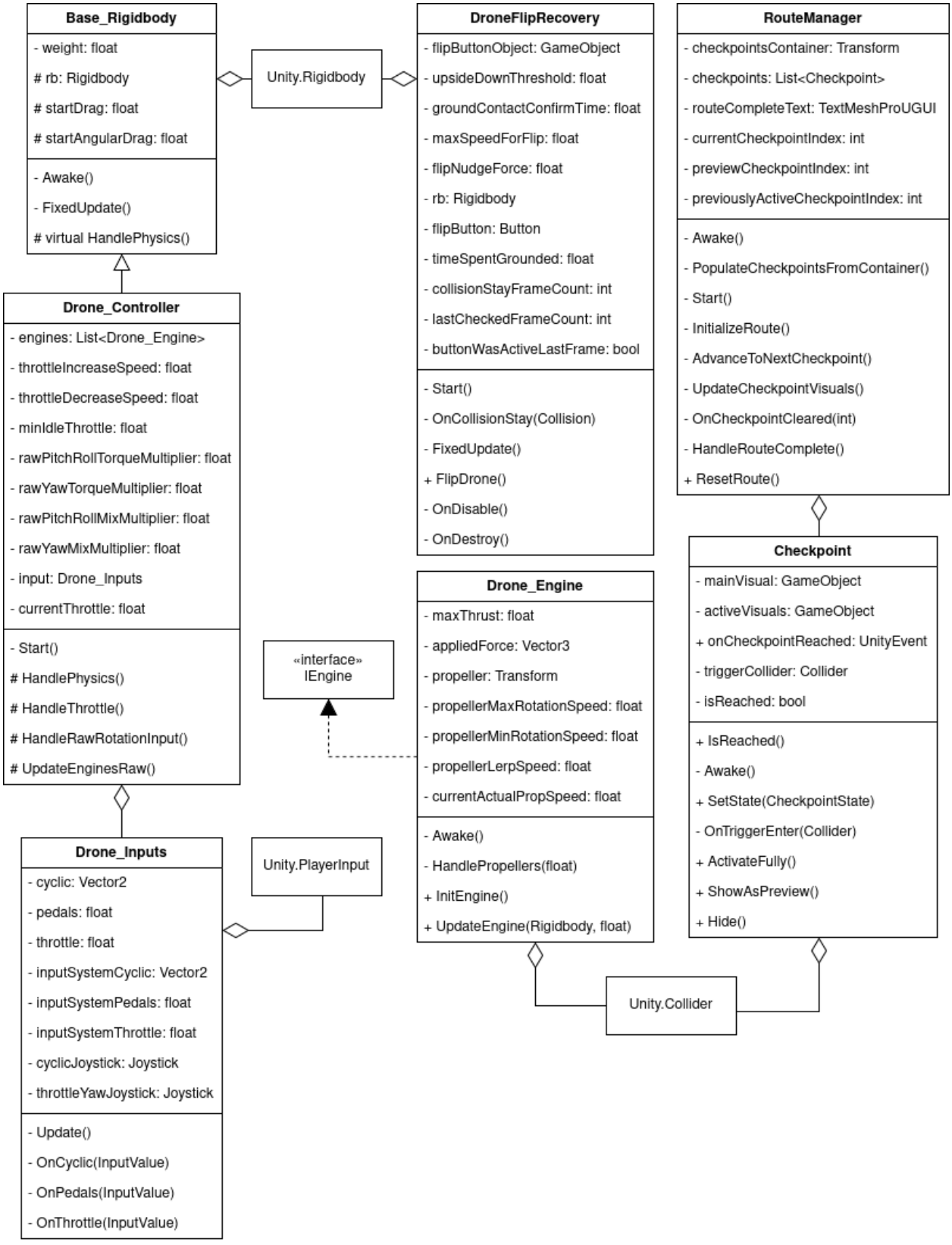


Рисунок 2.2 – Діаграма класів

Система складається з кількох ключових класів, кожен з яких відповідає за певні аспекти симуляції.

Base_Rigidbody: служить базовим класом для об’єктів, що потребують фізичних характеристик Rigidbody. Він інкапсулює загальні налаштування Rigidbody та цикл оновлення фізичного кадру.

Атрибути:

– weight: float: застосовується до властивості rb.mass під час ініціалізації, щоб вплинути на фізичну поведінку під дією таких сил, як гравітація та тяга двигуна;
– rb: Rigidbody: посилання на компонент Unity.Rigidbody, ініціалізований в Awake() – це критично важливо для фізичних взаємодій;
– startDrag: float: застосовується для встановлення опору за замовчуванням поступальному руху в повітрі;
– startAngularDrag: float: застосовується для встановлення опору обертальному руху за замовчуванням.

Операції:

– Awake(): void: ініціалізує посилання rb, викликаючи GetComponent<Rigidbody>();
– FixedUpdate(): void: викликається Unity на кожному фіксованому кроці часу оновлення фізичного кадру – він, у свою чергу, викликає HandlePhysics();
– virtual HandlePhysics(): void: захищений віртуальний метод, призначений для перевизначення похідними класами (наприклад, Drone_Controller) для реалізації їхньої специфічної логіки оновлення фізичного стану.

Зв’язки:

– має зв’язок агрегації з Unity.Rigidbody, що вказує на те, що він володіє екземпляром Rigidbody;
– служить батьківським класом (узагальненням) для Drone_Controller.

Drone_Controller (успадковується від Base_Rigidbody): центральний клас, що керує поведінкою дрона в польоті, обробкою вхідних даних та оркестрацією двигунів.

Атрибути:

- engines: List<Drone_Engine>: список компонентів Drone_Engine, що представляють двигуни дрона;
- throttleIncreaseSpeed, throttleDecreaseSpeed, minIdleThrottle: float: параметри, що контролюють поведінку дросельної заслінки (тяги);
- rawPitchRollTorqueMultiplier, rawYawTorqueMultiplier, rawPitchRollMixMultiplier, rawYawMixMultiplier: float: параметри, що визначають чутливість необробленого вхідного обертання до крутного моменту та «змішування» двигунів (розподілу прикладених сил між двигунами);
- input: Drone_Inputs: посилання на компонент Drone_Inputs для отримання обробленого вхідного сигналу гравця;
- currentThrottle: float: поточний накопичений рівень дросельної заслінки (тяги) дрона.

Операції:

- Start(): void: ініціалізує посилання (наприклад, вхідні дані), встановлює початкові властивості Rigidbody (маса, опір тощо) та ініціалізує двигуни;
- HandlePhysics(): void (Перевизначає з Base_Rigidbody): основний цикл оновлення фізичного стану для дрона, що викликає підпорядковані методи обробки;
- HandleThrottle(): void: керує накопиченням та застосуванням дросельної заслінки на основі вхідних даних;
- HandleRawRotationInput(): void: обчислює крутний момент на основі необроблених даних гравця щодо тангажу, крену та рискування;
- UpdateEnginesRaw(): void: обчислює індивідуальну тягу для кожного двигуна на основі currentThrottle та необроблених команд обертання, а потім викликає UpdateEngine для кожного Drone_Engine.

Зв’язки:

– успадковується від Base_Rigidbody;
– має зв’язок композиції з Drone_Inputs, що вказує на керування екземпляром Drone_Inputs;
– має зв’язок агрегації з Drone_Engine (їх список).

Drone_Inputs: абстрагує та обробляє вхідні дані з різних джерел (система введення для клавіатури/геймпада, джойстики інтерфейсу користувача для мобільних пристроїв) та надає чисті вхідні значення Drone_Controller.

Атрибути:

– cyclic: Vector2, pedals: float, throttle: float: приватні поля резервного копіювання для публічних властивостей введення;
– inputSystemCyclic: Vector2, inputSystemPedals: float, inputSystemThrottle: float: зберігає необроблені значення з Callbacks системи введення Unity;
– cyclicJoystick: Joystick, throttleYawJoystick: Joystick: посилання на компоненти Joystick інтерфейсу користувача.

Операції:

– Update(): void: зчитує вхідні дані з джойстиків інтерфейсу користувача (якщо активні) та об’єднує їх зі значеннями системи введення;
– OnCyclic(значення InputValue): void, OnPedals(значення InputValue): void, OnThrottle(значення InputValue): void: методи зворотного виклику, що викликаються компонентом Unity.PlayerInput, коли запускаються відповідні дії.

Зв’язки: має агрегаційний зв’язок з Unity.PlayerInput, оскільки вимагає, щоб цей компонент отримував події системи введення.

IEngine (Інтерфейс): визначає контракт для будь-якого двигуна дрона, забезпечуючи стандартизований спосіб оновлення.

Операції:

– InitEngine(): void;
– UpdateEngine(Rigidbody rb, float thrustInput): void.

Drone_Engine (реалізує Iengine): представляє один двигун/пропелер на дроні. Він застосовує тягу та керує візуальним обертанням його пропелера.

Атрибути:

– maxThrust: float: максимальна сила, яку може створити цей двигун;
– appliedForce: Vector3: зберігає розраховану силу для візуалізації гізмо (візуального об’єкту);
– propeller: Transform: посилання на об’єкт Transform моделі візуального пропелера;
– propellerMaxRotationSpeed, propellerMinRotationSpeed, propellerLerpSpeed: float: параметри, що керують візуальною анімацією пропелера;
– currentActualPropSpeed: float: поточна згладжена візуальна швидкість пропелера.

Операції:

– Awake(): void: ініціалізує currentActualPropSpeed;
– HandlePropellers(float thrustInput): void: обчислює та застосовує обертання до Transform пропелера на основі тяги та згладжування;
– InitEngine(): void: реалізує метод інтерфейсу;
– UpdateEngine(Rigidbody rigidbody, float thrustInput): void: обчислює та застосовує силу до Rigidbody дрона в позиції двигуна за допомогою AddForceAtPosition. Викликає HandlePropellers.

Зв’язки:

– реалізує інтерфейс IEngine;
– має агрегаційний зв’язок з Unity.Collider.

DroneFlipRecovery: керує логікою виявлення перевернутого дрона на землі та надає механізм інтерфейсу користувача для його повернення у вертикальне положення.

Атрибути:

– flipButtonObject: GameObject: посилання на GameObject кнопки інтерфейсу користувача;
– rb: Rigidbody: посилання на Rigidbody дрона;

– upsideDownThreshold, groundContactConfirmTime, maxSpeedForFlip, flipNudgeForce: float: налаштовувані параметри для виявлення та відновлення;
– flipButton: Button: посилання на компонент UI.Button;
– timeSpentGrounded: float: таймер стабільного контакту із землею;
– collisionStayFrameCount: int, lastCheckedFrameCount: int: лічильники для надійного виявлення зіткнень у різних кадрах;
– buttonWasActiveLastFrame: bool: змінна стану для запобігання мерехтіння інтерфейсу користувача під час певних сценаріїв пробудження фізичного стану об’єктів.

Операції:

– Start(): void: ініціалізує посилання та додає Listener до flipButton;
– OnCollisionStay(Collision collision): void: збільшує collisionStayFrameCount при фізичному контакті;
– FixedUpdate(): void: містить основну логіку для перевірки умов зіткнення, керування викликом WakeUp() для Rigidbody та визначення видимості кнопки;
– FlipDrone(): void: публічний метод, що викликається подією OnClick кнопки інтерфейсу користувача для виконання переорієнтації дрона;
– OnDisable(): void, OnDestroy(): void: очищення Listeners кнопок інтерфейсу користувача.

Зв’язки: має агрегацію з Unity.Rigidbody.

RouteManager: керує послідовністю контрольних точок, контролює їх стани (активний, попередній перегляд, прихований) та обробляє перебіг та завершення маршруту.

Атрибути:

– checkpointsContainer: Transform: посилання на батьківський GameObject для автоматичного заповнення контрольних точок;
– checkpoints: List<Checkpoint>: упорядкований список контрольних точок, що визначають маршрут;
– routeCompleteText: TextMeshProUGUI: посилання на текстовий

елемент інтерфейсу користувача для повідомлення про завершення;

– currentCheckpointIndex: int, previewCheckpointIndex: int, previouslyActiveCheckpointIndex: int: змінні стану для відстеження перебігу маршруту.

Операції:

– Awake(): void: викликає PopulateCheckpointsFromContainer;
– PopulateCheckpointsFromContainer(): void: заповнює список контрольних точок з дочірніх елементів checkpointsContainer;
– Start(): void: викликає InitializeRoute;
– InitializeRoute(): void: скидає стан маршруту, приховує всі контрольні точки, налаштовує Listeners та активує першу контрольну точку;
– AdvanceToNextCheckpoint(): void: переходить до наступної контрольної точки, оновлює індекси та викликає UpdateCheckpointVisuals, обробляє завершення маршруту;
– UpdateCheckpointVisuals(): void: встановлює відповідний стан (прихований, попередній перегляд, активний) для відповідних контрольних точок;
– OnCheckpointCleared(int indexCleared): void: обробник подій, що викликається контрольною точкою, коли вона досягнута;
– HandleRouteComplete(): void: керує діями після завершення маршруту, включаючи відображення routeCompleteText;
– ResetRoute(): void: дозволяє скинути маршрут до початкового стану.

Зв’язки: має агрегацію з Checkpoint.

Checkpoint: представляє окрему точку маршруту. Керує її візуальним виглядом та виявляє, коли дрон проходить через неї.

Атрибути:

– mainVisual: GameObject, activeVisuals: GameObject: посилання на GameObjects, що контролюють зовнішній вигляд точки;
– onCheckpointReached: UnityEvent: публічна подія, що спрацьовує при спрацьовуванні контрольної точки, на яку підписаний RouteManager;

– triggerCollider: Collider: посилання на тригерний колайдер точки;
– isReached: bool: відстежує, чи цю контрольну точку було успішно пройдено в поточному циклі активації.

Операції:

– Awake(): void: ініціалізує triggerCollider та встановлює початковий стан на Hidden;
– SetState(CheckpointState newState): void: основний метод для зміни візуального та інтерактивного стану контрольної точки;
– OnTriggerEnter(Collider other): void: виявляє проходження дрона, встановлює isReached та викликає onCheckpointReached;
– ActivateFully(): void, ShowAsPreview(): void, Hide(): void: методи зручності, що викликають SetState.

Зв’язки: має агрегацію з Unity.Collider.

Ця структура класів утворює логічну основу симулятора FPV-дрона. Drone_Controller та пов’язані з ним класи двигунів керують основною фізичною поведінкою та механікою польоту, керуючись абстрактними вхідними даними від Drone_Inputs. Класи RouteManager та Checkpoint співпрацюють, щоб забезпечити цілеспрямований ігровий процес за маршрутами польоту. Нарешті, DroneFlipRecovery пропонує утиліту для покращення взаємодії користувача після аварій дрону. Кожен клас має чітко визначені обов’язки, а їхня взаємодія регулюється за допомогою викликів методів, обробки подій та спільного доступу до компонентної архітектури Unity, зокрема до компонентів Rigidbody та Collider для взаємодії між об’єктами. Ця модульна конструкція спрямована на чіткість, зручність обслуговування та майбутнє розширення.

2.4 Фізичне проєктування

Існує потреба визначити концептуальну фізичну модель, що буде використано для симулятора FPV-дрона. Мета полягає в досягненні

реалістичного досвіду польоту, що імітує ключові сили та динаміку, що керують польотом квадрокоптера в режимі «Акро». Модель повинна забезпечувати однозначну основу для подальшої реалізації фізичної поведінки дрона.

Усі фізичні взаємодії повинні керуватися вбудованим 3D-фізичним двигуном Unity, використовуючи його компонент Rigidbody. Основні принципи моделі наведено нижче.

Моделювання на основі Rigidbody. Фізична присутність дрона та його реакція на сили повинні керуватися компонентом Unity Rigidbody. Цей компонент обробляє масу, інерцію, гравітацію та виявлення зіткнень.

Конфігурація квадрокоптера. На цьому етапі проєкту модель припускатиме стандартну конфігурацію квадрокоптера «X» з чотирма двигунами/пропелерами.

Фокус на акро-режимі. Основний режим польоту, що імітується, має бути аналогічним режиму «Акро» на реальних FPV-дронах. У цьому режимі введення даних за допомогою джойстика користувача керує бажаною швидкістю обертання навколо кожної осі, і дрон не вирівнюється автоматично.

Спрощена аеродинаміка. Не забуваючи необхідність відчуття реалізму, складні аеродинамічні ефекти, такі як вплив землі на віддзеркалення повітряних мас, відштовхуваних пропелерами, стани вихрового кільця або точні розрахунки підйомної сили/опіру аеродинамічного профілю вважаються поза межами цієї ітерації для підтримки обчислювальної продуктивності та спрощення реалізації. Ключові аеродинамічні ефекти, такі як повітряні лінійний та кутовий опори, слід апроксимувати на достатньому реалістичному рівні.

Рух дрона визначатиметься сумою кількох сил та динамічних моделей.

Гравітація. Постійна сила, спрямована вниз, що діє на центр мас дрона. Гравітація буде оброблятися безпосередньо фізичним двигуном Unity шляхом увімкнення опції «Використовувати гравітацію» на компоненті Rigidbody дрона. Величина та напрямок будуть взяті з глобальних налаштувань фізичного рушія Unity (-9,81 м/с² по осі Y). *Ефект*: змушує дрон

прискорюватися вниз, якщо цьому не протидіє достатня тяга, спрямована вгору.

Тяга. Сила, спрямована вгору, що генерується чотирма пропелерами. Величина тяги від кожного двигуна буде індивідуально керована. Кожен з чотирьох імітованих двигунів застосовуватиме вектор сили. Напрямок цієї сили буде вздовж локальної осі «вгору» кожного окремого екземпляру двигуна/пропелера. Це означає, що при нахилі дрона також нахиляється вектор тяги. Величина сили від кожного двигуна буде пропорційна його розрахованій команді тяги (отриманій із поєднання загальної тяги та змішування двигунів). Сили будуть прикладатися до положення кожного двигуна за допомогою Rigidbody.AddForceAtPosition(). Це критично важливо, оскільки різниця тяги між двигунами, що прикладається до їхніх відповідних положень, генеруватиме крутний момент для обертання. *Ефект*: протидіє силі тяжіння для підйомної сили та забезпечує рушійну силу для всіх маневрів.

Опір повітря. Сила, яка протидіє руху дрона в повітрі, збільшуючись зі швидкістю. Це критично важливо для реалістичного уповільнення та кінцевої швидкості. Наведемо деталі реалізації двох компонентів опору повітря.

Лінійний опір. Компонент Unity Rigidbody має властивість drag. Ця властивість буде використана для імітації лінійного опору повітря, що протистоїть поступальній швидкості. Її значення слід налаштувати для досягнення реалістичного відчуття.

Кутовий опір. Компонент Unity Rigidbody також має властивість angularDrag. Це є критично важливим для демпфування обертань, особливо в режимі Acro-mode, дозволяючи дрону зупинити обертання, коли вхідні дані керування центровані. Його значення слід налаштувати для реалістичної реакції. *Ефект*: уповільнює поступальний та обертальний рух дрона, запобігає нескінченному прискоренню та робить керування менш «слизьким».

Застосування крутного моменту. Обертальний рух (тангаж, крен, рискання) досягається шляхом застосування крутного моменту до Rigidbody дрона. Наведемо деталі реалізації крутного моменту.

Прямий крутний момент (режим Acro). Вхідні дані користувача (після обробки та масштабування) визначатимуть цільову кутову швидкість для кожної осі (тангаж, крен, риськування). Система керування повинна розрахувати різницю між поточною кутовою швидкістю жорсткого корпусу та цільовою кутовою швидкістю. Ця похибка буде помножена на коефіцієнт підсилення, щоб визначити необхідний коригувальний крутний момент. Цей крутний момент буде застосовано до Rigidbody за допомогою AddRelativeTorque().

Крутний момент від диференціальної тяги. Застосування різних рівнів тяги до окремих двигунів за допомогою AddForceAtPosition генерує крутний момент. Алгоритм змішування двигунів розроблений для цілеспрямованого створення цих диференціалів тяги для досягнення бажаних команд обертання, які відповідають крутним моментам, розрахованим вище. Два методи генерації крутного моменту (пряме AddRelativeTorque та диференціальна тяга через AddForceAtPosition від змішування двигунів) повинні працювати узгоджено. *Ефект*: змушує дрон обертатися навколо осей тангажу, крену та рискання.

Інерція. Опір дрона змінам його обертального стану. Важчі або більші дрони (або ті, чия маса розподілена далі від центру) матимуть вищу інерцію. Обробляється Unity.Rigidbody на основі його маси та розподілу пов’язаних з ним компонентів Collider (які визначають тензор інерції). Rigidbody.centerOfMass буде явно встановлено на геометричний центр дрона, щоб забезпечити передбачувану поведінку. *Ефект*: робить дрон «важким» під час обертання; швидкі рухи джойстиком не призведуть до миттєвих змін обертання.

Змішування двигунів. Вирішальний процес перетворення бажаної загальної тяги та бажаних команд обертання (тангажу, крену, рискання) у конкретні команди тяги для кожного з (чотирьох) окремих двигунів. Буде використано набір арифметичних формул. Для стандартної конфігурації квадрокоптера «X» значення baseThrottle буде визначено на основі введення дроселя гравцем. Значення pitchCommand, rollCommand та yawCommand

будуть отримані з введених гравцем даних за допомогою джойстика (масштабованих множниками чутливості). Остаточна команда тяги кожного двигуна буде сумою/відніманням цих компонентів: Individual_Motor_Thrust = baseThrottle +/- pitchContribution +/- rollContribution +/- yawContribution (зі знаками, що залежать від відносного положення двигуна). Знаки гарантують, що, наприклад, команда позитивного тангажу (ніс вгору) відповідно зменшує тягу на передніх двигунах і збільшує її на задніх. Аналогічно для крену та рискання, де рискання залежить від різниці реакцій крутного моменту між пропелерами, що обертаються за годинниковою стрілкою та проти годинникової стрілки. *Ефект*: дозволяє зіставити чотири керуючі входи (дросель, тангаж, крен, рискання) з чотирма двигунами таким чином, щоб забезпечити бажану поведінку польоту та обертання.

Модель зіткнення. Визначає, як дрон фізично взаємодіє з навколишнім середовищем. Дрон матиме один або кілька компонентів Collider, приєднаних до його ієрархії, що визначають його фізичну форму для цілей зіткнення. Об’єкти середовища також матимуть компоненти Collider. Фізичний двигун Unity оброблятиме виявлення зіткнень та реагування на них (наприклад, підстрибування, зупинка). Спеціальна ігрова логіка (наприклад, система DroneFlipRecovery) відстежуватиме стани зіткнень (OnCollisionStay) для запуску ігрових подій. *Ефект*: запобігає проходженню дрона крізь тверді об’єкти та забезпечує основу для виявлення зіткнень та реагування на них.

Цикл моделювання. Усі фізичні розрахунки, застосування сили та застосування крутного моменту відбуватимуться в циклі FixedUpdate() Unity, щоб забезпечити послідовну та детерміновану фізичну поведінку незалежно від частоти кадрів.

Підсумовуючи, концептуальна фізична модель симулятора FPV-дрона базується на використанні фізики твердого тіла (Rigidbody) Unity, застосовуючи реалістичні сили, такі як гравітація, тяга кількох двигунів та опір повітря. Динаміка обертання буде досягнута завдяки точному застосуванню крутного моменту, керована введенням користувача та за допомогою алгоритму змішування двигунів, який перетворює бажані маневри

на окремі команди двигунів. Цей підхід, з ретельним налаштуванням параметрів, має на меті забезпечити автентичний та захопливий досвід польоту в «акро-режимі», формуючи міцну та однозначну основу для основної ігрової механіки симулятора.

2.5 Висновки другого розділу

Другий розділ був присвячений комплексному проєктуванню інформаційної системи симуляції керування FPV-дроном. На цьому етапі було виконано детальний аналіз та формулювання функціональних і нефункціональних вимог, що дозволило чітко окреслити очікувану поведінку, можливості та якісні характеристики майбутнього програмного продукту. Визначені вимоги лягли в основу подальших проєктних рішень.

У рамках концептуального проєктування було розроблено діаграму прецедентів, яка наочно відображає основні сценарії взаємодії користувача з системою, включаючи запуск симулятора, керування дроном, проходження маршрутів та відновлення після аварій. Це дозволило верифікувати повноту охоплення користувацьких потреб.

На етапі логічного проєктування було створено діаграму класів, що відображає архітектуру ключових програмних компонентів, їх атрибути, методи та взаємозв’язки в контексті обраного рушія Unity. Таке структурування програмної логіки забезпечує модульність, полегшує подальшу розробку, тестування та потенційне розширення функціоналу симулятора.

Таким чином, проведена в цьому розділі робота з аналізу вимог та детального проєктування створила теоретичний та структурний фундамент для безпосередньої реалізації FPV-симулятора, мінімізуючи ризики на етапі розробки та забезпечуючи відповідність кінцевого продукту поставленим цілям та вимогам доступності й ефективності.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Програмна реалізація процесу керування дроном

У цьому розділі детально описано використану архітектуру програмного забезпечення та стратегію реалізації системи керування дроном у симуляторі FPV Drone. У ньому описано, як вхідні дані гравця фіксуються, обробляються та перетворюються на фізичні дії, що виконуються симульованим дроном.

Система розроблена модульною, з підтримкою різних пристроїв введення та забезпеченням реалістичного польоту в «акро-режимі» на основі фізичної моделі, визначеної раніше.

Ключові компоненти включають систему введення Unity для отримання вхідних даних, клас DroneInputs для обробки вхідних даних, клас Drone_Controller для логіки польоту та застосування фізичної моделі, а також класи Drone_Engine для тяги на рівні двигуна та візуального зворотного зв’язку. Основою керування дроном є точне захоплення та інтерпретація намірів гравця.

Основним механізмом для захоплення необроблених вхідних даних з різних пристроїв (клавіатури, геймпада та основою для сенсорних обгорток введення) є новий пакет Unity Input System.

Компонент PlayerInput приєднаний до GameObject дрона. Цей компонент налаштовано за допомогою активу Input Actions. Актив Input Actions визначає абстрактні дії, такі як “Cyclic” (для тангажу/кренування), «Педалі» (для рискання) та «Тяга». Ці дії потім зіставляються з певними фізичними елементами керування (наприклад, “Cyclic”, зіставленим з лівим джойстиком на геймпаді або клавішами WASD).

Компонент PlayerInput налаштовано на виклик подій (або надсилання повідомлень) під час виконання цих дій.

Компонент DroneInputs показано на рисунку 3.1.

```
using UnityEngine;
using UnityEngine.InputSystem;

[RequireComponent(typeof(PlayerInput))]
public class DroneInputs : MonoBehaviour
{

    private Vector2 cyclic;
    private float pedals;
    private float throttle;

    public Vector2 Cyclic { get => cyclic; }
    public float Pedals { get => pedals; }
    public float Throttle { get => throttle; }

    [Header("Mobile Touch Controls")]
    [Tooltip("Assign the UI Joystick for Pitch/Roll (Cyclic).")]
    [SerializeField] private Joystick cyclicJoystick;
    [Tooltip("Assign the UI Joystick for Throttle/Yaw (Pedals).")]
    [SerializeField] private Joystick throttleYawJoystick;

    private Vector2 _inputSystemCyclic;
    private float _inputSystemPedals;
    private float _inputSystemThrottle;

    void Update () {
        Vector2 currentCyclic = _inputSystemCyclic;
        float currentPedals = _inputSystemPedals;
        float currentThrottle = _inputSystemThrottle;

        #if UNITY_ANDROID || UNITY_IOS || UNITY_EDITOR
        if (cyclicJoystick != null && cyclicJoystick.gameObject.activeInHierarchy)
        {
            if (cyclicJoystick.Direction.sqrMagnitude > 0.01f)
            {
                 currentCyclic = cyclicJoystick.Direction;
            }
        }

        if (throttleYawJoystick != null && throttleYawJoystick.gameObject.activeInHierarchy)
        {
            if (throttleYawJoystick.Direction.sqrMagnitude > 0.01f)
            {
                currentThrottle = throttleYawJoystick.Vertical;
                currentPedals = throttleYawJoystick.Horizontal;
            }
        }
        #endif

        cyclic = currentCyclic;
        pedals = currentPedals;
        throttle = currentThrottle;
    }

    private void OnCyclic(InputValue value)
    {
        _inputSystemCyclic = value.Get<Vector2>();
    }

    private void OnPedals(InputValue value)
    {
        _inputSystemPedals = value.Get<float>();
    }

    private void OnThrottle(InputValue value)
    {
        _inputSystemThrottle = value.Get<float>();
    }
}
```

Рисунок 3.1 – Компонент DroneInputs

Цей скрипт діє як приймач і обробник вхідних даних від компонента PlayerInput та екранних елементів керування мобільними пристроями. Він забезпечує чистий, абстрактний шар вхідних значень для Drone_Controller. Він містить такі методи, як OnCyclic(InputValue), OnPedals(InputValue) та OnThrottle(InputValue). Ці методи автоматично викликаються компонентом PlayerInput, коли запускаються відповідні їм дії введення. Вони зберігають отримані вхідні значення в приватних полях резервного копіювання. Для платформи Android DroneInputs містить посилання на компоненти джойстика інтерфейсу користувача (cyclicJoystick, throttleYawJoystick). У своєму методі Update() він зчитує значення відхилення з цих екранних джойстиків. DroneInputs надає кінцеві, оброблені вхідні значення через публічні властивості лише для читання: Cyclic (Vector2), Pedals (float) та Throttle (float). Drone_Controller зчитує ці властивості для отримання команд користувача.

Клас Drone_Controller, що успадковується від Base_Rigidbody (який надає посилання rb Rigidbody та керує викликом циклу FixedUpdate до HandlePhysics), є мозком дрона (див. рис. 3.2). Метод ініціалізації отримує посилання на компонент DroneInputs, налаштовує властивості Rigidbody, важливі для польоту: mass, linearDamping, angularDamping, useGravity та centerOfMass, ініціалізує список компонентів Drone_Engine, або шляхом прямого призначення Inspector, або шляхом знаходження їх як дочірніх. HandlePhysics метод організовує оновлення для кожного логічного кадру для керування дроном. Він викликає спеціалізовані методи обробника послідовно.

Метод керування тягою зчитує значення Throttle з DroneInputs.Throttle. Це вхідне значення визначає швидкість зміни внутрішнього значення currentThrottle дрона. CurrentThrottle (значення від 0 до 1) накопичується з часом на основі throttleInput, throttleIncreaseSpeed та throttleDecreaseSpeed. Якщо throttleInput дорівнює нулю, currentThrottle залишається незмінним. minIdleThrottle забезпечує мінімальну вихідну потужність, навіть коли currentThrottle близький до нуля, що сприяє підвищеній стабільності та постійному візуальному обертанню пропелера.

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

[RequireComponent(typeof(DroneInputs))]
public class Drone_Controller : Base_Rigidbody
{
  [Header("Engines & Setup")]
    [SerializeField] private List<Drone_Engine> engines = new List<Drone_Engine>(4);

    [Header("Throttle Control")]
    [SerializeField] private float throttleIncreaseSpeed = 2.0f;
    [SerializeField] private float throttleDecreaseSpeed = 3.0f;
    [SerializeField] private float minIdleThrottle = 0.05f;

    [Header("Raw Rotation Control (No PID)")]
    [Tooltip("Multiplier for player PITCH/ROLL input -> applied torque strength.")]
    [SerializeField] private float rawPitchRollTorqueMultiplier = 0.1f;
    [Tooltip("Multiplier for player YAW input -> applied torque strength.")]
    [SerializeField] private float rawYawTorqueMultiplier = 0.5f;
    [Tooltip("Multiplier for player PITCH/ROLL input -> motor mix differential strength.")]
    [SerializeField] private float rawPitchRollMixMultiplier = 0.1f;
    [Tooltip("Multiplier for player YAW input -> motor mix differential strength.")]
    [SerializeField] private float rawYawMixMultiplier = 0.5f;


    private DroneInputs input;
    private float currentThrottle = 0f;

  void Start()
  {
    input = GetComponent<DroneInputs>();
    rb.mass = 1.5f;
        rb.linearDamping = 0.6f;
        rb.angularDamping = 3.0f;
        rb.useGravity = true;
        rb.centerOfMass = Vector3.zero;
        if (engines == null || engines.Count == 0)
        {
            engines = GetComponentsInChildren<Drone_Engine>().ToList();
            if (engines.Count != 4)
            {
                Debug.LogError("Drone_Controller expects exactly 4 engines assigned or found
as children!");
            }
        }
        foreach (var engine in engines)
        {
            engine.InitEngine();
        }
  }

    protected override void HandlePhysics()
    {
      HandleThrottle();
        HandleRawRotationInput();
        UpdateEnginesRaw();
    }
```

Рисунок 3.2 – Компонент Drone_Controller,

аркуш 1

```
    protected virtual void HandleThrottle()
    {
        float throttleInput = input.Throttle;

        float changeSpeed = (throttleInput > 0) ? throttleIncreaseSpeed :
throttleDecreaseSpeed;
        if (Mathf.Abs(throttleInput) > 0.01f)
        {
             float effectiveChangeSpeed = (throttleInput > 0) ? throttleIncreaseSpeed :
throttleDecreaseSpeed;
             currentThrottle += throttleInput * effectiveChangeSpeed * Time.fixedDeltaTime;
        }

        currentThrottle = Mathf.Clamp01(currentThrottle);
    }

    protected virtual void HandleRawRotationInput()
    {
        float pitchInput = input.Cyclic.y;
        float rollInput = -input.Cyclic.x;
        float yawInput = input.Pedals;

        Vector3 rawTorque = new Vector3(
            pitchInput * rawPitchRollTorqueMultiplier,
            yawInput * rawYawTorqueMultiplier,
            rollInput * rawPitchRollTorqueMultiplier
        );

        rb.AddRelativeTorque(rawTorque, ForceMode.Force);
    }

    protected virtual void UpdateEnginesRaw()
    {
        float baseThrottle = Mathf.Lerp(minIdleThrottle, 1.0f, currentThrottle);

        float pitchInput = input.Cyclic.y;
        float rollInput = input.Cyclic.x;
        float yawInput = input.Pedals;

        float pitchCommand = Mathf.Clamp(pitchInput * rawPitchRollMixMultiplier, -1f, 1f);
        float rollCommand  = Mathf.Clamp(rollInput * rawPitchRollMixMultiplier, -1f, 1f);
        float yawCommand   = Mathf.Clamp(yawInput * rawYawMixMultiplier, -1f, 1f);

        float thrustFL = baseThrottle - pitchCommand + rollCommand - yawCommand;
        float thrustFR = baseThrottle - pitchCommand - rollCommand + yawCommand;
        float thrustBL = baseThrottle + pitchCommand + rollCommand + yawCommand;
        float thrustBR = baseThrottle + pitchCommand - rollCommand - yawCommand;

        engines[0].UpdateEngine(rb, Mathf.Clamp01(thrustFL));
        engines[1].UpdateEngine(rb, Mathf.Clamp01(thrustFR));
        engines[2].UpdateEngine(rb, Mathf.Clamp01(thrustBL));
        engines[3].UpdateEngine(rb, Mathf.Clamp01(thrustBR));
    }
}
```

Рисунок 3.2, аркуш 2

Метод керування обертанням реалізує керування обертанням в “Acro Mode”. Він зчитує input.Cyclic (для команд тангажу та крену) та input.Pedals (для команди рискання) з DroneInputs. Ці необроблені вхідні значення

масштабуються за допомогою відповідних параметрів множників (rawPitchRollTorqueMultiplier, rawYawTorqueMultiplier). Отримані масштабовані значення використовуються для побудови Vector3 rawTorque. Цей rawTorque застосовується безпосередньо до Rigidbody дрона за допомогою rb.AddRelativeTorque(rawTorque, ForceMode.Force). Ця команда змушує дрон обертатися навколо своїх локальних осей зі швидкістю, що залежить від вхідної величини та налаштованих множників, без автоматичного вирівнювання. Rigidbody.angularDamping має вирішальне значення для зупинки обертання, коли вхідні дані центровані.

Метод Engines Handling визначає значення baseThrottle шляхом перемикання між minIdleThrottle та повною потужністю (1.0) на основі поточного значення Throttle, розрахованого в HandleThrottle(), та знову зчитує input.Cyclic та input.Pedals. Ці необроблені вхідні дані масштабуються за допомогою rawPitchRollMixMultiplier та rawYawMixMultiplier для створення pitchCommand, rollCommand та yawCommand. Ці команди представляють бажаний внесок диференціальної тяги для обертання. Набір арифметичних формул поєднує baseThrottle з pitchCommand, rollCommand та yawCommand для обчислення кінцевої цільової тяги для кожного з чотирьох двигунів (наприклад, thrustFL = baseThrottle - pitchCommand + rollCommand - yawCommand). Знаки в цих формулах є специфічними для стандартної конфігурації квадрокоптера «X» та визначають, як кожен двигун сприяє досягненню бажаних маневрів тангажу, крену та риськування. Розрахована тяга для кожного двигуна (фіксована між 0 та 1) потім передається до методу UpdateEngine() відповідного екземпляра Drone_Engine.

Інтерфейс Iengine визначає контракт з методами InitEngine() та UpdateEngine(Rigidbody rb, float thrustInput). Це дозволяє враховувати потенційні майбутні варіації типів двигунів, зберігаючи при цьому узгоджений інтерфейс для Drone_Controller.

Клас Drone_Engine, показаний на рисунку 3.3, представляє один блок двигуна та пропелера на дроні.

```
using UnityEngine;

[RequireComponent(typeof(BoxCollider))]
public class Drone_Engine : MonoBehaviour, IEngine
{
  [Header("Engine Properties")]
    [SerializeField] private float maxThrust = 10f;

    [Header("Propeller Properties")]
    [SerializeField] private Transform propeller;
    [SerializeField] private float propellerMaxRotationSpeed = 6000f;
    [SerializeField] private float propellerMinRotationSpeed = 100f;
    [SerializeField] private float propellerLerpSpeed = 2f;

    private float currentActualPropSpeed = 0f;
    private Vector3 appliedForce;

    void Awake()
    {
        currentActualPropSpeed = propellerMinRotationSpeed;
    }

    public void InitEngine() { }

    public void UpdateEngine(Rigidbody rb, float thrustInput)
    {
      thrustInput = Mathf.Clamp01(thrustInput);
        appliedForce = transform.up * thrustInput * maxThrust;
        rb.AddForceAtPosition(appliedForce, transform.position, ForceMode.Force);

        HandlePropellers(thrustInput);
    }

    void HandlePropellers(float thrustInput)
    {
        if(!propeller)
        {
            return;
        }

        float targetPropSpeed = Mathf.Lerp(propellerMinRotationSpeed,
propellerMaxRotationSpeed, thrustInput);

        currentActualPropSpeed = Mathf.Lerp(
            currentActualPropSpeed,
            targetPropSpeed,
            Time.deltaTime * propellerLerpSpeed
        );

        float frameRotation = currentActualPropSpeed * Time.deltaTime;
        propeller.Rotate(Vector3.up, frameRotation);
    }
}
```

Рисунок 3.3 – Компонент Drone_Engine

Даний клас Drone_Engine:

– отримує нормалізований thrustInput (від 0 до 1) від Drone_Controller;
– обчислює фактичний вектор сили: transform.up * thrustInput * maxThrust. Сила спрямована вздовж локальної осі двигуна вгору;

- прикладає цю силу до основного Rigidbody дрона в певній позиції transform.position двигуна за допомогою rb.AddForceAtPosition(appliedForce, transform.position, ForceMode.Force) – застосування сил у різних положеннях дозволяє диференціальній тязі створювати крутний момент для обертання;
- керує візуальним обертанням пов’язаного з Transform пропелера;
- обчислює цільову візуальну швидкість обертання, переходячи між propellerMinRotationSpeed та propellerMaxRotationSpeed на основі thrustInput;
- плавно інтерполює поточну ActualPropSpeed до цієї цільової швидкості, використовуючи propellerLerpSpeed для створення візуальної інерції;
- застосовує обертання до трансформації пропелера в кожному кадрі.

Система керування дроном реалізована за допомогою багаторівневої архітектури, яка розділяє збір вхідних даних, обробку вхідних даних, логіку польоту високого рівня та механіку двигунів низького рівня. Система введення Unity забезпечує гнучкий вхідний сигнал пристрою, DroneInputs стандартизує це для контролера, Drone_Controller реалізує основну фізичну модель польоту (тяга, крутний момент для обертання, змішування двигунів), а екземпляри Drone_Engine застосовують результуючі сили та анімують пропелери. Ця модульна конструкція сприяє реалістичному управлінню в «акро-режимі», підтримує різні методи введення та дозволяє цілеспрямоване налаштування характеристик польоту.

3.2 Кросплатформне компілювання застосунку

У цьому розділі описано процедури та технічні аспекти компіляції та пакування симулятора FPV-дрона для цільових платформ: Windows, macOS, Linux (разом – настільні платформи) та Android. Хоча ігровий рушій Unity забезпечує надійний кросплатформний конвеєр збірки, для забезпечення

оптимальної продуктивності, сумісності та зручності користувача необхідно враховувати конкретні конфігурації, вимоги до обробки ресурсів та пакування для кожної з них.

Фундальний процес збірки для всіх цільових платформ ініціюється через вікно «Налаштування збірки» Unity. Ключові кроки включають вибір цільової платформи, забезпечення включення всіх необхідних ігрових сцен до збірки, налаштування атрибутів, специфічних для платформи, у налаштуваннях Player, вибір між типами збірки “Debug” та “Release”, і, нарешті, ініціювання збірки для створення пакета програми або її безпосереднього розгортання.

Збірка для настільних платформ, як правило, простіша через меншу кількість апаратних обмежень. Загальні налаштування програвача на цих платформах включають налаштування роздільної здатності та параметрів представлення, таких як повноекранний режим, розміри екрана за замовчуванням та зміна розміру вікна. Тут також налаштовуються значки програм та заставки. Щодо технічних налаштувань, вибір шляху візуалізації та колірного простору (гама або лінійний) впливає на візуальний вивід та продуктивність. Unity зазвичай автоматично вибирає відповідні графічні API, такі як DirectX для Windows, Metal для macOS та Vulkan/OpenGL Core для Linux, хоча можливі ручні зміни. Скриптовий сервер може бути Mono для швидших ітерацій розробки або IL2CPP для оптимізованих збірок випусків, при цьому 64-бітна (x86_64) архітектура є стандартною для Windows та Linux, тоді як macOS підтримує універсальні бінарні файли як для x86_64, так і для Apple Silicon (ARM64).

Для Windows результат збірки складається з файлу .exe та пов’язаної з ним папки *_Data, які можна розповсюджувати як ZIP-архів або через інсталятори, такі як Inno Setup або NSIS. Підписання коду виконуваного файлу цифровим сертифікатом рекомендовано для публічного розповсюдження, щоб уникнути попереджень безпеки.

Збірки macOS створюють пакет .app. Можна створювати універсальні збірки, що підтримують як Intel, так і Apple Silicon. Для розповсюдження поза Mac App Store програма потребує нотаріального засвідчення Apple, що

включає підписання коду та його надсилання до автоматизованої служби Apple. Розповсюдження через Mac App Store накладає додаткові вимоги щодо прав доступу та забезпечення.

Збірки Linux генерують виконуваний файл і папку *_Data, які зазвичай розповсюджуються в архівах .zip або .tar.gz. Користувачам може знадобитися встановити дозволи на виконання для файлу програми та переконатися, що в ньому є загальні системні бібліотеки. Формати пакування, такі як Flatpak або Snap, можуть спростити керування залежностями для кінцевих користувачів.

Компіляція для Android вимагає детальнішої конфігурації, специфічної для платформи, через різноманітне апаратне забезпечення та ОС. У налаштуваннях програвача назва пакета програми має бути унікальним ідентифікатором зворотного домену. Версіонування здійснюється за допомогою рядка версії та зростаючого цілочисельного коду версії пакета. Мінімальний рівень API та цільовий рівень API визначають сумісність з ОС, причому Google Play має вимоги щодо новизни для останнього.

Для оптимальної продуктивності та безпеки коду на Android сервер сценаріїв має бути налаштовано на IL2CPP. Найважливіше, що цільові архітектури повинні включати як ARMv7 (32-розрядну), так і ARM64 (64-розрядну), щоб забезпечити широку сумісність пристроїв та відповідати вимогам Google Play щодо 64-розрядної версії нових програм та оновлень. Зазвичай використовуються графічні API, такі як OpenGLES3 та Vulkan, причому “Auto Graphics API” часто вибирає найкращий варіант, зазвичай надаючи пріоритет Vulkan, якщо він доступний. ASTC є кращим форматом стиснення текстур для якості та розміру, а ETC2 є резервним варіантом.

Для релізних збірок обов’язковий власний цифровий підпис (сховище ключів), який створюється за допомогою диспетчера сховищ ключів Unity. Цей файл сховища ключів життєво важливий для оновлень програм у магазині Google Play і повинен бути надійно скопійований. Unity може виводити або традиційний APK (Android Package Kit), або, що бажано для Google Play, AAB (Android App Bundle). AAB дозволяють динамічній доставці Google Play

надавати оптимізовані APK, адаптовані до окремих пристроїв користувачів, зменшуючи розміри завантажень.

Елементи, специфічні для платформи Android, включають керування дозволами в маніфесті Android (хоча Unity обробляє багато налаштувань за замовчуванням), адаптацію інтерфейсу користувача до різних розмірів екрана за допомогою Canvas Scaler (налаштований на Масштабування відповідно до розміру екрана) та реалізацію сенсорного керування на екрані, такого як джойстики інтерфейсу користувача, якими керує скрипт DroneInputs. Ретельна оптимізація продуктивності, включаючи профілювання на цільових пристроях для керування викликами малювання, кількістю полігонів, складністю шейдерів та фізичними накладними витратами, має першорядне значення через широкий спектр можливостей пристроїв Android та потенційну можливість теплового тротлінгу.

Після компіляції ретельне тестування на кожній цільовій платформі та репрезентативному діапазоні обладнання є важливим для виявлення та усунення специфічних для платформи помилок, вузьких місць у продуктивності або розбіжностей у вхідних даних. Розповсюдження відбуватиметься через відповідні канали для кожної платформи: прямі завантаження або цифрові вітрини, такі як Steam для комп’ютерів, а також Google Play Store або інші магазини додатків для Android.

Інтерфейс запущеного програмного забезпечення після компіляції під операційну систему Linux зображено на рисунку 3.4.

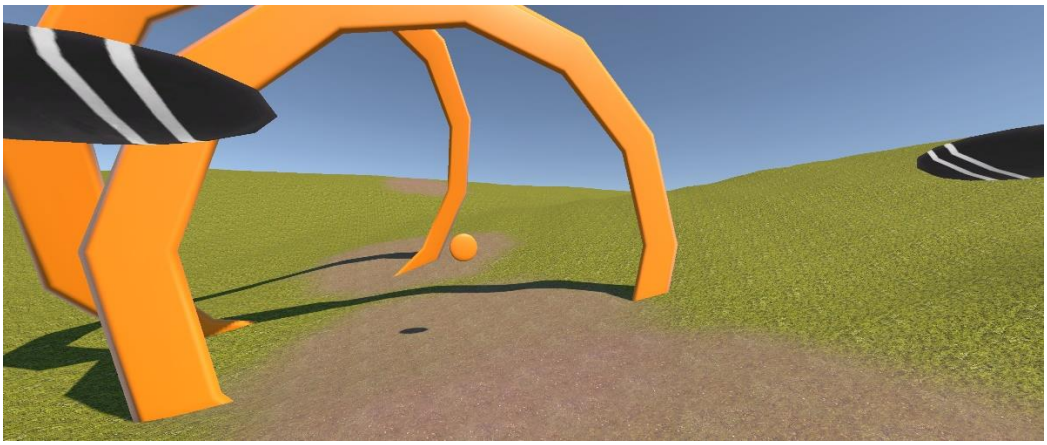


Рисунок 3.4 – Інтерфейс запущеного застосунку

Успішна компіляція та пакування симулятора FPV Drone для цільових платформ вимагає ретельного налаштування в конвеєрі збірки Unity. Збірки для комп'ютерів зосереджені на стандартному пакуванні виконуваних файлів та виборі API, тоді як збірки для Android вимагають уваги до підтримки архітектури (ARMv7/ARM64), вихідних форматів (APK/AAB), цифрового підпису та ретельної оптимізації продуктивності. Дотримання цих специфічних для платформи деталей та найкращих практик має вирішальне значення для надання функціонального, сумісного та оптимізованого продукту користувачам Windows, macOS, Linux та Android.

3.3 Висновки третього розділу

У третьому розділі описано процес програмної реалізації інформаційної системи симуляції керування віддаленими пристроями, перетворюючи проєктні рішення, розроблені в попередньому розділі, на функціонуючий програмний продукт. Було успішно реалізовано ключовий функціонал керування FPV-дроном, включаючи обробку вхідних даних з різних пристроїв за допомогою системи Unity Input System, імплементацію класів DroneInputs, Drone_Controller та Drone_Engine для симуляції поведінки польоту в «акро-режимі», а також візуального зворотного зв'язку. Модульна архітектура системи забезпечує чіткий розподіл відповідальності між компонентами, що сприяє реалістичному управлінню та гнучкості системи.

Окрім основної логіки симуляції, було розглянуто та реалізовано процедури кросплатформного компілювання та пакування готового застосунку для цільових платформ: Windows, macOS, Linux та Android. Це включало налаштування специфічних для кожної платформи параметрів збірки, управління залежностями та оптимізацію для забезпечення сумісності й належної продуктивності.

Таким чином, на етапі програмної реалізації було створено робочий прототип FPV-симулятора, що відповідає визначеним функціональним та

нефункціональним вимогам, з акцентом на реалістичну фізичну модель керування та доступність на широкому спектрі пристроїв. Це підтверджує ефективність обраних інструментів розробки та архітектурних підходів, створюючи основу для подальшого тестування та вдосконалення продукту.

ВИСНОВКИ

У рамках даної кваліфікаційної роботи було розроблено інформаційну систему симуляції керування віддаленими пристроями, зокрема FPV-дронами, що вирішує проблему обмеженої доступності існуючих симуляторів через високі системні вимоги та вартість. Головною метою було створення доступного програмного продукту для тренування базових навичок польоту на персональних комп’ютерах та мобільних пристроях з Android.

На першому етапі було проаналізовано ринок FPV-симуляторів, що підтвердило потребу в розробці та зроблено вибір рушія Unity, обґрунтований його можливостями для роботи з фізичною моделлю та кросплатформністю. Другий етап включив детальне проєктування системи, у тому числі: визначення функціональних та нефункціональних вимог, створення діаграми прецедентів та діаграми класів. Третій етап полягав у програмній реалізації, включаючи компоненти логіки керування дроном в “Acro-Mode”, та забезпеченні кросплатформного компілювання для настільних і мобільних операційних систем.

У результаті створено інформаційну систему, що забезпечує доступний інструмент для тренування керування FPV-дронами. Практична значущість полягає у наданні програмного продукту для індивідуального навчання, освітніх закладів та невеликих компаній без значних інвестицій.

Графічна складова була спрощена для продуктивності, втім існують перспективи для її покращення, розширення функціоналу (запис польотів, онлайн-таблиці, багатокористувацький режим), підтримки спеціалізованих контролерів та інтеграції з VR. Розроблена система є міцною основою для майбутніх розширень, сприяючи популяризації FPV-технологій та безпечному їх використанню.

ПЕРЕЛІК ПОСИЛАНЬ

1. Somerville A., Lynar T., Joiner K., Wild G. Use of Simulation for Pre-Training of Drone Pilots. *Drones*. Vol. 8(11), 640. P. 1-14. URL: https://doi.org/10.3390/drones8110640 (дата звернення: 03.03.2025).

2. Savaş T., Özdemir U., Esen Y. E. The Role of Flight Simulators in Pilot Training. *Journal of Aviation*. 2025. 9(1). P. 137-145. URL: https://doi.org/10.30518/jav.1588557 (дата звернення: 03.03.2025).

3. Top FPV Drone Simulators for Mastering Your Skills. *SRIZFLY*. URL: https://srizfly.com/top-fpv-drone-simulators-for-mastering-your-skills/ (дата звернення: 03.03.2025).

4. Liftoff Simulator Review (Is This the Best FPV Simulator?). *FPVCraft*. URL: https://fpvcraft.com/liftoff-simulator-review/ (дата звернення: 03.03.2025).

5. Features. *VelociDrone*. URL: https://www.velocidrone.com/features (дата звернення: 04.03.2025).

6. VelociDrone FPV Racing Simulator. *Team Blacksheep*. URL: https://www.team-blacksheep.com/simulator (дата звернення: 04.03.2025).

7. The Drone Racing League Simulator. *EpicGames Store*. URL: https://store.epicgames.com/en-US/p/the-drone-racing-league-simulator (дата звернення: 05.03.2025).

8. PyFlyt – UAV Simulation Environments for Reinforcement Learning Research / J.J. Tai and etc. *Journal of Latex Class Files*. 2021. Vol. 14, no. 8. P. 1-10. URL: https://doi.org/10.48550/arXiv.2304.01305 (дата звернення: 05.03.2025).

9. FPV SkyDive : FPV Drone Simulator. *Steam*. URL: https://store.steampowered.com/app/1278060/FPV_SkyDive__FPV_Drone_Simulator/ (дата звернення: 05.03.2025).

10. Nikolaiev M., Novotarskyi M. Comparative Review of Drone Simulators. *Information, Computing and Intelligent Systems*. 2024. No. 4. P. 79-99.

URL: https://doi.org/10.20535/2786-8729.4.2024.300614 (дата звернення: 06.03.2025).

11. DJI Flight Simulator Compatible Controllers Guide. *SRIZFLY*. URL: https://srizfly.com/dji-flight-simulator-compatible-controllers-guide/ (дата звернення: 06.03.2025).

12. Won J., Hong G. Research on Smart Construction Education Training Contents Using a Drone Simulator. *Journal of Multimedia Information System.* 2022. 9(4). P. 345-354. URL: https://doi.org/10.33851/JMIS.2022.9.4.345 (дата звернення: 07.03.2025).

13. Mairaj A., Baba A. I., Javaid A. Y. Application Specific Drone Simulators: Recent Advances and Challenges. *Simulation Modelling Practice and Theory*. 2019. P. 100-117. URL: https://doi.org/10.1016/j.simpat.2019.01.004 (дата звернення: 07.03.2025).

14. Survey of Simulators for Aerial Robots: An Overview and In-Depth Systematic Comparisons / C. A. Dimmig and etc. *IEEE Robotics & Automation Magazine*. 2024. P. 2-15. URL: https://doi.org/10.1109/MRA.2024.3433171 (дата звернення: 07.03.2025).

15. A Survey on Open-Source Simulation Platforms for Multi-Copter UAV Swarms / Z. Chen and etc. *Robotics*. 2023. Vol. 12(2), 53. P. 1-24. URL: https://doi.org/10.3390/robotics12020053 (дата звернення: 10.03.2025).

16. Carretta T. R., Dunlap R. D. Transfer of Training Effectiveness in Flight Simulation: 1986 to 1997. *United States Air Force Research Laboratory*. 1998. URL: https://doi.org/10.21236/ada362818 (дата звернення: 10.03.2025).

17. Physics solutions for game development. *Unity*. URL: https://unity.com/solutions/programming-physics (дата звернення: 11.03.2025).

18. Enhanced physics performance for smooth gameplay. *Unity*. URL: https://unity.com/how-to/enhanced-physics-performance-smooth-gameplay (дата звернення: 11.03.2025).

19. Physics in Unreal Engine. *Epic Developer Community*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/chaos-physics-overview (дата звернення: 11.03.2025).

20. Physics introduction. *Godot Docs*. URL: https://docs.godotengine.org/en/stable/tutorials/physics/physics_introduction.html (дата звернення: 12.03.2025).

21. Unity: Understanding URP, HDRP, and Built-In Render Pipeline. *Wayline*. URL: https://www.wayline.io/blog/unity-understanding-urp-hdrp-built-in (дата звернення: 12.03.2025).

22. Introduction to URP for advanced creators (Unity 6 edition). *Unity*. URL: https://unity.com/resources/introduction-to-urp-advanced-creators-unity-6 (дата звернення: 12.03.2025).

23. Unity проти Unreal Engine – який рушій обрати для гри та чому. Зважуємо всі за та проти з розробниками. *Gamedev.DOU.UA*. URL: https://gamedev.dou.ua/articles/unity-or-unreal-engine/ (дата звернення: 13.03.2025).

24. Unreal vs Unity vs Godot. *Ruah Tech Solutions*. URL: https://ruahtech.com.au/unreal-vs-unity-vs-godot/ (дата звернення: 13.03.2025).

25. Воробець Д. С. Ігровий рушій для розробки інді ігор на основі OpenGL : кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник В. І. Заяц. Запоріжжя : ЗНУ, 2023. 93 с. URL: https://dspace.znu.edu.ua/jspui/handle/12345/12502 (дата звернення: 13.03.2025).

26. Gamedev.DOU.UA. Власний Game Engine: як і навіщо створювати ігровий рушій. *Gamedev.DOU.UA*. URL: https://gamedev.dou.ua/blogs/how-and-why-to-create-a-game-engine/ (дата звернення: 14.03.2025).