Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

ДОСЛІДЖЕННЯ ТА РОЗРОБКА ШАБЛОНУ ПРОЕКТУВАННЯ ДЛЯ ANDROID-ДОДАТКІВ З VULKAN API

RESEARCH AND DEVELOPMENT OF A DESIGN PATTERN FOR ANDROID APPLICATIONS WITH VULKAN API

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗм-21
Лопух В. В.
Керівник:
к.т.н., доцент
Сичук В. А.

Кваліфікаційну роботу
допущено до захисту
«___» _____ 20__ р.
Гарант освітньої програми:
к.т.н., доцент Суринович О. М.
_____

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *магістр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«___» _____ 202__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ДРУГОГО (МАГІСТЕРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Лопуху Владиславу Володимировичу

1. Тема кваліфікаційної роботи: Дослідження та розробка шаблону проектування для Android-додатків з Vulkan API
Керівник роботи: Сичук Віктор Анатолійович, доцент, к.т.н.
затверджені наказом закладу вищої освіти від «29» березня 2025 року № 190/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи: 04 грудня 2025 р.
3. Вихідні дані до роботи *технічне та програмне забезпечення ЕОМ*
4. Зміст розрахунково-пояснювальної записки: *аналіз проблематики створення шаблону проектування Android додатків з Vulkan API, обґрунтування технологій і реалізацію 2 мобільних додатків з 2 різними підходами в проектуванні, експериментальне дослідження результативності програмного забезпечення*
5. Перелік графічного матеріалу 7 рисунків, 3 таблиці, 3 лістинги коду

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Сичук В. А.*		
Теоретичне дослідження та практична реалізація	*Сичук В. А.*		
Експериментальне дослідження системи	*Сичук В. А.*		
Нормоконтроль	*Повстяна Ю. С.*		
Гарант ОП	*Андрущак І. Є.*		
Показник запозичень тексту	___%		
Академічна доброчесність	*Сичук В. А.*		

7. Дата видачі завдання *«02 квітня 2025 р.*

КАЛЕНДАРНИЙ ПЛАН

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Сичук В. А.*		
Теоретичне дослідження та практична реалізація	*Сичук В. А.*		
Експериментальне дослідження системи	*Сичук В. А.*		
Нормоконтроль	*Повстяна Ю. С.*		
Гарант ОП	*Андрущак І. Є.*		
Показник запозичень тексту	___%		
Академічна доброчесність	*Сичук В. А.*		

Здобувач вищої освіти _____ Лопух В. В.

Керівник кваліфікаційної роботи _____ Сичук В. А.

АНОТАЦІЯ

Лопух В. В. Дослідження та розробка шаблону проектування для Android-додатків з Vulkan API. Рукопис.

Кваліфікаційна робота магістра ОП «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота магістра складається зі вступу, трьох розділів, висновків і пропозицій та списку використаних джерел.

У першому розділі здійснено аналіз проблематики інтеграції низькорівневого графічного API Vulkan в операційну систему Android, виявлено конфлікт життєвих циклів та сформульовано завдання дослідження. В другому розділі обґрунтовано шаблон проектування із сегрегацією ресурсів, спроектовано UML-діаграми системи, а також описано програмну реалізацію компонентів на базі Android Game Development Kit. У третьому розділі проведено експериментальне дослідження ефективності розробленого рішення порівняно з монолітним підходом за показниками навантаження на процесор та використання пам'яті,. У висновках узагальнено інформацію, відображену у попередніх частинах. Результати розробки демонструють можливості розробленого шаблону проектування для підвищення стабільності та продуктивності мобільних графічних систем.

Ключові слова: Android, Vulkan API, шаблон проектування, рендеринг, GameActivity, керування ресурсами, оптимізація.

ABSTRACT

Lopukh V. V. Research and development of a design pattern for Android applications with Vulkan API. Manuscript.

Master's qualification work OP "Software Engineering". Lutsk National Technical University. Lutsk, 2025.

The master's qualification work consists of an introduction, three sections, conclusions and proposals, and a list of sources used.

The first section analyzes the issues of integrating the low-level graphics API Vulkan into the Android operating system, identifies a life cycle conflict, and formulates the research task. The second section justifies the design pattern with resource segregation, designs UML diagrams of the system, and describes the software implementation of components based on the Android Game Development Kit. The third section conducts an experimental study of the effectiveness of the developed solution compared to the monolithic approach in terms of processor load and memory usage. The conclusions summarize the information reflected in the previous parts. The development results demonstrate the capabilities of the developed architectural pattern to improve the stability and performance of mobile graphics systems.

Keywords: Android, Vulkan API, design pattern, rendering, GameActivity, resource management, optimization.

ЗМІСТ

ВСТУП

Актуальність теми. Програмне забезпечення для роботи з мобільною 3D графікою є невід'ємною складовою сучасної індустрії розваг та професійних застосунків. Еволюційний розвиток мобільних обчислювальних систем характеризується переходом до низькорівневих графічних інтерфейсів, таких як Vulkan, які надають прямий контроль над апаратними ресурсами. Головною проблемою програмних забезпечень, які використовують цей API в екосистемі Android, є критична невідповідність між статичною природою графічного ядра та динамічним життєвим циклом операційної системи. Існуючі архітектурні рішення часто не враховують цієї специфіки, що призводить до нестабільної роботи програм, складності їх підтримки та неефективного використання ресурсів пристрою.

Метою роботи є підвищення ефективності та відмовостійкості мобільних графічних систем шляхом розробки спеціалізованого шаблону проектування, який забезпечує оптимальне керування ресурсами Vulkan API в середовищі Android.

Об'єкт дослідження – процес розробки нативних графічних додатків для операційної системи Android.

Предмет дослідження – шаблон проектування для Android додатків з Vulkan API.

Завдання, які необхідно виконати:

– провести критичний аналіз існуючих архітектурних рішень та інструментів нативної розробки для виявлення проблем у керуванні низькорівневими ресурсами;

– розробити та обґрунтувати шаблон проектування, який забезпечує модульність та відмовостійкість графічної системи;

– створити функціональне програмне забезпечення на базі розробленого шаблону та монолітну версію для порівняння;

– провести валідацію та тестування розроблених програмних засобів;

– проаналізувати отримані результати та сформулювати висновки щодо ефективності розробленого шаблону порівняно з монолітним підходом.

Наукова новизна роботи полягає у вирішенні фундаментального технічного конфлікту між статичною об'єктною моделлю Vulkan API та динамічним життєвим циклом Android-додатків. Автором було розроблено та теоретично обґрунтовано новий шаблон проектування, який базується на впровадженні стратегії суворої сегрегації ресурсів. Суть цього підходу полягає у чіткому поділі графічних об'єктів на дві незалежні групи: статичний стан асетів та ефемерний стан презентації. Такий підхід дозволив ефективно розірвати жорстку залежність графічного ядра від мінливості віконної системи Android.

Практична цінність роботи полягає у створенні закінченої інформаційної системи, що функціонує як високопродуктивне ядро рендерингу для нативних додатків на базі Vulkan API в середовищі Android. Розроблений шаблон вирішує проблему інтеграції низькорівневого графічного API з асинхронним життєвим циклом мобільної ОС, забезпечуючи стабільність роботи графічних систем та зменшуючи навантаження на апаратні ресурси. Результати роботи можна використовувати як початкову базу для створення прототипів мобільних додатків, які використовують Vulkan API. У науковому плані результати готові до використання як методологічна база для подальших досліджень шаблонів проектування та архітектур у сфері розробки графічних, мобільних додатків.

Це дослідження пройшло апробацію у V всеукраїнській науково-технічній конференції молодих вчених, аспірантів та студентів «Комп'ютерні ігри мультимедіа як інноваційний підхід до комунікації - 2025» [1], та у Second International Scientific and Practical Conference «Progressive Approaches in Science and Engineering» [2].

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМАТИКИ ЗА ТЕМОЮ РОБОТИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1 Огляд і аналіз предметної області проблеми, результатів існуючих теоретичних та експериментальних досліджень

Еволюційний розвиток мобільних обчислювальних систем та графічних технологій характеризується стійкою тенденцією до мінімізації рівнів абстракції між програмним кодом та апаратною архітектурою графічних процесорів. Протягом тривалого часу загальним стандартом для реалізації тривимірної графіки в операційній системі Android виступав OpenGL ES, архітектура якого базується на концепції глобальної машини станів. Проте, із зростанням складності графічних сцен та вимог до реалістичності зображення такий підхід виявив критичні обмеження, створюючи суттєве навантаження на центральний процесор через необхідність постійної валідації станів та значні накладні витрати з боку графічного драйвера [3]. У відповідь на ці технологічні виклики та необхідність підвищення ефективності взаємодії апаратного та програмного забезпечення, компанія Khronos Group розробила та представила Vulkan API. Цей стандарт спроектований для забезпечення прямого, низькорівневого контролю над ресурсами GPU, що дозволяє мінімізувати програмну абстракцію. Корпорація Google офіційно визначила Vulkan як пріоритетний графічний API для екосистеми Android [3], що поступово витісняє застарілі специфікації OpenGL ES завдяки можливості ефективнішого розділення задач та використання багатоядерних архітектур сучасних мобільних процесорів.

Впровадження Vulkan надає розробникам суттєві переваги, серед яких: кардинальне зниження навантаження на центральний процесор під час обробки графічних команд, реалізація новітніх стратегій оптимізації продуктивності та підтримка передових графічних технік, таких як технологія трасування променів у реальному часі. Однак досягнення такої ефективності

супроводжується значним зростанням архітектурної складності, оскільки Vulkan класифікується як API з високим рівнем деталізації коду, що вимагає опису кожного аспекту роботи графічного конвеєра. На відміну від OpenGL ES, драйвери Vulkan не виконують автоматичних, прихованих оптимізацій таких як керування станами або повторне використання об'єктів конвеєра, покладаючи повну відповідальність за реалізацію цих механізмів безпосередньо на прикладне програмне забезпечення, що дозволяє досягти вищої продуктивності порівняно з OpenGL ES при умові коректної інженерної реалізації [4].

Зазначена специфіка вимагає необхідність у розробці та застосуванні нових, адаптованих архітектурних шаблонів проектування, здатних забезпечити ефективне спрощення складності ініціалізації та високої деталізації коду Vulkan. Це вимагає впровадження систем явного керування пам'яттю, синхронізацією потоків та життєвим циклом ресурсів, що є критичною умовою для створення високопродуктивних, надійних та модульних Android-додатків, які повною мірою використовують обчислювальний потенціал сучасного графічного апаратного забезпечення без втрати керованості кодової бази.

Сучасні архітектурні рішення, що застосовуються в ігрових рушіях та графічних бібліотеках, зосереджуються на абстрагуванні низькорівневих Vulkan-операцій, керуванні ресурсами та мультипотоковості, дотримуючись модульного підходу. Для спрощення роботи зі складним та об'ємним кодом Vulkan API інженери використовують багаторівневу систему програмних надбудов, які приховують технічні деталі за зрозумілим інтерфейсом. Замість написання окремих модулів для кожного типу графічних даних, застосовуються універсальні шаблони для буферів та текстур. Різниця між ними визначається лише налаштуваннями доступу та формату, що дозволяє уникнути накопичення зайвого коду та робить структуру програми значно простішою для підтримки.

Сучасні методи рендерингу спрямовані на те, щоб перекласти основне навантаження з центрального процесора безпосередньо на відеокарту. Оптимізація роботи також досягається шляхом правильного групування команд, щоб відеокарта не витрачала час на часте перемикання режимів. Крім того,

Vulkan дозволяє виконувати допоміжні обчислення у фоновому режимі, використовуючи вільні ресурси графічного процесора паралельно з малюванням основного зображення, що суттєво покращує загальну продуктивність додатка.

Аналіз архітектурного підходу до побудови систем рендерингу на одному відкритому проекті на платформі Github свідчить, що ключовим вектором інженерних рішень є подолання бар'єра складності Vulkan API через створення багаторівневих шарів абстракції [5]. У сучасній практиці розробки графічних рушіїв, зокрема тих, що мають відкритий вихідний код, домінує тенденція до використання об'єктно-орієнтованого програмування для інкапсуляції низькорівневої логіки. Фундаментальна методологія полягає у проектуванні класів-обгорток, таких як Buffer, Texture, Pipeline, які слугують інтерфейсом для відповідних нативних об'єктів Vulkan [6]. Це дозволяє автоматизувати критичні процеси керування життєвим циклом ресурсів, спираючись на ідіому RAII, яка гарантує коректну ініціалізацію при створенні об'єкта та детерміноване звільнення ресурсів при виході з області видимості.

З метою забезпечення кросплатформної сумісності без зниження продуктивності, розробники часто відмовляються від динамічного поліморфізму та патернів приховання реалізації, які вносять накладні витрати на етапі виконання [6]. Натомість застосовується стратегія компіляційного заміщення, де створюються окремі класи з ідентичними іменами та публічними інтерфейсами для кожного графічного API. Вибір конкретної реалізації відбувається на етапі препроцесингу за допомогою платформних макросів, що забезпечує прямий виклик функцій та максимальну швидкодію кінцевого бінарного коду.

Тим не менш, впровадження високорівневих абстракцій несе в собі значні ризики, головним з яких є втрата прямого контролю над апаратними особливостями GPU. Надмірна інкапсуляція може приховувати від розробника критично важливі важелі оптимізації, такі як специфічні прапорці типів пам'яті або тонкі налаштування бар'єрів синхронізації. Крім того некомпетентне

проектування базової архітектури, позбавлене чіткої стратегії, може призвести до формування розмитої архітектури, яка намагаючись бути універсальною, стає неефективною та складною для масштабування.

В аспекті платформної інтеграції нативних обчислювальних систем в екосистему Android, корпорація Google впровадила спеціалізований інструментарій Android Game Development Kit, який виступає фундаментом для побудови високопродуктивних додатків. Ключовим архітектурним компонентом цього набору є бібліотека GameActivity, яка позиціонується як сучасна, еволюційна заміна застарілого класу NativeActivity [7]. Цей модуль виконує функцію низькорівневого інтерфейсного моста, що забезпечує інтероперабельність між керованим кодом середовища Java/Kotlin та нативним кодом C++, гарантуючи оптимізовану маршрутизацію системних подій життєвого циклу додатку безпосередньо у нативний простір.

Попри функціональну значущість, GameActivity надає виключно базові структурні примітиви для ініціалізації середовища виконання, фактично обмежуючись механізмом трансляції системних команд до нативного контролера. Суттєвим обмеженням даного підходу є відсутність регламентованої архітектурної моделі для організації Vulkan-підсистеми, що залишає розробнику задачу самостійного проектування логіки обробки цих подій. Бібліотека не пропонує вбудованого вирішення критичного архітектурного конфлікту, зумовленого агресивною політикою керування ресурсами ОС Android, де при переведенні додатку у фоновий режим відбувається примусове знищення об'єкта поверхні рендерингу VkSurfaceKHR. Ця специфіка диктує необхідність реалізації складних, відмовостійких алгоритмів динамічного знищення та повного відновлення ланцюжка обміну, що є критичним для забезпечення безперервності графічного контексту.

Аналіз високорівневих архітектурних патернів у контексті мобільної розробки вимагає чіткого розмежування сфер їх застосування, оскільки класичні шаблони проектування Android, такі як MVC, MVP, MVVM та MVI, перш за все орієнтовані на забезпечення модульності, тестованості та

масштабованості шарів користувацького інтерфейсу та бізнес-логіки. Серед них домінуючу позицію займає патерн Model-View-ViewModel, який забезпечує сувору сегрегацію презентаційної логіки від візуального відображення через впровадження проміжного компонента – ViewModel. Цей компонент працює як посередник, передаючи стан системи через потоки даних, на зміни в яких автоматично реагує інтерфейс користувача для миттєвого оновлення екрана. Поєднана з патерном Observer організація створює середовище для побудови реактивних інтерфейсів, де будь-які зміни в моделі даних миттєво та автоматично відображаються на екрані, при цьому забезпечуючи високу тестованість коду завдяки незалежності ViewModel від специфічних Android API.

Паралельно з цим набуває значного поширення архітектурний патерн Model-View-Intent, який демонструє високу сумісність із сучасними декларативними UI-фреймворками завдяки реалізації принципу односпрямованого потоку даних. В рамках цієї парадигми взаємодія користувача з системою формалізується у вигляді намірів, які ініціюють оновлення єдиного, незмінного та передбачуваного стану моделі, що згодом рендериться відображенням. Такий підхід гарантує детермінованість станів системи, що критично важливо для налагодження складних сценаріїв взаємодії, а також суттєво підвищує загальну масштабованість та модульність програмного забезпечення шляхом чіткої ізоляції побічних ефектів та мутацій стану.

Однак, незважаючи на беззаперечну ефективність цих високорівневих патернів у доменах керування бізнес-логікою та станом інтерфейсу, вони є недоцільними при спробі адаптації до завдань керування низькорівневими апаратними ресурсами Vulkan API. Фундаментальна проблема полягає у тому, що MVVM та MVI не враховують специфіку життєвого циклу графічного конвеєра та не надають механізмів для архітектурного розмежування ефемерних ресурсів презентації від стійких, інваріантних ресурсів. Ця архітектурна прогалина унеможливлює забезпечення належної відмовостійкості та

ефективного відновлення контексту рендерингу після системних подій життєвого циклу Android без створення спеціалізованого, нативного шаблону проектування. Такий шаблон повинен функціонувати на рівні системної інтеграції, використовуючи Android Game Development Kit як базовий інструментарій для прямого мосту між життєвим циклом ОС та графічним ядром.

Підсумовуючи вищевикладене, можна стверджувати, що на момент написання магістерської роботи інтерфейс Vulkan API зайняв позицію стандарту для розробки високошвидкісних графічних систем на платформі Android. Його здатність надавати прямий доступ до апаратних можливостей відеокарти є ключем до максимальної продуктивності, однак саме ця перевага стає головним викликом при інтеграції в мобільне середовище. Фундаментальна складність полягає у необхідності вручну керувати кожним аспектом роботи графічного процесора в умовах, коли операційна система Android постійно змінює стан вікна програми. Така невідповідність між жорсткими вимогами Vulkan та динамічною природою мобільної платформи створює високий бар'єр входження для розробників і часто призводить до нестабільної роботи програм, що не мають чіткої структури. Аналіз наявних технічних рішень засвідчив, що існуючі підходи здебільшого лише спрощують написання коду, але не пропонують надійної стратегії для вирішення конфлікту життєвих циклів. Саме цей технологічний розрив обумовлює критичну необхідність розробки спеціалізованого шаблону проектування, який зміг би гарантувати стабільність та ефективність керування ресурсами Vulkan у специфічних умовах Android.

1.2 Огляд і аналіз методів та засобів розробки шаблону проектування для Android-додатків з Vulkan API для вирішення проблеми дослідження

Історично так склалося, що розробники високопродуктивних ігор на Android намагалися максимально дистанціюватися від Java-середовища,

використовуючи NDK. Стандартним рішенням протягом багатьох років залишався клас NativeActivity. NativeActivity дозволяє додатку працювати повністю в нативному коді, де основний цикл виконується в окремому потоці, створеному допоміжною бібліотекою android_native_app_glue. Однак, NativeActivity є невід'ємною частиною фреймворку Android, це означає що будь-які помилки, обмеження або особливості реалізації цього класу жорстко прив'язані до версії операційної системи, встановленої на пристрої користувача. Розробник не може виправити баг у NativeActivity або додати нову функціональність, не чекаючи оновлення прошивки, що є критичним недоліком у екосистемі Android [8], крім того, NativeActivity ускладнює інтеграцію з сучасними UI-компонентами Android. Сучасні мобільні додатки часто потребують відображення WebView для угод користувача, рекламних банерів AdMob, або використання нативних діалогових вікон для покупок. Оскільки NativeActivity захоплює все вікно для рендерингу, накладання стандартних віджетів Android вимагає складних маніпуляцій з ієрархією View, що часто призводить до конфліктів фокусу та обробки дотиків [8].

Відповіддю на ці виклики стала поява бібліотеки GameActivity, яка є частиною Android Game Development Kit та розповсюджується через Jetpack. З точки зору шаблону проектування, використання GameActivity є безальтернативним стандартом для нових проектів на момент написання кваліфікаційної роботи.

Архітектурно GameActivity наслідує AppCompatActivity, що надає доступ до сучасних компонентів Android та забезпечує зворотну сумісність до старих версій Android. Ключова відмінність полягає у способі розповсюдження, GameActivity компілюється статично у додаток, що дозволяє розробникам отримувати виправлення помилок та нові функції кожні два тижні, незалежно від циклу оновлення ОС Android [8]. З точки зору рендерингу, GameActivity використовує SurfaceView, що дозволяє створювати композитні інтерфейси, де шар Vulkan-рендеру співіснує з шарами стандартного Android UI. Це вирішує проблему інтеграції клавіатур, полів вводу та системних діалогів. Для

розробника програмного засобу це означає зміну структури проекту: замість єдиної бібліотеки, що завантажується системою, додаток тепер будується як комбінація Java/Kotlin-частини, що керує GameActivity, та C++ бібліотеки, яка реалізує логіку рендерингу та ігрового процесу [8].

Управління ресурсами у Vulkan покладається на ручне створення та знищення об'єктів. Відсутність виклику деструктора неминуче призводить до витоків пам'яті, що на мобільних пристроях з обмеженим обсягом RAM швидко спричиняє аварійне завершення роботи через спрацювання механізму OOM Killer. Для вирішення цієї проблеми у мові C++ застосовується ідіома RAII, яка пов'язує життя ресурсу з областю видимості змінної.

Мобільні обчислювальні системи архітектурно спираються на концепцію уніфікованої пам'яті, в рамках якої центральний та графічний процесори спільно використовують єдиний фізичний пул оперативної пам'яті DRAM. Ця фундаментальна особливість відрізняє мобільні платформи від класичних десктопних рішень з дискретними відеокартами, що володіють виділеною відеопам'яттю VRAM. Незважаючи на фізичну монолітність пам'яті в UMA, специфікація Vulkan на Android реалізує логічне розмежування типів пам'яті для оптимізації доступу, виділяючи тип VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, який зазвичай репрезентує пам'ять, що кешується графічним процесором, та тип VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT, який дозволяє центральному процесору відображати пам'ять у свій адресний простір для запису даних.

У контексті оптимізації передачі даних, класичний метод завантаження текстур, який передбачає створення проміжного буфера з подальшим копіюванням даних командою vkCmdCopyBufferToImage, створює надлишкову подвійну копію даних, що є неефективним для UMA-архітектур. Впровадження розширення VK_EXT_host_image_copy, яке увійшло до стандарту Vulkan 1.4, вирішує цю проблему, надаючи центральному процесору можливість записувати дані безпосередньо у лінійну або тайлову структуру зображення через оптимізовану функцію vkCopyMemoryToImageEXT. Такий підхід

дозволяє суттєво зекономити обсяг оперативної пам'яті та знизити навантаження на пропускну здатність шини даних шляхом усунення зайвих транзакцій копіювання [9].

Для забезпечення ефективного керування алокаціями в умовах системних обмежень індустріальним стандартом де-факто визнано використання бібліотеки Vulkan Memory Allocator. Цей інструмент вирішує проблему жорсткого ліміту драйвера на кількість об'єктів пам'яті шляхом застосування стратегії субалокації, де бібліотека запитує великі блоки пам'яті та самостійно фрагментує їх для окремих ресурсів, що є критичною умовою для забезпечення стабільності роботи Android-додатків [10].

Процес інженерії складних програмних систем для візуалізації вимагає імплементації багаторівневих засобів контролю якості та аналізу, що обумовлено архітектурною парадигмою Vulkan API, де відповідальність за валідацію даних перекладена з драйвера на розробника заради мінімізації накладних витрат процесора. У екосистемі Android це завдання вирішується шляхом інтеграції розвиненого інструментарію діагностики, фундаментом якого є валідаційні шари, зокрема VK_LAYER_KHRONOS_validation. Цей програмний компонент, що підключається до ланцюжка викликів API як частина APK або завантажується динамічно, виконує перехоплення команд та їх верифікацію на відповідність специфікації у реальному часі [11].

Емпірична верифікація продуктивності та пошук архітектурних вузьких місць здійснюються за допомогою спеціалізованого програмного комплексу Android GPU Inspector, який є індустріальним стандартом профілювання від Google. Цей інструмент дозволяє проводити детальний моніторинг апаратних лічильників, аналізуючи рівень завантаженості виконавчих блоків GPU, ефективність використання пропускної здатності шини пам'яті та часові характеристики взаємодії між потоками центрального процесора та чергами команд GPU. Використання AGI є незамінним етапом для виявлення системних аномалій, таких як термальний тротлінг, що призводить до динамічного зниження тактових частот та падіння продуктивності [12].

1.3 Постановка завдання на кваліфікаційну роботу магістра

Актуальність дослідження зумовлена об'єктивними обмеженнями існуючих підходів до розробки високопродуктивних графічних програмних засобів на платформі Android, що використовують низькорівневий API Vulkan. Тому основними завданнями на кваліфікаційну роботу є:

- проведення критичного аналіу існуючих архітектурних рішень та інструментів нативної розробки для виявлення проблем у керуванні низькорівневими ресурсами;
- розробка та обґрунтування шаблону проектування, який забезпечує модульність та відмовостійкість графічної системи;
- створення функціонального програмного забезпечення на базі розробленого шаблону та монолітну версію для порівняння;
- проведення валідації та тестування розроблених програмних забезпечень;
- аналіз отриманих результатів та формулювання висновків щодо ефективності розробленого шаблону порівняно з монолітним підходом.

РОЗДІЛ 2
ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ ШАБЛОНУ ПРОЕКТУВАННЯ ДЛЯ ANDROID ДОДАТКІВ З VULKAN API

2.1 Обґрунтування вибору шляхів, технологій, алгоритмів і засобів вирішення поставленого завдання

Необхідність розробки нового шаблону зумовлена технічним конфілктом між стабільною моделлю роботи Vulkan та динамічною природою операційної системи Android. Vulkan функціонує як інструмент з повним ручним керуванням, де створення базових компонентів, таких як логічний пристрій або графічні конвеєри, є вкрай ресурсомістким процесом що вимагає значного часу процесора на компіляцію шейдерів та виділення пам'яті. Натомість Android сприймає вікно додатка як тимчасовий елемент, миттєво знищуючи поверхню рендерингу при звичайному повороті екрана або зміні системної теми. У класичній реалізації це призводить до ситуації, коли фундаментальні ресурси, розраховані на весь час роботи програми, вимушено перестворюються через тривіальні зміни інтерфейсу. Це не лише спричиняє візуальні артефакти та затримки але й значно підвищує енергоспоживання мобільного пристрою через надлишкові цикли процесора.

В архітектурах із жорстким зв'язуванням логіки та презентації ця невідповідність змушує програму постійно перезавантажувати все графічне ядро слідом за змінами вікна, що не лише критично знижує продуктивність через повторні обчислення, але й провокує аварійні завершення роботи через спроби запису у вже неіснуючі буфери. Щоб уникнути цього сценарію, розроблений шаблон повинен впровадити механізм, який розриває жорстку залежність між часом життя графічних ресурсів та станом екрана. Головна ідея полягає у чіткому поділі всіх об'єктів на дві незалежні групи: статичне ядро (стан асетів) та динамічну оболонку (стан презентації). Це дозволить зберігати важке ядро системи в оперативній пам'яті навіть під час перезапуску візуальної

оболонки, гарантуючи стабільність та швидкодію. Ці групи об'єктів зображені на таблиці 2.1.

Таблиця 2.1 – дві головні групи об'єктів у розроблюваному шаблоні проектування

Характеристика	Стан асетів	Стан презентації
Життєвий цикл	Тривалий (весь час життя процесу LifecycleManager)	Ефемерний (залежить від APP_CMD_INIT_WINDOW / TERM_WINDOW)
Ключові об'єкти	VkPipeline, VkBuffer, VkImage, VkDescriptorSetLayout	VkSwapchainKHR, VkFramebuffer, VkImageView, VkSemaphore
Поведінка при паузі	Залишаються в пам'яті	Знищуються та звільняють дескриптори вікон

Підтримка такої архітектури на рівні коду має реалізуватись через суворий поділ файлів на інтерфейс та реалізацію. У заголовкових файлах опис має лише чисті правила взаємодії класів, без підключення громіздких бібліотек Vulkan, що робить їх легкими та зрозумілими. Уся ж інша робота та важкі команди драйвера мають ховатись у файлах реалізації. Такий підхід дозволить значно прискорити час компіляції проекту та буде гарантувати, що зміни у внутрішніх налаштуваннях графіки не вимагатимуть перезбірки всієї логіки мобільного додатку.

У новому шаблоні проектування центральним елементом управління повинен виступати модуль LifecycleManager, який виконує функцію головного координатора системи. Цей компонент має слугувати єдиною точкою входу для всіх системних сигналів операційної системи Android, що надходять через інтерфейси android_main або GameActivity. Основне призначення менеджера повинно полягати у перехопленні технічних повідомлень системи та їх трансляції у логічно завершені команди для підсистем рендерингу, що дозволяє ізолювати графічне ядро від специфіки платформи. Функціонування менеджера базується на логіці складного скінченного автомата, який чітко регламентує послідовність дій при зміні станів додатка. Для гарантування детермінованої

поведінки програмного комплексу в архітектурі реалізовано скінченний автомат, що базується на трьох фундаментальних станах та строго регламентованих умовах переходу між ними. Початковим етапом функціонування є стан Core Initialized, вхід у який відбувається автоматично при запуску процесу додатку та успішному завершенні конструктора LifecycleManager. У цьому режимі система виконує одноразову ініціалізацію базових компонентів VulkanCore та ResourceManager, завантажуючи в оперативну пам'ять інваріантні ресурси стан асетів та кеш конвеєрів. Характерною рисою цього стану є повна готовність графічного пристрою до обчислень при відсутності активної поверхні для виводу зображення, що переводить цикл обробки подій у режим очікування системних команд. Перехід до стану Rendering Active ініціюється отриманням системної команди APP_CMD_INIT_WINDOW, що сигналізує про готовність віконної підсистеми. У відповідь на цю подію архітектура створює абстракцію поверхні VkSurfaceKHR та розгортає SwapchainManager, формуючи стан презентації, при цьому тайм-аут опитувача подій ALooper встановлюється на нульове значення для забезпечення максимальної пропускної здатності кадрового циклу та безперервної роботи методу RenderEngine::drawFrame. При отриманні команди APP_CMD_TERM_WINDOW, зумовленої згортанням додатку або зміною орієнтації екрана, система переходить у стан Suspended, що супроводжується виконанням блокуючого виклику vkDeviceWaitIdle для гарантованої синхронізації GPU. Цей процес передбачає деструкцію виключно об'єктів стану презентації зі збереженням стану асетів у пам'яті та зміною тайм-ауту ALooper на -1, що переводить центральний процесор у режим глибокого енергозбереження до моменту відновлення вікна або завершення процесу.

На етапі первинної ініціалізації має відбутись одноразове створення фундаментальних об'єктів ядра VulkanCore, зокрема VkInstance та логічного пристрою VkDevice. Паралельно з цим ініціалізується менеджер ресурсів ResourceManager, який негайно завантажує з диска попередньо збережений кеш конвеєрів. Ця операція є критичною, оскільки дозволяє уникнути повторної

компіляції шейдерів і значно прискорити холодний старт програми, при цьому створені об'єкти залишаються в пам'яті до повного завершення роботи процесу.

При отриманні сигналу про створення вікна, система переходить до формування стану презентації. Менеджер передає отриманий від Android вказівник на нативне вікно у ядро Vulkan для створення абстракції поверхні VkSurfaceKHR. Після цього активується SwapchainManager, який налаштовує ланцюжок обміну зображеннями і запускається основний цикл рендерингу. Така послідовність гарантує, що графічний вивід почнеться лише тоді, коли екран повністю готовий до відображення.

Найбільш відповідальним моментом є обробка втрати вікна, яка вимагає суворої синхронізації для запобігання аваріям. Перед будь-якими діями LifecycleManager повинен примусово зупинити роботу графічного процесора викликом vkDeviceWaitIdle. Це гарантує, що GPU завершив усі поточні задачі та не звертається до пам'яті, яку планується звільнити. Лише після повної зупинки відбувається безпечне знищення ефемерних об'єктів – ланцюжка обміну та поверхні рендерингу. При цьому критично важливо, що ResourceManager та VulkanCore залишаються недоторканими в оперативній пам'яті, що дозволяє додатку миттєво відновити роботу при поверненні користувача без необхідності повторного завантаження ресурсів.

Модуль VulkanCore виступає фундаментальним базисом розроблюваного шаблону проектування, виконуючи роль незмінного контейнера для глобальних об'єктів Vulkan, що забезпечує стабільність графічного контексту незалежно від зовнішніх збурень. У рамках реалізації на рівні Data, цей компонент відповідає за створення та утримання ресурсномістких констант системи, ізолюючи їх від циклічних перезапусків активності Android. Критичним етапом ініціалізації є алгоритмічний вибір фізичного пристрою, який базується на аналізі доступного апаратного забезпечення. Хоча в мобільному сегменті зазвичай доступний єдиний інтегрований GPU, код зобов'язаний верифікувати його відповідність системним вимогам, перевіряючи підтримку обов'язкових розширень, таких як VK_KHR_swapchain для роботи з ланцюжком обміну та

VK_KHR_android_surface для інтеграції з віконною системою, а також апаратні ліміти, наприклад maxImageDimension2D [13].

Для забезпечення максимальної ефективності паралельних обчислень, досліджуваний шаблон повинен вимагати суворої сегрегації потоків виконання на рівні апаратних черг. VulkanCore має здійснювати детальний аналіз властивостей сімейств черг GPU для ідентифікації та ініціалізації окремих індексів для графічних операцій, презентації зображення на екран та асинхронних обчислень. Таке розділення дозволяє рушію рендерингу ефективно розподіляти навантаження, уникаючи блокування графічного конвеєра важкими обчислювальними задачами або очікуванням вертикальної синхронізації [14].

Невід'ємною складовою інфраструктури є допоміжний клас Utils, розташований у шарі реалізації, який забезпечує критично важливі сервісні функції для конфігурації та діагностики VulkanCore. Зокрема він відповідає за інтеграцію шарів валідації через налаштування дебаг-месенджера vkCreateDebugUtilsMessengerEXT, що перехоплює повідомлення драйвера та помилки API. Цей механізм транслює внутрішні логи Vulkan безпосередньо у системний журнал Android Logcat, надаючи розробнику можливість відстежувати некоректну поведінку та помилки синхронізації в реальному часі безпосередньо в середовищі розробки [15]. Додатково методи класу Utils виконують превентивну перевірку сумісності середовища через функцію checkDeviceExtensionSupport, яка гарантує наявність специфічних розширень, таких як VK_KHR_maintenance, необхідних для апаратної інверсії системи координат та коректного відображення сцени на екранах мобільних пристроїв.

Модуль ResourceManager у структурі шаблону проектування виконує функцію гаранта стабільності та цілісності стану асетів, забезпечуючи ефективне керування незмінними ресурсами графічної підсистеми. На рівні доменної логіки, клас надає високорівневі інтерфейси, такі як loadTexture або loadModel, які повертають оптимізовані дескриптори, повністю інкапсулюючи низькорівневу складність роботи з об'єктами VkImage та VkDeviceMemory.

Технічна реалізація менеджера у шарі даних передбачає використання спеціалізованих стратегій алокації пам'яті, зокрема інтеграцію бібліотеки Vulkan Memory Allocator для уникнення фрагментації. Процес завантаження ресурсів реалізується через механізм проміжних буферів, розташованих у пам'яті, видимій для CPU. ResourceManager спочатку розміщує дані у тимчасовому буфері, після чого формує команду асинхронного трансферу vkCmdCopyBufferToImage у локальну пам'ять пристрою, забезпечуючи когерентність даних за допомогою відповідних бар'єрів пам'яті. Для запобігання блокуванню основного потоку рендерингу, операції завантаження виконуються через виділену чергу передачі, що дозволяє досягти високого рівня паралелізму між обчисленнями CPU та DMA-операціями GPU.

Критичним елементом оптимізації часу запуску додатку у розроблюваному шаблоні преоктування є вбудований механізм керування кешем конвеєрів. Оскільки створення об'єктів VkPipeline супроводжується ресурсомісткою компіляцією шейдерів, ResourceManager може реалізувати алгоритми серіалізації бінарних даних кешу у постійне сховище пристрою, а саме внутрішню директорію Android-контексту, при завершенні роботи. Під час наступної ініціалізації системи LifecycleManager викликає процедуру десеріалізації, передаючи збережений бінарний блоб у структуру VkPipelineCacheCreateInfo. Це дозволяє драйверу Vulkan виконати перевірку хеш-суми і у разі валідності кешу пропустити етап компіляції, що, в теорії, може забезпечити скорочення часу ініціалізації графічної підсистеми.

Модуль SwapchainManager має функціонувати як найбільш динамічний та чутливий компонент нового шаблону, оскільки він зобов'язаний адаптивно реагувати на будь-які зміни фізичних параметрів поверхні рендерингу, забезпечуючи цілісність стану презентації. Процедура перестворення ланцюжка обміну, яка імплементується у методі recreateSwapchain, базується на алгоритмі суворої синхронізації для запобігання гонитві даних. Першим критичним кроком є виклик функції vkDeviceWaitIdle, що переводить пристрій у режим повного простою, незважаючи на певну ресурсномісткість цієї операції, в

контексті мобільних платформ вона є необхідним компромісом, який гарантує нульовий рівень помилок типу VK_ERROR_DEVICE_LOST під час зміни орієнтації екрана. Після стабілізації стану драйвера відбувається опитування актуальних геометричних параметрів поверхні через vkGetPhysicalDeviceSurfaceCapabilitiesKHR, оскільки на пристроях Android розмір області рендерингу може динамічно варіюватися залежно від наявності системних панелей навігації або вирізів дисплея. Ініціалізація нового об'єкта VkSwapchainKHR передбачає передачу дескриптора попереднього ланцюжка як параметра oldSwapchain, що дозволяє драйверу оптимізувати повторне використання внутрішніх ресурсів пам'яті перед остаточним оновленням залежних VkImageView та VkFramebuffer.

Модуль RenderEngine у складі нового шаблону виступає виконавчим ядром графічної підсистеми, реалізуючи фундаментальний цикл побудови зображення drawFrame, який архітектурно суворо відокремлений від високорівневої ігрової логіки або бізнес-правил додатку. Для забезпечення максимальної утилізації обчислювальних потужностей GPU та мінімізації простоїв конвеєра, рушій імплементує стратегію паралельної обробки кадрів, використовуючи механізми подвійної або потрійної буферизації ресурсів синхронізації. Цей підхід базується на каскадному використанні примітивів VkSemaphore для внутрішньої синхронізації черг GPU та VkFence для координації доступу між CPU та GPU. Процес рендерингу розпочинається з виклику vkWaitForFences, який блокує центральний процесор до моменту завершення обробки попередніх кадрів, гарантуючи безпеку запису нових команд у командні буфери без ризику перезапису даних, що ще використовуються графічним процесором. Критичним етапом циклу є запит доступного індексу зображення з ланцюжка обміну через функцію vkAcquireNextImageKHR. Саме на цьому етапі RenderEngine повинен демонструвати свою надійність. У випадку повернення коду помилки VK_ERROR_OUT_OF_DATE_KHR, що сигналізує про невідповідність параметрів поверхні, рушій не має аварійно зупиняється, а повинен ініціювати

процедуру самовідновлення. Він буде викликати метод SwapchainManager::recreate для перестворення ланцюжка обміну та пропускати поточний кадр, відновлюючи стабільну роботу вже у наступному циклі. Після успішного отримання зображення має відбуватися відправка буфера команд на виконання та фінальний запит на презентацію результату, що завершує логічний цикл кадру.

Математичне забезпечення сцени та керування проекційними перетвореннями покладено на клас Camera, який на рівні доменної логіки надає абстракції для маніпулювання видовими матрицями та параметрами перспективи. На рівні роботи з даними цей модуль має реалізувати високоефективний механізм оновлення уніформ-буферів через техніку персистентного мапінгу. Замість витратних системних викликів vkMapMemory та vkUnmapMemory у кожному кадрі, що створюють значні накладні витрати через взаємодію з ядром ОС та сторінками віртуальної пам'яті, пам'ять мапується єдиноразово при ініціалізації. Оновлення матриць у кожному кадрі здійснюється через пряме копіювання пам'яті за отриманим вказівником, що забезпечує максимальну пропускну здатність шини даних. Окрім того, клас Camera виконує важливу функцію адаптації координатних просторів, інкапсулюючи фундаментальні відмінності між стандартами OpenGL та Vulkan. Оскільки Vulkan використовує праву систему координат з віссю Y, спрямованою вниз, та діапазоном глибини від 0 до 1, на відміну від OpenGL, де Y вгору, а глибина від -1 до 1, пряме використання стандартних математичних бібліотек може призвести до інвертованого зображення або некоректного відсікання геометрії. Camera автоматично інтегрує у проекційну матрицю спеціальну коригуючу матрицю, роблячи цю низькорівневу особливість прозорою для розробника та забезпечуючи коректну візуалізацію сцени без необхідності ручного втручання у шейдерний код.

З метою забезпечення комплексної візуалізації та формальної верифікації розроблюваного шаблону проектування, у рамках інженерного проектування було розроблено набір графічних моделей уніфікованою мовою моделювання,

який складається з двох схем: структурної діаграми класів та поведінкової діаграми послідовності.

Аналіз структури шаблону, представленої на першій UML-діаграмі, зображеній на рисунку 2.1, демонструє чіткий поділ програми на п’ять основних модулів.

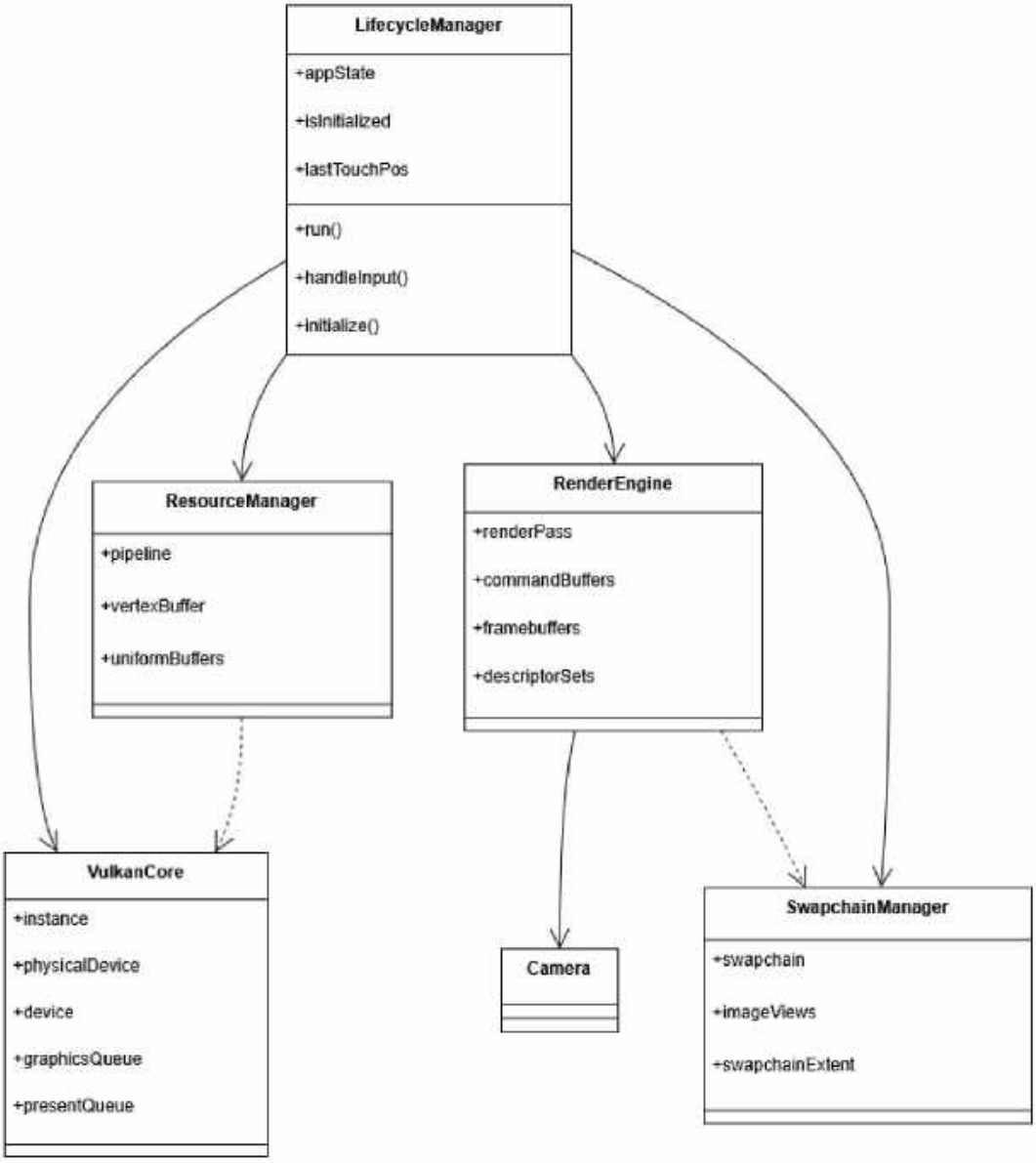

Рисунок 2.1 – UML діаграма класів нового шаблону проектування

Головним керуючим елементом системи виступає клас LifecycleManager, який виконує роль диригента. Він відповідає за загальний стан додатка,

зберігаючи посилання на нативне середовище у полі appState та контролюючи процес запуску через прапорець isInitialized. Саме цей клас містить головний цикл програми та методи для обробки дотиків екрана, координуючи роботу всіх інших компонентів системи. Фундаментом для графіки слугує клас VulkanCore. Його завдання – один раз на старті налаштувати зв'язок з відеокартою. Він зберігає найважливіші системні об'єкти: екземпляр Vulkan, обраний фізичний графічний чіп та логічний пристрій. Також він відповідає за доступ до черг команд, через які відправляються завдання на виконання. Далі архітектура розділяється на два напрямки керування даними. Клас SwapchainManager опікується тимчасовими ресурсами, які залежать від екрана і які потрібно перестворювати при повороті девайса. Натомість клас ResourceManager відповідає за константи – 3D-моделі та налаштування графічного конвеєра, які завантажуються в пам'ять один раз і не зникають при зміні розміру вікна.

Безпосереднім малюванням займається клас RenderEngine. Він відокремлений від системної рутини і зосереджений суто на формуванні зображення. Цей модуль використовує налаштування проходу рендерингу та записує команди малювання у спеціальні буфери. Логіка зв'язків тут вибудувана ієрархічно: LifecycleManager керує всіма менеджерами, менеджери ресурсів використовують VulkanCore для доступу до фізичного пристрою, а RenderEngine збирає все разом – бере пристрій від ядра, моделі від менеджера ресурсів та картинки від менеджера ланцюжка обміну, щоб намалювати кадр. Додатково використовується об'єкт Camera, який допомагає RenderEngine правильно розрахувати кут огляду та перспективу для коректного відображення сцени.

Для детального аналізу алгоритмів роботи системи на рисунку 2.2 наведено діаграму послідовності. Вона слугує ключем до розуміння динамічної моделі, ілюструючи складну оркестрацію процесів ініціалізації та рендерингу. Діаграма візуалізує потік керування, демонструючи, як ключові компоненти шаблону координують свої дії у відповідь на зовнішні події життєвого циклу, та

пояснює механізм забезпечення стабільності роботи графічного ядра на етапі виконання.

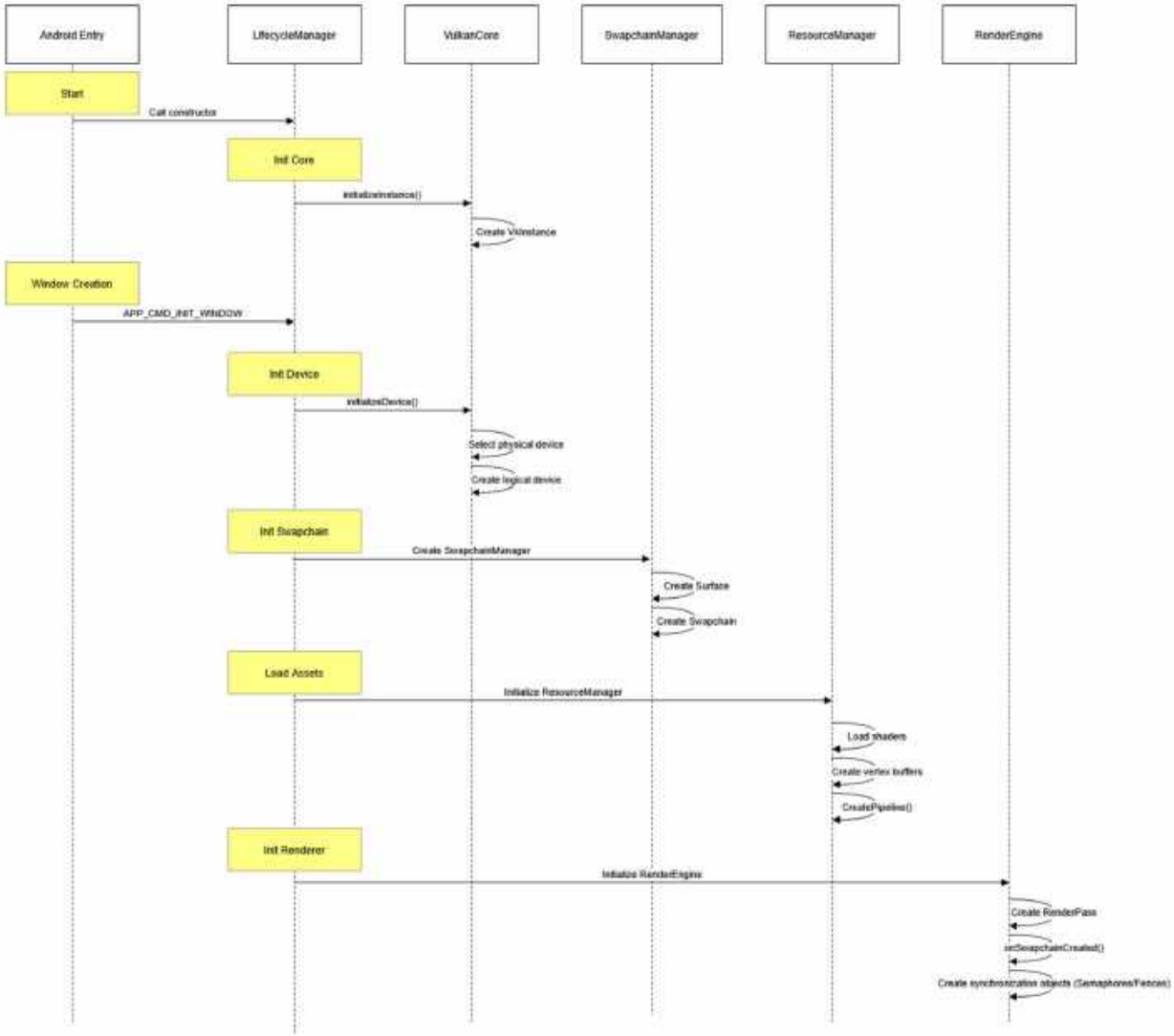

Рисунок 2.2 – UML діаграма послідовностей нового шаблону проектування

Процес розгортання системи розпочинається з точки входу Android, яка ініціює виклик конструктора головного керуючого компонента – LifecycleManager. Першим кроком цього менеджера є запуск ядра системи через ініціалізацію VulkanCore, що включає виклик методу initializeInstance() для створення базового об’єкта VkInstance та налаштування глобального контексту Vulkan API.

Наступний етап є визначальним для інтеграції графічного конвеєра з віконною підсистемою мобільної платформи та розпочинається з отримання системної команди APP_CMD_INIT_WINDOW. Ця подія передає додатку нативний дескриптор вікна, наявність якого дозволяє LifecycleManager перейти до ініціалізації логічного пристрою шляхом виклику методу initializeDevice. У рамках цієї процедури відбувається вибір оптимального фізичного пристрою, створення абстракції логічного пристрою та отримання дескрипторів апаратних черг команд, необхідних для подальшого виконання графічних операцій. Після успішного налаштування пристрою архітектура розгалужується на ініціалізацію підсистем презентації та керування ресурсами. LifecycleManager створює екземпляр SwapchainManager, який відповідає за формування стану презентації: створення поверхні рендерингу та безпосередньо ланцюжка обміну. Паралельно з цим ініціалізується ResourceManager, який завантажує стан асетів, зчитуючи шейдерні програми у форматі SPIR-V, створюючи вершинні буфери та компілюючи графічний конвеєр. Важливо зазначити, що для оптимізації цього ресурсомісткого процесу ResourceManager зобов'язаний використовувати механізм VkPipelineCache, що дозволяє значно прискорити старт додатка. Фінальна стадія ініціалізації присвячена налаштуванню виконавчого механізму RenderEngine та об'єктів синхронізації. LifecycleManager активує рушій рендерингу, який створює об'єкт проходу рендерингу та, реагуючи на подію onSwapchainCreated, конфігурує залежні від екрана ресурси, такі як фреймбуфери. Одночасно створюються примітиви синхронізації – семафори та огорожі, які є критично необхідними для забезпечення коректної черговості виконання команд між центральним та графічним процесорами, запобігаючи станам гонитви. Таким чином, детальний розгляд діаграми верифікує критичну роль LifecycleManager у забезпеченні стабільності застосунку. Діючи як центральний координатор, цей клас вибудовує ієрархічний ланцюжок ініціалізації, що унеможливлює виникнення звернення до неініціалізованої пам'яті. Завдяки цьому досягається безпечна інтеграція різнорідних

компонентів системи в єдиний конвеєр, де кожен модуль набуває активного стану лише після успішної підготовки його залежностей.

2.2 Практична реалізація об'єкта проектування

Технологічний базис розробки спирається на парадигми суворої модульності, сегрегації відповідальності та мінімізації накладних витрат центрального процесора. У якості фундаментального інструментарію для платформної інтеграції імплементовано Android Game Development Kit, де ключовим компонентом виступає бібліотека GameActivity. Цей модуль функціонує як низькорівневий комунікаційний міст, що гарантує надійну трансляцію системних сигналів, зокрема команд ініціалізації та деструкції віконного контексту у простір нативного C++ коду через механізм android_native_app_glue.

Для забезпечення заявлених показників модульності та практичної імплементації архітектурної парадигми сегрегації ресурсів, фізична архітектура програмного проекту була структурована у форматі ієрархічної системи дискретних одиниць трансляції. Вихідний код проекту піддався логічному діленню на три функціональні рівні. Перший рівень, що відповідає за платформну оркестрацію, представлений файлами LifecycleManager.h та LifecycleManager.cpp.

Даний модуль містить точку входу android_main та повністю інкапсулює логіку низькорівневої взаємодії з операційною системою через бібліотеку GameActivity.

Рівень графічного ядра, імплементований у файлах VulkanCore.cpp та RenderEngine.cpp, несе відповідальність за ініціалізацію фундаментальних об'єктів, таких як VkInstance та VkDevice, а також за безпосереднє виконання циклу рендерингу. Третій рівень, а саме керування ресурсами реалізовано через спеціалізовані модулі ResourceManager.cpp для адміністрування довготривалих активів та SwapchainManager.cpp для керування ефемерними ресурсами

презентації. Зазначена організація файлової системи, на відміну від монолітних архітектур, базується на суворому розмежуванні заголовкових файлів, що містять виключно декларації інтерфейсів, та файлів реалізації, де приховані ресурсомісткі підключення бібліотек Vulkan API. Такий підхід дозволяє ефективно зменшити ризик виникнення циклічних залежностей та оптимізувати час компіляції, хоча й формує розгалужену систему перехресних посилань, яка вимагає керування передачею вказівників між модулями для забезпечення їхньої когерентної взаємодії.

Важливо підкреслити, що оскільки GameActivity надає виключно базові структурні примітиви без визначення високорівневої логіки, реалізація нового шаблону проектування фокусується на надбудові спеціалізованого керуючого шару, здатного ефективно інтерпретувати ці події. Центральною архітектурною гіпотезою та ключовим методичним рішенням при цьому виступає формалізована різниця графічних ресурсів Vulkan. Система реалізує чітке розмежування між станом презентації та станом асетів. Перша категорія охоплює ефемерні об'єкти, такі як ланцюжки обміну та фреймбуфери, життєвий цикл яких жорстко синхронізовано з фізичною поверхнею рендерингу, що вимагає їх динамічного перестворення при будь-яких змінах конфігурації вікна. Натомість категорія стану асетів включає обчислювально вартісні ресурси – шейдерні модулі, геометричні буфери, текстури та кеш конвеєрів, які інкапсулюються у захищеному сховищі та зберігають свою валідність незалежно від варіативності віконної підсистеми, забезпечуючи тим самим стабільність виконання та швидке відновлення контексту.

Фізична реалізація розробленого шаблону проектування втілена у наборі вихідних файлів проекту – LifecycleManager.cpp, ResourceManager.cpp, SwapchainManager.cpp та VulkanCore.cpp, які сукупно формують виконавчу базу системи. Інтеграція цього програмного комплексу з операційним середовищем виконується за допомогою інструментарію Android Game Development Kit, де використання бібліотеки GameActivity та нативних Android-бібліотек

забезпечує необхідний рівень кросплатформної сумісності та низькорівневого доступу до апаратних ресурсів.

Ініціалізація взаємодії з операційною системою Android реалізується через точку входу android_main, розташовану у файлі реалізації LifecycleManager.cpp, яка виступає мостом між середовищем виконання Android Runtime та нативним C++ кодом додатку. Архітектурна модель відмовляється від використання стандартного циклу керування на базі Java-активності на користь прямої ініціалізації екземпляра класу LifecycleManager, якому як критична залежність передається покажчик на структуру стану android_app, після чого управління передається методу run(), що інкапсулює логіку головного виконавчого циклу. Алгоритмічний базис цього циклу побудований на механізмі безперервного опитування черги системних подій із використанням нативного інтерфейсу ALooper.

Ключовим елементом оптимізації продуктивності у цьому процесі є застосування функції ALooper_pollOnce зі стратегією динамічної адаптації тайм-ауту, яка дозволяє балансувати між чутливістю інтерфейсу та енергоспоживанням. Логіка керування визначає, що у стані готовності графічної підсистеми до рендерингу тайм-аут встановлюється у нульове значення, перемикаючи систему в неблокуючий режим для забезпечення максимальної частоти кадрів, тоді як у стані простою параметр змінюється на -1, переводячи потік у режим очікування переривань для збереження заряду батареї. Маршрутизація отриманих подій здійснюється через виклик колбек-функції process, яка транслює низькорівневі системні сигнали у значущі команди методу handleAppCmd, де застосовується switch-конструкція для суворого контролю переходів між станами скінченного автомата додатку, фрагмент цього коду зображений у лістингу 2.1.

Лістинг 2.1 – Реалізація обробки системних команд життєвого циклу у методі handleAppCmd

```
void LifecycleManager::handleAppCmd(int32_t cmd) {
    switch (cmd) {
```

```
            case APP_CMD_INIT_WINDOW:
                //...
                break;
            case APP_CMD_TERM_WINDOW:
                //...
                renderEngine->onSwapchainDestroyed();
                swapchainManager.reset();
                isReady = false;
                break;
    }
}
```

Кінець лістингу 2.1

Паралельно з обробкою команд життєвого циклу, архітектура реалізує високочастотний конвеєр обробки користувацького вводу, використовуючи механізм подвійної буферизації android_app_swap_input_buffers для забезпечення атомарності доступу до даних. У цьому процесі сирі події сенсорного екрану, такі як AMOTION_EVENT_ACTION_MOVE, підлягають миттєвій математичній трансформації у векторні дельти зміщення. Ці обчислені значення передаються безпосередньо в інтерфейс processTouch, що дозволяє досягти чіткого архітектурного розмежування між низькорівневим шаром драйвера вводу та високорівневою логікою маніпуляції камерою або об'єктами сцени.

Створений файл VulkanCore відповідає за фундамент графічної системи. Процес починається зі створення головного об'єкта VkInstance. На відміну від програм для ПК, тут система чітко вказує, що працює саме на Android, автоматично підключаючи необхідні інструменти для виведення зображення на мобільний екран. Також на цьому етапі, підключається система пошуку помилок, яка допомагає виявляти проблеми в коді ще до того, як вони призведуть до збою. Наступним кроком є вибір фізичного обладнання – конкретного графічного чіпа, на якому будуть виконуватися обчислення. Замість того, щоб сліпо використовувати перший знайдений пристрій, алгоритм перевіряє всі доступні варіанти на відповідність вимогам. Він шукає відеокарту, яка гарантовано вміє працювати з графічними чергами та підтримує виведення картинки на дисплей. Лише після успішної перевірки система переходить до

створення логічного пристрою. На фінальному етапі обраний фізичний чіп перетворюється на програмний інтерфейс – логічний пристрій VkDevice. Система налаштовує спеціальні канали зв'язку для малювання графіки та показу зображення на екрані. Посилання на ці черги зберігаються в пам'яті, завдяки чому інші частини програми, такі як менеджер ресурсів чи двигун рендерингу, можуть миттєво відправляти завдання на відеокарту, не витрачаючи час на повторний пошук каналів зв'язку.

Файл ResourceManager відповідає за ефективну взаємодію з пам'яттю відеокарти, шукаючи баланс між швидкістю завантаження ресурсів та продуктивністю їх відображення. Для передачі ресурсомістких даних, таких як 3D-геометрія, використовується стратегія проміжного буфера, відома. Оскільки центральний процесор фізично не може напряму писати у надшвидку відеопам'ять, система спершу копіює дані у доступну системну пам'ять. Одразу після цього формується команда для графічного процесора, яка змушує його самостійно перенести ці дані у свою локальну пам'ять. Такий підхід хоч і виглядає складнішим, гарантує, що під час малювання відеокарта матиме миттєвий доступ до інформації без затримок. Процес підготовки графічного конвеєра реалізовано як чітку послідовність кроків. Спершу система зчитує бінарний код шейдерів і запаковує їх у програмні модулі. Далі налаштовуються фіксовані параметри, такі як правила змішування кольорів або порядок з'єднання вершин. Важливим архітектурним рішенням є використання динамічних станів для розміру області малювання: це дозволяє змінювати розмір вікна додатка без необхідності повністю руйнувати та перестворювати весь складний графічний конвеєр. Для прискорення програми задіяно механізм кешування, система намагається завантажити вже скомпільовані драйвером дані з диска, що дозволяє пропустити ресурсомісткий етап обробки шейдерів при повторних запусках. Керування доступом шейдерів до змінних даних, таких як матриці повороту камери, здійснюється через систему наборів дескрипторів. Цей механізм працює як контракт інтерфейсу, він повідомляє відеокарті, за якою адресою знаходитиметься буфер з актуальними даними. Це дає

можливість оновлювати положення об'єктів у просторі, просто перезаписуючи дані в пам'яті, без необхідності втручатися в логіку роботи самих шейдерних програм.

RenderEngine функціонує як головне виконавче ядро системи, реалізуючи високоефективний алгоритм подвійної буферизації. Ця архітектурна парадигма дозволяє розпаралелити обчислювальні процеси центрального та графічного процесорів. Використання константи MAX_FRAMES_IN_FLIGHT забезпечує конвеєрний режим роботи: поки відеокарта завершує рендеринг поточного кадру, процесор вже готує команди для наступного, що мінімізує час простою обладнання. Метод drawFrame регламентує суворий порядок виконання графічних операцій, базуючись на використанні примітивів синхронізації Vulkan для координації взаємодії між центральним та графічним процесорами. На початковому етапі циклу виконується примусове блокування хост-потоку за допомогою функції vkWaitForFences, що виступає гарантом повного звільнення ресурсів поточного кадру від попередніх обчислювальних навантажень та забезпечує безпеку пам'яті перед початком нових операцій запису. Після розблокування ініціюється процедура асинхронного отримання індексу наступного доступного зображення з ланцюжка обміну через виклик vkAcquireNextImageKHR, при цьому часова координація готовності поверхні до рендерингу делегується семафорам, які сигналізують GPU про можливість безпечного початку малювання. Завершується дана послідовність передачею сформованого командного буфера безпосередньо у чергу виконання графічного пристрою за допомогою функції vkQueueSubmit, що інтегрує підготовлені команди у загальний потік обробки. Процедура recordCommandBuffer ініціюється для кожного кадру окремо, що дозволяє підтримувати гнучкість графічного конвеєра. На відміну від статичних методів, де всі параметри фіксуються заздалегідь, даний алгоритм, який зображено в лістингу 2.2, активно використовує механізм динамічних станів для налаштування області перегляду та відсікання.

Лістинг 2.2 – Динамічне налаштування області перегляду під час запису командного буфера

```
VkViewport viewport{};
viewport.width = static_cast<float>(swapchainManager->getExtent().width);
//...
vkCmdSetViewport(commandBuffer, 0, 1, &viewport);
```

Кінець лістингу 2.2

Такий підхід дозволяє адаптувати зображення під зміну розміру вікна або орієнтації пристрою відразу, без необхідності виконувати ресурсомістку перезбірку об'єкта VkPipeline, що є критичним фактором для забезпечення плавності роботи на мобільних платформах.

Взаємодія між компонентами RenderEngine та SwapchainManager реалізується через алгоритм реактивної обробки станів презентації, що виступає гарантом стабільності роботи додатку в умовах динамічної зміни конфігурації дисплея. На етапі ініціалізації SwapchainManager проводить глибокий аналіз можливостей графічної поверхні через функцію querySwapChainSupport, застосовуючи підхід до вибору параметрів, де найвищий пріоритет надається формату кольору VK_FORMAT_B8G8R8A8_SRGB у поєднанні з нелінійним колірним простором, а також режиму презентації, який забезпечує механізм потрійної буферизації для усунення артефактів розриву зображення при мінімальній затримці вводу, автоматично перемикаючись на стандартний режим у разі апаратної несумісності.

Геометричні параметри ланцюжка обміну обчислюються шляхом адаптації фізичних розмірів вікна ANativeWindow до жорстких апаратних лімітів графічного процесора з використанням функції нормалізації std::clamp. Фундаментальним механізмом забезпечення відмовостійкості є імплементація циклу обробки специфічних кодів повернення VK_ERROR_OUT_OF_DATE_KHR або VK_SUBOPTIMAL_KHR під час операцій отримання зображення та його презентації. При знаходженні цих станів RenderEngine негайно перериває обробку поточного кадру та ініціює

протокол аварійного відновлення через метод recreate, який виконує повну синхронізацію пристрою викликом vkDeviceWaitIdle, безпечне знищення застарілих ресурсів та реініціалізацію ланцюжка обміну з подальшою перебудовою всіх залежних фреймбуферів через систему зворотних викликів.

Для забезпечення інтуїтивно зрозумілого та прецизійного візуального аналізу тривимірних об'єктів у сцені, програмна система імплементує алгоритм орбітальної камери, логіка якої повністю інкапсульована у класі Camera. На відміну від механіки камер вільного польоту, дана реалізація базується на жорсткій фіксації вектора погляду на заданій цільовій точці, що обмежує рух точки спостереження поверхнею віртуальної сфери навколо об'єкта інтересу. Позиціонування спостерігача у тривимірному просторі розраховується шляхом математичного перетворення сферичних координат, що визначаються радіусом, де r це є дистанція до цілі, кутом рискання θ та кутом тангажу ϕ, у класичні декартові координати.

Алгоритм оновлення векторів орієнтації updateCameraVectors використовує тригонометричні залежності для обчислення компонентів позиції. Такий підхід гарантує, що камера перманентно залишається на фіксованій відстані від центру геометричного об'єкта. Інтерактивна взаємодія з користувачем забезпечується методом processTouchMovement, який реалізує обробку сенсорного вводу шляхом конвертації векторів зміщення пальця по площині екрана (ΔX, ΔY) у відповідні зміни кутових величин огляду сферичної системи. Для забезпечення стабільності орієнтації в алгоритм інтегровано механізм примусового обмеження кута тангажу у діапазоні від 89° до 89°. Це технічне рішення запобігає виникненню ефекту математичної сингулярності, що виникає, коли вектор погляду стає колінеарним до глобального вектора worldUp, що призвело б до втрати одного ступеня свободи та непередбачуваного обертання камери. Фінальна трансформація координатного простору виконується за допомогою стандартного методу бібліотеки лінійної алгебри glm::lookAt, який на основі розрахованої позиції спостерігача, координат цілі та ортогонального вектора up генерує видову матрицю, що

трансформує світові координати об'єктів у локальний простір камери для подальшої растеризації.

Для забезпечення детермінованої та коректної інтерпретації графічним процесором масиву геометричних даних, що надходять з буфера пам'яті, у допоміжному класі Utils реалізовано суворий механізм опису розмітки вхідних даних. Архітектурно це рішення спирається на техніку чергування даних, яка передбачає послідовне зберігання всіх атрибутів однієї вершини у єдиному безперервному блоці пам'яті. Такий підхід, на відміну від роздільних масивів атрибутів, суттєво підвищує ефективність використання кеш-пам'яті GPU завдяки принципу локальності посилань, оскільки при обробці конкретної вершини всі її параметри завантажуються в кеш однією транзакцією шини пам'яті.

Регламентація темпу подачі даних до вхідного асемблера здійснюється через метод Vertex::getBindingDescription, який формує дескриптор прив'язки. Цей опис визначає, що потік даних надходить через єдиний вхідний слот із кроком вибірки, який строго дорівнює розміру C++ структури Vertex, гарантуючи коректне зміщення покажчика при переході до наступного елемента. Встановлення параметра частоти вводу VK_VERTEX_INPUT_RATE_VERTEX інструктує драйвер про необхідність інкрементування індексу вибірки після обробки кожної окремої вершини, що є стандартною поведінкою для геометричного рендерингу.

Деталізація внутрішньої структури вершинних даних та їх мапінг на змінні шейдера реалізується через метод Vertex::getAttributeDescriptions, який визначає конфігурацію двох ключових атрибутів. Перший атрибут – позиція вершини, прив'язується до локації 0 у шейдері та використовує формат VK_FORMAT_R32G32B32_SFLOAT, що відповідає математичному вектору vec3. Другий атрибут – колір, прив'язується до локації 1 з аналогічним форматом даних. Для забезпечення абсолютної переносимості коду та захисту від специфічного для компілятора вирівнювання полів структури, обчислення зсуву атрибута кольору відносно початку вершини виконується за допомогою

макроса offsetof, що гарантує точну відповідність розмітки пам'яті очікуванням графічного конвеєра.

Окремим інженерним завданням при інтеграції математичних бібліотек є адаптація координатних просторів, зумовлена відмінностями між стандартами OpenGL та Vulkan. Оскільки бібліотека GLM історично спроектована для OpenGL, де вісь Y у нормалізованих координатах пристрою спрямована вгору, а Vulkan використовує систему з віссю Y, спрямованою вниз, пряме використання матриць призводить до перевернутого зображення. У модулі RenderEngine цю проблему вирішено шляхом алгоритмічної модифікації матриці проекції, операція інверсії діагонального елемента ubo.proj[1][1]*=−1 виконує геометричне дзеркальне відображення простору відсікання по вертикалі. Це дозволяє використовувати стандартні алгоритми GLM, забезпечуючи коректну орієнтацію фінального зображення без необхідності втручання у шейдерний код або модифікації вихідної геометрії 3D-моделей.

Стратегія забезпечення якості у розробленому шаблоні базується на синергетичному поєднанні інструментальних засобів глибокої діагностики Vulkan API та структурних рішень, спрямованих на забезпечення високої тестопридатності коду. Враховуючи, що фінальні релізні версії драйверів Vulkan заради продуктивності позбавлені механізмів перевірки помилок, компонент VulkanCore імплементує захисну парадигму через інтеграцію стандартного шару валідації VK_LAYER_KHRONOS_validation. У методі createInstance реалізовано алгоритм умовної конфігурації, який на етапі ініціалізації формує список активних шарів та, у разі ввімкнення режиму налагодження, реєструє об'єкт VkDebugUtilsMessengerEXT. Цей механізм виконує функцію перехоплювача, який фільтрує повідомлення від драйвера за ступенем критичності та маршрутизує їх через функцію зворотного виклику debugCallback безпосередньо у системний журнал logcat, що дозволяє розробнику миттєво ідентифікувати порушення специфікації, які не призводять до негайного краху, але загрожують стабільності.

Для реалізації вимоги щодо емпіричного аналізу продуктивності, система інтегрує нативні засоби трасування Android через бібліотеку trace.h, що дозволяє здійснювати інструментування коду для подальшої візуалізації у профайлерах типу Android GPU Inspector. У критичних секціях коду розміщено спеціальні маркери: зокрема, у методі LifecycleManager::handleAppCmd виклик функції ATrace_beginAsyncSection з міткою Swapchain_Recreation_Start фіксує початок ресурсомісткої операції перебудови ланцюжка обміну, тоді як у RenderEngine::drawFrame виклик ATrace_endAsyncSection позначає завершення циклу рендерингу. Такий підхід дозволяє отримати деталізовані часові діаграми виконання, точно виміряти затримки при реконфігурації графічного конвеєра та корелювати випадки втрати плавності з конкретними системними подіями або вузькими місцями в алгоритмах.

Високий рівень архітектурної тестопридатності шаблону досягається завдяки суворому дотриманню патерну ін'єкції залежностей при проектуванні конструкторів ключових компонентів. Наприклад, сигнатура конструктора RenderEngine, яка зображена на лістингу 2.3, дозволяє у тестовому середовищі підміняти реальні залежності на Mock-об'єкти або заглушки, що емулюють поведінку графічного процесора.

Лістинг 2.3 – Використання ін'єкції залежностей у конструкторі RenderEngine

```
RenderEngine::RenderEngine(VulkanCore& core, ResourceManager& resManager)
: vulkanCore(core), resourceManager(resManager) { ... }
```

Кінець лістингу 2.3

Це відкриває можливість проведення ізольованого модульного тестування логіки запису команд та керування станами рендерингу без необхідності розгортання додатку на фізичному пристрої чи емуляторі. Аналогічний підхід застосовано у LifecycleManager, який інкапсулює стан android_app, що дозволяє симулювати надходження кодів життєвого циклу у метод handleAppCmd та верифікувати реакцію системи відокремлено від фізичної оболонки Activity.

У розділі здійснено комплексне проектування та програмну реалізацію додатку для операційної системи Android на базі графічного API Vulkan. Головним науково-практичним результатом роботи стала розробка шаблону проектування, заснованого на принципі суворої сегрегації ресурсів на статичний стан асетів та динамічний стан презентації, що дозволило ефективно ізолювати ресурсомісткі об'єкти від нестабільного життєвого циклу активності.

Паралельно було імплементовано алгоритм безпечного керування станом із обов'язковою синхронізацією CPU-GPU під час перестворення ланцюжка обміну, який гарантує стійкість системи до критичних помилок драйвера при зміні конфігурації пристрою. Функціональність додатку розширена впровадженням механізмів оптимізації, зокрема системою кешування конвеєрів а також глибокою інтеграцією з інструментарієм AGDK для забезпечення надійної обробки подій. Отриманий програмний продукт вирішує поставлені архітектурні завдання та повністю готовий до етапу експериментальних досліджень ефективності.

РОЗДІЛ 3
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РЕЗУЛЬТАТИВНОСТІ ШАБЛОНУ ПРОЕКТУВАННЯ ДЛЯ ANDROID ДОДАТКІВ З VULKAN API

3.1 Методика проведення дослідження

Для проведення об'єктивного порівняння ефективності розробленого шаблону проектування та монолітного підходу, де весь код зосереджено в одному місці, було виконано серію практичних експериментів на смартфоні Samsung M12. Цей пристрій обладнано графічним процесором Mali-G52, який, на відміну від флагманських чіпів Adreno, має менше публічної документації щодо глибокого профілювання, тому методика вимірювань була адаптована під його можливості. Головною метою дослідження стало отримання точних цифр по трьом ключовим показникам: наскільки сильно програма навантажує центральний процесор, скільки оперативної пам'яті вона споживає та наскільки швидко програма відновлюється.

На підготовчому етапі було створено дві версії тестового додатка, які візуально нічим не відрізняються і відтворюють однакову 3D-сцену. Різниця прихована у коді: перша версія побудована на базі розробленого шаблону проектування з чітким розподілом завдань між менеджерами, а друга – це суцільний код, де вся логіка змішана в одному класі. Щоб інструменти аналізу могли б аналізувати працюючі програми, обидва додатки були скомпільовані зі спеціальним налаштуванням android:debuggable=true, що відкриває доступ до внутрішніх системних метрик. Перед початком замірів швидкості було перевірено додаток на відсутність прихованих помилок у коді, примусово увімкнувши через комп'ютер функцію валідаційних шарів Vulkan.

Безпосередній збір даних здійснювався за допомогою інструменту Android GPU Inspector у режимі System Profiling, який дозволяє записати поведінку системи протягом певного часу, наприклад 15-30 секунд активної роботи додатка. Процес налаштування починається з підключення пристрою та

вибору цільового додатка зі списку доступних процесів, після чого необхідно ретельно налаштувати параметри збору даних. Оскільки графічні процесори Mali мають свої особливості, критично важливим є вибір правильних джерел даних: було активовано опції CPU Usage для відстеження навантаження на кожне ядро процесора та Memory Usage для моніторингу виділення оперативної пам'яті.

Особливу увагу при налаштуванні слід приділити метрикам графічної підсистеми. У налаштуваннях AGI необхідно активувати збір даних від Surface Flinger – системного сервісу Android, який відповідає за вивід зображення на екран. Для GPU Mali-G52, який підтримує стандартні лічильники продуктивності, також доцільно обрати доступні метрики GPU Counters, що дозволять оцінити загальну завантаженість відеоядра. Після завершення конфігурації запускається запис трасування, під час якого на екрані смартфона виконується тестовий сценарій, а всі телеметричні дані зберігаються у файл для подальшого детального аналізу та порівняння двох архітектурних підходів.

Для високоточного вимірювання латентності критичних операцій, зокрема часу відновлення контексту презентації, методика передбачає використання інструментування коду через нативну бібліотеку trace.h. У ключових точках алгоритму розміщено асинхронні маркери трасування. Це дозволяє на часовій шкалі профілювальника чітко ізолювати інтервал блокування процесора викликом vkDeviceWaitIdle від загального часу виконання кадру та отримати точні дані синхронізації в мілісекундах.

Експериментальна фаза збору даних базувалася на проведенні серії тестів у режимі системного профілювання за допомогою інструментарію Android GPU Inspector. Для забезпечення статистичної чистоти експерименту кожен сеанс вимірювання тривав фіксовані 30 секунд, при цьому критично важливою умовою була стабілізація термального стану мобільного пристрою. Щоб запобігти спотворенню результатів через механізми термального тротлінгу, який автоматично знижує частоти процесора при перегріві, здійснювався безперервний моніторинг тактових частот CPU та GPU, а між тестами

забезпечувалися періоди охолодження апаратної частини. Окрім стандартного прогону, методика включає сценарій стрес-тестування життєвого циклу для перевірки відмовостійкості. Цей сценарій передбачає серію циклів примусового перемикання додатку у фоновий режим з частотою одне перемикання за 2 секунди. Метою цього етапу є фіксація можливих станів гонитви даних та аварійних завершень драйвера, які характерні для асинхронної природи Android. Конфігурація профілювальника була налаштована на агрегацію комплексного набору даних, що включав лічильники продуктивності графічного ядра, точні часові інтервали обробки кадрів та діаграми планування потоків, що дозволило отримати повну картину взаємодії програмного забезпечення з апаратними ресурсами.

Аналіз навантаження на центральний процесор виконувався шляхом дослідження графіків планування потоків з акцентом на розподіл задач між енергоефективними та продуктивними ядрами. Головним завданням було відстеження ефективності винесення важких операцій, а саме завантаження ресурсів, компіляція шейдерів, з головного потоку на фонові потоки. Це дозволяє оцінити, чи вдалося розробленому шаблону розвантажити критичне для системи Ядро 7, яке зазвичай обслуговує UI потік та перенести навантаження на вільні обчислювальні потужності інших ядер.

Аналіз ефективності роботи з пам'яттю здійснювався через моніторинг метрик пропускної здатності та загальної ефективності алокацій у AGI. Окрема увага приділялася категорії Graphics Cache та Native Heap. Методика передбачає порівняння обсягу споживаної пам'яті при завантаженні ідентичних асетів для перевірки ефективності роботи модуля ResourceManager.

3.2 Обробка та аналіз отриманих результатів

Обробка масиву емпіричних даних із використанням діагностичного інструментарію Android GPU Inspector у режимі системного профілювання, дала можливість проведення комплексної кількісної верифікації ефективності

розробленого шаблону проектування у порівнянні з базовою монолітною реалізацією. Аналіз проводився за трьома ключовими напрямками: навантаження на центральний процесор, ефективність використання пам'яті та часові характеристики відновлення контексту.

Першим етапом дослідження стало порівняння загальної утилізації процесорних потужностей. Детальний аналіз агрегованих статистичних даних засвідчив перевагу розробленого шаблону. Зокрема, середній показник завантаження центрального процесора для нового шаблону склав 13,16 %, тоді як для монолітного шаблону цей показник фіксувався на рівні 15,81 %.

Абсолютне зниження навантаження склало 2,65 %, що в перерахунку на відносні величини демонструє оптимізацію ресурсів на рівні 16,76 %. Візуалізація характеру планування потоків, яка зображена на рисунку 3.1 дозволяє оцінити щільність виконання інструкцій.

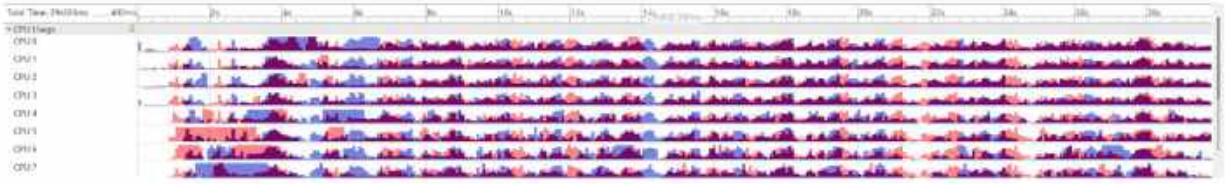

Рисунок 3.1 – Навантаження на ядра під час тестування у програмі Android GPU Inspector

Як видно з наведених трейсів, монолітна архітектура характеризується щільним заповненням таймлайну активними потоками, що свідчить про частіші виклики драйвера та блокування. Натомість, позитивна динаміка зниження навантаження у розробленому шаблоні є наслідком інтеграції механізму VkPipelineCache у модуль ResourceManager. Це дозволило системі уникнути ресурсомісткої процедури повторної компіляції шейдерів та зменшити пікові сплески навантаження на CPU, які чітко простежуються на графіках монолітного рішення під час ініціалізації.

Для глибшого розуміння природи оптимізації було проведено дослідження розподілу навантаження по окремих обчислювальних ядрах, результати якого знаходяться на рисунку 3.2.

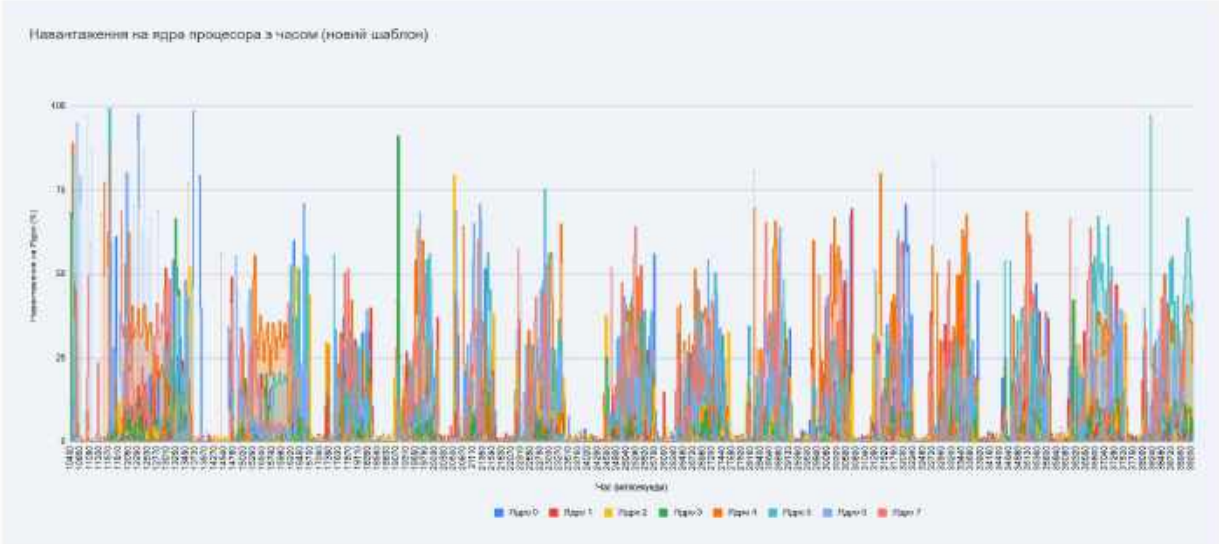

Рисунок 3.2 – Навантаження на ядра процесора з часом (новий шаблон)

Як видно з графіка нового шаблону, пікові навантаження розподілені рівномірно, а загальний рівень активності є помірним. На противагу цьому, рисунок 3.3 демонструє поведінку монолітної архітектури.

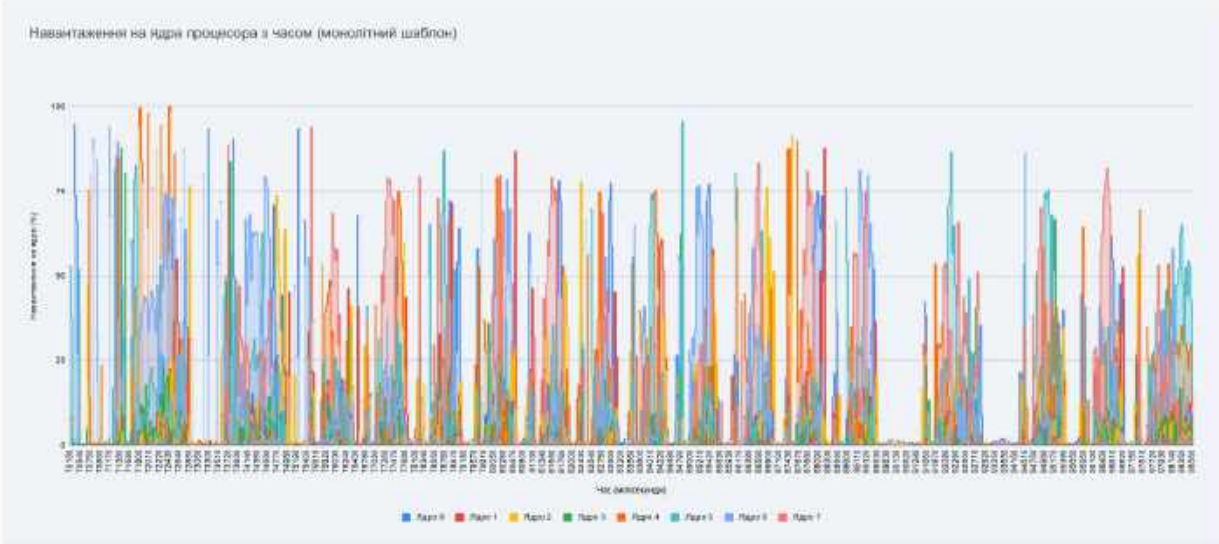

Рисунок 3.3 – Навантаження на ядра процесора з часом (монолітний шаблон)

Кількісні показники були сформовані на основі експорту сирих телеметричних даних з профілювальника Android GPU Inspector. Під час сесії профілювання для кожного фізичного ядра процесора (CPU 0 – CPU 7) фіксувався окремий потік даних активності.

Точні дані, які були виведені з програми Android GPU Inspector, через вбудовану можливість виводити дані за допомогою SQL запитів, наведено в таблиці 3.1, де отримані масиви значень були експортовані для зовнішньої обробки у середовище табличного процесора Microsoft Excel. Для отримання статистично значущого результату, було виконано математичний розрахунок середнього арифметичного значення навантаження для кожного ядра за весь період виконання тестового сценарію.

Таблиця 3.1 – Порівняльний аналіз навантаження на ядра CPU

Ядро процесора	Навантаження (монолітний шаблон), %	Навантаження (новий шаблон), %	Абсолютна різниця, %	Відносна різниця, %
Ядро 0	13,16	7,71	-5,45	-41,41
Ядро 1	8,22	6,89	-1,33	-16,18
Ядро 2	8,49	6,37	-2,12	-24,97
Ядро 3	7,65	7,42	-0,23	-3,01
Ядро 4	20,69	21,56	+0,87	+4,20
Ядро 5	18,86	18,92	+0,06	+0,32
Ядро 6	21,24	17,19	-4,05	-19,07
Ядро 7	28,16	19,24	-8,92	-31,68
Середнє по CPU	15,81	13,16	-2,65	-16,76

Порівнюючи отримані значення, можна помітити значно щільнішу активність та вищі амплітуди навантаження у монолітному варіанті. Кількісний аналіз цих даних підтверджує візуальні спостереження: найбільш значуще зменшення навантаження відбулося на Ядрі 7, де показник зменшився з 28,16 % до 19,24 % (відносна різниця -31,68 %), та на Ядрі 0 (зниження на 41,41 %).

Зниження активності на ядрах, які відповідають за системну рутину та основний потік рендерингу, є наслідком інтеграції механізму VkPipelineCache у

модуль ResourceManager. Це дозволило уникнути ресурсомісткої компіляції шейдерів під час ініціалізації.

Наступним етапом стала оцінка споживання оперативної пам’яті. Порівняння середніх показників виявило, що монолітна реалізація утримувала в середньому 60,05 мегабайт RAM, тоді як новий шаблон знизив цей показник до 58,20 МБ (оптимізація 3,18 %).

Значно виразнішу ефективність новий шаблон продемонстрував у сегменті використання кеш-пам’яті. Впровадження модульного підходу дозволило скоротити цей обсяг з 44,27 мегабайт до 38,72 мегабайт, що становить суттєву відносну різницю у 14,33 %. Динаміка накопичення пам’яті візуалізована на рисунку 3.4.

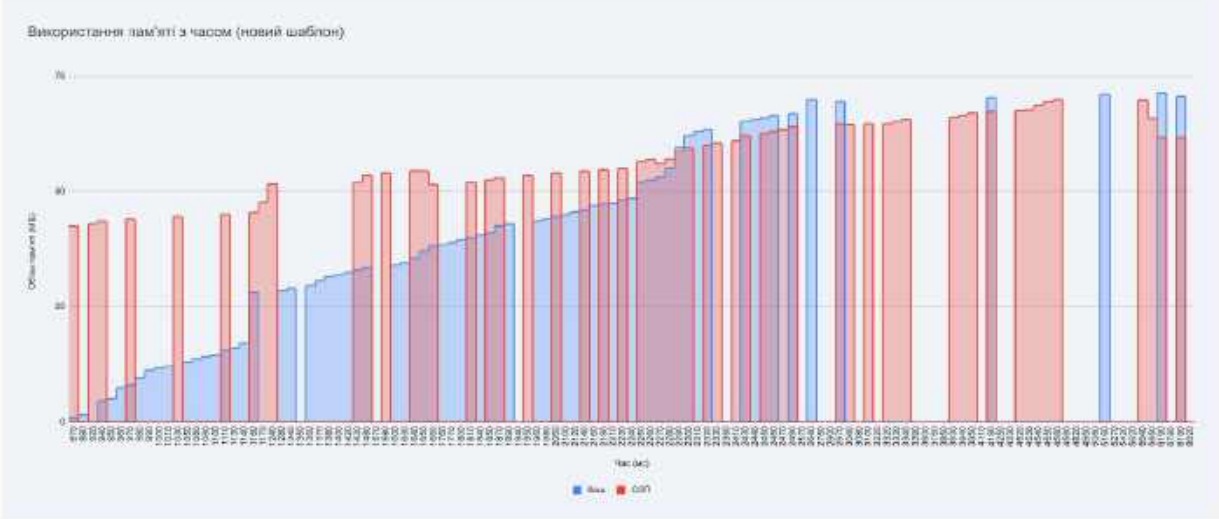

Рисунок 3.4 – Використання пам’яті з часом (новий шаблон)

На представленому графіку спостерігається стабільний рівень алокацій без різких стрибків, що свідчить про ефективну роботу ResourceManager. Синя гістограма (Кеш) демонструє монотонне зростання з поступовим ефектом, що свідчить про поступове завантаження та утримання графічних активів у пам'яті резидента. Для порівняння різниці, на рисунку 3.5 наведено графік використання пам’яті для монолітного шаблону, де попри загальну схожість з попереднім графіком, вимальовується різниця у використаних мегабайтах.

Рисунок 3.5 – Використання пам’яті з часом (монолітний шаблон)

Аналіз рисунка 3.5 демонструє вищий базовий рівень споживання ресурсів. Це є наслідком відсутності у моноліті структурних патернів на кшталт Flyweight, що призводить до дублювання ідентичних текстур та геометричних даних у пам’яті. Зниження навантаження на пам’ять є критичним для графічних процесорів Mali-G52, оскільки це зменшує тиск на шину даних та локальну пам’ять. Точні дані, виведені через SQL-запит у програмі Android GPU Inspector, наведено в таблиці 3.2.

Таблиця 3.2 – Порівняльний аналіз використання пам’яті

Пам’ять	Середня кількість мегабайт (новий шаблон)	Середня кількість мегабайт (монолітний шаблон)	Абсолютна різниця, %	Відносна різниця, %
Кеш	38,72	44,27	-5,55	-14,33
ОЗП	58,20	60,05	-1,85	-3,18

Емпіричні дані, що демонструють різницю у часі відновлення контексту, були отримані шляхом аналізу часової шкали у Android GPU Inspector. Для точної ідентифікації досліджуваного процесу використовувалося інструментування коду нативними маркерами, де тривалість операції

фіксувалася за системним тегом Swapchain_Recreation_Start. Виявлені результати демонструють різницю у часі відновлення контексту між монолітною реалізацією (119,679 мілісекунд) та розробленим шаблоном (276,433 мілісекунд), ілюструють вплив архітектурної декомпозиції на швидкодію системи. Зафіксоване відставання розробленого шаблону на 156,754 мілісекунд є наслідком ускладнення внутрішньої структури програми.

Вища швидкість відновлення у монолітному шаблоні пояснюється високим рівнем зв'язності коду. У цій реалізації вся логіка рендерингу, керування ресурсами та обробка віконних подій зосереджені у єдиному просторі імен, часто в межах одного класу. Це забезпечує компонентам прямий доступ до пам'яті та змінних один одного без необхідності використання проміжних інтерфейсів. Відсутність ієрархічних бар'єрів дозволяє процесору виконувати інструкції лінійно, мінімізуючи накладні витрати на перемикання контексту між функціями.

Натомість додаток з новим шаблоном демонструє більшу латентність через свою розподілену файлову структуру. Система розбита на ізольовані фізичні модулі, які мають непряме сполучення один з одним. Кожна операція відновлення вимагає проходження сигналу через ланцюжок абстракцій: від перехоплення події у головному циклі, через маршрутизацію у відповідний менеджер, до виклику конкретних методів ядра Vulkan. Така багаторівнева організація, де об'єкти змушені комунікувати через систему вказівників та віртуальних функцій, створює додаткове навантаження на стек викликів, що сумарно збільшує час виконання операції порівняно з прямолінійним кодом моноліту.

Створений у результаті проведеної роботи шаблон проектування являє собою закінчену інформаційну систему, що функціонує як високопродуктивне ядро рендерингу, спроектоване для нативних додатків на базі Vulkan API в операційному середовищі Android. Практична цінність та сфера впровадження даного рішення зосереджені на системному вирішенні фундаментальної інженерної проблеми – інтеграції низькорівневого, статично типізованого

Vulkan API з асинхронним та динамічним життєвим циклом мобільної операційної системи. Шаблон позиціонується як архітектурно верифікована, стабільна та енергоефективна програмна основа, що дозволяє мінімізувати навантаження на розробників при створенні складних графічних систем. Сфера практичного застосування отриманих результатів охоплює широкий спектр завдань, починаючи від розробки графічних рушіїв та фреймворків, де новий шаблон виступає як модульний шар абстракції. Цей шар ефективно інкапсулює надмірну складність Vulkan API, створюючи надійний фундамент для побудови спеціалізованих бібліотек або інтеграції в існуючі графічні екосистеми. Крім того, архітектура є критично важливою для класу високопродуктивних додатків, таких як системи автоматизованого проектування, програмні засоби доповненої реальності та професійні візуалізатори, які висувають жорсткі вимоги до обчислювальної ефективності та мінімізації накладних витрат на центральний процесор. Завдяки чіткому розмежуванню логіки рендерингу та механізмів платформної інтеграції, шаблон забезпечує високий потенціал для створення кросплатформних рішень, значно спрощуючи процес портування графічної підсистеми на інші операційні системи з підтримкою Vulkan.

Успішна імплементація та подальше масштабування результатів дослідження, втілених у розробленому шаблоні проектування, неможливі без суворого дотримання комплексу технічних передумов та стандартів, що формують екосистему розробки. Фундаментальною технічною вимогою виступає обов'язкова інтеграція компонентів Android Game Development Kit, серед яких центральну роль відіграє бібліотека GameActivity. Цей компонент функціонує як критично важливий комунікаційний міст між керованим середовищем Java/Kotlin та нативним C++ кодом, забезпечуючи трансляцію системних подій операційної системи та гарантуючи коректну обробку переривань без втрати контексту виконання.

З архітектурної точки зору, життєздатність та гнучкість системи безпосередньо залежать від дотримання методології об'єктно-орієнтованого проектування SOLID, з особливим акцентом на принцип єдиної

відповідальності. Інженерам необхідно забезпечувати слабку зв'язаність між модулями при розширенні функціональних можливостей, що є критичною умовою для збереження модульності, тестопридатності та запобігання деградації архітектури ядра рендерингу. Крім того, незважаючи на те, що розроблений шаблон забезпечує ефективну високорівневу інкапсуляцію механізмів життєвого циклу, він не скасовує імперативної природи Vulkan API. Розробка вимагає від програміста суворого дотримання принципів явного керування алокаціями пам'яті та ручної синхронізації потоків виконання, оскільки ігнорування цих низькорівневих аспектів на рівні прикладної логіки може нівелювати переваги стабільності та продуктивності, закладені в основу розробленого шаблону.

ВИСНОВКИ

У магістерській кваліфікаційній роботі успішно вирішено актуальне науково-прикладне завдання, спрямоване на підвищення ефективності та забезпечення експлуатаційної відмовостійкості графічних систем на мобільній платформі Android через розробку спеціалізованого шаблону проектування для Vulkan API.

Було проведено критичний аналіз існуючих архітектурних рішень та інструментів нативної розробки для виявлення проблем у керуванні низькорівневими ресурсами. Проведений аналіз предметної області дозволив ідентифікувати та формалізувати фундаментальний технічний конфлікт між статичною природою об'єктної моделі Vulkan API та асинхронним, динамічним життєвим циклом Android-додатків. Встановлено, що класичні архітектурні патерни високого рівня, такі як MVVM або MVI, а також стандартні засоби інтеграції на кшталт NativeActivity, не забезпечують необхідної гнучкості для детермінованого керування низькорівневими апаратними ресурсами, що у традиційних реалізаціях неминуче призводить до критичних помилок втрати контексту пристрою та неконтрольованих витоків пам'яті.

Був розроблений та обґрунтований шаблон проектування, який забезпечує модульність та відмовостійкість системи. Цей підхід передбачає чіткий поділ графічних об'єктів на статичний стан асетів, призначений для тривалого зберігання даних, та ефемерний стан презентації, що підлягає динамічному оновленню, що дозволило ефективно розірвати жорстку залежність графічного ядра від волатильності віконної системи.

Було створено функціональне програмне забезпечення на базі розробленого шаблону та аналогічне ПЗ з монолітним підходом у проектуванні, яке було використано для порівняльного дослідження.

Була проведена валідація та тестування розроблених програмних засобів, за допомогою програмного забезпечення для аналізу та тестування мобільних додатків – Android GPU Inspector. Тестувались такі метрики як навантаження на

CPU, використання оперативної пам'яті мобільного пристрою, а також швидкість відновлення розроблених додатків.

Були проаналізовані отримані, з проведеного тестування, результати та були сформовані висновки, щодо ефективності розробленого шаблону у порівнянні з монолітним підходом. Ефективність запропонованого рішення доведена експериментальним шляхом на базі апаратної платформи Samsung M12 з графічним процесором Mali-G52. Порівняльний аналіз продемонстрував зниження середнього навантаження на центральний процесор на 16,76 % (з 15,81 % до 13,16 %). Також зафіксовано зменшення використання графічного кешу на 14,33 % завдяки механізмам дедуплікації ресурсів. Хоча час відновлення контексту у новому шаблоні склав 276,433 мілісекунд проти 119,679 мілісекунд у монолітній версії, встановлено, що ця затримка у 156 мілісекунд є необхідною архітектурною платою за безпеку пам'яті та відсутність критичних помилок драйвера.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Лопух В. В. Проектування архітектури Android-додатку для тривимірної візуалізації даних з використанням Vulkan SDK. V всеукраїнська науково-технічна конференція молодих вчених, аспірантів та студентів «Комп'ютерні ігри і мультимедіа як інноваційний підхід до комунікації - 2025». Одеса, Україна, 25-26 вересня, 2025. Одеса: ОНТУ, 2025. С. 336-337.

2. Лопух В. В., Сичук В. А. Тестування шаблону проектування для Android-додатків з Vulkan API. Second International Scientific and Practical Conference «Progressive Approaches in Science and Engineering». Copenhagen, Denmark, November 26-28, 2025. Copenhagen: International Scientific Unity, 2025. P. 277-279.

3. Use Vulkan for graphics. URL: https://developer.android.com/games/develop/vulkan/overview (дата звернення: 04.08.2025).

4. Vulkan design guidelines. URL: https://developer.android.com/ndk/guides/graphics/design-notes (дата звернення: 15.08.2025).

5. Kool - A Vulkan / WebGPU / OpenGL graphics engine written in Kotlin. URL: https://github.com/kool-engine/kool (дата звернення: 04.09.2025).

6. An Opinionated Post on Modern Rendering Abstraction Layers. URL: https://alextardif.com/RenderingAbstractionLayers.html (дата звернення: 08.09.2025).

7. Android Game Development Kit. URL: https://developer.android.com/games/agdk/overview (дата звернення: 27.09.2025).

8. GameActivity. URL: https://developer.android.com/games/agdk/game-activity (дата звернення: 02.10.2025).

9. Shahbaz Y. Vulkan 1.4: Faster app loads, less stutter and less Memory Usage. URL: https://medium.com/androiddevelopers/vulkan-1-4-faster-app-loads-less-stutter-and-l

ess-memory-usage-host-image-copy-is-a-game-53c57e531f5d (дата звернення: 04.10.2025).

10.Halás T. Easy Vulkan. URL: https://excel.fit.vutbr.cz/submissions/2022/008/8.pdf (дата звернення: 14.10.2025).

11.Vulkan validation layers on Android. URL: https://developer.android.com/ndk/guides/graphics/validation-layer (дата звернення: 20.10.2025).

12.Android GPU Inspector quickstart. URL: https://developer.android.com/agi/start (дата звернення: 23.10.2025).

13.Vulkan Shaders. URL: https://vkguide.dev/docs/new_chapter_2/vulkan_shader_drawing/ (дата звернення: 26.10.2025).

14.Compute Shader - Vulkan Tutorial. URL: https://vulkan-tutorial.com/Compute_Shader (дата звернення: 29.10.2025).

15.Vulkan Memory Allocator. URL: https://gpuopen.com/vulkan-memory-allocator/ (дата звернення: 02.11.2025).