Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система мульти-API для 3D-рендеринга
(тема)

Виконав:
здобувач 4 року навчання
групи ПЗПІ-21-2

Вадим ЧАН
(власне ім'я, прізвище)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник доц. кафедри ПІ Андрій РАБОТЯГОВ
(посада, власне ім'я, прізвище)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
Кафедра Програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Чану Вадиму Ха
(прізвище, ім'я, по батькові)

1. Тема роботи Програмна система мульти-API для 3D-рендеринга
Затверджена наказом по університету від 19.05.2025 № 397Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 16.06.2025
3. Вихідні дані до роботи Розробити програмну систему мульти-API для 3D-рендеринга з підтримкою графічних API DirectX 12 та Vulkan, модульною архітектурою на основі Entity Component System, інтегрованим редактором, системою асинхронного завантаження ресурсів та динамічного оновлення шейдерів, використовуючи мову програмування C++20, HLSL для шейдерів та систему збирання CMake.
4. Перелік питань, що потрібно опрацювати в роботі Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	21.05.2025	*виконано*
2	Створення специфікації ПЗ	23.05.2025	*виконано*
3	Проектування ПЗ	24.05.2025	*виконано*
4	Розробка ПЗ	26.05.2025	*виконано*
5	Тестування ПЗ	29.05.2025	*виконано*
6	Оформлення пояснювальної записки	30.05.2025	*виконано*
7	Підготовка презентації та доповіді	03.06.2025	*виконано*
8	Попередній захист	10.06.2025	*виконано*
9	Нормоконтроль, рецензування	10.06.2025	*виконано*
10	Здача роботи у електронний архів	11.06.2025	*виконано*
11	Допуск до захисту у зав. кафедри	14.06.2025	*виконано*

Дата видачі завдання «8» «квітня» 2025р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. кафедри ПІ Андрій РАБОТЯГОВ
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра, 109 сторінок, 71 рисунків, 14 джерел.

КОМП'ЮТЕРНА ГРАФІКА, МОДУЛЬНА АРХІТЕКТУРА, МУЛЬТИ-API, РЕНДЕРИНГ, C++20, CMAKE, DIRECTX, VULKAN

Об'єкт розробки – програмна система мульти-API для 3D-рендеринга з підтримкою графічних API.

Мета розробки – розробка програмної системи мульти-API для 3D-рендеринга з гнучкою модульною архітектурою рушія для 3D-рендеринга з уніфікованим інтерфейсом для графічних API.

Метод рішення – застосування сучасних стандартів C++20, системи збирання CMake та використання патернів проектування для створення модульної архітектури з чітким розподілом відповідальності між компонентами.

У результаті виконання кваліфікаційної роботи розроблена програмна система для 3D-рендеринга, яка підтримує різні графічні API через єдиний інтерфейс, забезпечує оптимальну продуктивність, має низький поріг входження та мінімальні залежності від зовнішніх бібліотек.

ABSTRACT

COMPUTER GRAPHICS, MODULAR ARCHITECTURE, MULTI-API, RENDERING, C++20, CMAKE, DIRECTX, VULKAN

Object of development – a multi-API software system for 3D rendering with support for various graphics APIs, providing cross-platform capabilities and high performance on different devices.

The purpose of development – creating a flexible, modular architecture for a 3D rendering engine with a unified interface for various graphics APIs, simplifying the development of three-dimensional applications and ensuring their efficient operation on different platforms.

The solution method – application of modern C++20 standards, CMake build system, and design patterns to create a modular architecture with clear separation of responsibilities between components.

As a result, an architecture of a software system for 3D rendering has been designed, which supports various graphics APIs through a unified interface, provides optimal performance, has a low entry threshold, and minimal dependencies on external libraries.

ЗМІСТ

ВСТУП

Метою даної роботи є створення програмної системи мульти-API для 3D-рендеринга, яка забезпечить розробникам графічних додатків можливість використовувати єдиний інтерфейс для роботи з різними графічними API (DirectX 12, Vulkan) на різних платформах. Система спрямована на вирішення проблеми фрагментації графічних технологій, зниження порогу входження для розробників та забезпечення високопродуктивного рендеринга незалежно від цільової платформи.

Програмна система реалізується у вигляді повноцінного рушія для 3D-рендеринга з модульною архітектурою та чітким розподілом відповідальності між компонентами. Інтерфейс системи є інтуїтивно зрозумілим та легким у використанні, що дозволить розробникам зосередитись на специфіці своїх додатків, а не на особливостях графічних API.

Система надає можливість легкого налаштування візуалізаційних пайплайнів, управління сценою, що базується на Entity Component System, та інтегрованого середовища редактора з повним набором інструментів для розробки та відлагодження.

Для створення даної програмної системи використовуються мова C++ (стандарт C++20), система збирання CMake, системи управління ресурсами Vulkan Memory Allocator та D3D12 Memory Allocator, бібліотека Cgltf для завантаження 3D-моделей, ImGui та ImGuizmo для інтерфейсу редактора, EnTT для реалізації Entity Component System, spdlog для системи логування.

Результатом роботи є високопродуктивна, гнучка та розширювана програмна система, яка спрощує розробку крос-платформних 3D-додатків, забезпечує високу якість візуалізації та ефективне використання ресурсів на різному обладнанні, що робить її конкурентоспроможною альтернативою існуючим рішенням на ринку 3D-графіки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

3D-рендеринг є ключовим компонентом сучасних комп'ютерних технологій, що забезпечує створення тривимірних графічних зображень на основі математичних моделей. Ця технологія знаходить широке застосування у розробці відеоігор, комп'ютерній анімації, архітектурній візуалізації, медичній діагностиці [1], інженерному проектуванні та інших галузях, де необхідно візуалізувати віртуальні об'єкти з високою точністю та реалістичністю [2].

Розвиток галузі 3D-рендеринга розпочався у 1960-х роках з простих каркасних моделей, проте значний розвиток відбувся у 1980-х та 1990-х роках із появою перших графічних акселераторів [3]. Розвиток індустрії відеоігор, поява таких технологій як шейдери та програмовані графічні конвеєри у 2000-х роках значно розширили можливості 3D-рендеринга та зробили його більш доступним. Сьогодні ми спостерігаємо стрімкий розвиток графічних технологій, де ключовими трендами є реалістичність, продуктивність (рендеринг у реальному часі) та доступність (крос-платформність). У паралельних дослідженнях 3D-обробка сигналів використовується як додатковий фактор захисту в системах інформаційної безпеки [4].

Дослідження компанії Grand View Research показують, що глобальний ринок 3D-рендеринга оцінювався у 4,21 мільярда доларів у 2024 році і прогнозується його зростання до 11,3 мільярдів доларів до 2030 року, з середньорічним темпом зростання (CAGR) близько 17,9%. Це зростання пов'язане із підвищеним попитом на високоякісну візуалізацію у різних галузях, розвитком технологій віртуальної та доповненої реальності, а також поширенням хмарних рішень для рендеринга [5].

Сьогодні основними технологіями, що використовуються для 3D-рендеринга, є різноманітні графічні API, включаючи DirectX, Vulkan, OpenGL та Metal. Кожен з цих інтерфейсів має свої особливості, переваги та недоліки, що впливають на їх застосування на різних платформах. Водночас досліджуються спеціалізовані алгоритми імпульсних перетворень для сегментації клітинних

зображень [6]. Наприклад, DirectX традиційно використовується в екосистемі Windows, тоді як Vulkan пропонує крос-платформне рішення з низькорівневим доступом до графічного обладнання.

Одним із головних технічних викликів у галузі 3D-рендеринга є розробка системи, яка підтримує різні графічні API одночасно. Це особливо важливо для забезпечення крос-платформності програмного забезпечення, зокрема між Windows, де традиційно використовується DirectX, та іншими операційними системами, де переважає Vulkan або OpenGL. Сучасні рендеринг-системи повинні підтримувати різноманітні режими візуалізації (каркасний, суцільний, з відображенням нормалей) та різні пайплайни обробки для забезпечення гнучкості в роботі з графікою.

Іншою важливою проблемою є фрагментація апаратного та програмного забезпечення. Різноманітність графічних процесорів (GPU), операційних систем та графічних API створює складнощі для розробників, які прагнуть забезпечити однакову якість та продуктивність на різних платформах. Це призводить до необхідності створення абстрактних шарів над низькорівневими API, щоб спростити розробку та забезпечити портативність програмного забезпечення.

Крім того, сучасні рендер-системи стикаються з проблемою масштабованості – здатністю ефективно використовувати наявні ресурси на різноманітному обладнанні, від портативних пристроїв до високопродуктивних робочих станцій. Це вимагає розробки адаптивних алгоритмів та оптимізації використання пам'яті та обчислювальних ресурсів.

У контексті цих викликів, розробка універсальної мульти-API системи для 3D-рендеринга є актуальним завданням, що може вирішити ряд існуючих проблем. Така система дозволить абстрагуватися від низькорівневих особливостей різних графічних API, забезпечити крос-платформність, спростити процес розробки та оптимізувати використання ресурсів на різному обладнанні [7].

1.2 Виявлення та вирішення проблем

Після детального аналізу предметної галузі 3D-рендеринга важливо дослідити існуючі рішення на ринку, щоб виявити їхні обмеження та визначити напрямки вдосконалення. Розглянемо три провідні рушії для 3D-графіки та їхні особливості.

Open 3D Engine (O3DE) – це відносно новий рушій з відкритим кодом, підтримуваний Amazon та Linux Foundation. Він представляє собою амбітну спробу створити повноцінну платформу для 3D-розробки з підтримкою сучасних графічних API та крос-платформністю. Ключовими перевагами O3DE є багатий набір інструментів для розробки, гнучка компонентна система та вбудована підтримка хмарних сервісів. Однак, незважаючи на корпоративну підтримку, рушій має ряд суттєвих недоліків. O3DE страждає від проблем зі стабільністю та продуктивністю, особливо в порівнянні з більш зрілими рішеннями. Його складна архітектура створює високий поріг входження для нових користувачів, а значна кількість залежностей ускладнює процес збирання та інтеграції. Часто навіть досвідчені програмісти потребують тижнів, щоб зрозуміти базові принципи роботи рушія [8].

Unreal Engine, розроблений Epic Games, є потужним комерційним рушієм, який широко використовується як в ігровій індустрії, так і для візуалізації в архітектурі, кіно та інших галузях. Його незаперечними перевагами є висока якість графіки, розвинені інструменти візуального програмування (Blueprints) та обширна документація. Проте, Unreal Engine також має свої обмеження. Його висока продуктивність досягається за рахунок значних вимог до апаратного забезпечення, що робить його менш доступним для розробників з обмеженими ресурсами. Власна система збирання та управління залежностями Unreal Engine часто ускладнює інтеграцію з зовнішніми інструментами та бібліотеками. Це створює замкнуту екосистему, яка, хоча і потужна, але обмежує гнучкість розробників. Модифікація внутрішніх механізмів рендеринга вимагає глибокого розуміння архітектури рушія, що ускладнює його використання для нестандартних задач рендеринга [9].

Godot Engine позиціонується як повністю відкрита альтернатива комерційним рушіям, що пропонує крос-платформний розвиток без ліцензійних відрахувань. Його інтегроване середовище розробки та інтуїтивна система вузлів роблять Godot привабливим для інді-розробників та малих команд. Однак, на відміну від його сильних можливостей для 2D-розробки, 3D-функціональність Godot досі залишається обмеженою. Хоча версія 4.0 принесла значні покращення, включаючи підтримку Vulkan та оновлений конвеєр рендеринга, система все ще відстає від конкурентів у плані продуктивності та якості візуалізації складних 3D-сцен [10]. Екосистема Godot значно менша, ніж у комерційних конкурентів, що обмежує доступність готових ресурсів та плагінів для 3D-розробки.

Аналіз цих рушіїв виявляє наступні проблеми, які потрібно вирішити:

1) баланс між доступністю та продуктивністю: існуючі рішення або мають високий поріг входження через складність API, або не забезпечують достатньої продуктивності для складних 3D-проектів;
2) недостатня крос-платформність. Більшість рушіїв надають різний рівень підтримки для різних операційних систем та графічних API, що змушує розробників адаптувати свої проекти для кожної цільової платформи;
3) складні системи збирання та управління залежностями. Багато рушіїв використовують власні комплексні системи, тісно інтегровані з конкретним продуктом, що ускладнює включення зовнішніх бібліотек та інструментів;
4) обмежена гнучкість для нестандартних задач. Архітектура існуючих рушіїв часто не дозволяє легко впроваджувати нестандартні техніки рендеринга без глибокої модифікації ядра системи.

На основі проведеного аналізу можемо виділити ключові вимоги до нової програмної системи мульти-API для 3D-рендеринга:

1) спрощення та доступність у використанні системи для новачків як головний пріоритет при розробки системи (на відміну від складних комерційних рушіїв);

2) використання сучасних стандартів C++ з простими та зрозумілими інтерфейсами;
3) модульна архітектура з чіткими інтерфейсами між компонентами, що спрощує розуміння та розширення;
4) сучасна система збирання з мінімальною кількістю кроків для початку роботи;
5) підтримка різних графічних API з уніфікованим та інтуїтивним інтерфейсом.

1.3 Постановка задачі

Сформулюємо основні задачі роботи:

1) створення доступної для новачків архітектури рушія, що спрощує розуміння принципів 3D-рендеринга та уникає надмірної складності комерційних рушіїв;
2) розробка гнучкої архітектури із забезпеченням легкої адаптації до різних графічних API (DirectX 12, Vulkan) через зрозумілий абстрактний шар;
3) впровадження освітніх інструментів для профілювання та моніторингу продуктивності системи з візуалізацією етапів роботи рушія;
4) забезпечення високої якості візуалізації з підтримкою різних режимів рендеринга та інтуїтивними налагоджувальними інструментами;
5) розробка простої та зрозумілої системи збирання на основі CMake з мінімальною кількістю кроків для початку роботи;
6) реалізація прозорого абстрактного шару над графічними API, що дозволяє зосередитися на логіці рендеринга замість деталей API;
7) створення дружнього до користувача інтегрованого редактора з інтуїтивним інтерфейсом та навчальними можливостями;
8) впровадження системи асинхронного завантаження ресурсів з індикацією статусу завантаження;
9) реалізація системи динамічного оновлення шейдерів для швидкого експериментування та навчання;

10) використання зрозумілих архітектурних патернів та сучасних стандартів C++20 з акцентом на читабельність коду.

В результаті реалізації поставлених задач буде створена програмна система, що забезпечує доступний вхід у сферу 3D-графіки для новачків через спрощену архітектуру та зрозумілі інтерфейси, водночас зберігаючи оптимальний баланс між функціональністю, продуктивністю та зручністю використання для розробки різноманітних 3D-додатків.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Програмна система повинна складатися з наступних компонентів:

1) абстракція графічних API (Render Hardware Interface): система має надавати єдиний, зрозумілий інтерфейс для взаємодії з різними графічними API, такими як DirectX 12 та Vulkan. Це забезпечить легкість інтеграції нових технологій та розширюваність архітектури без потреби переписування існуючого коду у разі додавання підтримки нових API;
2) система проходів рендеринга: компонент повинен відповідати за управління візуалізаційним конвеєром через систему рендер-пасів, включаючи базовий прохід (BasePass), прохід налагодження (DebugPass) з підтримкою різних режимів візуалізації (каркасний, відображення нормалей, overdraw) та фінальний прохід (FinalPass);
3) система управління ресурсами: компонент має забезпечувати ефективне завантаження, зберігання та управління ресурсами через спеціалізовані менеджери, включаючи асинхронне завантаження 3D-моделей та текстур, кешування ресурсів та систему завантаження через абстрактні інтерфейси;
4) абстракція рівня додатку: система має забезпечувати абстрактні інтерфейси для операцій, які взаємодіють з операційною системою, наприклад, створення вікон або управління життєвим циклом додатку через SDL;
5) система управління сценами: реалізація архітектури Entity Component System на основі EnTT для інтуїтивного управління об'єктами у сцені без складних ієрархій класів, з підтримкою зрозумілої серіалізації сцен у JSON формат для можливості ручного редагування;
6) інтегроване середовище редактора: розробка редактора на основі ImGui з панеллю ієрархії сцени, інспектором об'єктів, інструментами трансформації через ImGuizmo, viewport для відображення результатів рендеринга та засобами моніторингу продуктивності;

7) система логування та діагностики: компонент має забезпечувати детальне логування через spdlog з підтримкою різних рівнів деталізації для ефективного відстеження та діагностики проблем;
8) система подій та вводу: реалізація механізму для обробки подій від вікна, клавіатури та миші через SDL, що забезпечить гнучкий інтерфейс для взаємодії користувача з системою;
9) система конфігурації: компонент для управління налаштуваннями системи через JSON файли з підтримкою динамічного вибору графічного API та режиму роботи програми;
10) система динамічного оновлення шейдерів: реалізація hot reload функціональності для автоматичного перекомпілювання та застосування змін у шейдерах під час роботи програми.

Функціональні вимоги до системи включають:

1) підтримка графічних API:
   - реалізація підтримки DirectX 12 для Windows з використанням D3D12 Memory Allocator;
   - реалізація підтримки Vulkan з використанням Vulkan Memory Allocator;
   - можливість динамічного вибору графічного API через конфігураційний файл;
2) управління ресурсами рендеринга:
   - завантаження 3D-моделей з підтримкою форматів glTF через Cgltf та інших форматів через Assimp;
   - асинхронне завантаження ресурсів у фоновому потоці з callback-сповіщеннями;
   - завантаження та обробка текстур із різних форматів через STB та DirectX Texture;
   - система кешування ресурсів для запобігання повторному завантаженню;
   - управління матеріалами та шейдерами з підтримкою hot reload;

3) візуалізація та рендеринг:
   - базовий рендеринг сцени з підтримкою освітлення та матеріалів;
   - налагоджувальні режими візуалізації (каркасний, нормалі, overdraw);
   - система математичних трансформацій з підтримкою різних координатних систем;
4) інтегроване середовище редактора:
   - відображення ієрархії сцени з можливістю вибору та редагування об'єктів;
   - інспектор об'єктів для налаштування компонентів Entity Component System з інтуїтивним інтерфейсом;
   - візуальні інструменти трансформації (gizmo) для переміщення, обертання та масштабування без необхідності ручного введення координат;
   - viewport з інтеграцією результатів рендеринга та миттєвим відображенням змін;
   - система створення, завантаження та збереження сцен з простим файловим форматом;
   - навчальні можливості через візуалізацію різних етапів рендеринга та налагоджувальні режими;
5) інструменти розробки та навчання:
   - інтеграція з Tracy профайлером для наочного моніторингу CPU та GPU продуктивності;
   - відображення метрик продуктивності в реальному часі з доступними поясненнями;
   - система логування з різними рівнями деталізації для розуміння внутрішніх процесів;
   - динамічне оновлення шейдерів з автоматичною перебудовою для швидкого експериментування;

- налагоджувальні режими візуалізації з освітніми можливостями для розуміння різних аспектів рендеринга;

6) система подій та взаємодії:
   - обробка подій вікна (зміна розміру, переміщення, фокус);
   - обробка подій введення (клавіатура, миша) з підтримкою контекстів;
   - система конфігурації через JSON файли.

Нефункціональні вимоги включають:

1) продуктивність та ефективність:
   - підтримка стабільної частоти кадрів на середньостатистичному обладнанні;
   - ефективне управління ресурсами GPU через спеціалізовані алокатори пам'яті;
   - оптимізація рендеринга через систему проходів та групування команд;
2) надійність та якість архітектури:
   - обробка критичних помилок з відповідним логуванням;
   - модульна архітектура з чітким розділенням відповідальності компонентів;
   - здатність до адаптації під різні сценарії використання;
3) доступність та освітня цінність:
   - мінімальні залежності від системних бібліотек через використання vcpkg та FetchContent;
   - підтримка різних конфігурацій апаратного забезпечення;
   - інтуїтивний інтерфейс редактора з низьким порогом входження для новачків;
   - чітка архітектура з мінімальною кількістю абстракцій для легшого розуміння принципів роботи;
   - освітня цінність через прозорість архітектурних рішень та уникнення надмірної складності;
4) простота розробки та навчання:

- чітка та зрозуміла структура коду з дотриманням сучасних стандартів C++20;
- мінімальна кількість рівнів абстракції для легшого розуміння архітектури;
- гнучка система збірки з підтримкою опціональних компонентів та простим процесом налаштування;
- використання загальновідомих патернів проектування замість власних складних рішень;
- детальна документація архітектурних рішень з освітньою метою.

Для реалізації програмної системи буде використано наступний технологічний стек:

1) мова програмування та стандарти:
   - C++20 для основної кодової бази;
   - CMake як система збирання проекту з підтримкою FetchContent;
   - vcpkg для управління залежностями;
   - Visual Studio як основне інтегроване середовище розробки (IDE);
2) графічні API та супутні технології:
   - DirectX 12 для Windows з підтримкою DirectX Agility SDK;
   - Vulkan для кросплатформної підтримки;
   - D3D12 Memory Allocator для управління пам'яттю DirectX;
   - Vulkan Memory Allocator для управління пам'яттю Vulkan;
   - DirectX Shader Compiler (DXC) для компіляції HLSL шейдерів у DXIL та SPIRV;
3) робота з ресурсами:
   - Cgltf для завантаження glTF моделей;
   - Assimp для підтримки додаткових форматів 3D-моделей;
   - STB та DirectX Texture для завантаження та обробки зображень;
   - RapidJSON для серіалізації та десеріалізації даних сцен;
4) інтерфейс та взаємодія:

- SDL для роботи з вікнами та системою вводу;
- ImGui з docking підтримкою для інтерфейсу редактора;
- ImGuizmo для візуальних інструментів трансформації об'єктів;

5) системні компоненти:
   - spdlog для системи логування;
   - EnTT для реалізації Entity Component System;
   - власна математична бібліотека з підтримкою SIMD-оптимізацій;
   - wtr::watcher для моніторингу змін файлів (hot reload);
   - Tracy для профілювання продуктивності CPU та GPU.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML проєктування ПЗ

Моделювання програмної системи мульти-API для 3D-рендеринга доцільно здійснювати з використанням діаграм UML, які дозволяють візуалізувати структурні та поведінкові аспекти системи. В цьому підрозділі представлено ключову діаграму класів, яка ілюструє архітектурні компоненти спроектованої системи та їх взаємодію.

Найважливішою діаграмою для представлення структури даної системи є діаграма класів, яка показує основні компоненти системи рендеринга, їхні поля, методи та взаємозв'язки. Спроектована діаграма класів показана на рисунку 4.1.

Діаграма класів системи розділена на декілька логічних блоків, що відображають основні підсистеми програмного забезпечення. Перший блок представляє керуючі компоненти, такі як Engine, SceneManager та Window. Engine виступає центральною сутністю, що координуватиме роботу всіх інших компонентів. Він міститиме посилання на Renderer, Window, Scene та інші важливі підсистеми.

Важливою частиною архітектури є Scene та SceneManager, які відповідатимуть за управління сценами та об'єктами в них. SceneManager керуватиме множиною сцен та забезпечуватиме механізм переключення між ними, в той час як Scene міститиме entityRegistry для управління всіма сутностями в поточній сцені відповідно до парадигми Entity Component System.

Центральним компонентом рендеринга є клас Renderer, який інкапсулюватиме логіку рендеринга та взаємодії з графічними API. Він матиме посилання на Device, що абстрагуватиме роботу з конкретним графічним API, та FrameResources, що управлятиме ресурсами, необхідними для рендеринга кадру. Renderer також координуватиме роботу різних рендер-пасів: BasePass, DebugPass та FinalPass.

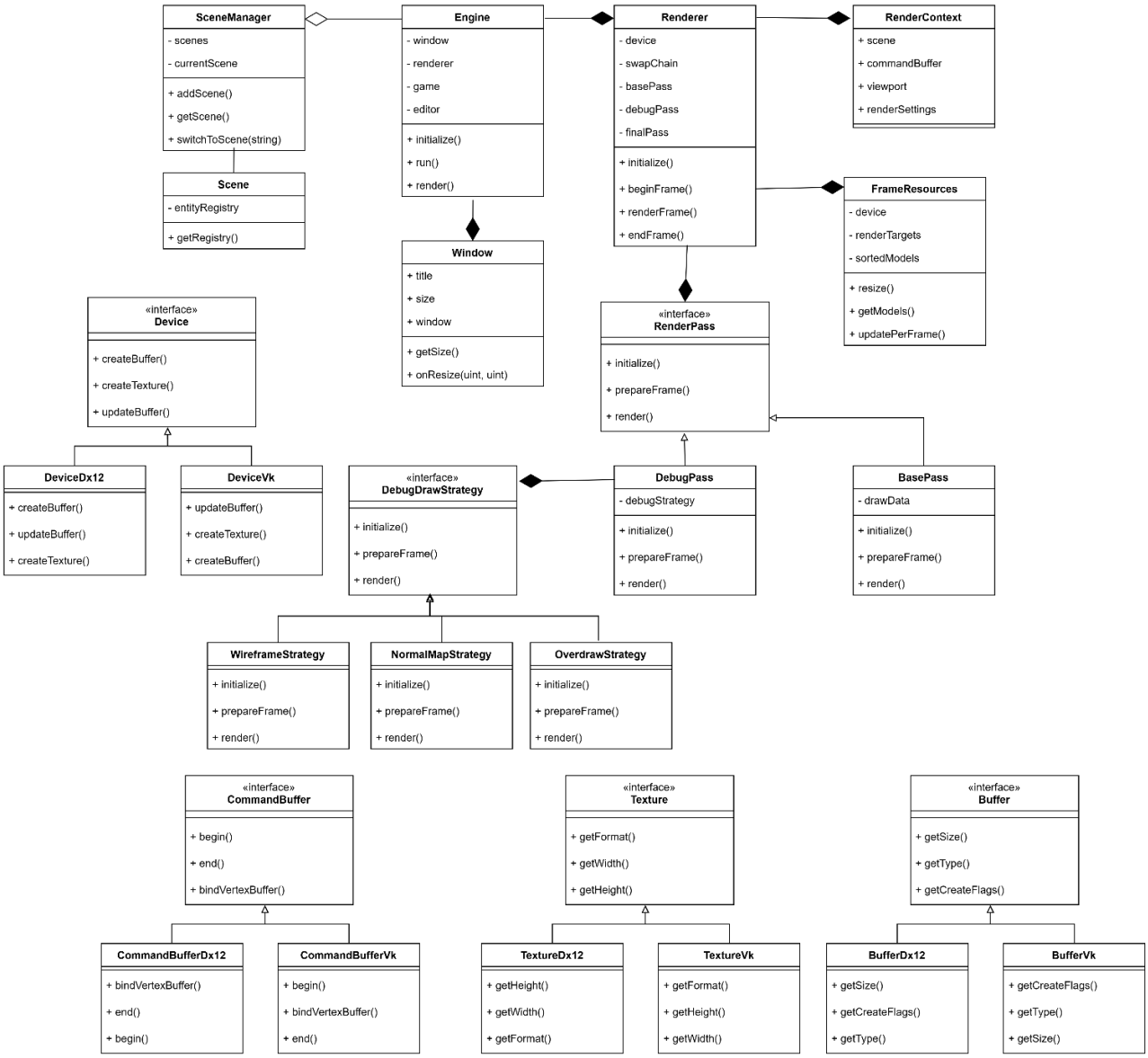

Рисунок 3.1 – UML діаграма класів (виконано самостійно)

Найбільш цікавою частиною системи є абстракція графічних API, що буде реалізована через інтерфейси Device, Buffer, Texture, CommandBuffer та інші. Вони матимуть конкретні реалізації для різних API, таких як DeviceVk та DeviceDx12, що дозволить системі працювати з різними графічними API без зміни високорівневого коду [11].

RenderPass представлятиме етап рендеринга, який міститиме логіку для рендеринга певного аспекту сцени. BasePass відповідатиме за основний рендеринг сцени, DebugPass - за режими візуалізації для налагодження, а FinalPass - за фінальну обробку зображення. DebugPass використовуватиме патерн Стратегія

через DebugDrawStrategy, який матиме конкретні реалізації для різних режимів відображення: WireframeStrategy, NormalMapVisualizationStrategy, VertexNormalVisualizationStrategy та ShaderOverdrawStrategy.

Важливою частиною системи будуть компоненти, які відповідатимуть за управління ресурсами. FrameResources управлятиме ресурсами, необхідними для рендеринга кадру, включаючи буфери, текстури та дескриптори. RenderResourceManager відповідатиме за створення та управління графічними ресурсами, такими як буфери, текстури, сэмплери та пайплайни.

Система також включатиме підсистему обробки подій через класи EventManager, WindowEventManager, ApplicationEventManager та InputManager, які відповідатимуть за обробку різних типів подій від користувача та системи.

3.2 Проектування архітектури ПЗ

Архітектура програмної системи базуватиметься на модульному підході з чітким розподілом відповідальності між компонентами. Ця архітектура забезпечить гнучкість, розширюваність та можливість підтримки різних графічних API. Система складатиметься з декількох ключових шарів та модулів, кожен з яких виконуватиме специфічну функцію.

Основу архітектури складатиме ядро системи (Engine), яке координуватиме роботу всіх інших модулів та забезпечуватиме точку входу в програму. Ядро міститиме цикл оновлення та рендеринга, керуватиме ініціалізацією та звільненням ресурсів. Воно також відповідатиме за інтеграцію з іншими системами, такими як управління сценами, обробка вводу та рендеринг.

Наступним важливим шаром буде абстракція операційної системи, яка надаватиме уніфікований інтерфейс для роботи з ресурсами ОС, такими як вікна, вхідні пристрої та системні події. Це дозволить системі працювати на різних платформах без зміни високорівневого коду.

Ключовою особливістю системи буде абстракція графічного API, яка дозволить підтримувати різні графічні API, такі як Vulkan та DirectX 12, через спільний інтерфейс. Ця абстракція буде реалізована за допомогою інтерфейсів для

основних графічних ресурсів: пристрою, буферів, текстур, шейдерів, пайплайнів та командних буферів. Кожен інтерфейс матиме конкретні реалізації для кожного з підтримуваних API.

Архітектура системи також включатиме шар управління ресурсами, який відповідатиме за створення, управління та звільнення графічних ресурсів. Цей шар включатиме менеджери для різних типів ресурсів: буферів, текстур, шейдерів, моделей та матеріалів.

Важливою частиною архітектури буде система рендеринга, яка буде організована навколо концепції рендер-пасів. Кожен рендер-пас відповідатиме за певний етап рендеринга сцени: BasePass для основного рендеринга, DebugPass для режимів візуалізації для налагодження, та FinalPass для фінальної обробки зображення. Система рендеринга також включатиме компоненти для управління камерою, світлом та іншими аспектами сцени.

Додатково система використовуватиме Entity Component System (ECS) для організації об'єктів сцени та їх властивостей. Цей підхід дозволить гнучко комбінувати різні властивості об'єктів без необхідності створення складних ієрархій класів. Система ECS складатиметься з компонентів, які зберігатимуть дані (наприклад, Transform, Camera, Mesh), та систем, які оброблятимуть ці дані (наприклад, CameraSystem, MovementSystem, RenderSystem).

Для комунікації між компонентами система використовуватиме патерн Observer через систему подій. Це дозволить компонентам реагувати на події, не будучи безпосередньо пов'язаними один з одним. Система подій включатиме обробники для різних типів подій: клавіатури, миші, вікна та додатку.

Для демонстрації процесу рендеринга кадру спроектовано діаграму послідовності, показану на рисунку 4.2. Ця діаграма ілюструє взаємодію між основними компонентами системи під час рендеринга одного кадру.

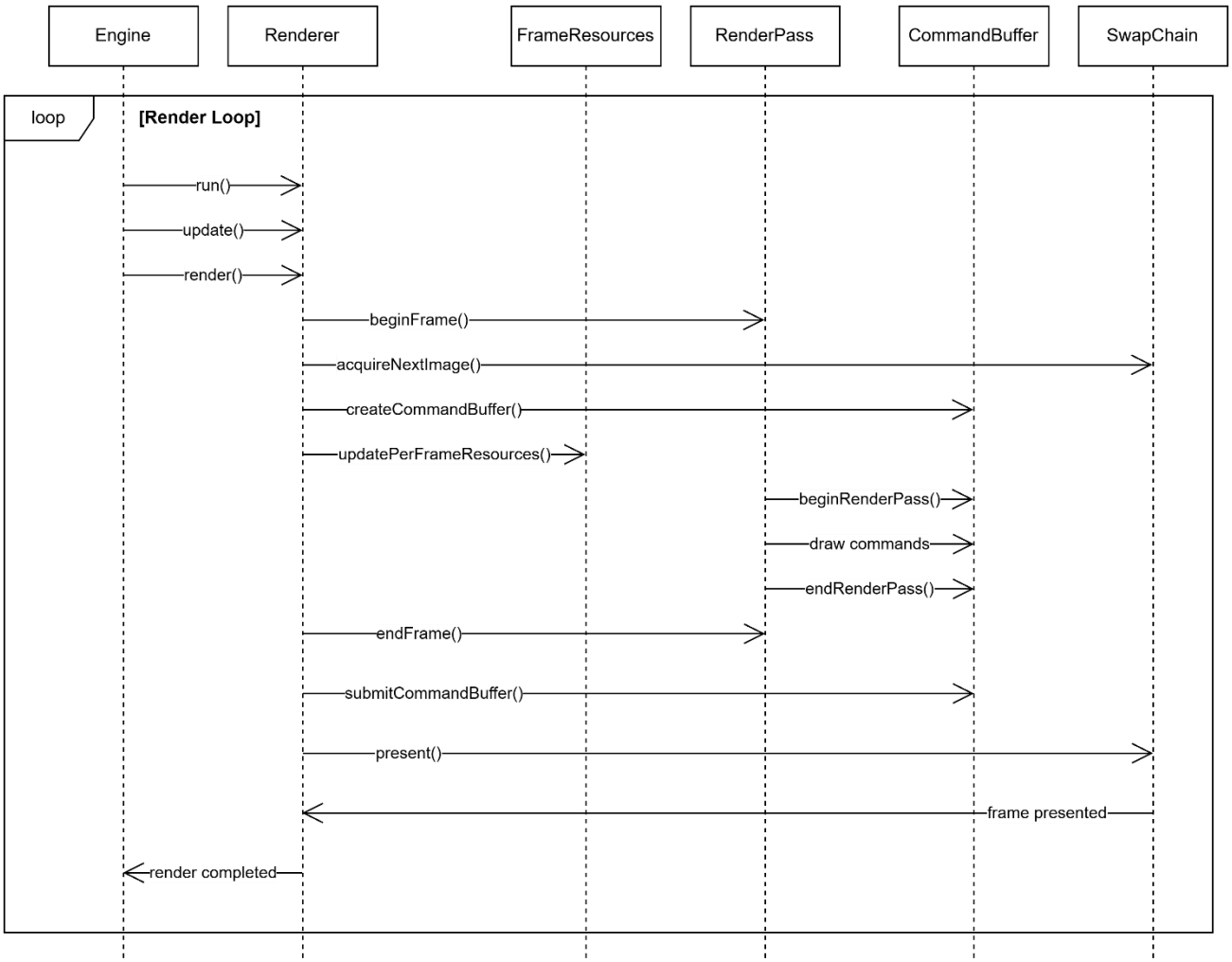

Рисунок 3.2 – Діаграма послідовності (виконано самостійно)

Процес рендеринга кадру починатиметься з виклику Engine::run(), який запускатиме цикл рендеринга. Далі виконуватиметься Engine::update(), що оновлюватиме стан рендерера, та Engine::render(), що запускатиме процес рендеринга поточного кадру. Renderer::beginFrame() виконуватиме початкову підготовку до рендеринга: отримуватиме наступне зображення з SwapChain, створюватиме CommandBuffer та оновлюватиме ресурси для поточного кадру через FrameResources::updatePerFrameResources().

Після підготовки ресурсів виконуватиметься підготовка рендер-пасів через RenderPass::prepareFrame(). Далі для кожного рендер-пасу виконуватиметься RenderPass::render(), який починатиме рендер-пас (CommandBuffer::beginRenderPass()), встановлюватиме стан пайплайну,

виконуватиме команди рендеринга та завершуватиме рендер-пас (CommandBuffer::endRenderPass()).

Останнім етапом буде виклик Renderer::endFrame(), який виконуватиме відправку командного буфера на виконання GPU та презентацію готового кадру через SwapChain::present(). Цей процес буде виконуватись кожен кадр, що забезпечить плавну анімацію та оновлення сцени.

Для забезпечення гнучкості, розширюваності та підтримки різних графічних API в системі буде використано ряд патернів проектування, кожен з яких вирішуватиме певну архітектурну проблему.

Entity Component System буде основним патерном для організації об'єктів сцени [12]. В цьому патерні об'єкти (Entity) складатимуться з набору компонентів (Component), які міститимуть дані, але не матимуть логіки. Логіка обробки даних міститиметься в системах (System), які оперуватимуть компонентами. Цей патерн дозволить гнучко комбінувати різні властивості об'єктів без необхідності створення складних ієрархій класів. В системі ECS буде реалізовано через бібліотеку EnTT, яка надає ефективну реалізацію реєстру сутностей (Registry) та засоби для управління компонентами та системами.

Factory Method використовуватиметься для створення графічних ресурсів, таких як буфери, текстури, шейдери та пайплайни. Цей патерн інкапсулюватиме створення конкретних об'єктів і дозволить підкласам визначати, який клас створювати. В системі цей патерн буде реалізований через методи createBuffer(), createTexture() та інші в класі Device, які створюватимуть конкретні реалізації ресурсів для кожного графічного API.

Strategy патерн використовуватиметься для реалізації різних стратегій рендеринга в DebugPass. Через цей патерн можна буде легко змінювати алгоритм рендеринга без зміни класу, який його використовує. Конкретні стратегії, такі як WireframeStrategy, NormalMapVisualizationStrategy, VertexNormalVisualizationStrategy та ShaderOverdrawStrategy, наслідуватимуть спільний інтерфейс DebugDrawStrategy і зможуть бути легко замінені одна на одну.

Service Locator використовуватиметься для доступу до глобальних сервісів, таких як ConfigManager, FileWatcherManager, InputManager, без прямих залежностей між компонентами. Цей патерн дозволить компонентам отримувати доступ до сервісів через глобальний реєстр, що спростить управління залежностями та тестування. У системі цей патерн буде реалізований через клас ServiceLocator, який надаватиме методи для реєстрації та отримання сервісів.

Adapter патерн використовуватиметься для забезпечення сумісності між різними інтерфейсами. В системі цей патерн застосовуватиметься для адаптації різних графічних API до єдиного інтерфейсу. Наприклад, класи DeviceVk та DeviceDx12 адаптуватимуть API Vulkan та DirectX 12 відповідно до спільного інтерфейсу Device.

Observer патерн використовуватиметься для системи подій, яка дозволить компонентам реагувати на події без прямих залежностей між ними. В системі цей патерн буде реалізований через класи EventManager, WindowEventManager, ApplicationEventManager та InputManager, які надаватимуть можливість реєстрації обробників подій та розсилки подій зареєстрованим обробникам.

В проєктуванні програмної системи буде застосовано ряд важливих принципів, які забезпечать її якість, гнучкість та підтримуваність.

Принципи SOLID будуть ключовими при проєктуванні архітектури системи [13]. Принцип єдиної відповідальності (Single Responsibility Principle) буде застосовано до класів системи, кожен з яких відповідатиме за конкретну функцію. Наприклад, клас Renderer відповідатиме лише за рендеринг, не займаючись управлінням сценами чи обробкою вводу.

Принцип відкритості-закритості (Open-Closed Principle) буде реалізований через використання інтерфейсів та абстрактних класів, які можна буде розширювати без зміни існуючого коду. Наприклад, система підтримки різних графічних API буде реалізована через інтерфейси Device, Buffer, Texture та інші, які матимуть конкретні реалізації для кожного API. Додавання підтримки нового API не вимагатиме зміни існуючого коду, а лише створення нових реалізацій цих інтерфейсів.

Принцип підстановки Лісков (Liskov Substitution Principle) буде дотриманий завдяки ретельному проєктуванню ієрархій класів та інтерфейсів. Наприклад, класи DeviceVk та DeviceDx12 зможуть бути використані замість інтерфейсу Device без зміни поведінки програми.

Принцип розділення інтерфейсу (Interface Segregation Principle) буде застосований через створення малих, специфічних інтерфейсів, які відповідатимуть конкретним потребам клієнтів. Наприклад, інтерфейси Buffer, Texture, Sampler та інші визначатимуть мінімальний набір методів, необхідних для роботи з відповідними ресурсами.

Принцип інверсії залежностей (Dependency Inversion Principle) буде реалізований через залежність високорівневих модулів від абстракцій, а не від конкретних реалізацій. Наприклад, Engine залежатиме від інтерфейсів Device, Buffer, Texture та інших, а не від їх конкретних реалізацій для певного API.

Окрім SOLID, у проєктуванні системи будуть застосовані інші важливі принципи. Принцип KISS (Keep It Simple, Stupid) буде дотримано через створення простих, зрозумілих класів з чітко визначеною відповідальністю. Принцип YAGNI (You Aren't Gonna Need It) буде застосовано через уникнення додавання функціональності, яка не є необхідною для задоволення поточних вимог. Принцип SoC (Separation of Concerns) буде реалізовано через розділення системи на модулі, кожен з яких відповідатиме за певний аспект функціональності.

Ці принципи в поєднанні з використаними патернами проєктування забезпечать гнучку, розширювану архітектуру, яка зможе бути легко адаптована до нових вимог та технологій.

3.3 Огляд алгоритмів та методів

Для ілюстрації основного алгоритму рендеринга кадру спроєктовано діаграму діяльності, показану на рисунку 4.3. Ця діаграма представляє послідовність кроків, необхідних для рендеринга одного кадру в системі.

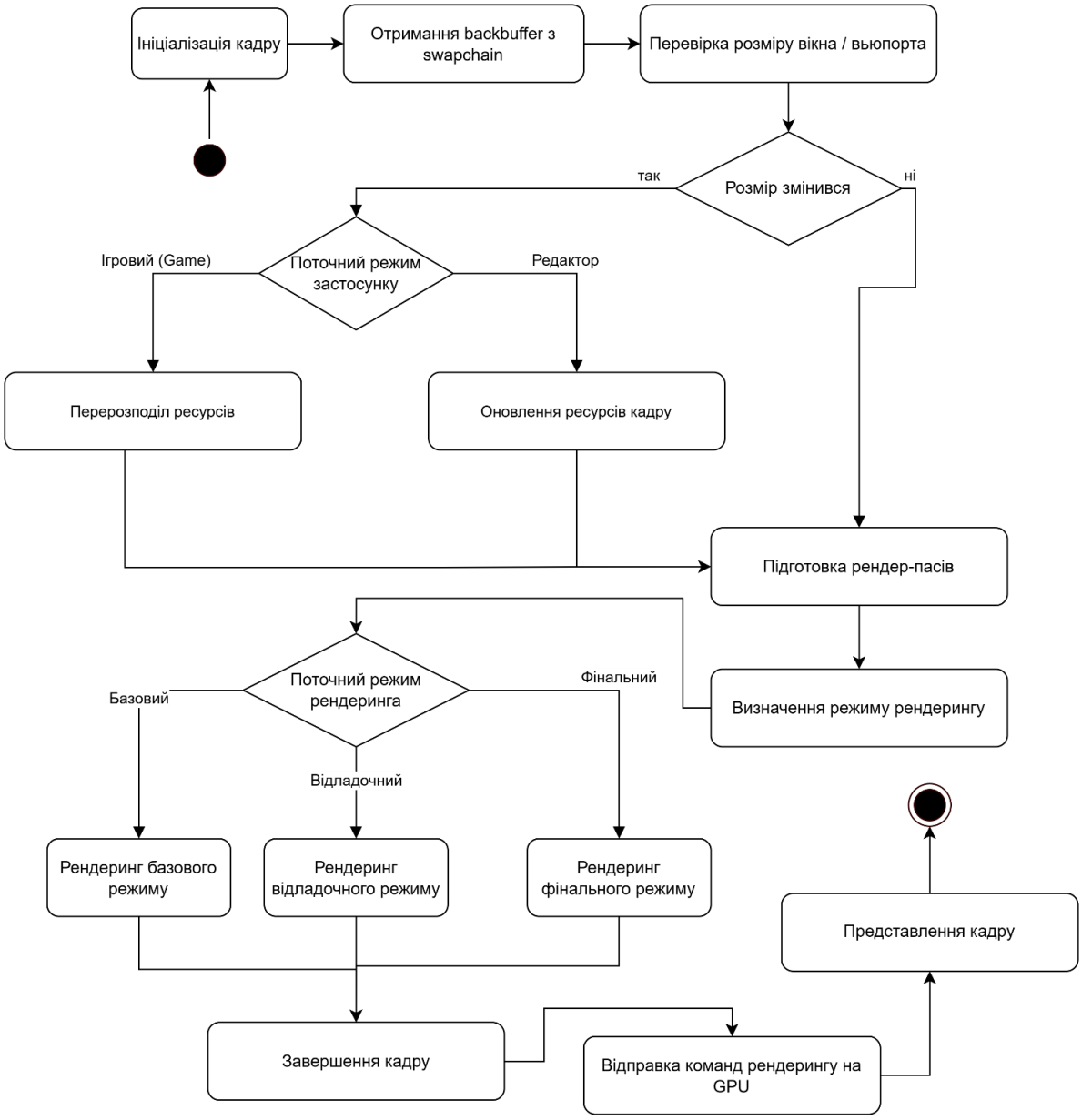

Рисунок 3.3 – Діаграма діяльності (виконано самостійно)

Процес рендеринга кадру починатиметься з ініціалізації кадру, що включатиме підготовку необхідних ресурсів та стану системи. Далі виконуватиметься отримання зображення з SwapChain, яке буде використовуватися

для рендеринга. Після цього система перевірятиме, чи змінився розмір вікна або видового екрану. Якщо так, виконуватиметься перерозподіл ресурсів, включаючи буфери, текстури та дескриптори. Якщо ні, система переходитиме до оновлення ресурсів кадру, що включатиме оновлення матриць камери, буферів трансформацій об'єктів та інших параметрів.

Після оновлення ресурсів виконуватиметься підготовка рендер-пасів, яка включатиме налаштування пайплайнів, буферів, текстур та дескрипторів для кожного рендер-пасу. Далі система визначатиме режим рендеринга, який може бути одним з трьох: базовий, відладочний або фінальний. Залежно від обраного режиму виконуватиметься відповідний рендер-пас: BasePass, DebugPass або FinalPass.

Після завершення рендер-пасу виконуватиметься завершення кадру, що включатиме звільнення тимчасових ресурсів та підготовку системи до наступного кадру. Далі команди рендеринга відправлятимуться на виконання GPU, і готовий кадр представлятиметься на екрані.

Діаграма діяльності допомагає зрозуміти загальний потік виконання процесу рендеринга та взаємодію між різними компонентами системи.

Ключовим елементом у процесі 3D-рендеринга є математична модель трансформації координат з одного простору в інший. У розробленій системі реалізовано повний ланцюжок трансформацій від локальних координат об'єкта до нормалізованих координат пристрою, які потім використовуються для растеризації.

Одним із ключових аспектів системи, що проєктується, буде абстрагування різних графічних API (DirectX 12, Vulkan) для забезпечення єдиного інтерфейсу для розробника. Це дозволить створювати програми, які зможуть працювати з різними графічними API без зміни високорівневого коду.

Основою абстракції буде інтерфейс Device, який представлятиме графічний пристрій і надаватиме методи для створення та управління графічними ресурсами. Цей інтерфейс матиме конкретні реалізації для кожного підтримуваного API: DeviceVk для Vulkan та DeviceDx12 для DirectX 12. Кожна реалізація

інкапсулюватиме детальну роботу з відповідним API, надаючи єдиний інтерфейс для інших компонентів системи.

Аналогічний підхід застосовуватиметься до інших графічних ресурсів: буферів, текстур, шейдерів, пайплайнів, командних буферів тощо. Кожен тип ресурсу матиме інтерфейс (Buffer, Texture, Shader тощо) та конкретні реалізації для кожного API (BufferVk, TextureVk, ShaderVk тощо для Vulkan та BufferDx12, TextureDx12, ShaderDx12 тощо для DirectX 12).

Для перемикання між різними API система використовуватиме Factory Method патерн через функцію g_createDevice(), яка створюватиме конкретну реалізацію Device на основі переданого параметра API. Це дозволить легко переключатися між різними API без зміни високорівневого коду.

Для конвертації між типами даних різних API система використовуватиме допоміжні функції, такі як getTextureFormatVk, getTextureFormatDx12, getImageLayoutVk, getResourceLayoutDx12 та інші. Ці функції перетворюватимуть типи даних з абстрактного інтерфейсу системи в конкретні типи даних API і навпаки.

Для управління станом ресурсів система використовуватиме абстракцію ResourceLayout, яка представлятиме стан ресурсу (наприклад, ShaderReadOnly, ColorAttachment, DepthStencilAttachment). Ця абстракція матиме відповідні перетворення для кожного API: VkImageLayout для Vulkan та D3D12_RESOURCE_STATES для DirectX 12.

Такий підхід дозволить ефективно абстрагувати різні графічні API та забезпечити єдиний інтерфейс для розробника, що значно спростить розробку кросплатформних додатків та експериментування з різними графічними API.

Ефективне управління графічними ресурсами є критичним аспектом продуктивної системи рендеринга. Спроєктована система міститиме ряд алгоритмів та методів для управління різними типами ресурсів: буферами, текстурами, шейдерами, моделями та матеріалами.

Для управління буферами система використовуватиме BufferManager, який надаватиме методи для створення, оновлення та звільнення буферів. Буфери

можуть бути вершинними (для вершинних даних), індексними (для індексів), константними (для шейдерних параметрів) або сховищними (для загального доступу). BufferManager також забезпечуватиме кешування буферів для уникнення повторного створення однакових буферів.

Управління текстурами здійснюватиметься через TextureManager, який надаватиме методи для завантаження, створення та управління текстурами. Текстури можуть бути завантажені з файлів різних форматів, створені як рендер-таргети або глибинні буфери, або створені програмно. TextureManager також забезпечуватиме кешування текстур та управління їх життєвим циклом.

Для управління шейдерами система використовуватиме ShaderManager, який надаватиме методи для завантаження, компіляції та кешування шейдерів. Шейдери можуть бути компільовані з HLSL-коду в SPIR-V для Vulkan або DXIL для DirectX 12. ShaderManager також підтримуватиме гаряче перезавантаження шейдерів під час розробки, що значно прискорюватиме процес налагодження.

Для управління моделями та матеріалами система використовуватиме ModelManager та MaterialManager відповідно. Ці менеджери надаватимуть методи для завантаження, створення та управління моделями та матеріалами. Моделі та матеріали можуть бути завантажені з різних форматів файлів, таких як OBJ, FBX, MTL тощо.

Для оптимізації використання пам'яті система застосовуватиме декілька методів:

1) використання спеціалізованих алокаторів пам'яті для графічних ресурсів, таких як VulkanMemoryAllocator для Vulkan та D3D12MemoryAllocator для DirectX 12. Ці алокатори оптимізуватимуть використання пам'яті GPU через стратегії суб-алокації, що дозволятиме уникнути фрагментації пам'яті та зменшити кількість алокацій;

2) система використовуватиме пули об'єктів для ресурсів, які часто створюються та знищуються, таких як командні буфери та дескриптори. Наприклад, CommandAllocatorManager для DirectX 12 та CommandPoolManager для

Vulkan управлятимуть пулами командних буферів, що дозволятиме ефективно перевикористовувати їх замість створення нових;

3) система застосовуватиме методи відкладеного звільнення ресурсів, що дозволятиме уникнути звільнення ресурсів, які ще використовуються GPU. Це досягатиметься через контроль синхронізації між CPU та GPU з використанням об'єктів Fence (для DirectX 12) та VkFence (для Vulkan).

Для забезпечення високої продуктивності система включатиме методи для профілювання та оптимізації рендеринга. Ці методи дозволять ідентифікувати та усунути «вузькі місця» в процесі рендеринга.

Для профілювання часу виконання система використовуватиме клас Stopwatch, який надаватиме високоточне вимірювання часу. Цей клас використовуватиметься для вимірювання часу виконання різних операцій, таких як оновлення стану, рендеринг, завантаження ресурсів тощо.

Система також включатиме TimingManager, який обчислюватиме та зберігатиме важливі метрики продуктивності, такі як час кадру (frame time), кількість кадрів на секунду (FPS) та загальний час виконання. Ці метрики використовуватимуться для моніторингу продуктивності та ідентифікації проблем.

Редактор (Editor) міститиме панель Performance, яка відображатиме метрики продуктивності у режимі реального часу. Ця панель включатиме поточний FPS, час кадру та графік часу кадру за останні кадри, що дозволить візуально ідентифікувати спади продуктивності.

Для оптимізації рендеринга система використовуватиме декілька методів.

1) сортування моделей за матеріалами (batching) для мінімізації зміни стану рендеринга. Моделі з однаковими матеріалами групуватимуться разом, що дозволить зменшити кількість викликів API для зміни шейдерів, текстур та інших параметрів;

2) система використовуватиме інстансовий рендеринг для ефективного рендеринга багатьох однакових об'єктів. Замість створення окремого набору команд рендеринга для кожного об'єкта, система групуватиме об'єкти з однаковою геометрією та використовуватиме інстансовий рендеринг для них;

3) система оптимізуватиме використання командних буферів через їх перевикористання та групування команд. Замість створення нових командних буферів для кожного кадру, система перевикористовуватиме існуючі, що зменшуватиме навантаження на CPU;

4) система використовуватиме рендер-паси для оптимізації порядку операцій рендеринга та переходів між ними. Рендер-паси дозволятимуть графічному драйверу оптимізувати використання пам'яті та порядок виконання операцій, що особливо важливо для мобільних платформ та тайлових GPU.

Для виявлення та вирішення проблем з рендерингом система включатиме різні режими візуалізації для налагодження, такі як Wireframe (для відображення каркасної моделі), NormalMapVisualization (для візуалізації нормальних мап), VertexNormalVisualization (для візуалізації вершинних нормалей) та ShaderOverdraw (для виявлення надлишкового рендеринга). Ці режими дозволять візуально ідентифікувати проблеми з геометрією, освітленням та продуктивністю.

Всі ці методи в поєднанні дозволять ефективно профілювати та оптимізувати процес рендеринга, забезпечуючи високу продуктивність системи на різних платформах та з різними графічними API.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Архітектурне розділення на шари

Одним із ключових програмних рішень при розробці програмної системи мульти-API для 3D-рендерінга стало впровадження багатошарової архітектури, що забезпечує чітке розділення відповідальностей між різними компонентами системи. Цей підхід базується на принципі Separation of Concerns, який дозволяє ізолювати різні аспекти функціональності системи та забезпечити високий рівень модульності і підтримуваності коду.

Архітектура системи побудована згідно з ієрархічною структурою, де кожен шар має чітко визначені обов'язки та взаємодіє лише з суміжними шарами через добре визначені інтерфейси. Верхній рівень представлений шаром додатку (Application Layer), що відповідає за логіку конкретного застосунку та обробку користувацького вводу. Наступний рівень складає ядро рушія (Engine Layer), який координує роботу всіх підсистем та забезпечує головний цикл виконання програми. Шар рендеринга (Rendering Layer) інкапсулює всю логіку, пов'язану з відображенням графіки, включаючи управління ресурсами та організацію проходів рендеринга. Найнижчий рівень становить шар апаратної абстракції рендеринга (Render Hardware Interface Layer), який забезпечує уніфікований доступ до різних графічних API.

Така багатошарова організація забезпечує ефективну ізоляцію змін та спрощує процес тестування окремих компонентів системи. Кожен шар може розвиватися незалежно, що дозволяє команді розробників працювати паралельно над різними аспектами функціональності без ризику конфліктів. Додатково, чітке розмежування відповідальностей значно спрощує діагностику проблем та оптимізацію продуктивності, оскільки кожен шар має обмежену область впливу. Архітектура також забезпечує можливість заміни реалізації будь-якого шару без впливу на інші компоненти системи.

Шар додатку реалізовано через клас Application, який успадковується від базового інтерфейсу та надає можливість створення конкретних застосунків з

власною логікою. На рисунку 4.1 представлено основну структуру цього класу. Цей шар інкапсулює всю логіку, специфічну для конкретного застосунку, включаючи обробку користувацького вводу, управління камерою та взаємодію зі сценою.

```
class Application {
  public:
  enum class Action {
    MoveForward,
    MoveBackward,
    MoveLeft,
    MoveRight,
    MoveUp,
    MoveDown,
    Count,
  };

  Application() {}
  virtual ~Application() {}

  void setup();
  void update(float deltaTime);
  void linkWithEngine(const Engine& engine);
  void onMouseMove(int32_t xOffset, int32_t yOffset);

  private:
  void setupInputHandlers();

  Window* m_window_ = nullptr;
  float m_movingSpeed_ = 10.0f;
  float m_mouseSensitivity_ = 0.1f;
  std::bitset<static_cast<size_t>(Action::Count)> m_actionStates_;
  Scene* m_scene_ = nullptr;
};
```

Рисунок 4.1 – Програмний код класу Application шару додатку (виконано самостійно)

Особливо важливим аспектом реалізації класу Application є використання енумерації Action для категоризації дій користувача та ефективне управління їх станом через std::bitset. Цей підхід дозволяє швидко перевіряти активні дії користувача та обробляти одночасні комбінації клавіш без необхідності складних умовних конструкцій. Використання значення Count в енумерації забезпечує автоматичне визначення розміру bitset, що робить систему легко розширюваною

при додаванні нових типів дій. Параметри movingSpeed та mouseSensitivity надають можливість тонкого налаштування поведінки камери та руху об'єктів відповідно до потреб конкретного застосунку.

Цей шар інкапсулює всю логіку, специфічну для конкретного застосунку, включаючи обробку користувацького вводу, управління камерою та взаємодію зі сценою. Ядро системи представлене класом Engine, який координує роботу всіх підсистем та забезпечує їх правильну ініціалізацію і взаємодію. На рисунку 4.2 показано ключові методи цього класу.

```
1   class Engine {
2     public:
3     Engine() = default;
4     ~Engine();
5
6     auto initialize() -> bool;
7     void render();
8     void run();
9     void setGame(Application* game);
10
11    Window* getWindow() const { return m_window_.get(); }
12
13    private:
14    void processEvents_();
15    void update_(float deltaTime);
16
17    bool m_isRunning_{false};
18    gfx::renderer::ApplicationRenderMode m_applicationMode;
19    std::unique_ptr<Window> m_window_;
20    std::unique_ptr<gfx::renderer::Renderer> m_renderer_;
21    std::unique_ptr<Editor> m_editor_;
22    Application* m_application_;
23  };
```

Рисунок 4.2 – Програмний код класу Engine ядра системи (виконано самостійно)

Рушій використовує патерн Service Locator для управління залежностями між компонентами, що забезпечує слабке зв'язування та полегшує тестування окремих модулів. Ініціалізація сервісів відбувається в методі initialize(), де реєструються всі необхідні менеджери та підсистеми.

Шар рендеринга організовано навколо класу Renderer, який координує процес відображення кадрів та управляє графічними ресурсами. Для забезпечення гнучкості та ефективності процесу візуалізації було прийнято рішення про структурування рендеринга через систему проходів (render passes). Кожен прохід відповідає за конкретний аспект візуалізації та може працювати незалежно від інших, що дозволяє легко додавати нові ефекти, змінювати порядок обробки та оптимізувати продуктивність системи. На рисунку 4.3 представлено структуру основного класу рендерера.

```
class Renderer {
  public:
  Renderer() = default;

  ~Renderer() { waitForAllFrames_(); }

  bool initialize(Window* window, rhi::RenderingApi api);

  RenderContext beginFrame(Scene* scene, const RenderSettings& renderSettings);
  void          renderFrame(RenderContext& context);
  void          endFrame(RenderContext& context);

  rhi::Device* getDevice() const { return m_device.get(); }
  RenderResourceManager* getResourceManager() const { return m_resourceManager.get(); }

  private:
  void setupRenderPasses_();
  void waitForAllFrames_();

  std::unique_ptr<rhi::Device> m_device;
  std::unique_ptr<rhi::SwapChain> m_swapChain;
  std::unique_ptr<RenderResourceManager> m_resourceManager;
  std::unique_ptr<FrameResources> m_frameResources;

  std::unique_ptr<BasePass> m_basePass;
  std::unique_ptr<FinalPass> m_finalPass;
  std::unique_ptr<DebugPass> m_debugPass;
};
```

Рисунок 4.3 – Програмний код класу Renderer шару рендеринга (виконано самостійно)

Рендерер організує процес відображення через серію проходів рендеринга, кожен з яких відповідає за конкретний аспект візуалізації. Архітектура проходів базується на патерні Strategy, де кожен прохід реалізує специфічний алгоритм відображення. Це рішення забезпечує високу модульність системи та спрощує додавання нових технік рендеринга без необхідності модифікації існуючого коду.

Базовий прохід (BasePass) відповідає за основне рендеринг геометрії сцени з урахуванням освітлення, матеріалів та текстур. Цей прохід виконує найбільший обсяг роботи у процесі формування кадру та забезпечує реалістичне відображення 3D-об'єктів. На рисунку 4.4 показано ключові методи цього класу.

```
class BasePass : public RenderPass {
  public:
  BasePass() = default;

  ~BasePass() override { cleanup(); }

  void initialize(rhi::Device*           device,
                  RenderResourceManager* resourceManager,
                  FrameResources*        frameResources,
                  rhi::ShaderManager*    shaderManager) override;

  void resize(const math::Dimension2Di& newDimension) override;
  void prepareFrame(const RenderContext& context) override;
  void render(const RenderContext& context) override;
  void cleanup() override;

  private:
  void setupRenderPass_();
  void updateInstanceBuffer_(RenderModel* model,
                             const std::vector<math::Matrix4f<>>& matrices,
                             ModelBufferCache& cache);
  void prepareDrawCalls_(const RenderContext& context);

  std::unordered_map<RenderModel*, ModelBufferCache> m_instanceBufferCache;
  std::vector<DrawData> m_drawData;
};
```

Рисунок 4.4 – Програмний код класу BasePass основного проходу рендеринга (виконано самостійно)

BasePass використовує техніку інстансингу для ефективного рендеринга множинних копій однакових об'єктів. Метод updateInstanceBuffer_() оновлює буфери з матрицями трансформацій для кожної моделі, а prepareDrawCalls_() підготовлює команди малювання з урахуванням матеріалів та дескрипторних наборів. Це забезпечує оптимальну продуктивність при роботі зі складними сценами, що містять велику кількість об'єктів.

Фінальний прохід (FinalPass) виконує копіювання результату рендеринга з внутрішнього кольорового буфера у буфер відображення. Цей прохід активується лише у ігровому режимі додатку, коли необхідно вивести фінальне зображення на

екран. У режимі редактора результат залишається у внутрішньому буфері для подальшої обробки інтерфейсом користувача. На рисунку 4.5 представлено реалізацію цього проходу.

```
class FinalPass : public RenderPass {
  public:
  FinalPass() = default;

  ~FinalPass() override { cleanup(); }

  void initialize(rhi::Device*           device,
                  RenderResourceManager* resourceManager,
                  FrameResources*        frameResources,
                  rhi::ShaderManager*    shaderManager) override;

  void render(const RenderContext& context) override;

  private:
  rhi::Device*           m_device          = nullptr;
  RenderResourceManager* m_resourceManager = nullptr;
  FrameResources*        m_frameResources  = nullptr;
};
```

Рисунок 4.5 – Програмний код класу FinalPass фінального проходу рендеринга (виконано самостійно)

Простота реалізації FinalPass обумовлена його специфічною функцією - він лише копіює текстуру з одного буфера в інший. Метод render() перевіряє режим роботи додатку та виконує копіювання тільки у випадку ігрового режиму, що забезпечує правильне відображення результатів рендеринга для кінцевого користувача.

Організація системи проходів рендеринга дозволяє легко розширювати функціональність візуалізації шляхом додавання нових проходів без модифікації існуючого коду. Кожен прохід працює з унифікованим RenderContext, що забезпечує консистентність даних та спрощує інтеграцію нових алгоритмів рендеринга. Послідовність виконання проходів контролюється рендерером, який може динамічно змінювати порядок обробки залежно від поточних налаштувань

візуалізації. Така архітектура особливо корисна для реалізації складних ефектів, які потребують множинних етапів обробки.

Найбільш складним компонентом системи проходів є DebugPass, який забезпечує різноманітні режими налагодження та візуалізації. Цей прохід використовує патерн Strategy для динамічного вибору алгоритму відображення залежно від поточного режиму дебагу. На рисунку 4.6 показано структуру цього класу.

```
class DebugPass : public RenderPass {
  public:
  DebugPass();
  ~DebugPass() override;

  void initialize(rhi::Device*           device,
                  RenderResourceManager* resourceManager,
                  FrameResources*        frameResources,
                  rhi::ShaderManager*    shaderManager) override;

  void prepareFrame(const RenderContext& context) override;
  void render(const RenderContext& context) override;
  void cleanup() override;

  bool isExclusive() const override;

  private:
  void createDebugStrategy_();

  std::unique_ptr<DebugDrawStrategy> m_debugStrategy;
  RenderMode m_currentRenderMode = RenderMode::Solid;
};
```

Рисунок 4.6 – Програмний код класу DebugPass для налагоджувального рендеринга (виконано самостійно)

DebugPass підтримує кілька режимів візуалізації, включаючи каркасний рендеринг (wireframe), візуалізацію карт нормалей, відображення перевитрат шейдерів (shader overdraw) та візуалізацію освітлення. Кожен режим реалізований через окрему стратегію, що успадковується від базового класу DebugDrawStrategy. Метод createDebugStrategy_() динамічно створює відповідну стратегію залежно від поточного режиму рендеринга, забезпечуючи гнучкість системи налагодження.

Практичне застосування цих режимів візуалізації значно прискорює процес виявлення та усунення проблем під час розробки графічних додатків. Каркасний режим дозволяє аналізувати топологію мешів та виявляти проблеми з геометрією, візуалізація нормалей допомагає діагностувати некоректне освітлення об'єктів, а режим shader overdraw виявляє неефективні ділянки рендеринга з надмірним перекриттям піксельних шейдерів. Інтеграція системи налагодження з графічним редактором забезпечує можливість миттєвого переключення між режимами через користувацький інтерфейс, що дозволяє розробникам швидко порівнювати різні аспекти візуалізації сцени без перезапуску програми.

Для підтримки різних режимів дебагу використано патерн Strategy, що дозволяє динамічно змінювати алгоритми візуалізації. На рисунку 4.7 показано базовий інтерфейс стратегій дебагу.

```
class DebugDrawStrategy {
  public:
  DebugDrawStrategy() = default;
  virtual ~DebugDrawStrategy() = default;

  virtual void initialize(rhi::Device* device,
                          RenderResourceManager* resourceManager,
                          FrameResources* frameResources,
                          rhi::ShaderManager* shaderManager) = 0;

  virtual void resize(const math::Dimension2Di& newDimension) = 0;
  virtual void prepareFrame(const RenderContext& context) = 0;
  virtual void render(const RenderContext& context) = 0;
  virtual void cleanup() = 0;

  virtual bool isExclusive() const { return false; }
};
```

Рисунок 4.7 – Програмний код базового інтерфейсу стратегій дебагу (виконано самостійно)

Цей підхід забезпечує легке додавання нових режимів візуалізації без модифікації існуючого коду. Конкретні стратегії, такі як WireframeStrategy або

NormalMapVisualizationStrategy, реалізують специфічні алгоритми рендеринга для різних цілей налагодження.

Система налагоджувальних режимів інтегрована з інтерфейсом редактора, дозволяючи розробникам швидко перемикатися між різними режимами візуалізації під час роботи над проектом. Це значно прискорює процес діагностики проблем з освітленням, геометрією та продуктивністю рендеринга. Можливість динамічного переключення між режимами без перезапуску програми забезпечує безперервний workflow розробки та тестування. Додатково, система підтримує одночасну роботу кількох режимів дебагу, що дозволяє комплексно аналізувати різні аспекти процесу рендеринга в межах одного сеансу налагодження.

Найнижчий рівень архітектури представлений шаром Render Hardware Interface, який абстрагує специфічні особливості різних графічних API. Цей шар використовує патерн Abstract Factory для створення об'єктів, специфічних для конкретного API, та патерн Adapter для уніфікації інтерфейсів. На рисунку 4.8 показано приклад фабричної функції для створення пристрою рендеринга.

```
1  std::unique_ptr<rhi::Device> g_createDevice(rhi::RenderingApi api,
2                                              const rhi::DeviceDesc& desc) {
3    switch (api) {
4      case rhi::RenderingApi::Vulkan:
5        return std::make_unique<rhi::DeviceVk>(desc);
6      case rhi::RenderingApi::Dx12:
7        return std::make_unique<rhi::DeviceDx12>(desc);
8      default:
9        return nullptr;
10   }
11 }
```

Рисунок 4.8 – Програмний код фабричної функції створення пристрою Render Hardware Interface (виконано самостійно)

Такий підхід дозволяє системі підтримувати множинні графічні API без необхідності змін у вищих шарах архітектури. Кожна реалізація API інкапсулює всі специфічні деталі взаємодії з відповідною графічною бібліотекою, забезпечуючи уніфікований інтерфейс для шару рендеринга. Розроблена архітектура системи

рендеринга забезпечує високу продуктивність завдяки оптимізації кожного етапу візуалізації, спрощує процес налагодження через спеціалізовані режими відображення та дозволяє легко розширювати функціональність додавання нових проходів без порушення існуючої структури.

4.2 Система абстракції графічних API

Розробка сучасних графічних додатків стикається з необхідністю підтримки різних графічних API, зокрема Vulkan та DirectX 12, кожен з яких має свої особливості та переваги на різних платформах. Для вирішення цієї проблеми було прийнято рішення про створення системи Render Hardware Interface (Render Hardware Interface) - шару абстракції, який забезпечує уніфікований доступ до функцій графічного апаратного забезпечення незалежно від конкретного API.

Система Render Hardware Interface побудована на основі патерна Abstract Factory, який дозволяє створювати сімейства пов'язаних об'єктів без прив'язки до конкретних класів їх реалізації. Цей підхід забезпечує можливість динамічного вибору графічного API під час виконання програми та спрощує процес додавання підтримки нових API у майбутньому. Архітектура системи включає базові абстрактні інтерфейси для всіх ключових графічних ресурсів та операцій, а також конкретні реалізації для кожного підтримуваного API.

Центральним елементом системи є інтерфейс Device, який представляє логічний пристрій рендеринга та відповідає за створення всіх графічних ресурсів. Цей інтерфейс використовує патерн Factory Method для створення об'єктів, специфічних для конкретного API, забезпечуючи при цьому єдиний програмний інтерфейс для клієнтського коду. На рисунку 4.9 показано основну структуру базового інтерфейсу Device.

Інтерфейс Device визначає повний набір методів для створення та управління графічними ресурсами, включаючи буфери, текстури, шейдери, конвеєри рендеринга та об'єкти синхронізації. Використання чисто віртуальних методів забезпечує строгу типізацію та гарантує, що всі конкретні реалізації надають повну функціональність, необхідну для роботи системи рендеринга.

Архітектура Render Hardware Interface значно спрощує процес розробки та підтримки кросплатформних графічних додатків, дозволяючи розробникам зосередитися на логіці рендеринга замість специфічних деталей API. Система автоматично обирає оптимальні шляхи виконання операцій для кожного конкретного API, забезпечуючи максимальну продуктивність без необхідності ручної оптимізації. Додатково, уніфікований інтерфейс дозволяє легко тестувати та порівнювати продуктивність різних графічних API на одній і тій же кодовій базі. Така абстракція також створює надійну основу для майбутнього розширення системи підтримкою додаткових API без порушення існуючої архітектури.

```
class Device {
  public:
  Device(const DeviceDesc& desc)
      : m_window_(desc.window) {}
  virtual ~Device() = default;
  virtual RenderingApi getApiType() const = 0;
  virtual std::unique_ptr<Buffer>              createBuffer(const BufferDesc& desc) = 0;
  virtual std::unique_ptr<Texture>             createTexture(const TextureDesc& desc) = 0;
  virtual std::unique_ptr<Sampler>             createSampler(const SamplerDesc& desc) = 0;
  virtual std::unique_ptr<Shader>              createShader(const ShaderDesc& desc) = 0;
  virtual std::unique_ptr<GraphicsPipeline>    createGraphicsPipeline(const GraphicsPipelineDesc& desc) = 0;
  virtual std::unique_ptr<DescriptorSetLayout> createDescriptorSetLayout(const DescriptorSetLayoutDesc& desc) = 0;
  virtual std::unique_ptr<DescriptorSet>       createDescriptorSet(const DescriptorSetLayout* layout) = 0;
  virtual std::unique_ptr<RenderPass>          createRenderPass(const RenderPassDesc& desc) = 0;
  virtual std::unique_ptr<Framebuffer>         createFramebuffer(const FramebufferDesc& desc) = 0;
  virtual std::unique_ptr<CommandBuffer>       createCommandBuffer(const CommandBufferDesc& desc = CommandBufferDesc()) = 0;
  virtual std::unique_ptr<Fence>               createFence(const FenceDesc& desc = FenceDesc()) = 0;
  virtual std::unique_ptr<Semaphore>           createSemaphore() = 0;
  virtual std::unique_ptr<SwapChain>           createSwapChain(const SwapchainDesc& desc) = 0;
  virtual void updateBuffer(Buffer* buffer, const void* data, size_t size, size_t offset = 0) = 0;
  virtual void updateTexture(Texture* texture, const void* data, size_t dataSize, uint32_t mipLevel = 0, uint32_t arrayLayer = 0) = 0;
  virtual void submitCommandBuffer(CommandBuffer* cmdBuffer, Fence* signalFence = nullptr,
                                   const std::vector<Semaphore*>& waitSemaphores = {},
                                   const std::vector<Semaphore*>& signalSemaphores = {}) = 0;
  virtual void waitIdle() = 0;
  private:
  const Window* const m_window_;
};
```

Рисунок 4.9 – Програмний код базового інтерфейсу Device системи Render Hardware Interface (виконано самостійно)

Кожен конкретний тип графічного ресурсу також має свою ієрархію класів, побудовану за тим же принципом. Наприклад, для представлення буферів використовується базовий абстрактний клас Buffer та його конкретні реалізації BufferVk і BufferDx12 для Vulkan та DirectX 12 відповідно. Цей підхід забезпечує консистентність архітектури та спрощує розуміння коду.

Ієрархічна організація графічних ресурсів дозволяє ефективно управляти життєвим циклом об'єктів та забезпечувати типобезпечність операцій з ресурсами. Базові класи інкапсулюють загальні властивості та поведінку, такі як розмір, тип та прапори створення, які є актуальними для всіх API, тоді як конкретні реалізації додають специфічну для API логіку управління пам'яттю та взаємодії з драйверами. Такий розподіл відповідальностей дозволяє системі автоматично обирати найбільш оптимальні стратегії роботи з ресурсами для кожного графічного API. Додатково, використання віртуального деструктора у базових класах гарантує коректне вивільнення ресурсів незалежно від конкретного типу реалізації.

На рисунку 4.10 представлено структуру базового класу Buffer.

```
class Buffer {
  public:
  Buffer(const BufferDesc& desc)
      : m_desc_(desc) {}
  virtual ~Buffer() = default;
  uint64_t getSize() const { return m_desc_.size; }
  BufferType getType() const { return m_desc_.type; }
  BufferCreateFlag getCreateFlags() const { return m_desc_.createFlags; }
  const BufferDesc& getDesc() const { return m_desc_; }
  protected:
  BufferDesc m_desc_;
};
```

Рисунок 4.10 – Програмний код базового класу Buffer системи Render Hardware Interface (виконано самостійно)

Базовий клас Buffer інкапсулює загальні властивості всіх типів буферів, такі як розмір, тип та прапорці створення, забезпечуючи уніфікований доступ до цієї інформації незалежно від конкретної реалізації API. Конкретні реалізації, такі як BufferVk та BufferDx12, розширюють цю функціональність специфічними для API деталями, включаючи управління пам'яттю та взаємодію з драйверами.

Особливо важливим аспектом системи Render Hardware Interface є уніфікація енумерацій та констант, які можуть значно відрізнятися між різними API. Для вирішення цієї проблеми створено систему перетворення енумерацій, яка використовує статичні таблиці відповідностей для конвертації між внутрішніми типами Render Hardware Interface та специфічними для API константами. На

рисунку 4.11 показано приклад такої системи перетворення для форматів текстур у DirectX 12.

```
static const std::unordered_map<TextureFormat, DXGI_FORMAT> textureFormatMappingToDXGI = {
  { TextureFormat::Rgb8,       DXGI_FORMAT_R8G8B8A8_UNORM     },
  { TextureFormat::Rgba8,      DXGI_FORMAT_R8G8B8A8_UNORM     },
  { TextureFormat::Rgba16f,    DXGI_FORMAT_R16G16B16A16_FLOAT },
  { TextureFormat::Rgba32f,    DXGI_FORMAT_R32G32B32A32_FLOAT },
  { TextureFormat::D24S8,      DXGI_FORMAT_D24_UNORM_S8_UINT  },
  { TextureFormat::D32,        DXGI_FORMAT_D32_FLOAT          }
};

DXGI_FORMAT g_getTextureFormatDx12(TextureFormat textureFormat) {
  return getEnumMapping(textureFormatMappingToDXGI, textureFormat, DXGI_FORMAT_UNKNOWN);
}
```

Рисунок 4.11 – Програмний код системи перетворення енумерацій для DirectX 12 (виконано самостійно)

Система використовує патерн Strategy для реалізації різних підходів до управління ресурсами в залежності від обраного API. Кожна конкретна реалізація інкапсулює специфічну логіку створення, управління та знищення ресурсів, що дозволяє оптимально використовувати можливості кожного API, зберігаючи при цьому уніфікований інтерфейс.

Додатково система Render Hardware Interface використовує патерн Adapter для забезпечення сумісності між концепціями, які по-різному реалізовані в різних API. Наприклад, концепція render pass у Vulkan не має прямого аналога в DirectX 12, тому клас RenderPassDx12 емулює цю функціональність через управління станом render target та операціями очищення.

Впровадження системи Render Hardware Interface значно спрощує розробку графічних додатків, дозволяючи розробникам зосередитися на логіці рендеринга замість деталей конкретних API. Система забезпечує портативність коду між різними платформами та графічними API, полегшує тестування та налагодження за рахунок уніфікованих інтерфейсів, а також створює основу для майбутнього розширення підтримки додаткових графічних API без необхідності модифікації клієнтського коду.

4.3 Система математичних трансформацій та управління координатними системами

Розробка ефективної системи 3D-рендеринга потребує комплексного підходу до управління математичними трансформаціями координат, що забезпечують перетворення геометричних даних від локальних координат об'єктів до фінального відображення на екрані. Прийняте рішення базується на створенні власної математичної бібліотеки, оптимізованої під специфічні потреби графічного рушія з підтримкою SIMD-інструкцій для прискорення обчислень.

Основою системи трансформацій є математична модель послідовного перетворення координат, що виражається загальною формулою:

$$\overrightarrow{\mathrm{v_{NDC}}} = \frac{\overrightarrow{\mathrm{v_{object}}} \cdot M \cdot V \cdot P}{\mathrm{w}}, \quad (1)$$

де $\overrightarrow{v_{object}}$ – координати вершин в локальному просторі об'єкта;

M – матриця моделі (Model matrix);

V – матриця виду (View matrix);

P – матриця проекції (Projection matrix);

$\overrightarrow{\mathrm{v_{NDC}}}$ – координати вершин в нормалізованому просторі пристрою;

w – четверта координата точки після застосування матриць (використовується для перспективного ділення).

Матриця моделі перетворює координати з локального простору об'єкта у світовий простір, включаючи операції масштабування, обертання та переміщення:

$$M = S \cdot R \cdot T \quad (2)$$

де S – матриця масштабування;

R – матриця обертання;

T – матриця переміщення.

Матриця виду перетворює координати зі світового простору в простір камери, представляючи обернену матрицю трансформації камери:

$$V = (C)^{-1} \tag{3}$$

де C – матриця, що представляє положення та орієнтацію камери в світовому просторі.

Матриця проекції залежить від типу проекції. Для перспективної проекції у лівосторонній системі координат з діапазоном глибини 0-1:

$$\begin{pmatrix} \frac{1}{\tan\frac{fov}{2}*aspect} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{fov}{2}} & 0 & 0 \\ 0 & 0 & \frac{far+near}{near-far} & -1 \\ 0 & 0 & \frac{2*near*far}{near-far} & 0 \end{pmatrix} \tag{4}$$

де fov – поле зору (field of view) у радіанах;

$aspect$ – співвідношення сторін вікна (ширина/висота);

$near$ – відстань до ближньої площини відсічення;

far – відстань до дальньої площини відсічення.

Перспективне ділення завершує трансформацію згідно з формулою (1), переводячи координати з простору відсічення в нормалізований простір пристрою:

$$\overrightarrow{v_{NDC}} = \left(\frac{x_{clip}}{w_{clip}}, \frac{y_{clip}}{w_{clip}}, \frac{z_{clip}}{w_{clip}}\right) \tag{5}$$

де x_{clip},y_{clip}, z_{clip} – координати вершин після застосування матриці проекції;

w_{clip}– четверта координата вершин після застосування матриці проекції, яка використовується для створення ефекту перспективи.

Програмна реалізація цієї математичної моделі базується на спеціалізованій математичній бібліотеці з підтримкою різних координатних систем. Реалізація матриці проекції згідно з формулою (4) представлена на рисунку 4.12.

```
template <typename T, Options Option = Options::RowMajor>
auto g_perspectiveLhZo(T fovY, T aspect, T nearZ, T farZ)
    -> Matrix<T, 4, 4, Option> {
  assert(std::abs(aspect - std::numeric_limits<T>::epsilon())
         > static_cast<T>(0));
  const T tanHalfFovY = std::tan(fovY / static_cast<T>(2));
  Matrix<T, 4, 4, Option> perspeciveMatrix;
  const T scaleX = static_cast<T>(1) / (tanHalfFovY * aspect);
  const T scaleY = static_cast<T>(1) / (tanHalfFovY);
  const T scaleZ          =   farZ / (farZ - nearZ);
  const T translateZ      = -(farZ * nearZ) / (farZ - nearZ);
  const T handednessScale =   static_cast<T>(1);
  if constexpr (Option == Options::RowMajor) {
    perspeciveMatrix
      scaleX,  T(),     T(),         T(),
      T(),     scaleY,  T(),         T(),
      T(),     T(),     scaleZ,      handednessScale,
      T(),     T(),     translateZ,  T();
  }
  return perspeciveMatrix;
}
```

Рисунок 4.12 – Програмний код функції створення лівосторонньої перспективної проекції (виконано самостійно)

Шаблонна природа функції g_perspectiveLhZo дозволяє використовувати її з різними типами чисел з плаваючою крапкою, забезпечуючи гнучкість у виборі точності обчислень залежно від потреб конкретного застосунку. Використання constexpr if забезпечує компіляційну оптимізацію для різних варіантів організації матриць, дозволяючи компілятору генерувати найбільш ефективний код для кожної конфігурації. Функція включає перевірку валідності аспектного співвідношення через assert, що запобігає виникненню некоректних матриць проекції та потенційних помилок рендеринга. Ця реалізація забезпечує повну сумісність з різними графічними API, які можуть мати відмінні конвенції щодо організації матриць та систем координат.

Система трансформацій об'єктів реалізована через компонент Transform, що інкапсулює параметри позиціонування згідно з формулою (2). На рисунку 4.13 показано структуру компонента та функцію обчислення матриці трансформації.

```cpp
struct Transform {
  math::Vector3Df translation;
  math::Vector3Df rotation;
  math::Vector3Df scale = math::Vector3Df(1.0f, 1.0f, 1.0f);
  bool isDirty = false;
};

inline math::Matrix4f<> calculateTransformMatrix(const Transform& transform) {
  math::Matrix4f<> transformMatrix;
  transformMatrix = math::g_scale(transform.scale);
  auto pitch = math::g_degreeToRadian(transform.rotation.x());
  auto yaw   = math::g_degreeToRadian(transform.rotation.y());
  auto roll  = math::g_degreeToRadian(transform.rotation.z());
  auto q = math::Quaternionf::fromEulerAngles(pitch, yaw, roll,
                                              math::EulerRotationOrder::XYZ);
  auto rotationMatrix = math::Matrix4f<>::Identity();
  auto temp = q.toRotationMatrix();
  for (std::size_t i = 0; i < 3; ++i) {
    for (std::size_t j = 0; j < 3; ++j) {
      rotationMatrix(i, j) = temp(i, j);
    }
  }
  transformMatrix *= rotationMatrix;
  math::g_addTranslate(transformMatrix, transform.translation);
  return transformMatrix;
}
```

Рисунок 4.13 – Програмний код компонента Transform та обчислення матриці трансформації (виконано самостійно)

Функція calculateTransformMatrix реалізує формулу (2), послідовно застосовуючи масштабування, обертання через кватерніони та переміщення. Використання кватерніонів замість прямого обчислення матриці обертання з кутів Ейлера забезпечує стабільність та уникнення проблем «gimbal lock». Додатково, такий підхід забезпечує більш ефективні інтерполяційні операції для анімацій та плавних переходів між різними орієнтаціями об'єктів.

Практичне застосування математичних трансформацій відбувається у вершинних шейдерах, де реалізується повний ланцюжок перетворень координат

згідно з формулою (1). На рисунку 4.14 представлено фрагмент vertex shader, що демонструє використання цієї формули.

```
VSOutput main(VSInput input)
{
    VSOutput output = (VSOutput) 0;
    float4x4 worldMatrix = mul(input.Instance, ModelParam.ModelMatrix);
    float4 worldPos = mul(worldMatrix, float4(input.Position, 1.0));
    output.WorldPos = worldPos.xyz;
    output.Position = mul(ViewParam.VP, worldPos);
    float3x3 normalMat =
    {
        normalize(worldMatrix[0].xyz),
        normalize(worldMatrix[1].xyz),
        normalize(worldMatrix[2].xyz)
    };
    output.Normal = normalize(mul(normalMat, input.Normal));
    output.Tangent = normalize(mul(normalMat, input.Tangent));
    output.Bitangent = normalize(mul(normalMat, input.Bitangent));
    return output;
}
```

Рисунок 4.14 – Програмний код vertex shader з реалізацією трансформацій координат (виконано самостійно)

У цьому коді послідовно застосовуються трансформації згідно з формулою (1): спочатку перетворення з локального простору у світовий через множення матриці інстансингу на матрицю моделі (компонент M формули), потім трансформація у простір відсічення через матрицю view-projection (компоненти V та P формули). Перспективне ділення згідно з формулою (5) виконується автоматично апаратурою GPU після завершення vertex shader.

Система камер автоматично генерує необхідні матриці на основі параметрів камери, реалізуючи формули (3) та (4). На рисунку 4.15 показано реалізацію системи оновлення камер.

Інтеграція системи камер з архітектурою Entity Component System забезпечує ефективне масштабування для сцен з множинними камерами та спрощує управління їх життєвим циклом. Використання компонентно-орієнтованого

підходу дозволяє динамічно додавати та видаляти камери без впливу на продуктивність системи рендеринга. Кешування обчислених матриць у компоненті CameraMatrices мінімізує повторні обчислення та підвищує загальну ефективність системи.

```
void CameraSystem::update(Scene* scene, float deltaTime) {
  Registry& registry = scene->getEntityRegistry();
  auto view = registry.view<Transform, Camera>();
  for (auto entity : view) {
    auto& transform = registry.get<Transform>(entity);
    auto& camera = registry.get<Camera>(entity);
    auto& matrices = registry.get<CameraMatrices>(entity);

    auto pitch = math::g_degreeToRadian(transform.rotation.x());
    auto yaw = math::g_degreeToRadian(transform.rotation.y());
    auto roll = math::g_degreeToRadian(transform.rotation.z());
    auto q = math::Quaternionf::fromEulerAngles(pitch, yaw, roll,
                                                math::EulerRotationOrder::XYZ);
    auto forward = math::g_forwardVector<float, 3>();
    auto direction = q.rotateVector(forward);
    matrices.view = math::g_lookToLh(transform.translation, direction, worldUp);

    if (camera.type == CameraType::Perspective) {
      float aspectRatio = camera.width / camera.height;
      matrices.projection = math::g_perspectiveLhZo(camera.fov, aspectRatio,
                                                    camera.nearClip, camera.farClip);
    }
  }
}
```

Рисунок 4.15 – Програмний код системи оновлення камер (виконано самостійно)

Функція g_lookToLh генерує матрицю виду відповідно до формули (3), а g_perspectiveLhZo створює матрицю проекції згідно з формулою (4). Це забезпечує коректну реалізацію повного ланцюжка трансформацій координат у системі рендеринга.

Розроблена система математичних трансформацій забезпечує повну підтримку різних координатних систем та конвенцій проекції, що дозволяє адаптувати рушій для роботи з різними графічними API. Використання кватерніонів для представлення обертань запобігає проблемам «gimbal lock» та забезпечує стабільність обчислень, а оптимізація через SIMD-інструкції значно підвищує продуктивність математичних операцій.

4.4 Система асинхронного завантаження ресурсів

Завантаження великих графічних ресурсів, наприклад, таких як 3D-моделі та текстури високої роздільності, може створювати значні часові затримки у роботі програми, особливо коли ці операції виконуються в головному потоці відображення. Для вирішення цієї проблеми було прийнято рішення про впровадження системи асинхронного завантаження ресурсів, яка дозволяє продовжувати рендеринг сцени навіть під час завантаження додаткових активів у фоновому режимі.

Архітектура системи спирається на окремий робочий потік, який незалежно від головного циклу рендеринга обробляє чергу запитів на завантаження ресурсів. Такий підхід забезпечує швидку реакцію програми та дає змогу користувачеві працювати з тими частинами сцени, що вже завантажені, поки інші елементи завантажуються у фоновому режимі. Для повідомлення про завершення операцій завантаження й автоматичного оновлення компонентів сутностей система використовує функції зворотного виклику.

Основний клас AssetLoader інкапсулює всю логіку асинхронного завантаження та управління чергою запитів. Клас використовує патерн Producer-Consumer для організації взаємодії між головним потоком, який додає запити на завантаження, та робочим потоком, який ці запити обробляє. Для забезпечення потокобезпечності використовуються мютекси та умовні змінні, що дозволяє ефективно координувати роботу між потоками без активного очікування. На рисунку 4.16 представлено структуру та основні методи цього класу.

Метод initialize() створює робочий потік, який виконує функцію workerFunction() у циклі до завершення роботи системи. Робочий потік використовує умовну змінну m_condVar для ефективного очікування нових запитів, що запобігає марному використанню процесорного часу. Метод loadModel() перевіряє, чи не знаходиться ресурс уже у черзі завантаження або чи не був він завантажений раніше, що запобігає дублюванню запитів та оптимізує використання пам'яті.

```cpp
class AssetLoader {
  public:
  using LoadCallback = std::function<void(bool success)>;

  AssetLoader() : m_running(false), m_workerThread(nullptr) {}

  void initialize() {
    if (m_running) return;
    m_running = true;
    m_workerThread = std::make_unique<std::thread>(&AssetLoader::workerFunction, this);
  }

  void loadModel(const std::filesystem::path& filepath, LoadCallback callback = nullptr) {
    if (!m_running) initialize();
    std::string assetKey = createAssetKey(AssetType::Model, filepath.string());
    {
      std::lock_guard<std::mutex> lock(m_queueMutex);
      if (m_pendingAssets.find(assetKey) != m_pendingAssets.end()) {
        if (callback) m_pendingCallbacks[assetKey].push_back(callback);
        return;
      }
      auto modelManager = ServiceLocator::s_get<RenderModelManager>();
      if (modelManager && modelManager->hasRenderModel(filepath)) {
        if (callback) callback(true);
        return;
      }
      AssetRequest request{AssetType::Model, filepath};
      m_pendingAssets[assetKey] = true;
      if (callback) m_pendingCallbacks[assetKey].push_back(callback);
      m_requestQueue.push(request);
      m_condVar.notify_one();
    }
  }

  private:
  void workerFunction() {
    while (m_running) {
      AssetRequest request;
      {
        std::unique_lock<std::mutex> lock(m_queueMutex);
        m_condVar.wait(lock, [this] { return !m_running || !m_requestQueue.empty(); });
        if (!m_running) break;
        if (!m_requestQueue.empty()) {
          request = m_requestQueue.front();
          m_requestQueue.pop();
        }
      }
      processRequest(request);
    }
  }

  void processRequest(const AssetRequest& request) {
    bool success = (request.type == AssetType::Model) ?
                   loadModelInternal_(request.path) : loadTextureInternal_(request.path);
    std::string assetKey = createAssetKey(request.type, request.path.string());
    std::vector<LoadCallback> callbacks;
    {
      std::lock_guard<std::mutex> lock(m_queueMutex);
      callbacks = std::move(m_pendingCallbacks[assetKey]);
      m_pendingAssets.erase(assetKey);
      m_pendingCallbacks.erase(assetKey);
    }
    for (const auto& callback : callbacks) callback(success);
  }

  std::atomic<bool> m_running;
  std::unique_ptr<std::thread> m_workerThread;
  mutable std::mutex m_queueMutex;
  std::condition_variable m_condVar;
  std::queue<AssetRequest> m_requestQueue;
  std::unordered_map<std::string, bool> m_pendingAssets;
  std::unordered_map<std::string, std::vector<LoadCallback>> m_pendingCallbacks;
};
```

Рисунок 4.16 – Програмний код класу AssetLoader для асинхронного завантаження ресурсів (виконано самостійно)

Система використовує спеціальний компонент ModelLoadingTag для позначення сутностей, які очікують завантаження моделі. Цей тег дозволяє відрізнити сутності у процесі завантаження від тих, що мають повністю завантажені ресурси, та забезпечує можливість відображення альтернативного контенту або індикаторів завантаження під час очікування. Після успішного завантаження моделі callback-функція автоматично видаляє тег ModelLoadingTag та додає компонент RenderModel* із посиланням на завантажений ресурс.

Використання хеш-таблиці m_pendingAssets дозволяє швидко перевіряти стан завантаження ресурсів та запобігати створенню дублікатних запитів. Кожен ресурс ідентифікується унікальним ключем, що складається з типу ресурсу та шляху до файлу. Система підтримує множинні callback-функції для одного ресурсу, що дозволяє кільком сутностям очікувати завантаження однієї моделі без створення окремих запитів для кожної з них.

Архітектура забезпечує graceful shutdown через встановлення атомарного прапора m_running у false та сигналізацію умовної змінної, що дозволяє робочому потоку коректно завершити обробку поточного запиту та вийти з циклу. Використання std::unique_ptr для управління потоком забезпечує автоматичне вивільнення ресурсів при знищенні об'єкта AssetLoader. Така система асинхронного завантаження значно покращує користувацький досвід, особливо при роботі зі складними сценами, що містять велику кількість деталізованих 3D-моделей та текстур високої роздільності.

4.5 Система динамічного оновлення шейдерів

Розробка та налагодження графічних ефектів потребує постійної модернізації кода шейдерів, що традиційно вимагає перезапуску програми після кожної зміни. Для підвищення продуктивності розробки було впроваджено систему динамічного оновлення шейдерів (hot reload), яка дозволяє автоматично перекомпілювати та застосовувати зміни в шейдерах під час роботи програми без її перезапуску.

Архітектура системи базується на трьох основних компонентах: менеджері відстеження файлів, менеджері шейдерів з підтримкою динамічного оновлення та системі планування оновлень графічних конвеєрів. Така структура забезпечує швидку реакцію на зміни у файлах шейдерів та безпечне оновлення графічних ресурсів з урахуванням особливостей багатопоточного виконання на GPU.

Основою системи є FileWatcherManager, який використовує бібліотеку wtr::watcher для моніторингу змін у файловій системі. Клас реалізує патерн Observer, відстежуючи модифікації файлів у заданих директоріях та повідомляючи про них через callback-функції. Для кожної директорії створюється окремий watcher, що дозволяє гнучко налаштовувати області моніторингу та уникати дублювання спостерігачів. На рисунку 4.17 показано структуру цього класу.

```
1   class FileWatcherManager {
2     public:
3     using Callback = std::function<void(const wtr::event&)>;
4     void addWatcher(const std::filesystem::path& dirPath,
5                     const Callback&              callback);
6     void removeWatcher(const std::filesystem::path& dirPath);
7     void removeAllWatchers();
8     private:
9     std::unordered_map<std::filesystem::path,
10                       std::unique_ptr<wtr::watcher::watch>>
11        m_watchers_;
12  };
```

Рисунок 4.17 – Програмний код класу FileWatcherManager для відстеження змін файлів (виконано самостійно)

Метод addWatcher() створює новий екземпляр wtr::watcher::watch для заданої директорії та реєструє callback-функцію, яка викликається при будь-яких змінах у файлах. Використання std::unique_ptr забезпечує автоматичне управління життєвим циклом, а unordered_map дозволяє швидко перевіряти існування спостерігачів для конкретних шляхів.

Координацію процесу динамічного оновлення здійснює HotReloadManager, який служить посередником між системою відстеження файлів та менеджером

шейдерів. Цей клас інкапсулює логіку фільтрації подій файлової системи, передаючи далі лише події модифікації файлів та ігноруючи створення, видалення чи перейменування. На рисунку 4.18 представлено його реалізацію.

```
class HotReloadManager {
  public:
  using Callback = std::function<void(const wtr::event&)>;
  void watchFileModifications(const std::filesystem::path& dirPath,
                              const Callback&               onChange);
};

void HotReloadManager::watchFileModifications(
    const std::filesystem::path& dirPath, const Callback& onChange) {
  auto fileWatcher = ServiceLocator::s_get<FileWatcherManager>();
  fileWatcher->addWatcher(dirPath, [onChange](const wtr::event& e) {
    if (e.effect_type == wtr::event::effect_type::modify) {
      onChange(e);
    }
  });
}
```

Рисунок 4.18 – Програмний код класу HotReloadManager для координації hot reload (виконано самостійно)

Ядром системи динамічного оновлення є ShaderManager, який поєднує функції завантаження, кешування та перекомпіляції шейдерів. Клас підтримує автоматичне визначення типу шейдера за розширенням файлу та використовує DxcUtil для компіляції HLSL-коду у відповідний байт-код для обраного графічного API. На рисунку 4.19 показано його структуру.

Архітектура HotReloadManager забезпечує децентралізовану обробку подій файлової системи, дозволяючи різним підсистемам незалежно реагувати на зміни у своїх ресурсах. Система використовує асинхронний підхід до обробки подій, що запобігає блокуванню головного потоку під час виконання операцій перекомпіляції та оновлення ресурсів. Координація між компонентами здійснюється через єдиний інтерфейс, що спрощує управління життєвим циклом спостерігачів та забезпечує

коректне завершення роботи системи. Така організація дозволяє легко додавати підтримку hot reload для нових типів ресурсів без модифікації існуючого коду координатора.

```
class ShaderManager {
  public:
  ShaderManager(Device* device, uint32_t maxFramesDelay, bool enableHotReload = true)
      : m_device_(device)
      , m_enableHotReload_(enableHotReload)
      , m_maxFramesDelay_(maxFramesDelay) {}

  Shader* getShader(const std::filesystem::path& path, const std::string& entryPoint = "main");
  void reloadShader(const std::filesystem::path& path);
  void registerPipelineForShader(Pipeline* pipeline, const std::filesystem::path& shaderPath);

  private:
  void watchDirectoryForChanges(const std::filesystem::path& filePath);
  auto createShaderObject(const std::filesystem::path& path, const std::string& entryPoint) -> std::unique_ptr<Shader>;
  ShaderStageFlag deduceStageFromPath(const std::filesystem::path& path);

  Device*                                                    m_device_;
  bool                                                       m_enableHotReload_;
  std::mutex                                                 m_mutex_;
  std::unordered_map<std::filesystem::path, std::unique_ptr<Shader>>       m_loadedShaders_;
  std::unordered_set<std::filesystem::path>                  m_watchedDirs_;
  std::unordered_map<std::filesystem::path, std::unordered_set<Pipeline*>> m_shaderPipelines_;
  uint32_t                                                   m_maxFramesDelay_;
};
```

Рисунок 4.19 – Програмний код класу ShaderManager з підтримкою динамічного оновлення (виконано самостійно)

Параметр maxFramesDelay має критичне значення для забезпечення синхронізації з GPU - він визначає кількість кадрів затримки перед перебудовою графічних конвеєрів після зміни шейдера. Це забезпечує завершення виконання всіх command buffer'ів, що використовують старий конвеєр, перед його заміною новим.

Правильне управління життєвим циклом графічних ресурсів є особливо критичним у мульти-API архітектурі, оскільки різні графічні API мають відмінні підходи до синхронізації та валідації ресурсів. Vulkan вимагає явного управління синхронізацією через семафори та бар'єри, тоді як DirectX 12 використовує fence-об'єкти для координації між CPU та GPU. Система hot reload повинна враховувати ці особливості, забезпечуючи коректну роботу незалежно від обраного графічного API. Також відстрочене оновлення конвеєрів дозволяє уникнути проблем з валідацією стану рендеринга, які можуть виникати при спробі заміни активно використовуваних ресурсів.

Метод reloadShader() виконує повний цикл оновлення шейдера: перекомпіляцію вихідного коду, оновлення внутрішнього стану шейдера та планування перебудови всіх пов'язаних графічних конвеєрів. Використання мютексу забезпечує потокобезпечність операцій, що особливо важливо для hot reload функціональності, яка може викликатися з різних потоків. На рисунку 4.20 представлено реалізацію цього методу.

```
void ShaderManager::reloadShader(const std::filesystem::path& path) {
  std::lock_guard<std::mutex> lock(m_mutex_);
  auto rel = std::filesystem::relative(path, std::filesystem::current_path());
  rel = std::filesystem::path(rel.generic_string());
  auto it = m_loadedShaders_.find(rel);
  if (it == m_loadedShaders_.end()) {
    GlobalLogger::Log(LogLevel::Warning, "Cannot reload shader, not loaded: " + path.string());
    return;
  }
  Shader* shader = it->second.get();
  auto backend = (m_device_->getApiType() == RenderingApi::Vulkan) ? ShaderBackend::SPIRV : ShaderBackend::DXIL;
  std::wstring wEntryPoint(shader->getEntryPoint().begin(), shader->getEntryPoint().end());
  auto         compiledShader = DxcUtil::s_get().compileHlslFile(path, shader->getStage(), wEntryPoint, backend);
  if (!compiledShader) {
    GlobalLogger::Log(LogLevel::Error, "Shader compilation failed: " + path.string());
    return;
  }
  auto                 data = static_cast<const uint8_t*>(compiledShader->GetBufferPointer());
  size_t               size = compiledShader->GetBufferSize();
  std::vector<uint8_t> newCode(data, data + size);
  shader->reinitialize(newCode);
  auto pipelineIt = m_shaderPipelines_.find(rel);
  if (pipelineIt != m_shaderPipelines_.end()) {
    for (Pipeline* pipeline : pipelineIt->second) {
      pipeline->scheduleUpdate(m_maxFramesDelay_);
    }
  }
  GlobalLogger::Log(LogLevel::Info, "Reloaded shader: " + path.string());
}
```

Рисунок 4.20 – Програмний код методу reloadShader для динамічної перекомпіляції (виконано самостійно)

Функція автоматично визначає цільовий бекенд компіляції (SPIRV для Vulkan або DXIL для DirectX 12) та використовує збережену точку входу шейдера для коректної перекомпіляції. Після успішної компіляції новий байт-код передається у метод reinitialize() шейдера, який оновлює внутрішні ресурси API-специфічних реалізацій.

Система планування оновлень конвеєрів реалізована через метод scheduleUpdate() у базовому класі Pipeline. Цей механізм використовує лічильник

кадрів для відстрочки перебудови конвеєра, що забезпечує синхронізацію з GPU та запобігає використанню застарілих ресурсів. На рисунку 4.21 показано цю систему.

```
class Pipeline {
  public:
  bool needsUpdate() const { return m_updateFrame == 0; }
  void scheduleUpdate(uint32_t delayFrames) { m_updateFrame = delayFrames; }
  void decrementUpdateCounter() {
    if (m_updateFrame > 0) {
      m_updateFrame--;
    }
  }
  virtual bool rebuild() = 0;
  protected:
  std::atomic<int32_t> m_updateFrame{-1};
};
```

Рисунок 4.21 – Програмний код системи планування оновлень Pipeline (виконано самостійно)

Використання std::atomic забезпечує потокобезпечний доступ до лічильника кадрів, що дозволяє безпечно планувати оновлення з одного потоку та перевіряти їх стан з іншого. Метод decrementUpdateCounter() викликається щокадру у головному циклі рендеринга, поступово зменшуючи лічильник до нуля, після чого конвеєр позначається як готовий до перебудови.

Важливим аспектом системи є обробка потенційних помилок під час перебудови конвеєрів, що може статися через некоректний код шейдерів або недоступність графічних ресурсів. Система реалізує механізм відкату до попередньої стабільної версії конвеєра у випадку невдалої перебудови, що гарантує безперервність рендеринга навіть при наявності помилок у оновленому коді. Додатково впроваджено систему логування всіх операцій оновлення, що дозволяє розробникам швидко ідентифікувати проблемні шейдери та відстежувати історію змін. Це особливо важливо в процесі експериментального розробки графічних ефектів, де частота змін може бути досить високою.

Інтеграція системи динамічного оновлення у процес рендеринга відбувається через метод updateScheduledPipelines() у RenderResourceManager, який викликається на початку кожного кадру перед рендерингом сцени. Це забезпечує своєчасне оновлення графічних конвеєрів та мінімізує вплив на продуктивність рендеринга.

Розроблена система динамічного оновлення шейдерів значно прискорює цикл розробки графічних ефектів, дозволяючи розробникам бачити результати змін у реальному часі без перезапуску програми. Архітектура системи забезпечує надійність через синхронізацію з GPU, гнучкість через підтримку різних графічних API та масштабованість через можливість розширення на інші типи ресурсів у майбутньому.

4.6 Система управління сценами

Сучасні 3D-додатки потребують ефективного механізму організації та управління різними сценами, що можуть містити складні ієрархії об'єктів, освітлення, камери та інші елементи візуалізації. Для вирішення цієї задачі було розроблено комплексну систему управління сценами, яка забезпечує створення, завантаження, збереження та динамічне перемикання між різними конфігураціями 3D-середовища без перезапуску програми.

Архітектурне рішення базується на поєднанні архітектури Entity Component System для представлення внутрішньої структури сцени з JSON-форматом для персистентного зберігання даних сцени. Цей підхід забезпечує гнучкість у описі складних сценаріїв візуалізації та зручність редагування конфігурацій сцен як програмно, так і через зовнішні текстові редактори.

Використання JSON-формату як основи для зберігання сцен дозволяє досягти високого рівня переносимості та сумісності між різними версіями системи. Текстовий характер формату забезпечує можливість версіонування сцен через системи контролю версій, що особливо важливо для командної розробки. Крім того, JSON-структура дозволяє легко інтегрувати систему з зовнішніми інструментами створення контенту та автоматизувати процеси імпорту сцен з

різних джерел. Поєднання з ECS архітектурою гарантує ефективну серіалізацію навіть складних сцен з тисячами об'єктів та їх взаємозв'язків.

Центральним елементом системи є клас Scene, який інкапсулює реєстр сутностей та їх компонентів, представляючи повний стан 3D-середовища у конкретний момент часу. Клас використовує бібліотеку EnTT для ефективного управління великою кількістю ігрових об'єктів та забезпечення високої продуктивності під час операцій пошуку та модифікації компонентів сутностей. На рисунку 4.22 показано структуру цього класу.

```
1   class Scene {
2     public:
3     Scene() = default;
4     Scene(Registry registry);
5     Registry& getEntityRegistry();
6     const Registry& getEntityRegistry() const;
7     void setEntityRegistry(Registry registry);
8     private:
9     Registry entityRegistry_;
10  };
```

Рисунок 4.22 – Програмний код класу Scene для представлення 3D-сцени (виконано самостійно)

Клас Scene забезпечує контейнерну функціональність для всіх елементів сцени через єдиний реєстр сутностей, що спрощує операції серіалізації та десеріалізації складних ієрархічних структур об'єктів. Використання типу Registry як псевдоніму для entt::registry забезпечує абстракцію від конкретної реалізації Entity Component System та можливість майбутніх змін без порушення інтерфейсу.

Управління множинними сценами здійснюється через клас SceneManager, який використовується через ServiceLocator для забезпечення глобального доступу до поточної активної сцени та централізованого управління життєвим циклом всіх завантажених сцен. Менеджер підтримує кешування завантажених сцен у пам'яті

для швидкого переключення між ними без повторного читання з диска. На рисунку 4.23 представлено структуру цього класу.

```
class SceneManager {
  public:
  SceneManager() = default;
  void addScene(const std::string& name, Registry registry);
  Scene* getScene(const std::string& name);
  void removeScene(const std::string& name);
  bool switchToScene(const std::string& name);
  Scene* getCurrentScene() const;
  std::string getCurrentSceneName() const;
  bool hasScene(const std::string& name) const;
  void clearAllScenes();
  private:
  std::unordered_map<std::string, std::unique_ptr<Scene>> scenes_;
  Scene* currentScene_ = nullptr;
  std::string currentSceneName_;
};
```

Рисунок 4.23 – Програмний код класу SceneManager для управління множинними сценами (виконано самостійно)

Архітектура SceneManager дозволяє ефективно керувати пам'яттю через використання std::unique_ptr для автоматичного вивільнення ресурсів при видаленні сцен. Метод switchToScene() забезпечує атомарне переключення між сценами з автоматичним оновленням поточного контексту рендеринга, що запобігає некоректним станам під час переходів.

Персистентне зберігання сцен реалізовано через систему серіалізації у JSON-формат, що забезпечує читабельність конфігураційних файлів та можливість їх ручного редагування. Клас SceneLoader відповідає за завантаження та десеріалізацію сцен з файлової системи з підтримкою валідації структури даних та обробки помилок формату. На рисунку 4.24 показано реалізацію цього класу.

Система завантаження сцен інтегрована з ConfigManager для централізованого управління конфігураційними файлами та забезпечення їх кешування у пам'яті. Використання реєстрації конвертерів для різних типів компонентів дозволяє легко розширювати підтримку нових типів даних без

модифікації основного коду завантажувача. Обробка помилок реалізована на кожному етапі процесу завантаження, від читання файлу до парсингу JSON та створення сутностей, що забезпечує стійкість системи до некоректних вхідних даних. Такий підхід гарантує, що навіть у випадку помилок завантаження окремих сцен, загальна функціональність системи управління сценами залишається працездатною.

```
class SceneLoader {
  public:
  static Scene* loadScene(const std::string& sceneName, SceneManager* sceneManager);
  static Scene* loadSceneFromFile(const std::filesystem::path& configPath,
                                  SceneManager* sceneManager,
                                  const std::string& customSceneName = "");
};
Scene* SceneLoader::loadSceneFromFile(const std::filesystem::path& configPath,
                                      SceneManager* sceneManager,
                                      const std::string& customSceneName) {
  auto configManager = ServiceLocator::s_get<ConfigManager>();
  auto config = configManager->getConfig(configPath);
  if (!config) {
    configManager->addConfig(configPath);
    config = configManager->getConfig(configPath);
    if (!config) {
      GlobalLogger::Log(LogLevel::Error, "Failed to load scene config: " + configPath.string());
      return nullptr;
    }
    config->registerConverter<Transform>(&g_loadTransform);
    config->registerConverter<Camera>(&g_loadCamera);
    config->registerConverter<Light>(&g_loadLight);
  }
  rapidjson::Document document;
  document.Parse(config->toString().c_str());
  if (document.HasParseError()) {
    GlobalLogger::Log(LogLevel::Error, "Failed to parse scene JSON: " + configPath.string());
    return nullptr;
  }
  std::string sceneName = customSceneName.empty() && document.HasMember("name")
    ? document["name"].GetString() : customSceneName;
  sceneManager->addScene(sceneName, Registry());
  auto scene = sceneManager->getScene(sceneName);
  auto& registry = scene->getEntityRegistry();
  if (document.HasMember("entities") && document["entities"].IsArray()) {
    for (auto& entityJson : document["entities"].GetArray()) {
      g_createEntityFromConfig(registry, entityJson);
    }
  }
  return scene;
}
```

Рисунок 4.24 – Програмний код класу SceneLoader для завантаження сцен (виконано самостійно)

SceneLoader використовує систему реєстрації конвертерів компонентів для автоматичного перетворення JSON-даних у відповідні C++ об'єкти. Цей підхід

забезпечує розширюваність системи завантаження для нових типів компонентів без модифікації основного коду завантажувача. Функція g_createEntityFromConfig() створює сутності та їх компоненти на основі декларативного опису у JSON-форматі.

Зворотний процес серіалізації реалізований через клас SceneSaver, який перетворює поточний стан сцени у структурований JSON-документ з підтримкою версіонування схеми та збереженням метаданих сцени. Система серіалізації автоматично обробляє всі типи компонентів, зареєстровані у ECS-реєстрі. На рисунку 4.25 представлено основні методи цього класу.

```
class SceneSaver {
  public:
  static bool saveScene(Scene* scene, const std::string& sceneName,
                        const std::filesystem::path& filePath);
  private:
  static void serializeRegistry(const Registry& registry,
                                rapidjson::Document& document,
                                rapidjson::MemoryPoolAllocator<>& allocator);
  static void serializeTransform(const Transform& transform,
                                 rapidjson::Value& componentValue,
                                 rapidjson::MemoryPoolAllocator<>& allocator);
};
void SceneSaver::serializeRegistry(const Registry& registry,
                                   rapidjson::Document& document,
                                   rapidjson::MemoryPoolAllocator<>& allocator) {
  rapidjson::Value entitiesArray(rapidjson::kArrayType);
  auto view = registry.view<entt::entity>();
  for (auto entity : view) {
    rapidjson::Value entityObject(rapidjson::kObjectType);
    std::string entityName = "Entity_" + std::to_string(static_cast<uint32_t>(entity));
    if (registry.all_of<Camera>(entity)) {
      entityName = "MainCamera";
    } else if (registry.all_of<RenderModel*>(entity)) {
      auto* model = registry.get<RenderModel*>(entity);
      if (model && !model->filePath.empty()) {
        entityName = "Model_" + model->filePath.filename().string();
      }
    } else if (registry.all_of<Light, DirectionalLight>(entity)) {
      entityName = "DirectionalLight";
    }
    entityObject.AddMember("name", rapidjson::Value(entityName.c_str(), allocator), allocator);
    rapidjson::Value componentsArray(rapidjson::kArrayType);
    if (registry.all_of<Transform>(entity)) {
      rapidjson::Value componentObject(rapidjson::kObjectType);
      componentObject.AddMember("type", rapidjson::Value("transform", allocator), allocator);
      const auto& transform = registry.get<Transform>(entity);
      serializeTransform(transform, componentObject, allocator);
      componentsArray.PushBack(componentObject, allocator);
    }
    entityObject.AddMember("components", componentsArray, allocator);
    entitiesArray.PushBack(entityObject, allocator);
  }
  document.AddMember("entities", entitiesArray, allocator);
}
```

Рисунок 4.25 – Програмний код класу SceneSaver для збереження сцен (виконано самостійно)

Механізм серіалізації автоматично генерує зрозумілі назви сутностей на основі їхнього складу компонентів, що робить файли сцен більш читабельними. Використання RapidJSON гарантує високу продуктивність серіалізації навіть у сценах із великою кількістю об’єктів. Структурований підхід до збереження компонентів забезпечує просте розширення формату для підтримки нових типів даних.

Інтеграція системи управління сценами з графічним редактором забезпечує інтуїтивний інтерфейс для роботи зі сценами через візуальні елементи управління. Редактор підтримує автоматичне сканування доступних файлів сцен, створення нових сцен з типовими налаштуваннями та плавне переключення між сценами з урахуванням стану завантаження асинхронних ресурсів.

Для забезпечення стабільності роботи система реалізує механізм відкладеного переключення сцен у випадках, коли поточна сцена містить ресурси, що завантажуються асинхронно. Це запобігає некоректним станам під час переходів та забезпечує цілісність даних системи рендеринга. Додатково впроваджено систему автоматичного збереження змін сцени з відображенням прогресу та результатів операції через користувацький інтерфейс.

Розроблена система управління сценами забезпечує гнучку організацію складних 3D-середовищ, ефективне використання пам'яті через кешування та відкладене завантаження ресурсів, а також зручні інструменти для дизайнерів та розробників для створення та модифікації ігрових рівнів та демонстраційних сцен.

4.7 Система інтегрованого редактора

Розробка інструментарію для візуального створення та редагування 3D-сцен потребувала впровадження комплексної системи інтегрованого редактора, яка забезпечує інтуїтивний інтерфейс для роботи з різноманітними елементами віртуального середовища. Система редактора побудована на принципі тісної інтеграції з архітектурою рендеринга та використовує бібліотеку ImGui для створення сучасного користувацького інтерфейсу з підтримкою docking-системи та багатовіконного режиму.

Архітектурне рішення базується на створенні єдиного класу Editor, який координує роботу всіх підсистем редагування та забезпечує взаємодію між різними компонентами через патерн Service Locator. Цей підхід дозволяє централізовано управляти станом редактора, забезпечити консистентність даних між різними панелями інтерфейсу та спростити процес додавання нових функціональних можливостей без порушення існуючої архітектури системи.

Клас Editor інкапсулює всю логіку візуального редагування, включаючи управління viewport'ом для відображення результатів рендеринга, систему gizmo для візуальної трансформації об'єктів, панелі інспектора властивостей сутностей та менеджер сцен з підтримкою створення, завантаження та збереження конфігурацій. Інтеграція з системою Render Hardware Interface забезпечує кросплатформну сумісність редактора з різними графічними API без необхідності модифікації інтерфейсного коду. На рисунку 4.26 представлено структуру цього класу.

```
class Editor {
  public:
  Editor()  = default;
  ~Editor() = default;
  bool initialize(Window*                      window,
                  gfx::rhi::RenderingApi       renderingApi,
                  gfx::rhi::Device*            device,
                  gfx::renderer::FrameResources* frameResources);
  void render(gfx::renderer::RenderContext& context);
  const gfx::renderer::RenderSettings& getRenderParams() const { return m_renderParams; }
  void onWindowResize(uint32_t width, uint32_t height);
  private:
  void renderMainMenu();
  void renderPerformanceWindow();
  void renderViewportWindow(gfx::renderer::RenderContext& context);
  void renderModeSelectionWindow();
  void renderSceneHierarchyWindow();
  void renderInspectorWindow();
  void renderGizmo(const math::Dimension2Di& viewportSize, const ImVec2& viewportPos);
  void handleGizmoInput();
  void handleEntitySelection(entt::entity entity);
  void addDirectionalLight();
  void addPointLight();
  void addSpotLight();
  void removeSelectedEntity();
  void createModelEntity(const std::filesystem::path& modelPath, const Transform& transform);
  void saveCurrentScene_();
  void switchToScene_(const std::string& sceneName);

  gfx::renderer::RenderSettings m_renderParams;
  bool        m_showSaveNotification = false;
  ElapsedTime m_notificationTimer;
  FrameTime   m_sceneSaveTimer;
};
```

Рисунок 4.26 – Програмний код класу Editor інтегрованого редактора (виконано самостійно)

Метод initialize() налаштовує інтеграцію з графічною підсистемою через створення контексту ImGuiRHIContext, який абстрагує специфічні особливості роботи з різними графічними API. Параметр renderingApi дозволяє адаптувати поведінку редактора під конкретний бекенд рендеринга, забезпечуючи оптимальну продуктивність та стабільність роботи на різних платформах. Ініціалізація системи gizmo через ImGuizmo::SetImGuiContext() забезпечує правильну взаємодію з контекстом ImGui для візуального редагування трансформацій об'єктів.

Центральним елементом роботи редактора є метод render(), який координує відображення всіх панелей інтерфейсу та забезпечує синхронізацію між різними компонентами системи. Архітектура методу побудована на принципі композиції окремих функцій рендеринга, що дозволяє легко додавати нові панелі або модифікувати існуючі без впливу на загальну структуру системи.

Система gizmo реалізована через інтеграцію з бібліотекою ImGuizmo та забезпечує візуальне редагування трансформацій об'єктів безпосередньо у viewport'i. Архітектура підтримує різні режими трансформації - переміщення, обертання та масштабування, з можливістю переключення між світовими та локальними координатами. Особливу увагу приділено обробці спеціальних типів сутностей, таких як джерела освітлення, для яких доступні лише певні типи трансформацій відповідно до їх фізичної природи.

Інспектор властивостей забезпечує детальне редагування параметрів вибраних сутностей через динамічно генерований інтерфейс, який автоматично адаптується до складу компонентів сутності. Система використовує патерн Strategy для обробки різних типів компонентів, що дозволяє легко розширювати функціональність інспектора для нових типів даних без модифікації основного коду.

Менеджмент сцен інтегрований з системою асинхронного завантаження ресурсів та забезпечує безпечне переключення між сценами з урахуванням стану завантаження моделей. Механізм відкладеного переключення через m_pendingSceneSwitch запобігає некоректним станам системи під час завантаження ресурсів та гарантує цілісність даних при переходах між сценами.

Система нотифікацій використовує таймери для відображення результатів операцій користувачеві з поступовим згасанням повідомлень. Це забезпечує неінвазивне інформування про виконання дій без перевантаження інтерфейсу та дозволяє користувачеві продовжувати роботу, отримуючи при цьому необхідну зворотну інформацію про стан системи.

Інтеграція з системою вводу реалізована через підписку на події клавіатури та миші, що дозволяє забезпечити швидкий доступ до часто використовуваних функцій через клавіатурні комбінації. Система контекстів вводу автоматично переключається між ігровим режимом та режимом редактора залежно від фокусу інтерфейсу, що запобігає конфліктам між різними обробниками подій.

Архітектура області перегляду (viewport) забезпечує відображення результатів рендеринга безпосередньо в панелі редактора через інтеграцію з -шаром та автоматичне управління розмірами текстур відповідно до розмірів панелі. Система підтримує динамічне зміна розмірів області перегляду з автоматичним оновленням відповідних графічних ресурсів, що забезпечує ефективне використання пам'яті та оптимальну якість відображення.

Розроблений інтегрований редактор значно прискорює процес створення та налагодження 3D-сцен, забезпечуючи інтуїтивний інтерфейс для роботи з складними віртуальними середовищами. Модульна архітектура системи дозволяє легко розширювати функціональність редактора додавання нових інструментів та панелей, а інтеграція з системою рендеринга забезпечує реалізм попереднього перегляду результатів у реальному часі.

4.8 Система конфігурації

Розробка гнучкої системи конфігурації є критично важливим аспектом програмної системи мульти-API для 3D-рендерінга, оскільки дозволяє налаштовувати різні параметри системи без необхідності перекомпіляції проекту. Система конфігурації забезпечує можливість динамічної зміни параметрів рендеринга, режиму роботи застосунку, налаштувань камери та інших важливих параметрів системи під час виконання програми.

Архітектура системи конфігурації базується на використанні JSON файлів для зберігання налаштувань, що забезпечує читабельність та простоту редагування параметрів. Основними компонентами системи є клас Config для роботи з окремими конфігураційними файлами, ConfigManager для централізованого управління множиною конфігурацій та RuntimeSettings для глобальних налаштувань системи.

Центральним елементом системи є конфігураційний файл налаштувань рушія, який дозволяє змінювати ключові параметри роботи системи. Цей файл містить основні налаштування для вибору API рендеринга, режиму роботи застосунку та світової системи координат. На рисунку 4.27 показано приклад такого конфігураційного файлу.

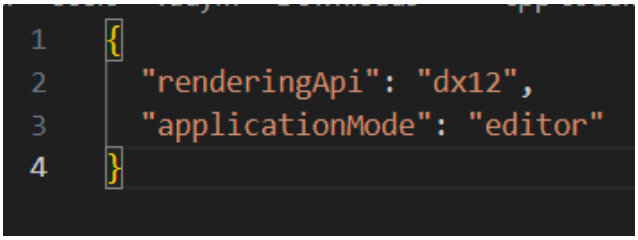

Рисунок 4.27 – Конфігураційний файл налаштувань рушія (виконано самостійно)

Наведений конфігураційний файл демонструє основні можливості системи щодо налаштування критично важливих аспектів роботи програми. Параметр «renderingApi» дозволяє вибирати між різними API рендеринга, такими як DirectX 12 або Vulkan, без необхідності перекомпіляції проекту. Цей параметр є центральним для мульти-API архітектури системи, оскільки визначає, який саме рендеринг бекенд буде використовуватися під час виконання програми.

Налаштування «applicationMode» забезпечує перемикання між режимами роботи застосунку, що дозволяє використовувати одну і ту ж збірку проекту як для розробки у редакторі, так і для фінального продукту. Режим «editor» активує додатковий функціонал для розробки, включаючи інструменти налагодження та візуальний редактор сцени, тоді як режим «game» оптимізує роботу для кінцевого користувача.

Реалізація системи конфігурації використовує типобезпечний підхід із застосуванням шаблонів C++ та концептів для забезпечення коректності типів під час компіляції. Асинхронне завантаження конфігурацій реалізовано для підвищення продуктивності системи, що дозволяє не блокувати основний потік виконання під час читання файлів з диска.

Переваги розробленої системи конфігурації включають швидкість експериментування з різними налаштуваннями, можливість створення різних профілів конфігурацій для різних сценаріїв використання, простоту налагодження через зрозумілий формат JSON та гнучкість розширення через додавання нових параметрів конфігурації без зміни коду програми.

4.9 Архітектура Entity Component System

Для організації ігрових об'єктів та їх взаємодії в програмній системі було прийнято рішення про використання архітектурного патерну Entity Component System, який забезпечує високу гнучкість у створенні складних ігрових сутностей та ефективне управління великою кількістю об'єктів у 3D-сценах. Цей підхід вирішує фундаментальні проблеми традиційної об'єктно-орієнтованої ієрархії класів, такі як жорстка зв'язаність компонентів, складність множинного наслідування та обмеження у композиції поведінки об'єктів.

Архітектура Entity Component System базується на трьох основних концепціях: сутності (Entity) як унікальні ідентифікатори об'єктів, компоненти (Component) як контейнери даних без логіки, та системи (System) як обробники логіки, що працюють з групами компонентів. Такий поділ забезпечує принцип Separation of Concerns на рівні архітектури додатку, дозволяючи незалежно розвивати дані та логіку їх обробки.

Реалізація Entity Component System у системі спирається на бібліотеку EnTT, яка забезпечує високопродуктивне управління сутностями та компонентами через оптимізовані структури даних та cache-friendly алгоритми обходу. Центральними елементами архітектури є тип Entity як псевдонім для entt::entity та Registry як псевдонім для entt::registry, що абстрагують конкретну реалізацію Entity Component

System та забезпечують можливість майбутніх змін без порушення інтерфейсу. На рисунку 4.28 показано базові типи системи.

```
1   using Entity = entt::entity;
2   using Registry = entt::registry;
```

Рисунок 4.28 – Програмний код базових типів Entity Component System (виконано самостійно)

Компоненти в системі представлені як прості структури даних без логіки, що інкапсулюють специфічні аспекти ігрових об'єктів. Компонент Transform відповідає за просторове позиціонування об'єктів у тривимірному просторі, Camera визначає параметри візуалізації сцени, Light інкапсулює властивості джерел освітлення, а RenderModel* містить посилання на графічні ресурси для відображення. Такий підхід забезпечує максимальну гнучкість у створенні різноманітних комбінацій властивостей об'єктів. На рисунку 4.29 представлено приклади основних компонентів системи.

Важливою особливістю реалізації компонентів є використання флагів стану, таких як isDirty, які дозволяють системам ефективно відстежувати зміни та виконувати оновлення лише за необхідності. Це значно оптимізує продуктивність системи, особливо у сценах з великою кількістю статичних об'єктів, які не потребують постійного перерахунку матриць трансформацій або параметрів освітлення. Компоненти також включають флаг enabled для динамічного включення та виключення функціональності без видалення компонента з сутності. Така архітектура забезпечує cache-friendly організацію даних, оскільки компоненти одного типу зберігаються в суміжних областях пам'яті, що покращує продуктивність систем при обробці великих наборів сутностей.

```cpp
struct Transform {
  math::Vector3Df translation;
  math::Vector3Df rotation;
  math::Vector3Df scale = math::Vector3Df(1.0f, 1.0f, 1.0f);
  bool isDirty = false;
};

struct Camera {
  CameraType type;
  float      fov;
  float      nearClip;
  float      farClip;
  float      width;
  float      height;
};

struct Light {
  math::Vector3Df color;
  float           intensity;
  bool            isDirty = true;
  bool            enabled = true;
};
```

Рисунок 4.29 – Програмний код прикладів компонентів ECS (виконано самостійно)

Системи в архітектурі Entity Component System реалізують всю логіку обробки компонентів та взаємодії між ними. Базовий інтерфейс IUpdatableSystem визначає контракт для систем, що потребують регулярного оновлення кожного кадру, забезпечуючи уніфікований підхід до управління життєвим циклом всіх систем у додатку. Метод update приймає посилання на сцену та час, що минув з попереднього кадру, що дозволяє системам здійснювати часозалежні обчислення. На рисунку 4.30 показано структуру базового інтерфейсу систем.

```cpp
class IUpdatableSystem {
  public:
  virtual ~IUpdatableSystem() = default;
  virtual void update(Scene* scene, float deltaTime) = 0;
};
```

Рисунок 4.30 – Програмний код базового інтерфейсу систем ECS (виконано самостійно)

Конкретні реалізації систем демонструють зручність даного архітектурного підходу через використання view-механізму EnTT для ефективного отримання сутностей з потрібними компонентами. MovementSystem обробляє переміщення об'єктів на основі компонентів Movement та Transform, LightSystem збирає інформацію про джерела освітлення та підготовлює дані для шейдерів, а CameraSystem оновлює матриці виду та проекції для камер. Кожна система працює незалежно від інших, що забезпечує модульність та спрощує тестування окремих аспектів функціональності. На рисунку 4.31 представлено приклад реалізації системи обробки руху об'єктів.

```
struct Movement {
  math::Vector3Df direction;
  float           strength;
};

void MovementSystem::update(Scene* scene, float deltaTime) {
  Registry& registry = scene->getEntityRegistry();
  auto view = registry.view<Transform, Movement>();
  for (auto entity : view) {
    auto& transform = view.get<Transform>(entity);
    auto& movement = view.get<Movement>(entity);
    transform.translation += movement.direction * movement.strength * deltaTime;
    movement = Movement{};
  }
}
```

Рисунок 4.31 – Програмний код системи обробки руху об'єктів (виконано самостійно)

Координацію роботи всіх систем здійснює клас SystemManager, який використовує патерн Composite для управління колекцією систем як єдиним об'єктом. Менеджер забезпечує централізоване додавання систем через метод addSystem, отримання посилань на конкретні системи через шаблонний метод getSystem та виконання циклу оновлення всіх систем через метод updateSystems. Використання std::unique_ptr для зберігання систем гарантує правильне управління пам'яттю та автоматичне вивільнення ресурсів при знищенні менеджера. На рисунку 4.32 показано реалізацію менеджера систем.

```cpp
class SystemManager {
  public:
  void addSystem(std::unique_ptr<IUpdatableSystem> system);

  template <typename T>
  T* getSystem() const {
    for (const auto& system : m_systems_) {
      T* typedSystem = dynamic_cast<T*>(system.get());
      if (typedSystem) {
        return typedSystem;
      }
    }
    return nullptr;
  }

  void updateSystems(Scene* scene, float deltaTime) {
    for (const auto& system : m_systems_) {
      system->update(scene, deltaTime);
    }
  }

  private:
  std::vector<std::unique_ptr<IUpdatableSystem>> m_systems_;
};
```

Рисунок 4.32 – Програмний код менеджера систем ECS (виконано самостійно)

Реалізована архітектура Entity Component System забезпечує високу продуктивність завдяки cache-friendly алгоритмам обходу компонентів, гнучкість у створенні складних ігрових об'єктів через композицію компонентів, масштабованість системи для обробки великої кількості сутностей та спрощення процесу розробки через чітке розділення даних та логіки. Архітектура дозволяє легко додавати нові типи компонентів та систем без модифікації існуючого коду, що забезпечує довгострокову підтримуваність та розширюваність системи 3D-рендеринга.

4.10 Система завантаження ресурсів через інтерфейси

Розробка сучасних графічних додатків потребує підтримки широкого спектру форматів файлів для 3D-моделей, текстур та матеріалів. Кожен формат має свої особливості та вимагає використання спеціалізованих бібліотек для завантаження та обробки даних. Для вирішення цієї проблеми було прийнято рішення про створення гнучкої системи завантаження ресурсів, яка базується на

абстрактних інтерфейсах та дозволяє легко додавати підтримку нових форматів без модифікації існуючого коду.

Архітектурне рішення спирається на поєднання патернів Abstract Factory та Strategy, що забезпечує можливість динамічного вибору відповідного завантажувача на основі типу файлу та його розширення. Система побудована навколо трьох основних абстрактних інтерфейсів, кожен з яких відповідає за конкретний тип ресурсів. Інтерфейс IModelLoader забезпечує завантаження геометричних даних моделей у CPU-пам'ять, IRenderModelLoader створює GPU-ресурси для рендеринга, а IMaterialLoader обробляє інформацію про матеріали та їх властивості.

Базовий інтерфейс для завантаження моделей визначає єдиний контракт для всіх можливих реалізацій, незалежно від використовуваної бібліотеки або формату файлу. Цей підхід гарантує консистентність API та спрощує процес інтеграції нових завантажувачів у систему. На рисунку 4.33 представлено структуру основного інтерфейсу для завантаження моделей.

```
1  class IModelLoader {
2      public:
3      virtual ~IModelLoader()                                                    = default;
4      virtual std::unique_ptr<Model> loadModel(const std::filesystem::path& filepath) = 0;
5  };
```

Рисунок 4.33 – Програмний код базового інтерфейсу IModelLoader для завантаження моделей (виконано самостійно)

Простота інтерфейсу забезпечує легкість його імплементації для різних бібліотек, при цьому приховуючи всі деталі реалізації від клієнтського коду. Метод loadModel приймає шлях до файлу та повертає унікальний вказівник на завантажену модель, що гарантує правильне управління пам'яттю та дозволяє передавати власність об'єкта між різними компонентами системи.

Для підтримки GPU-орієнтованого рендеринга система включає окремий інтерфейс IRenderModelLoader, який відповідає за створення графічних ресурсів та їх оптимізацію для конкретного API рендеринга. Цей підхід дозволяє розділити

логіку обробки геометричних даних та створення GPU-ресурсів, що підвищує гнучкість архітектури. На рисунку 4.34 показано структуру цього інтерфейсу.

```
class IRenderModelLoader {
  public:
  virtual ~IRenderModelLoader() = default;
  virtual std::unique_ptr<RenderModel> loadRenderModel(const std::filesystem::path& filepath,
                                                       std::optional<Model*>        outModel = std::nullopt) = 0;
};
```

Рисунок 4.34 – Програмний код інтерфейсу IRenderModelLoader для створення GPU-ресурсів (виконано самостійно)

Параметр outModel дозволяє отримати CPU-версію моделі разом з GPU-ресурсами, що може бути корисно для операцій, які потребують доступу до оригінальних геометричних даних. Використання std::optional забезпечує гнучкість у випадках, коли CPU-дані не потрібні.

Архітектура на основі інтерфейсів забезпечує винятковою гнучкістю системи завантаження ресурсів, дозволяючи одночасно підтримувати множинні бібліотеки для роботи з однаковими форматами файлів. Це рішення особливо важливе для 3D-рендеринга, оскільки різні бібліотеки можуть мати специфічні переваги для певних типів моделей або сценаріїв використання. Розробники можуть легко експериментувати з альтернативними реалізаціями завантажувачів без необхідності модифікації клієнтського коду, що значно прискорює процес оптимізації та налагодження. Така модульність також дозволяє створювати спеціалізовані збірки проекту з підтримкою лише необхідних форматів файлів, зменшуючи розмір фінального виконуваного файлу.

Конкретні реалізації інтерфейсів демонструють можливості системи щодо підтримки різних форматів файлів. Клас AssimpModelLoader використовує бібліотеку Assimp для завантаження широкого спектру 3D-форматів, включаючи OBJ, FBX та інші популярні формати. Реалізація інкапсулює всі деталі взаємодії з Assimp та перетворює дані у внутрішній формат системи. На рисунку 4.35 представлено фрагмент реалізації цього класу.

```cpp
class AssimpModelLoader : public IModelLoader {
  public:
  std::unique_ptr<Model> loadModel(const std::filesystem::path& filePath) override {
    auto scenePtr = AssimpSceneCache::getOrLoad(filePath);
    if (!scenePtr) {
      return nullptr;
    }
    const aiScene* scene = scenePtr.get();
    auto model      = std::make_unique<Model>();
    model->filePath = filePath;
    auto meshManager = ServiceLocator::s_get<MeshManager>();
    if (!meshManager) {
      GlobalLogger::Log(LogLevel::Error, "MeshManager not available in ServiceLocator.");
      return nullptr;
    }
    model->meshes.reserve(scene->mNumMeshes);
    for (unsigned int i = 0; i < scene->mNumMeshes; ++i) {
      aiMesh* ai_mesh = scene->mMeshes[i];
      auto mesh = processMesh(ai_mesh);
      Mesh* meshPtr = meshManager->addMesh(std::move(mesh), filePath);
      model->meshes.push_back(meshPtr);
    }
    return model;
  }
  private:
  std::unique_ptr<Mesh> processMesh(aiMesh* ai_mesh);
};
```

Рисунок 4.35 – Програмний код класу AssimpModelLoader для завантаження моделей через Assimp (виконано самостійно)

Реалізація використовує кешування сцен через AssimpSceneCache для оптимізації продуктивності при повторному завантаженні файлів. Інтеграція з MeshManager забезпечує централізоване управління геометричними даними та запобігає дублюванню ресурсів у пам'яті.

Паралельно з Assimp система підтримує сучасний формат glTF через окрему реалізацію CgltfModelLoader. Цей підхід демонструє гнучкість архітектури та можливість одночасної підтримки множинних бібліотек для різних форматів. На рисунку 4.36 показано ключові аспекти реалізації для glTF.

```cpp
class CgltfModelLoader : public IModelLoader {
  public:
  std::unique_ptr<Model> loadModel(const std::filesystem::path& filePath) override {
    auto scene = CgltfSceneCache::getOrLoad(filePath);
    if (!scene) {
      GlobalLogger::Log(LogLevel::Error, "Failed to load GLTF scene: " + filePath.string());
      return nullptr;
    }
    const cgltf_data* data = scene.get();
    auto model      = std::make_unique<Model>();
    model->filePath = filePath;
    auto meshManager = ServiceLocator::s_get<MeshManager>();
    for (size_t i = 0; i < data->meshes_count; ++i) {
      cgltf_mesh* gltf_mesh = &data->meshes[i];
      for (size_t j = 0; j < gltf_mesh->primitives_count; ++j) {
        auto mesh = processPrimitive(&gltf_mesh->primitives[j]);
        if (mesh) {
          Mesh* meshPtr = meshManager->addMesh(std::move(mesh), filePath);
          model->meshes.push_back(meshPtr);
        }
      }
    }
    return model;
  }
  private:
  std::unique_ptr<Mesh> processPrimitive(const cgltf_primitive* primitive);
};
```

Рисунок 4.36 – Програмний код класу CgltfModelLoader для завантаження glTF моделей (виконано самостійно)

Реалізація для glTF враховує специфіку цього формату, зокрема обробку примітивів та підтримку ієрархічної структури сцени. Використання того ж інтерфейсу IModelLoader дозволяє системі прозоро працювати з обома форматами без необхідності змін у клієнтському коді.

Координацію між різними завантажувачами здійснює клас ModelLoaderManager, який реалізує патерн Registry та забезпечує автоматичний вибір відповідного завантажувача на основі розширення файлу. Менеджер підтримує реєстрацію нових завантажувачів під час виконання програми та гарантує потокобезпечність операцій. На рисунку 4.37 представлено структуру цього класу.

```cpp
class ModelLoaderManager {
  public:
  void registerLoader(ModelType modelType, std::unique_ptr<IModelLoader> loader) {
    std::lock_guard<std::mutex> lock(mutex_);
    loaderMap_[modelType] = std::move(loader);
  }

  std::unique_ptr<Model> loadModel(const std::filesystem::path& filePath) {
    std::string extension = filePath.extension().string();
    std::transform(extension.begin(), extension.end(), extension.begin(), ::tolower);
    ModelType modelType = getModelTypeFromExtension(extension);
    if (modelType == ModelType::UNKNOWN) {
      GlobalLogger::Log(LogLevel::Error, "Unknown model type for extension: " + extension);
      return nullptr;
    }
    IModelLoader* loader = nullptr;
    {
      std::lock_guard<std::mutex> lock(mutex_);
      auto it = loaderMap_.find(modelType);
      if (it != loaderMap_.end()) {
        loader = it->second.get();
      }
    }
    if (loader) {
      return loader->loadModel(filePath);
    }
    GlobalLogger::Log(LogLevel::Error, "No loader found for model type with extension: " + extension);
    return nullptr;
  }
  private:
  std::unordered_map<ModelType, std::unique_ptr<IModelLoader>> loaderMap_;
  std::mutex mutex_;
};
```

Рисунок 4.37 – Програмний код класу ModelLoaderManager для управління завантажувачами (виконано самостійно)

Функція getModelTypeFromExtension забезпечує відображення розширень файлів на внутрішні типи моделей, що дозволяє системі автоматично визначати потрібний завантажувач. Використання std::transform для приведення розширення до нижнього регістру гарантує коректну обробку файлів незалежно від регістру їх імен.

Аналогічна архітектура застосована для завантаження матеріалів через інтерфейс IMaterialLoader та відповідний MaterialLoaderManager. Система матеріалів інтегрована з менеджерами текстур та зображень, що забезпечує автоматичне завантаження всіх пов'язаних ресурсів. На рисунку 4.38 показано приклад реалізації завантажувача матеріалів для Assimp.

```cpp
class AssimpMaterialLoader : public IMaterialLoader {
  public:
  std::vector<std::unique_ptr<Material>> loadMaterials(const std::filesystem::path& filePath) override {
    auto scenePtr = AssimpSceneCache::getOrLoad(filePath);
    if (!scenePtr) {
      return {};
    }
    const aiScene* scene = scenePtr.get();
    std::vector<std::unique_ptr<Material>> materials;
    for (unsigned int i = 0; i < scene->mNumMaterials; ++i) {
      aiMaterial* ai_material = scene->mMaterials[i];
      auto material = std::make_unique<Material>();
      material->filePath = filePath;
      processMaterialProperties(ai_material, material.get());
      loadTextures(ai_material, material.get(), filePath);
      materials.push_back(std::move(material));
    }
    return materials;
  }
  private:
  void processMaterialProperties(aiMaterial* mat, Material* material);
  void loadTextures(aiMaterial* mat, Material* material, const std::filesystem::path& basePath);
};
```

Рисунок 4.38 – Програмний код класу AssimpMaterialLoader для завантаження матеріалів (виконано самостійно)

Завантаження текстур інтегровано з системою управління зображеннями, що дозволяє автоматично обробляти різні формати текстур через відповідні завантажувачі. Метод loadTextures використовує ImageManager та TextureManager для створення GPU-ресурсів текстур та їх кешування.

Для підвищення продуктивності система включає кешуючі менеджери, які зберігають завантажені ресурси у пам'яті та запобігають повторному завантаженню ідентичних файлів. RenderModelManager поєднує функції кешування з автоматичним завантаженням через відповідні loader manager.

Архітектура кешування побудована з урахуванням багатопоточного доступу до ресурсів, що є критично важливим для сучасних графічних додатків. Система використовує стратегію lazy loading, коли ресурси завантажуються лише при першому зверненні, що значно зменшує час запуску програми та споживання пам'яті. Додатково реалізовано механізм подвійної перевірки (double-checked locking), який мінімізує overhead синхронізації при частих операціях читання з кешу. Така оптимізація особливо ефективна у сценаріях, коли одні й ті ж ресурси використовуються множинними об'єктами сцени.

На рисунку 4.39 представлено ключовий метод цього класу.

```
class RenderModelManager {
  public:
  RenderModel* getRenderModel(const std::filesystem::path& filepath, std::optional<Model*> outModel) {
    {
      std::shared_lock<std::shared_mutex> readLock(mutex_);
      auto it = renderModelCache_.find(filepath);
      if (it != renderModelCache_.end()) {
        return it->second.get();
      }
    }
    auto renderModelLoaderManager = ServiceLocator::s_get<RenderModelLoaderManager>();
    if (!renderModelLoaderManager) {
      GlobalLogger::Log(LogLevel::Error, "RenderModelLoaderManager not available in ServiceLocator.");
      return nullptr;
    }
    auto renderModel = renderModelLoaderManager->loadRenderModel(filepath, outModel);
    if (renderModel) {
      RenderModel* modelPtr = renderModel.get();
      {
        std::unique_lock<std::shared_mutex> writeLock(mutex_);
        auto it = renderModelCache_.find(filepath);
        if (it != renderModelCache_.end()) {
          return it->second.get();
        }
        renderModelCache_[filepath] = std::move(renderModel);
      }
      return modelPtr;
    }
    return nullptr;
  }
  private:
  std::unordered_map<std::filesystem::path, std::unique_ptr<RenderModel>> renderModelCache_;
  mutable std::shared_mutex mutex_;
};
```

Рисунок 4.39 – Програмний код методу getRenderModel з кешуванням ресурсів (виконано самостійно)

Використання shared_mutex дозволяє множинним потокам одночасно читати з кешу, блокуючи лише операції запису. Подвійна перевірка існування ресурсу після отримання write lock запобігає гонитві потоків та дублюванню завантаження.

Реєстрація завантажувачів відбувається під час ініціалізації системи через ServiceLocator, що забезпечує централізоване управління залежностями. Цей підхід дозволяє легко налаштовувати доступні формати залежно від конфігурації збірки проекту. Додавання підтримки нового формату вимагає лише створення відповідної реалізації інтерфейсу та її реєстрації у відповідному менеджері.

Розроблена система завантаження ресурсів через інтерфейси забезпечує високу гнучкість архітектури через можливість легкого додавання нових форматів та бібліотек, ефективне використання пам'яті завдяки кешуванню та уникненню дублювання ресурсів, потокобезпечність операцій завантаження та кешування, а також чітке розділення відповідальностей між завантаженням CPU та GPU ресурсів. Архітектура дозволяє розробникам сконцентруватися на специфіці конкретних форматів файлів, не турбуючись про інтеграцію з рештою системи рендеринга.

4.11 Система конфігурації збірки проекту

Розробка програмної системи мульти-API для 3D-рендеринга потребує створення гнучкої та масштабованої системи конфігурації збірки, яка б дозволяла адаптувати проект під різні платформи, графічні API та набори функціональних можливостей. Традиційні підходи до конфігурації C++ проектів часто призводять до складних і важкопідтримуваних систем збірки, особливо коли проект залежить від великої кількості сторонніх бібліотек та підтримує множинні конфігурації.

Прийняте архітектурне рішення базується на використанні CMake як основної системи збірки з розробкою власної системи опціональних компонентів, яка дозволяє точково включати або виключати функціональність під час конфігурації проекту. Цей підхід забезпечує модульність системи збірки, зменшує час компіляції та розмір фінального виконуваного файлу, а також спрощує підтримку різних конфігурацій для розробки та продуктивного використання.

Центральним елементом системи конфігурації є набір CMake опцій, які дозволяють контролювати включення сторонніх бібліотек та специфічних функціональних можливостей. Система організована у три основні категорії: опції для вибору графічних API, опції для включення сторонніх бібліотек та опції для додаткових інструментів розробки. На рисунку 4.40 показано основні опції конфігурації системи збірки.

```
# Choose which rendering API to use
option(USE_VULKAN  "Use Vulkan as the rendering API"                ON)
option(USE_OPENGL  "Use OpenGL as the rendering API (DEPRECATED!)"  OFF)
option(USE_DIRECTX "Use DirectX as the rendering API"          ${WIN32})
option(FORCE_RHI_API "Force specific RHI API at compile time (disables runtime selection)" OFF)

# Choose which third-party libraries to include in the build
option(BUILD_SDL          "Build the SDL library"       ON)
option(BUILD_GLFW         "Build the GLFW library"      OFF)
option(BUILD_SPDLOG       "Build the SPDLOG library"    ON)
option(BUILD_IMGUI        "Build the ImGui library"     ON)
option(BUILD_ASSIMP       "Build the Assimp library"    OFF)
option(BUILD_STB          "Build the STB library"       ON)
option(BUILD_MATH_LIBRARY "Build the Math Library"      ON)
option(BUILD_ENTT         "Build the EnTT library"      ON)
```

Рисунок 4.40 – Програмний код системи опцій конфігурації проекту (виконано самостійно)

Система використовує значення за замовчуванням, враховуючи особливості цільової платформи. Наприклад, опція USE_DIRECTX автоматично встановлюється в TRUE для Windows-платформ та FALSE для інших операційних систем. Це забезпечує коректну конфігурацію проекту без необхідності ручного налаштування в типових сценаріях використання.

Для управління складними залежностями між опціями використовується механізм CMakeDependentOption, який дозволяє автоматично включати або виключати певні компоненти на основі стану інших опцій. Цей підхід запобігає некоректним конфігураціям та спрощує процес налаштування системи збірки. На рисунку 4.41 представлено систему умовних залежностей між компонентами.

```
include(CMakeDependentOption)
cmake_dependent_option(BUILD_VULKAN_MEMORY_ALLOCATOR "Build Vulkan Memory Allocator" ON "USE_VULKAN"  OFF)
cmake_dependent_option(BUILD_D3D12_MEMORY_ALLOCATOR  "Build D3D12 Memory Allocator"  ON "USE_DIRECTX" OFF)
cmake_dependent_option(USE_DIRECTX_AGILITY           "Fetch DirectX 12 Agility SDK (and DirectX-Headers as dependent library)"  ON "USE_DIRECTX" OFF)
cmake_dependent_option(BUILD_IMGUIZMO                "Build the ImGuizmo library" ON "BUILD_IMGUI" OFF)
cmake_dependent_option(BUILD_TRACY                         "Build Tracy profiler" ON  "USE_PROFILING"               OFF)
cmake_dependent_option(USE_CPU_PROFILING  "Enable CPU profiling (requires Tracy)" ON  "USE_PROFILING;BUILD_TRACY"     OFF)
```

Рисунок 4.41 – Програмний код системи умовних залежностей між компонентами (виконано самостійно)

Система автоматично включає Vulkan Memory Allocator лише при активації підтримки Vulkan, D3D12 Memory Allocator для DirectX, а ImGuizmo компілюється

тільки за наявності ImGui. Такий підхід забезпечує логічну консистентність конфігурації та запобігає помилкам збірки через відсутність необхідних залежностей.

Одним з найбільш складних аспектів системи конфігурації є управління сторонніми бібліотеками. Система використовує FetchContent API для автоматичного завантаження та інтеграції зовнішніх залежностей під час конфігурації проекту. Цей підхід забезпечує воспроизводимість збірки та усуває необхідність ручного управління зовнішніми залежностями. На рисунку 4.42 показано реалізацію системи автоматичного завантаження залежностей.

```
include(FetchContent)
set(FETCHCONTENT_BASE_DIR ${CMAKE_SOURCE_DIR}/third_party)

if(BUILD_SPDLOG)
    message(STATUS "Fetching SPDLOG...")
    FetchContent_Declare(
        spdlog
        GIT_REPOSITORY https://github.com/gabime/spdlog.git
        GIT_TAG v1.15.3
        GIT_SHALLOW    TRUE
        GIT_PROGRESS   TRUE
    )
endif()

if(BUILD_IMGUI)
    message(STATUS "Fetching IMGUI...")
    FetchContent_Declare(
        imgui
        GIT_REPOSITORY https://github.com/ocornut/imgui.git
        GIT_TAG v1.91.6-docking
    )
endif()
```

Рисунок 4.42 – Програмний код системи автоматичного завантаження залежностей (виконано самостійно)

Використання конкретних версій (Git tags) замість HEAD гілки забезпечує стабільність та воспроизводимість збірки. Параметри GIT_SHALLOW та GIT_PROGRESS оптимізують процес завантаження, зменшуючи об'єм завантажуваних даних та надаючи інформацію користувачеві про поточний стан операції.

Для Windows-специфічних бібліотек, таких як DirectX компоненти, система використовує VCPKG як менеджер пакетів. Це рішення обумовлено складністю ручної інтеграції Microsoft-специфічних SDK та бібліотек. Система автоматично визначає необхідність використання VCPKG та налаштовує відповідну інфраструктуру. На рисунку 4.43 представлено код інтеграції VCPKG для Windows-специфічних залежностей.

```
if(USE_DIRECTX OR USE_DIRECTX_SHADER_COMPILER OR USE_DIRECTX_TEX)
    set(USE_VCPKG TRUE)
else()
    set(USE_VCPKG FALSE)
endif()

if(USE_VCPKG)
    message(STATUS "Fetching vcpkg-cmake-integration...")
    FetchContent_Declare(
        vcpkg_cmake_integration
        GIT_REPOSITORY https://github.com/bitmeal/vcpkg-cmake-integration.git
        GIT_TAG bd73d80e5ba118f0db3f046a2d71472c84594a91
    )
    FetchContent_MakeAvailable(vcpkg_cmake_integration)
    file(COPY ${vcpkg_cmake_integration_SOURCE_DIR}/vcpkg.cmake DESTINATION ${CMAKE_SOURCE_DIR}/cmake)
    set(VCPKG_PARENT_DIR "${CMAKE_SOURCE_DIR}/third_party")
    set(VCPKG_VERSION edge)
    include(cmake/vcpkg.cmake)
endif()
```

Рисунок 4.43 – Програмний код інтеграції VCPKG для Windows-специфічних залежностей (виконано самостійно)

Система автоматично визначає архітектуру цільової платформи та налаштовує відповідний triplet для VCPKG, забезпечуючи коректну збірку бібліотек для конкретної конфігурації. Інтеграція з vcpkg-cmake-integration спрощує використання VCPKG у CMake проектах та забезпечує прозору інтеграцію з існуючою системою збірки.

Особливу увагу приділено системі компіляції та інтеграції ImGui з різними бекендами рендеринга. Система динамічно додає необхідні backend-файли залежно від обраних графічних API, забезпечуючи коректну роботу інтерфейсу користувача незалежно від конфігурації рендеринга. На рисунку 4.44 показано динамічну конфігурацію ImGui бекендів.

```
if(BUILD_IMGUI)
    FetchContent_MakeAvailable(imgui)

    set(IMGUI_SOURCES
        ${imgui_SOURCE_DIR}/imgui.cpp
        ${imgui_SOURCE_DIR}/imgui_draw.cpp
        ${imgui_SOURCE_DIR}/imgui_widgets.cpp
        ${imgui_SOURCE_DIR}/imgui_tables.cpp
    )
    if(USE_VULKAN)
        list(APPEND IMGUI_SOURCES ${imgui_SOURCE_DIR}/backends/imgui_impl_vulkan.cpp)
    endif()
    if(USE_DIRECTX)
        list(APPEND IMGUI_SOURCES ${imgui_SOURCE_DIR}/backends/imgui_impl_dx12.cpp)
    endif()
    if(BUILD_SDL)
        list(APPEND IMGUI_SOURCES ${imgui_SOURCE_DIR}/backends/imgui_impl_sdl2.cpp)
    endif()
    add_library(imgui_library STATIC ${IMGUI_SOURCES})
endif()
```

Рисунок 4.44 – Програмний код динамічної конфігурації ImGui бекендів (виконано самостійно)

Цей підхід дозволяє створити оптимізовану збірку ImGui, яка включає лише необхідні компоненти для обраної конфігурації системи рендеринга. Система автоматично налаштовує залежності та включає директорії для кожного обраного бекенду.

Система макросів компіляції автоматично генерується на основі обраних опцій конфігурації, дозволяючи коду C++ адаптуватися під час компіляції до доступних можливостей. Це забезпечує ефективну умовну компіляцію та виключення неактивного коду з фінального бінарного файлу.

Використання compile-time конфігурації замість runtime перевірок значно підвищує продуктивність системи, оскільки компілятор може застосовувати агресивні оптимізації та повністю видаляти недоступні гілки коду. Такий підхід також зменшує розмір виконуваного файлу та споживання оперативної пам'яті, що особливо важливо для графічних додатків з обмеженими ресурсами. Додатково, система макросів спрощує підтримку кодової бази, дозволяючи розробникам писати універсальний код, який автоматично адаптується під різні конфігурації без необхідності ручного управління умовними блоками. Це рішення також покращує

читабельність коду та зменшує ймовірність помилок, пов'язаних з некоректними runtime перевірками доступності функціональності.

На рисунку 4.45 представлено систему генерації макросів компіляції.

```
string(TOUPPER "${PROJECT_NAME}" PROJECT_UPPER)

if(BUILD_SDL)
    target_compile_definitions(${PROJECT_NAME} PRIVATE ${PROJECT_UPPER}_USE_SDL)
endif()

if(BUILD_IMGUI)
    target_compile_definitions(${PROJECT_NAME} PRIVATE ${PROJECT_UPPER}_USE_IMGUI)
endif()

if(USE_VULKAN)
    target_compile_definitions(${PROJECT_NAME} PRIVATE ${PROJECT_UPPER}_USE_VULKAN)
endif()

if(USE_DIRECTX AND WIN32)
    target_compile_definitions(${PROJECT_NAME} PRIVATE ${PROJECT_UPPER}_USE_DX12)
endif()
```

Рисунок 4.45 – Програмний код системи генерації макросів компіляції (виконано самостійно)

Розроблена система конфігурації збірки забезпечує високу гнучкість у налаштуванні проекту під різні потреби та платформи, автоматизує управління складними залежностями між компонентами, спрощує процес інтеграції нових сторонніх бібліотек та мінімізує ризики некоректних конфігурацій через систему умовних залежностей. Архітектура дозволяє розробникам легко створювати спеціалізовані збірки для різних сценаріїв використання, від мінімальних конфігурацій для тестування до повнофункціональних збірок з усіма доступними можливостями системи рендеринга.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Тестування є важливим етапом у розробці будь-якого програмного забезпечення, особливо коли мова йде про графічні системи та системи реального часу. Відсутність належного тестування може призвести до критичних помилок у рендерінгу, проблем з продуктивністю, некоректної роботи з різними графічними API та нестабільної поведінки системи під навантаженням. У контексті розробленої програмної системи мульти-API для 3D-рендерінга тестування набуває особливого значення через необхідність забезпечення коректної роботи з кількома графічними API одночасно.

Тестування графічних застосунків кардинально відрізняється від традиційних підходів до тестування програмного забезпечення. Класичні методології юніт-тестування та інтеграційного тестування не можуть бути безпосередньо застосовані до перевірки коректності візуального виходу системи рендерінга. Це зумовлено тим, що результат роботи графічної системи є візуальним за своєю природою і потребує спеціальних інструментів для аналізу послідовності графічних команд, стану графічного конвеєра та ресурсів GPU.

Для ефективного тестування розробленої системи було створено спеціальну тестову сцену, що дозволяє перевіряти роботу системи в умовах, максимально наближених до реального використання, при цьому забезпечуючи можливість детального аналізу кожного етапу процесу рендерінга. Тестова сцена містить базові геометричні об'єкти, різні типи джерел освітлення та матеріали, що дозволяє перевірити всі ключові компоненти системи рендерінга. До складу тестової сцени входять модель авокадо як основний об'єкт для тестування, двосторонній полігон для перевірки обробки матеріалів, а також три типи джерел світла: точкове, направлене та прожекторне освітлення для тестування різних алгоритмів затінення.

Першочерговим завданням тестування було забезпечення коректної ініціалізації та роботи обох підтримуваних графічних API - DirectX 12 та Vulkan. Для цього було налаштовано конфігураційну систему, що дозволяє динамічно перемикатися між API під час запуску програми. Процес ініціалізації системи

супроводжується детальним логуванням всіх критичних етапів, що дозволяє швидко виявляти проблеми конфігурації або несумісності з апаратним забезпеченням.

На рисунку 5.1 показано фрагмент логів ініціалізації системи, де видно послідовність завантаження компонентів системи, ініціалізації обраного графічного API та створення ресурсів рендерінга. З логів можна простежити, що система успішно обирає DirectX 12 як активний API відповідно до конфігурації, ініціалізує GPU профайлер, компілює шейдери та налаштовує всі необхідні компоненти для рендерінга.

```
[info] engine.cpp:103 | initialize() | Engine::initialize() started
[info] asset_loader.h:41 | AssetLoader() | AssetLoader created
[info] asset_loader.h:54 | initialize() | AssetLoader initialized
[info] input_context.cpp:9 | InputContextManager() | InputContextManager initialized with Game context
[info] config_manager.cpp:32 | addConfig() | Config loaded: config/resources/paths.json
[info] config_manager.cpp:32 | addConfig() | Config loaded: config/engine\settings.json
[info] engine.cpp:205 | initialize() | RHI API selected from config: dx12
[info] engine.cpp:218 | initialize() | GPU profiler created for dx12
[info] descriptor_dx12.cpp:566 | initialize() | Frame resources manager initialized with 2 frames
[info] dxc_util.h:109 | compileHlslCode() | Compiling shader for target: vs_6_6
[info] dxc_util.h:109 | compileHlslCode() | Compiling shader for target: ps_6_6
[info] renderer.cpp:328 | initializeGpuProfiler_() | GPU profiler initialized successfully
[info] renderer.cpp:81 | initialize() | Renderer initialized successfully
[info] light_system.cpp:27 | initialize() | LightSystem initialized
[info] imgui_rhi_context.cpp:356 | initializeDx12() | Successfully initialized ImGui DirectX 12 backend
[info] imgui_rhi_context.cpp:87 | initialize() | ImGui RHI context initialized successfully
[info] editor.cpp:59 | initialize() | Editor initialized successfully
[info] engine.cpp:331 | initialize() | Editor initialized successfully
[info] engine.cpp:343 | initialize() | Engine::initialize() completed
[info] application.cpp:22 | setup() | Application::setup() started
[info] config_manager.cpp:32 | addConfig() | Config loaded: config/scenes\scene.json
[info] component_loaders.cpp:172 | g_createEntityFromConfig() | Created entity: PointLight
[info] component_loaders.cpp:172 | g_createEntityFromConfig() | Created entity: Model_Avocado.gltf
[info] component_loaders.cpp:211 | g_processEntityComponents() | Starting async load for model: assets/models/gltf/2.0/Avocado\glTF\Avocado.gltf
[info] asset_loader.h:120 | loadModel() | Queued asset for loading: assets/models/gltf/2.0/Avocado\glTF\Avocado.gltf
[info] component_loaders.cpp:172 | g_createEntityFromConfig() | Created entity: MainCamera
[info] asset_loader.h:276 | loadModelInternal_() | Loading model: assets/models/gltf/2.0/Avocado\glTF\Avocado.gltf
[info] component_loaders.cpp:172 | g_createEntityFromConfig() | Created entity: DirectionalLight
[info] component_loaders.cpp:172 | g_createEntityFromConfig() | Created entity: Model_TwoSidedPlane.gltf
[info] component_loaders.cpp:211 | g_processEntityComponents() | Starting async load for model: assets/models/gltf/2.0/TwoSidedPlane/glTF/TwoSidedPlane.gltf
[info] asset_loader.h:120 | loadModel() | Queued asset for loading: assets/models/gltf/2.0/TwoSidedPlane/glTF/TwoSidedPlane.gltf
[info] component_loaders.cpp:172 | g_createEntityFromConfig() | Created entity: SpotLight
[info] application.cpp:41 | setup() | Application::setup() completed
```

Рисунок 5.1 – Логи ініціалізації системи з DirectX 12 API (виконано самостійно)

Важливим аспектом тестування було проведення порівняльного аналізу роботи системи з різними графічними API. Для цього використовувався інструмент RenderDoc, який дозволяє захоплювати та аналізувати послідовність графічних команд, стан конвеєра рендерінга та ресурси GPU. Такий аналіз дозволяє переконатися в тому, що абстрактний шар Render Hardware Interface коректно транслює команди рендерінга в специфічні виклики кожного API, зберігаючи при цьому ідентичність візуального результату.

На рисунках 5.2 та 5.3 показано Event Browser для DirectX 12 та Vulkan відповідно, де можна порівняти послідовність команд рендерінга для однієї і тієї ж

сцени. Незважаючи на різну внутрішню структуру команд та специфіку кожного API, загальна логіка рендерінга залишається незмінною, що підтверджує коректність реалізації абстрактного шару.

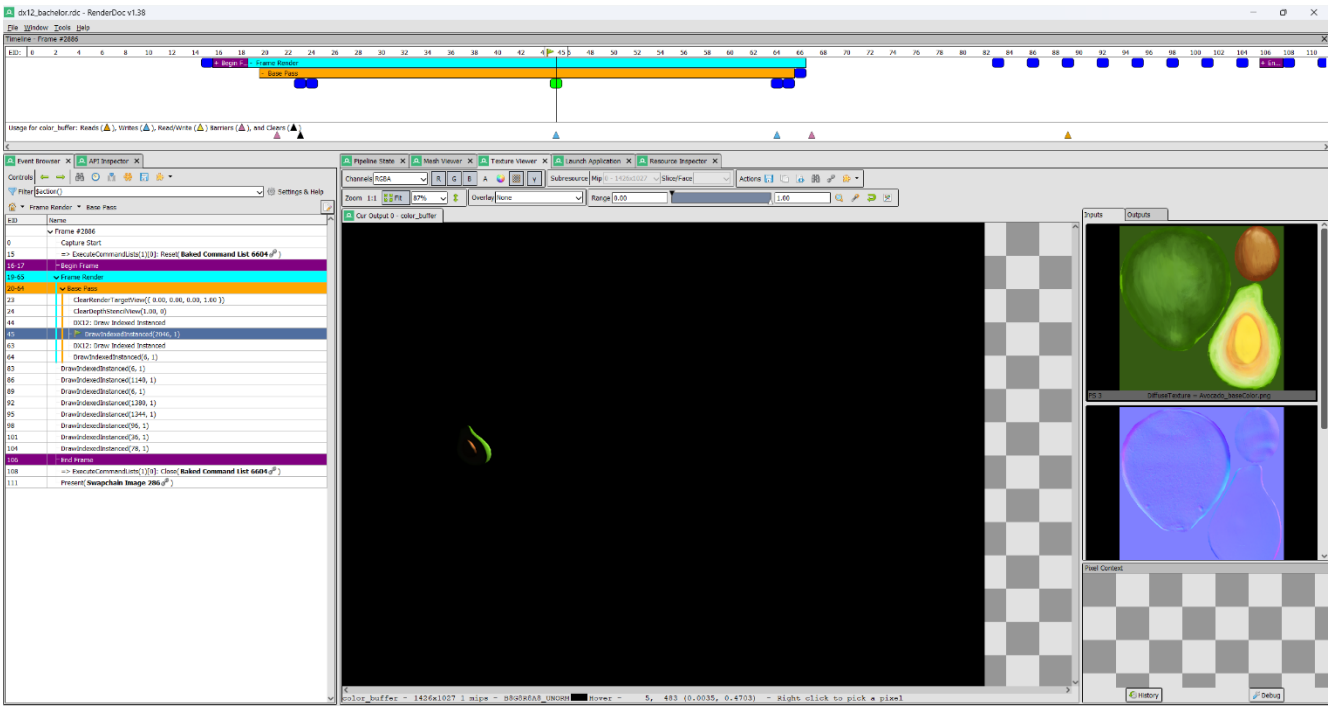

Рисунок 5.2 – Event Browser у RenderDoc для DirectX 12 API (виконано самостійно)

Аналіз послідовності команд DirectX 12 показав правильну організацію рендер-пасів, коректне встановлення render targets та ефективне використання command lists. Система демонструє оптимальну кількість draw calls та правильне групування команд для мінімізації перемикань стану графічного конвеєра. Особливу увагу було приділено перевірці синхронізації між кадрами та коректності роботи з дескрипторними купами. Важливо відзначити, що DirectX 12 надає більш прямий контроль над апаратним забезпеченням, що вимагає ретельної перевірки управління ресурсами та їх життєвим циклом.

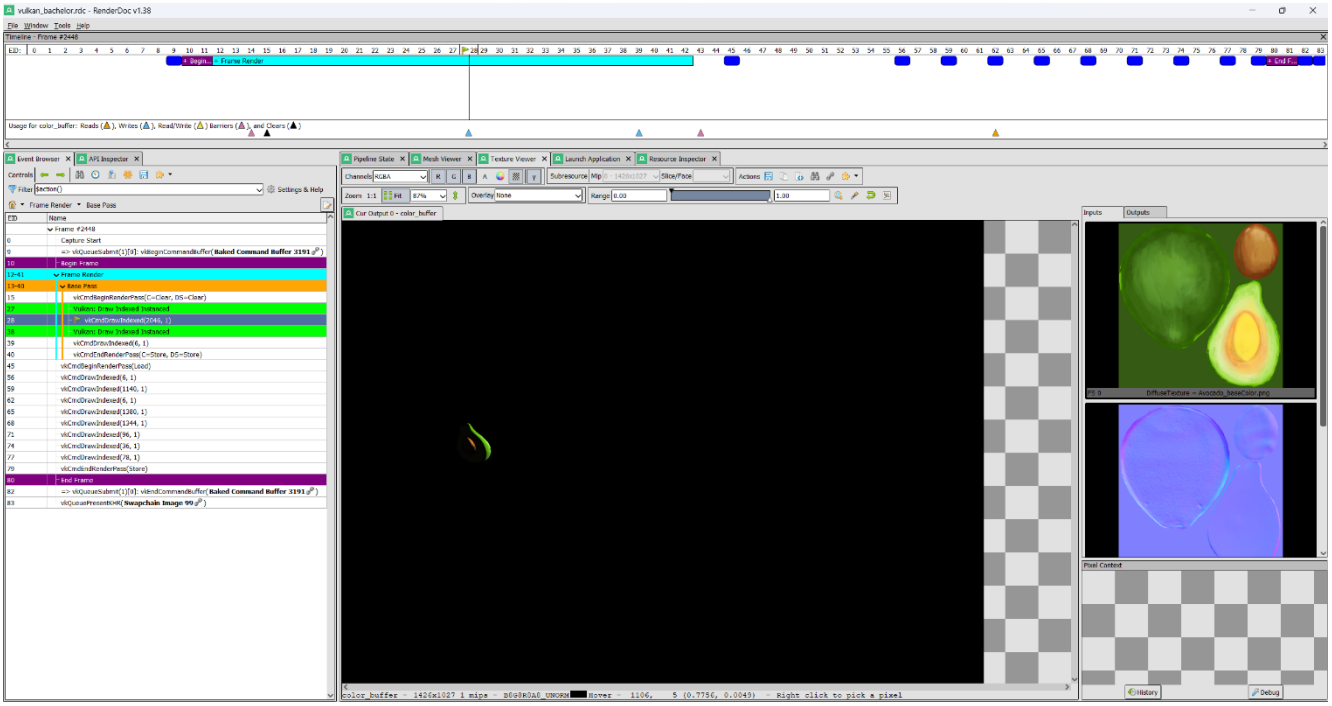

Рисунок 5.3 – Event Browser у RenderDoc для Vulkan API (виконано самостійно)

Порівняльний аналіз Vulkan implementation виявив схожу структуру команд, але з характерними особливостями цього API. Vulkan демонструє більш детальний контроль над синхронізацією через семафори та бар'єри пам'яті, що відображається у більш розгорнутій послідовності команд. Проте загальна архітектура рендерінга залишається ідентичною, що підтверджує ефективність абстрактного шару Rendering.

Детальний аналіз стану графічного конвеєра проводився через панель Pipeline State у RenderDoc. На рисунку 5.4 показано конфігурацію Input Assembler, де видно структуру вхідних даних для рендерінга тестової моделі авокадо. Система коректно налаштовує вершинні буфери з позиціями, текстурними координатами, нормалями та іншими атрибутами вершин. Правильна конфігурація Input Assembler є критично важливою для забезпечення коректної передачі геометричних даних до наступних етапів графічного конвеєра.

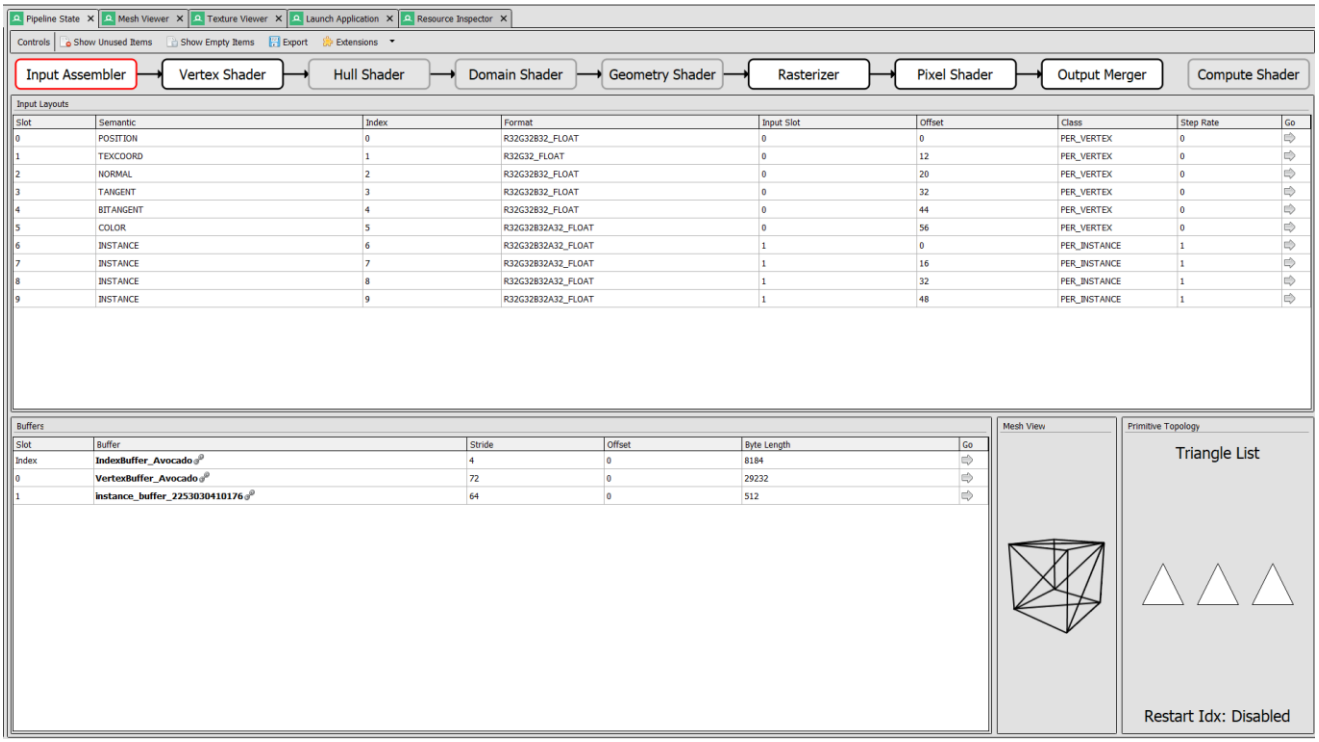

Рисунок 5.4 – Pipeline State Input Assembler з вхідними ресурсами для рендерінга моделі авокадо (виконано самостійно)

Верифікація Input Assembler підтвердила правильність формату вершинних даних та їх розміщення в пам'яті GPU. Всі необхідні атрибути вершин передаються коректно, включаючи позиції у тривимірному просторі, UV координати для текстурування, нормалі для освітлення та тангентні вектори для normal mapping. Індексні буфери також налаштовані правильно, що забезпечує ефективне використання пам'яті та оптимальну продуктивність рендерінга. На рисунку 5.5 показано ресурси, що використовуються пікселним шейдером, включаючи текстури та буфери констант для освітлення та матеріалів. Аналіз ресурсів піксельного шейдера є не менш важливим етапом, оскільки саме на цьому рівні відбувається фінальне формування кольору пікселів.

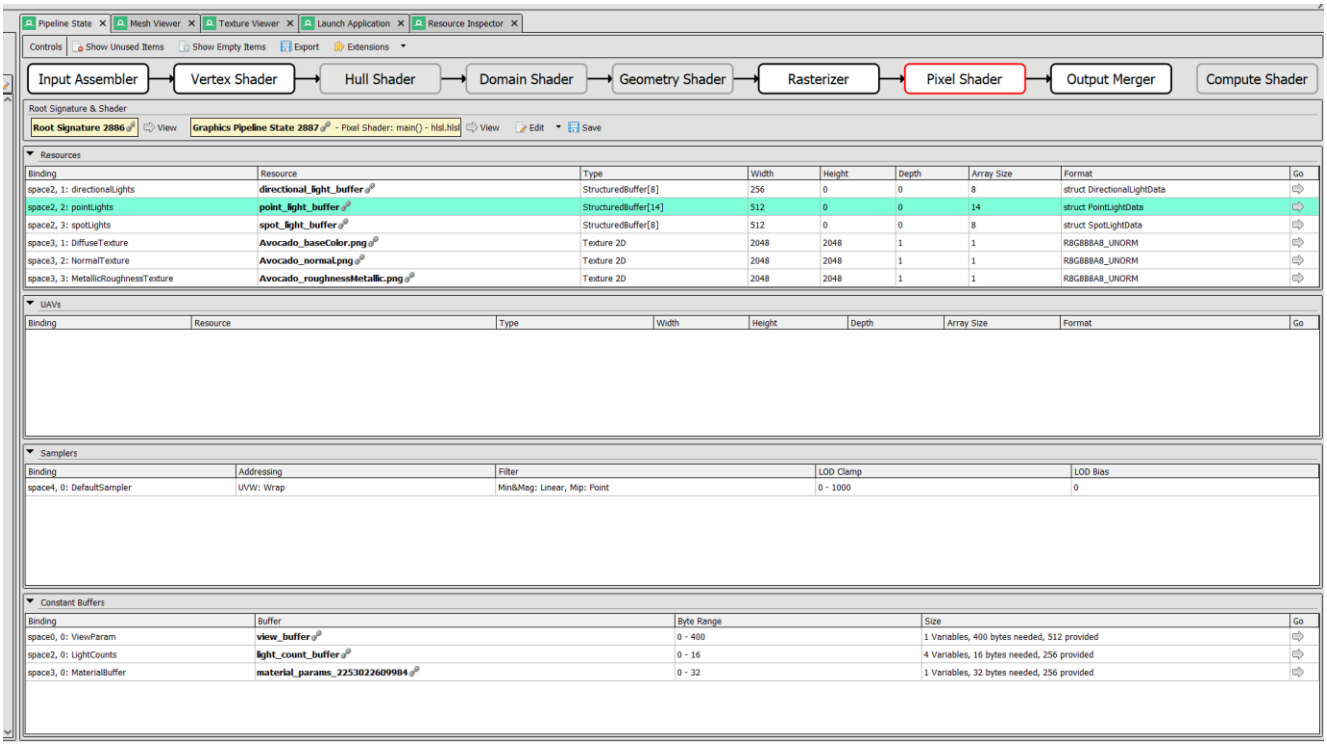

Рисунок 5.5 – Pipeline State Pixel Shader з ресурсами для затінення моделі авокадо (виконано самостійно)

Аналіз pixel shader resources показав правильну прив'язку всіх необхідних текстур та uniform buffers. Система коректно передає diffuse та normal map текстури, а також constant buffers з параметрами освітлення, матеріалів та view, projection матрицями. Особливо важливою є перевірка правильності sample states для текстур, що забезпечує якісну фільтрацію та відсутність артефактів при рендерінгу.

Mesh Viewer у RenderDoc дозволяє візуально перевірити коректність передачі геометричних даних до GPU. На рисунку 5.6 показано тривимірну модель авокадо в різних режимах відображення, що підтверджує правильність завантаження та обробки мешів у системі.

Візуальна інспекція геометрії є незамінним інструментом для виявлення потенційних проблем з топологією мешів або некоректним завантаженням 3D-моделей.

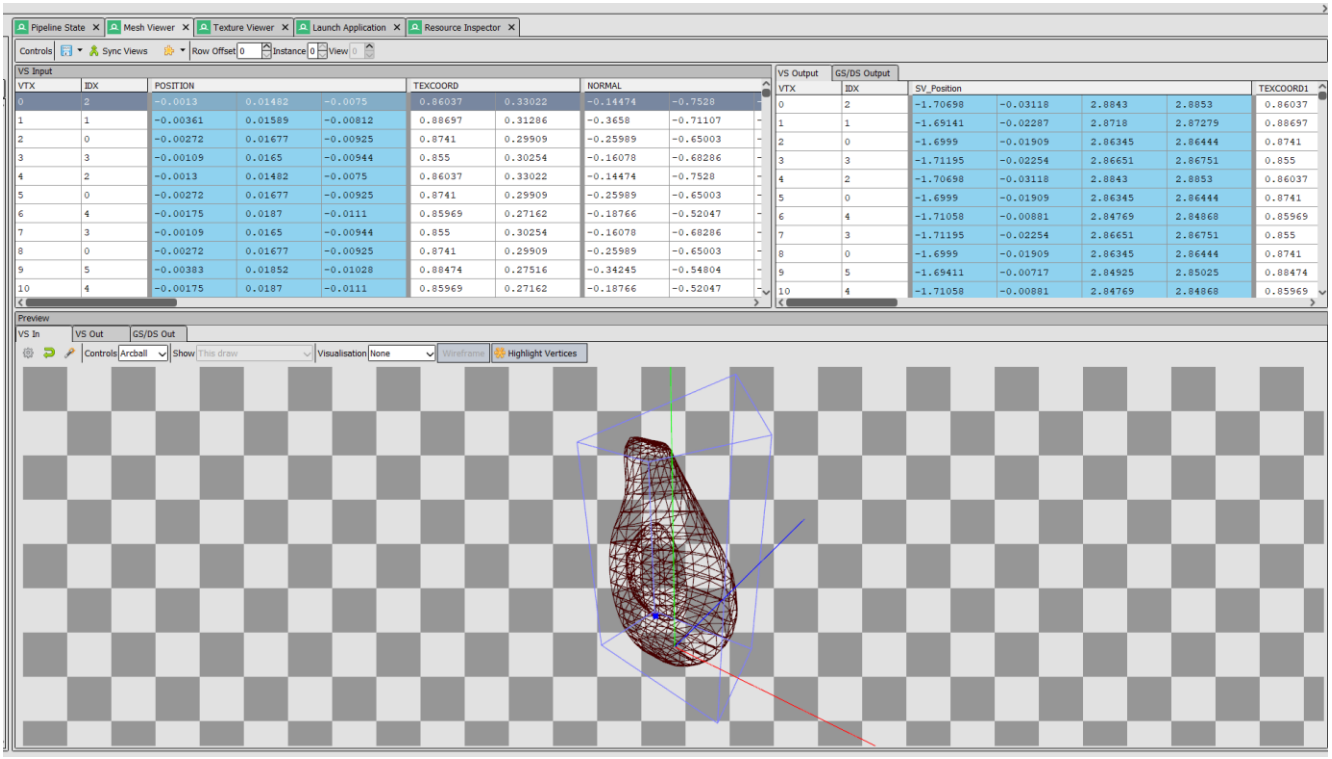

Рисунок 5.6 – Mesh Viewer з візуалізацією геометрії моделі авокадо (виконано самостійно)

Mesh Viewer надає можливість детального огляду геометрії на рівні окремих трикутників та вершин. Візуалізація підтвердила, що всі полігони моделі передаються коректно, без втрати даних або спотворень геометрії. Система правильно обробляє як прості меші з базовою геометрією, так і складні моделі з великою кількістю деталей. Перевірка різних режимів відображення, включаючи wireframe та solid modes, показала стабільну роботу з різними типами рендерінга.

Особливо важливим компонентом тестування було профілювання продуктивності системи за допомогою інтегрованого профайлера Tracy. Цей інструмент дозволяє відстежувати як CPU, так і GPU навантаження в реальному часі, виявляти вузькі місця в продуктивності та оптимізувати критичні ділянки коду. На рисунку 5.7 показано timeline профайлера, де видно розподіл навантаження між CPU та GPU зонами протягом рендерінга одного кадру.

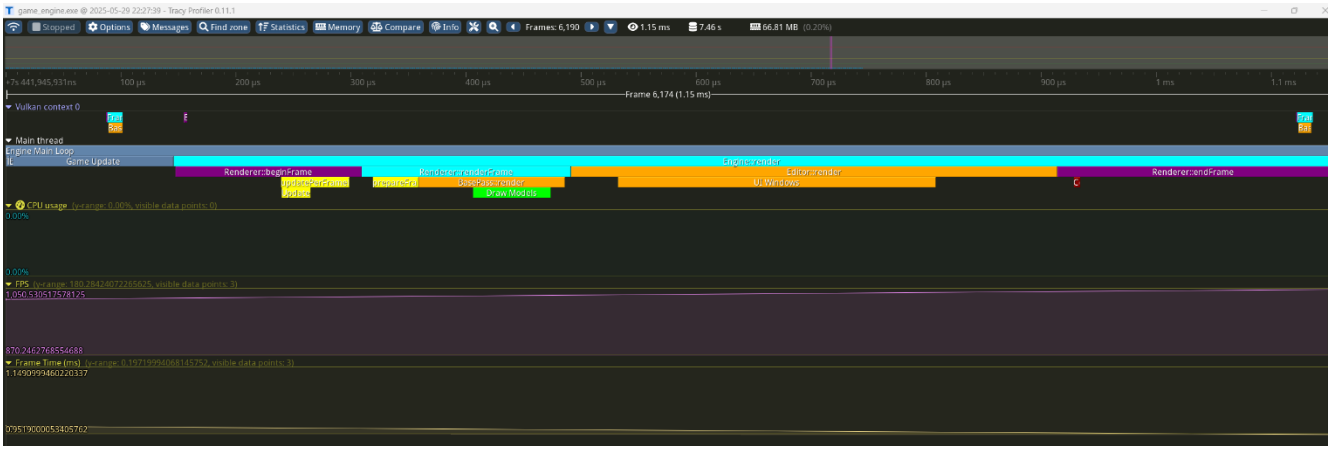

Рисунок 5.7 – Tracy профайлер timeline з CPU та GPU зонами під час рендерінга кадру (виконано самостійно)

Профілювання через Tracy виявило ефективний розподіл навантаження між CPU та GPU компонентами системи. Timeline демонструє мінімальні простої GPU та оптимальне перекриття CPU робіт з GPU виконанням. Особливо важливим є те, що система демонструє стабільний frame time без значних піків або провалів у продуктивності, що свідчить про відсутність критичних вузьких місць у реалізації.

Інтеграція Tracy профайлера в систему була реалізована через макроси CPU_ZONE та GPU_ZONE, що дозволяють легко додавати зони профілювання в код без значного впливу на продуктивність. Профайлер автоматично збирає статистику по часу виконання різних етапів рендерінга, від підготовки команд на CPU до виконання шейдерів на GPU.

Тестування також включало перевірку стабільності системи під різними навантаженнями та в різних режимах роботи. Система була протестована з різними типами геометрії, різною кількістю джерел світла, різними матеріалами та текстурами. Особлива увага приділялася тестуванню перемикання між різними режимами рендерінга в редакторі, такими як solid, wireframe, та різними режимами візуалізації для налагодження.

Результати тестування підтвердили стабільність та коректність роботи розробленої системи з обома графічними API. Система демонструє ідентичний візуальний результат при роботі з DirectX 12 та Vulkan, зберігаючи при цьому оптимальну продуктивність та ефективно використовуючи ресурси GPU..

ВИСНОВКИ

У ході виконання кваліфікаційної роботи її мета - розробка програмної системи мульти-API для 3D-рендеринга – виконана у повному обсязі.

Розробка проводилась з використанням мови програмування C++ зі стандартом C++20, системи збирання CMake та підтримкою графічних API DirectX 12 і Vulkan для забезпечення крос-платформної сумісності.

Було проведено детальний аналіз предметної галузі 3D-рендеринга, що включав дослідження існуючих рішень, таких як Open 3D Engine, Unreal Engine та Godot Engine, а також вивчення сучасних тенденцій розвитку графічних технологій. Це дозволило виявити основні недоліки існуючих систем та сформувати чіткі вимоги до розробки нової архітектури, орієнтованої на доступність для новачків та гнучкість використання.

Було створено модульну багатошарову архітектуру системи, яка забезпечує чітке розділення відповідальностей між компонентами. Розроблено комплект UML-діаграм, що ілюструють взаємодію основних підсистем: шару абстракції графічних API, системи рендеринга, управління ресурсами та інтегрованого редактора.

Впровадження архітектури Entity Component System забезпечило ефективне управління ігровими об'єктами та їх властивостями.

Система мульти-API рендеринга була оптимізована для забезпечення високої продуктивності та якості візуалізації на різному апаратному забезпеченні. Створений шар абстракції Render Hardware Interface дозволяє прозоро працювати з різними графічними API через єдиний інтерфейс, забезпечуючи ідентичність візуальних результатів. Впровадження системи асинхронного завантаження ресурсів та динамічного оновлення шейдерів значно покращує користувацький досвід та прискорює процес розробки.

Система розроблена з урахуванням принципів розширення та підтримування, що дозволяє легко адаптувати її до майбутніх потреб та технологічних змін. Модульна архітектура забезпечує можливість додавання підтримки нових

графічних API, розширення функціональності редактора та інтеграції додаткових інструментів розробки без модифікації існуючого коду.

Використання сучасних патернів проектування та стандартів C++20 гарантує довгострокове підтримування системи.

Розроблена програмна система надає можливість для навчання, експериментування та створення практичних інноваційних графічних додатків у галузі комп'ютерної графіки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Bionic image segmentation of cytology samples method / A. Rabotiahov та ін. 2018 14th international conference on advanced trends in radioelecrtronics, telecommunications and computer engineering (TCSET), м. Lviv-Slavske, 20–24 лют. 2018 р. 2018. URL: https://doi.org/10.1109/tcset.2018.8336289 (дата звернення: 01.06.2025).

2. 3D rendering market growth, trends and forecast | the brainy insights. Global Market Research Reports and Consulting Services | The Brainy Insights. URL: https://www.thebrainyinsights.com/report/3d-rendering-market-13936 (дата звернення: 09.05.2025).

3. Youvan D. C. From graphics to AI: the evolution of gpus and tensor boards. 2024. 36 с. (Препринт). URL: https://doi.org/10.13140/RG.2.2.23379.59684 (дата звернення: 09.05.2025).

4. Bionic model of blood cell segmentation based on impulse image transformation / R. Y. Bukhtiiarov та ін. Polish journal of medical physics and engineering. 2024. Т. 30, № 4. URL: https://doi.org/10.2478/pjmpe-2024-0027 (дата звернення: 01.06.2025).

5. Grand view research. Market Research Reports & Consulting | Grand View Research, Inc. URL: https://www.grandviewresearch.com/industry-analysis/3d-rendering-market (дата звернення: 09.05.2025).

6. Analysis of human speech as a protection tool in infocommunication systems / V. Lyashenko та ін. 2018 international scientific-practical conference problems of infocommunications. science and technology (PIC S&T), м. Kharkiv, Ukraine, 9–12 жовт. 2018 р. 2018. URL: https://doi.org/10.1109/infocommst.2018.8632156 (дата звернення: 01.06.2025).

7. Porting | vulkan | cross platform 3D graphics. The Khronos Group Inc. URL: https://www.khronos.org/vulkan/portability-initiative (дата звернення: 09.05.2025).

8. Some feedbacks about the O3DE building process. o3de/o3de discussion #3943. GitHub. URL: https://github.com/o3de/o3de/discussions/3943 (дата звернення: 09.05.2025).

9. Epic Developer Community Docs. Guidelines for Optimizing Rendering for Real-Time in Unreal Engine. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/guidelines-for-optimizing-rendering-for-real-time-in-unreal-engine (дата звернення: 09.05.2025).

10. 3D rendering limitations. Godot Engine documentation. URL: https://docs.godotengine.org/en/stable/tutorials/3d/3d_rendering_limitations.html (дата звернення: 09.05.2025).

11. Graphics API abstraction – wicked engine. Wicked Engine. URL: https://wickedengine.net/2021/05/graphics-api-abstraction/ (дата звернення: 09.05.2025).

12. Let's build an Entity Component System from scratch (part 1). Hexops' devlog. URL: https://devlog.hexops.com/2022/lets-build-ecs-part-1/ (дата звернення: 09.05.2025).

13. Tahan E. Applying SOLID principles in game development: creating a more flexible and sustainable code… Medium. URL: https://medium.com/@ezgitahann/applying-solid-principles-in-game-development-creating-a-more-flexible-and-sustainable-code-4f0a47be291f (дата звернення: 09.05.2025).

14. Архів роботи на GitHub. URL: https://github.com/vadymchan/2025_B_PI_PZPI-21-2_Chan_V_H (дата звернення: 10.06.2025).