Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри

_____ **Артем ВОЛОКИТА**
(підпис)

_____ _____ 2026 р.
(число) (місяць)

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем»
спеціальність 121 – «Інженерія програмного забезпечення»

ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ VULKAN API

Виконав
Ярошенко Олександр Сергійович, студент 4 курсу, група ІМ-21 _____
(підпис)

Керівник
Порєв Віктор Миколайович, кандидат технічних наук, доцент _____
(підпис)

Консультант
Нікольський Сергій Сергійович, асистент _____
(підпис)

Рецензент
Шимкович В.М., кандидат технічних наук _____
(підпис)

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

_____ **Ярошенко О. С.**
(підпис)

Київ – 2026 р.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень: Бакалавр
Освітньо-професійна програма: «Інженерія програмного забезпечення комп'ютерних систем»
Спеціальність: 121 – «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри

_____ **Артем ВОЛОКИТА**
(підпис)

_____ _____ 2026 р.
(число) (місяць)

ЗАВДАННЯ
на бакалаврський дипломний проєкт

Ярошенко Олександру Сергійовичу

1. Тема проєкту: Програмний двигун на основі Vulkan API
керівник Порєв Віктор Миколайович, кандидат технічних наук, доцент
затверджено наказом по університету від 5 червня 2026 року № 1212-с

2. Дата здачі закінченого проєкту 17 червня 2026 року
(число, місяць, рік)

3. Вихідні дані для проекту: технічна документація, теоретичні та статистичні дані, патенти на винахід, наукові публікації.

4. Зміст розрахунково-пояснювальної записки: опис предметної області і напрямків дослідження, аналіз та характеристика об'єкту досліджень, аналіз існуючих рішень, опис архітектури системи.

5. Перелік графічного матеріалу: принципова, структурна та функціональна схеми системи.

6. Консультанти розділів проєкту

№	Розділ	Консультант (прізвище та ініціали)	Підпис, дата	
			завдання видав	завдання прийняв
1	Нормоконтроль	Нікольський С. С.		

7. Дата видачі завдання 10 лютого 2026 року
(число, місяць, рік)

Календарний план

№	Назва етапу виконання дипломного проекту	Термін виконання	Примітка
1	Затвердження теми проєкту	02.02.2026-10.02.2026	
2	Вивчення та аналіз завдання	10.02.2026-02.04.2026	
3	Розробка архітектури та загальної структури системи	02.04.2026-19.04.2026	
4	Програмна реалізація системи	19.04.2026-21.05.2026	
5	Оформлення пояснювальної записки	21.05.2026-04.06.2026	
6	Передзахист	9.06.2026	
7	Захист	16.06.2026	

Виконавець _____ Ярошенко Олександр Сергійович
(підпис)

Керівник _____ Порєв Віктор Миколайович
(підпис)

АНОТАЦІЯ

Об'єм і структура. Проєкт складається зі вступу, трьох розділів, висновків, переліку умовних скорочень та списку літератури.

Актуальність теми. 3D-графіка є основою ігор, систем візуалізації та редакторів сцен; розробка власного рушія на базі сучасного низькорівневого Vulkan API з інтеграцією в Qt/QML є актуальною інженерною задачею.

Метою проєкту є розробка 3D рушія на основі Vulkan API з використанням Qt та QML для відображення тривимірних сцен у редакторі та автономному режимі.

Поставлені **задачі**:

1. проаналізувати застосування Vulkan API, Qt і QML у графічних застосунках;
2. спроєктувати структуру рушія та його основні модулі;
3. реалізувати Vulkan-контекст, графічний конвеєр і рендеринг сцени;
4. реалізувати завантаження моделей glTF, роботу з текстурами та камерою;
5. забезпечити відображення кадру у Qt/QML-інтерфейсі та окремий runtime-режим.

Ключові слова: *3D-графічний рушій, Vulkan API, Qt, QML, glTF, динамічний рендеринг, offscreen-рендеринг, графічний інтерфейс.*

ANNOTATION

The aim of this project is to develop a 3D engine based on the Vulkan API with Qt and QML for rendering three-dimensional scenes in an editor window and a standalone runtime mode.

Tasks:

1. analyze the use of Vulkan API, Qt, and QML in graphics applications;
2. design the structure of the 3D engine and its main modules;
3. implement the Vulkan context, graphics pipeline, and scene rendering;
4. implement glTF model loading, textures, camera control and scene navigation;
5. display the rendered image inside a Qt/QML interface and provide a standalone runtime mode.

Keywords: *3D engine, Vulkan API, Qt, QML, glTF, dynamic rendering, offscreen rendering, graphical user interface.*

Довідки	Формат	Позначення	Найменування	Кільк. листів	№ екземпляра	Додаток
	А4		Завдання на дипломний проект	4		
	А4	ІАЛЦ.467200.001 ОА	ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ	1		
			VULKAN API			
			Опис альбому			
	А4	ІАЛЦ.467200.002 ТЗ	ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ	6		
			VULKAN API			
			Технічне завдання			
	А4	ІАЛЦ.467200.003 ПЗ	ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ	75		
			VULKAN API			
			Пояснювальна записка			
	А4	ІАЛЦ.467200.004 Д1	ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ	2		
			VULKAN API			
			Принципова схема			
	А4	ІАЛЦ.467200.005 Д2	ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ	2		
			VULKAN API			
			Функціональна схема			

Зм.	Лист	№ докум.	Підп.	Дата
Розробив		Ярошенко О. С.		
Перевірив		Порєв В. М.		
Н. контр.		Нікольський С. С.		
Затвердив		Волокита А. М.		

ІАЛЦ.467200.001 ОА

Програмний двигун на основі Vulkan API
Опис альбому

Лит.	Аркуш	Аркушів
	1	2

НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ ІМ-21

Довідки	Формат	Позначення	Найменування	Кільк. листів	№ екземпляра	Додаток
	А4	ІАЛЦ.467200.006 Д3	ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ	2		
			VULKAN API			
			Структурна схема			
	А4	ІАЛЦ.467200.007 Д4	ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ	6		
			VULKAN API			
			Допоміжні таблиці			
	А4	ІАЛЦ.467200.008 Д5	ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ	34		
			VULKAN API			
			Текст програми			

ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ VULKAN API

Технічне завдання

ІА/ЛЦ.467200.002 ТЗ

Київ – 2026 р.

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування: програмний 3D рушій на основі Vulkan API з графічним інтерфейсом на базі Qt та QML.

Область застосування: інтерактивні 3D-застосунки, редактори сцен, системи візуалізації, навчальні та дослідницькі програмні засоби з комп’ютерної графіки.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського дипломного проєкту за освітньо-професійною програмою «Інженерія програмного забезпечення комп’ютерних систем» спеціальності 121 «Інженерія програмного забезпечення», затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даного проєкту є розробка програмного 3D-графічного рушія на основі Vulkan API для відображення тривимірних сцен у реальному часі, завантаження моделей формату glTF, керування сценами та інтеграції з графічним інтерфейсом на базі Qt та QML.

Призначення розробки полягає у створенні програмного продукту для роботи з тривимірними сценами, який забезпечує рендеринг, завантаження моделей, керування ресурсами, обробку подій та взаємодію з графічним інтерфейсом користувача.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є науково-технічна література з комп’ютерної графіки та проєктування програмних систем, документація Vulkan API, Qt та QML, матеріали щодо побудови систем рендерингу, керування сценами та ресурсами, а також публікації та технічна документація в мережі Інтернет.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

- Програмний продукт повинен забезпечувати ініціалізацію Vulkan API та відображення тривимірних сцен у реальному часі.
- У системі мають бути передбачені засоби ієрархічної організації сцени та керування її об’єктами.
- У системі мають бути передбачені засоби організації процесу рендерингу та взаємодії між його основними етапами.
- Програмний продукт повинен забезпечувати керування моделями, текстурами, матеріалами, шейдерами та іншими ресурсами.
- Продукт повинен підтримувати завантаження 3D-моделей формату glTF, обробку геометрії, матеріалів і текстур.
- Має бути реалізована система камер, базова навігація сценою та обробка вводу користувача.
- Має бути реалізований графічний інтерфейс користувача на базі Qt та QML для відображення сцени та взаємодії з нею.
- У програмному продукті мають бути передбачені засоби обміну подіями між основними підсистемами.
- Система повинна підтримувати окремий режим редактора та окремий runtime-режим запуску.

– Продукт повинен забезпечувати можливість подальшого розширення архітектури без суттєвої переробки базових модулів.

5.2. Вимоги до структури програмного продукту

– Програмний продукт повинен мати модульну структуру.
– У структурі програмного продукту мають бути передбачені підсистеми рендерингу, керування сценою, керування ресурсами, обробки подій та графічного інтерфейсу користувача.
– Програмний продукт повинен підтримувати режим редактора та окремий режим автономного запуску.
– Структура програмного продукту повинна забезпечувати можливість подальшого розширення функціональності.
– Програмний продукт повинен забезпечувати взаємодію між графічною підсистемою та інтерфейсом користувача.
– Організація програмного продукту повинна забезпечувати зручність супроводу, тестування та подальшої модифікації.

5.3. Вимоги до програмного забезпечення

– Операційна система GNU/Linux.
– Підтримка Vulkan 1.3 та встановлений Vulkan SDK.
– Qt 6.5 або вище з компонентами Core, Gui, Quick та Qml.
– Система збирання CMake.
– Компілятор C++ з підтримкою стандарту C++23.
– Пакет GLFW для автономного runtime-режиму.
– Засоби компіляції шейдерів, сумісні з проєктом, зокрема slangc.

5.4. Вимоги до апаратної частини

- Персональний комп’ютер або ноутбук на базі 64-бітного процесора.
- Оперативна пам’ять не менше 8 Гбайт.
- Відеоадаптер із підтримкою Vulkan 1.3.
- Вільний простір на накопичувачі не менше 1 Гбайт.
- Монітор або вбудований дисплей для роботи з графічним інтерфейсом редактора.

6. ЕТАПИ РОЗРОБКИ

6.1. Аналіз предметної області, сучасних 3D рушіїв та засобів Vulkan, Qt і QML.

6.2. Формування архітектури системи та проєктування основних модулів рушія.

6.3. Розробка базового Vulkan-контексту, графічного конвеєра та механізмів рендерингу.

6.4. Розробка засобів керування сценою, камерою та вводом користувача.

6.5. Розробка засобів організації рендерингу та керування ресурсами.

6.6. Розробка системи обробки подій та взаємодії між підсистемами програмного продукту.

6.7. Розробка графічного інтерфейсу на базі Qt/QML та інтеграція його з підсистемою рендерингу.

6.8. Реалізація підтримки завантаження моделей формату glTF, текстур і матеріалів.

6.9. Налагодження, тестування, оптимізація та підготовка документації.

ПРОГРАМНИЙ ДВИГУН НА ОСНОВІ VULKAN API

Пояснювальна записка

ІА/ЛЦ.467200.003 ПЗ

Київ – 2026 р.

ЗМІСТ

ВСТУП

Комп’ютерна графіка є однією з ключових галузей сучасної інформатики та програмної інженерії, оскільки лежить в основі ігрових застосунків, систем візуалізації, симуляторів, засобів автоматизованого проєктування, віртуальної та доповненої реальності. Особливе значення мають засоби формування тривимірних сцен у реальному часі, для яких критичними є продуктивність, керованість графічним конвеєром, ефективне використання апаратних ресурсів і можливість масштабування програмної архітектури [1, 2].

У сучасних умовах 3D-графічний рушій розглядається не лише як інструмент відображення геометрії, а як комплексна програмна система, що поєднує підсистеми рендерингу, керування сценою, ресурсами, матеріалами, освітленням, камерою, подіями та користувацькою взаємодією. Якість побудови такої системи значною мірою визначає ефективність подальшої розробки прикладних програмних продуктів, а також зручність їхнього супроводу та розширення [3].

Важливою тенденцією розвитку графічного програмного забезпечення є перехід від високорівневих інтерфейсів до низькорівневих графічних API, які надають розробнику точніший контроль над пам’яттю, синхронізацією, командними буферами та взаємодією з графічним процесором. Одним із найактуальніших рішень цього класу є Vulkan - кросплатформний графічний і обчислювальний API, орієнтований на високу продуктивність, передбачуваність виконання та явне керування ресурсами [4–6].

Актуальність теми дипломної роботи зумовлена потребою у створенні програмних засобів для побудови та дослідження сучасних 3D-графічних систем, а також практичною цінністю розробки власного графічного рушія як навчально-дослідницької платформи. Такий підхід дає змогу глибше вивчити принципи організації рендерингу в реальному часі, архітектуру графічних

підсистем і закономірності взаємодії між окремими компонентами складної програмної системи [2, 3].

Метою дипломної роботи є розробка програмного 3D-графічного рушія на основі Vulkan API для відображення тривимірних сцен у реальному часі, завантаження моделей, керування сценовими об’єктами та інтеграції з графічним інтерфейсом користувача.

Об’єктом дослідження є процес побудови програмних систем для відображення та керування тривимірними сценами.

Предметом дослідження є методи, засоби та архітектурні підходи до реалізації 3D-графічного рушія з використанням низькорівневого графічного API.

Для досягнення поставленої мети в роботі необхідно розв’язати такі завдання:

а) проаналізувати предметну область 3D-графіки, рендерингу в реальному часі та сучасних графічних API;

б) дослідити сучасні підходи до побудови 3D-графічних рушіїв і виконати порівняльний аналіз наявних рішень;

в) сформувати вимоги до програмного рушія та обґрунтувати архітектурні принципи його побудови;

г) реалізувати основні підсистеми рушія, необхідні для відображення тривимірної сцени, керування ресурсами та взаємодії з користувачем;

д) перевірити працездатність розробленого програмного рішення та оцінити можливості його подальшого розвитку.

У роботі використано методи аналізу науково-технічних джерел, порівняння сучасних програмних рішень, методи проєктування програмних систем, а також модульний та об’єктно-орієнтований підходи до розробки програмного забезпечення [1, 3].

Практичне значення одержаних результатів полягає у створенні програмного 3D-графічного рушія, який може використовуватися як навчальна, дослідницька або базова експериментальна платформа для подальшого розвитку засобів рендерингу, редакторів сцен і спеціалізованих графічних застосунків.

Дипломна робота складається зі вступу, трьох розділів, висновків, переліку умовних скорочень та списку використаних джерел. У першому розділі розглянуто теоретичні основи побудови 3D-графічних рушіїв, у другому - питання проєктування архітектури програмного рішення, у третьому - особливості реалізації та результати розробки.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ 3D-ГРАФІЧНИХ РУШІЇВ

1.1 Сутність та понятійний апарат розробки 3D-графічних рушіїв

Комп'ютерна графіка є галуззю інформатики, що охоплює методи створення, обробки, зберігання та відображення візуальної інформації засобами обчислювальної техніки. Одним із найважливіших її напрямів є тривимірна графіка, у межах якої цифрові об'єкти описуються в координатному просторі, а кінцеве зображення формується на основі математичних моделей геометрії, освітлення, матеріалів і спостереження [1, 2].

Тривимірна графіка широко використовується в ігровій індустрії, системах автоматизованого проєктування, візуальному моделюванні, медичних симуляторах, науковій візуалізації, системах підготовки персоналу, а також у засобах віртуальної та доповненої реальності. На відміну від статичного рендерингу, значна частина сучасних прикладних задач потребує відображення сцени в реальному часі, коли новий кадр формується за дуже обмежений проміжок часу. Саме ця вимога визначає характер архітектурних і алгоритмічних рішень у графічних рушіях [2, 7].

Під рендерингом у реальному часі доцільно розуміти процес синтезу зображення з такою швидкістю, яка забезпечує безперервне сприйняття динаміки сцени користувачем. На практиці це означає, що всі етапи підготовки даних, взаємодії з графічним процесором, виконання шейдерних програм і виведення кадру мають вкладатися в жорсткий часовий бюджет. Відповідно, у центрі уваги під час розробки графічних систем перебувають ефективність

керування пам’яттю, мінімізація накладних витрат, оптимізація передачі даних та раціональна організація команд виконання [2, 6].

Одним із базових понять теми є *3D-графічний рушій*. У загальному розумінні це програмна система або сукупність взаємопов’язаних модулів, що забезпечують представлення сцени, керування графічними ресурсами, обробку вводу, моделювання поведінки об’єктів та побудову зображення засобами графічного API. На відміну від окремої візуалізаційної бібліотеки, рушій зазвичай має комплексний характер і слугує основою для створення прикладних застосунків певного класу [3].

Не менш важливим є поняття *сцени*. Сцена являє собою структуроване подання сукупності об’єктів, що підлягають відображенню: геометрії, джерел світла, камер, матеріалів, текстур та допоміжних сутностей. Практична організація сцени часто передбачає ієрархію об’єктів, просторові перетворення, логічне групування сутностей та зв’язок між візуальним представленням і внутрішніми даними програми [1, 3].

Під *графічним конвеєром* розуміють послідовність етапів перетворення початкових даних сцени у готове растрове зображення. У сучасних системах до цього процесу входять опрацювання вершин, складання примітивів, растеризація, фрагментна обробка, тестування глибини, змішування кольору та інші операції. Вивчення графічного конвеєра є методологічно важливим, оскільки саме він визначає спосіб побудови шейдерних програм, розподіл відповідальності між центральним і графічним процесорами та структуру даних, що передаються в підсистему рендерингу [1, 2].

Ще одним ключовим поняттям є *графічний API* - програмний інтерфейс, що надає доступ до функцій графічного процесора та засобів формування зображення. Історично графічні API пройшли шлях від відносно абстрактних високорівневих інтерфейсів до низькорівневих засобів, які забезпечують явне керування командами, пам’яттю та синхронізацією. Для сучасних

високопродуктивних графічних систем особливий інтерес становлять саме низькорівневі API, серед яких Vulkan посідає одне з провідних місць [5, 8, 9].

Отже, понятійний апарат досліджуваної теми охоплює взаємопов'язані категорії: комп'ютерну графіку, тривимірну сцену, рендеринг у реальному часі, графічний конвеєр, графічний API та графічний рушій. Сукупність цих понять формує основу для подальшого аналізу історії розвитку галузі, порівняння сучасних рішень і вироблення методологічних принципів побудови власного 3D-рушія.

1.2 Історія розвитку та сучасний стан технологій тривимірної графіки і графічних API

Розвиток тривимірної комп'ютерної графіки безпосередньо пов'язаний із загальним прогресом обчислювальної техніки та спеціалізованих графічних прискорювачів. На ранніх етапах значна частина операцій побудови зображення виконувалася програмно або за допомогою жорстко визначених апаратних конвеєрів. Такий підхід забезпечував базову візуалізацію, проте істотно обмежував гнучкість у реалізації матеріалів, освітлення та спеціальних ефектів [1].

Важливою віхою в еволюції галузі стало поширення стандартизованих графічних API, зокрема OpenGL, який тривалий час виконував роль універсального кросплатформного інструмента для високопродуктивної графіки. OpenGL сприяв формуванню спільної моделі взаємодії між прикладною програмою та графічним обладнанням, а також став основою для численних навчальних і промислових рішень [8]. Паралельно на платформі Windows активно розвивалася лінійка Direct3D, що відіграла значну роль у становленні комерційних графічних застосунків та ігрових рушіїв [9].

Подальший розвиток графічних систем супроводжувався переходом від фіксованого конвеєра до програмованого. Запровадження вершинних і

фрагментних шейдерів дало змогу реалізовувати значно складніші моделі освітлення, анімації, постобробки та процедурної генерації ефектів. Це стало підґрунтям для якісного стрибка в реалістичності візуалізації, а також для поширення фізично обґрунтованих моделей матеріалів та освітлення [2, 7].

Зі зростанням складності сцен і збільшенням вимог до частоти кадрів дедалі гострішою ставала проблема накладних витрат на рівні драйверів та недостатньої керованості з боку прикладної програми. У відповідь на це з'явився новий клас низькорівневих API, орієнтованих на явне керування графічними ресурсами, синхронізацією та чергами виконання. Vulkan є одним із найхарактерніших представників цього підходу. На відміну від традиційніших API, він покладає більше відповідальності на розробника, але водночас відкриває ширші можливості оптимізації та контролю над процесом рендерингу [4–6].

Якщо розглядати сучасні графічні API з позицій практичного використання, доцільно порівнювати їх за рівнем абстракції, ступенем контролю над ресурсами, платформною прив'язкою, складністю освоєння та придатністю до високопродуктивного рендерингу. У цьому контексті OpenGL, Direct3D 12 і Vulkan репрезентують різні етапи еволюції графічних інтерфейсів та різні підходи до організації взаємодії між програмою і графічним процесором [5, 8, 9].

OpenGL характеризується відносно вищим рівнем абстракції та простішим входженням у розробку графічних застосунків. Саме тому він тривалий час залишався одним з основних інструментів навчання, прототипування та створення кросплатформних рішень. Водночас частина керування ресурсами та оптимізацій традиційно приховується на рівні драйвера, що обмежує передбачуваність продуктивності у складних високонавантажених сценах [8].

Direct3D 12, навпаки, орієнтований на нижчий рівень абстракції та на-

дає розробнику значно ширші можливості контролю над пам’яттю, списками команд і синхронізацією. Такий підхід є вигідним для побудови продуктивних графічних застосунків у середовищі Windows, однак супроводжується вищою складністю реалізації та тіснішою залежністю від екосистеми Microsoft [9].

Vulkan поєднує низькорівневий характер керування з кросплатформною моделлю використання. Він дає змогу зменшити накладні витрати драйвера, підтримує багатопотокову підготовку команд і забезпечує точніший контроль над життєвим циклом ресурсів. Разом із цим Vulkan висуває високі вимоги до дисципліни проєктування та якості програмної архітектури, оскільки значна частина відповідальності за коректність роботи переноситься на сторону розробника [4–6].

Узагальнене порівняння основних характеристик сучасних графічних API наведено у додатку Д4 (табл. Д4.1).

Проведене порівняння дає підстави стверджувати, що вибір графічного API безпосередньо залежить від цілей проєкту. Для навчальних або швидких прикладних рішень може бути достатнім використання OpenGL, для високопродуктивних Windows-орієнтованих систем доцільним є Direct3D 12, тоді як для створення сучасного кросплатформного 3D-рушія з високим ступенем контролю найбільш обґрунтованим вибором є Vulkan.

Сучасний стан галузі характеризується кількома тенденціями. По-перше, зростає роль кросплатформності та уніфікації інструментів розробки. По-друге, дедалі більшого значення набувають засоби багатопотокової підготовки команд, ефективного розподілу пам’яті й точного контролю над життєвим циклом ресурсів. По-третє, поширюються підходи, пов’язані з фізично коректним відображенням матеріалів, високодеталізованою геометрією, динамічним освітленням і гнучкими механізмами інтеграції графічної частини з іншими підсистемами програмного продукту [2, 7].

Таким чином, еволюція 3D-графіки демонструє перехід від відносно простих і жорстко фіксованих моделей візуалізації до складних, програмовано керованих і модульних систем. Саме в такому контексті доцільно розглядати сучасні графічні рушії, які поєднують теоретичні напрацювання з практичними інструментами розробки інтерактивних тривимірних застосунків.

1.3 Порівняльний аналіз сучасних 3D-графічних рушіїв

Сучасний ринок програмних засобів для розробки інтерактивної 3D-графіки представлений великою кількістю рушіїв, які відрізняються архітектурою, моделлю розповсюдження, рівнем абстракції, доступними інструментами та цільовими сферами застосування. Порівняльний аналіз таких систем є необхідним для розуміння їхніх сильних сторін, обмежень і доцільності створення власного спеціалізованого рішення.

Типову організацію робочого середовища сучасного 3D-графічного рушія ілюструє скріншот редактора, на якому видно вікно сцени, панелі ієрархії об’єктів, інспектор властивостей та засоби керування ресурсами.

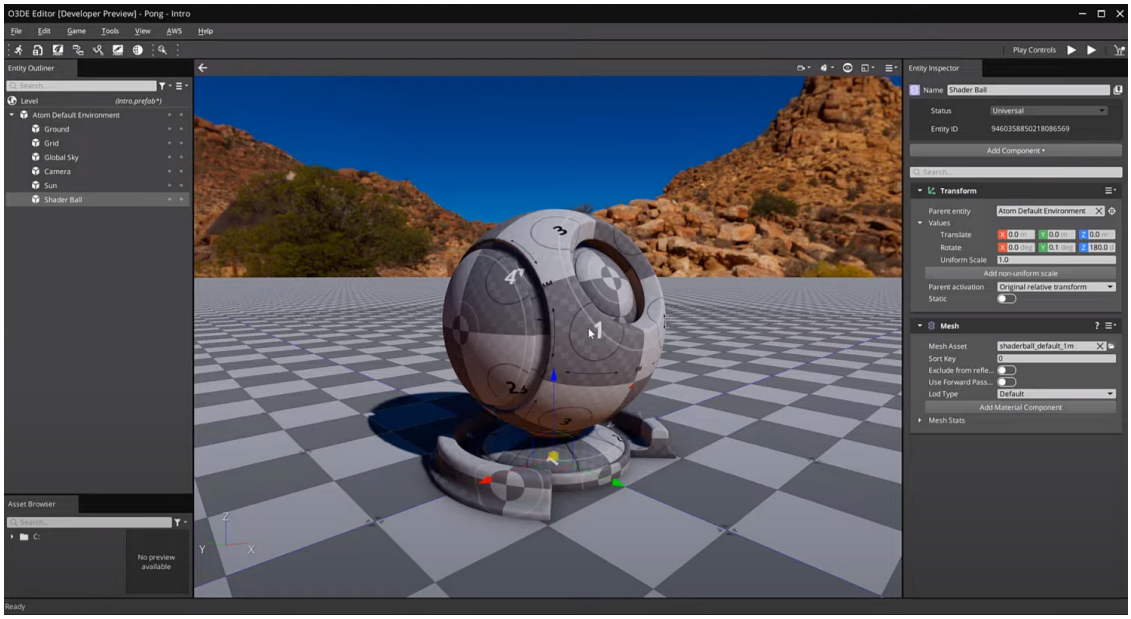


Рисунок 1.1 – Приклад інтерфейсу редактора сучасного 3D-графічного рушія

На рис. 1.1 наведено типовий вигляд середовища розробки, що полегшує подальше порівняння функціональних можливостей сучасних рушіїв.

1.3.1 Unreal Engine

Unreal Engine позиціонується як комплексний набір засобів для створення високоякісних інтерактивних продуктів у реальному часі. Рушій має розвинені інструменти для побудови віртуальних світів, створення візуальних ефектів, програмування логіки, роботи з анімацією, інтерфейсами та виробничими конвеєрами. Серед характерних рис Unreal Engine - орієнтація на високу візуальну якість, підтримка візуального програмування Blueprint та широкі можливості використання в ігрових, кінематографічних і візуалізаційних проєктах [10, 11].

Разом із цим Unreal Engine є складною багатокомпонентною системою, повноцінне опанування якої потребує значного часу. Для навчально-дослідницьких проєктів така масштабність не завжди є перевагою, оскільки

частина внутрішніх механізмів виявляється надмірно абстрагованою від безпосереднього вивчення низькорівневих аспектів рендерингу.

1.3.2 Unity

Unity є одним із найпоширеніших рушіїв для створення 2D- та 3D-застосунків на широкому спектрі платформ. Його перевагами вважаються багатоплатформність, зручний інструментарій, розвинена екосистема документації та навчальних матеріалів, а також підтримка як ігрових, так і неігрових сценаріїв використання. Unity пропонує розвинені засоби роботи зі сценою, скриптами, візуальними компонентами та оптимізацією продуктивності [12].

Unity добре підходить для швидкого створення прикладних рішень, однак у межах дослідження низькорівневих механізмів графічної підсистеми він також не завжди надає прямий доступ до всіх внутрішніх аспектів побудови рендера. Тому для задач, пов’язаних з вивченням архітектури власного рушія, Unity радше є корисним орієнтиром, ніж безпосередньою основою реалізації.

1.3.3 Godot

Godot Engine є вільним рушієм з відкритим вихідним кодом, що підтримує розробку 2D- і 3D-проєктів. До його характерних переваг належать відкрита модель розвитку, гнучка система сцен і вузлів, вбудовані засоби для 2D-графіки, підтримка 3D-рендерингу та зручність для індивідуальних розробників і невеликих команд. Додатково важливою є доступність рушія для освітніх цілей, оскільки відкритість коду полегшує дослідження внутрішньої організації системи [13].

Водночас Godot орієнтований на універсальність і широку аудиторію, а отже не завжди відповідає специфічним вимогам дослідницьких експериментів із низькорівневими графічними API. У контексті дипломного проєкту

він є цінним прикладом відкритої модульної системи, але не усуває потреби у створенні власної архітектури для глибшого контролю над рендерингом.

1.3.4 O3DE

Open 3D Engine (O3DE) є відкритим 3D-рушієм, що позиціонується як модульна, кросплатформна та спільнотно керована система для високоякісної графіки реального часу. Серед задекларованих властивостей O3DE - модульність, підтримка різних галузей застосування, орієнтація на відкритий розвиток та можливість розширення функціональності через окремі компоненти [14].

Перевагою O3DE є його архітектурна відкритість, однак через значний масштаб і багаторівневість цей рушій також може бути надто складним для задачі створення компактного навчального рішення, у якому важливо чітко простежити зв’язок між архітектурою, кодом і кінцевим результатом рендерингу.

Узагальнене порівняння розглянутих рушіїв наведено у додатку Д4 (табл. Д4.2).

Проведений аналіз дає підстави стверджувати, що сучасні рушії є потужними універсальними платформами, однак жоден із них не є повністю оптимальним для всіх класів задач. Для навчально-дослідницького проєкту доцільним є створення власного спеціалізованого 3D-рушія, оскільки це дає змогу зосередитися на вивченні архітектурних принципів, низькорівневих механізмів рендерингу та забезпечити контроль над складом і поведінкою програмної системи.

1.4 Методологічні засади побудови 3D-графічних рушіїв: класифікація, структура та основні компоненти

Методологія побудови 3D-графічного рушія базується на поєднанні загальних принципів інженерії програмного забезпечення та спеціалізованих вимог комп'ютерної графіки. Передусім такі системи можна класифікувати за сферою застосування на універсальні та спеціалізовані. Універсальні рушії орієнтовані на широкий клас задач і мають розвинений набір підсистем, тоді як спеціалізовані рішення створюються для конкретних прикладних або дослідницьких потреб і зазвичай мають компактнішу архітектуру [3].

За принципом внутрішньої організації графічні рушії можна розглядати як монолітні, модульні або компонентно-орієнтовані системи. Монолітний підхід спрощує початкову реалізацію, але ускладнює розширення та супровід. Модульна архітектура, навпаки, передбачає розподіл функціональності між відокремленими підсистемами, що зменшує зв'язність, підвищує придатність до тестування та полегшує подальшу модернізацію. Компонентний підхід особливо ефективний у задачах керування сутностями сцени, коли поведінка та стан об'єктів формуються з набору незалежних складових [3].

Базова структура сучасного 3D-графічного рушія, як правило, охоплює кілька ключових компонентів. Центральною є підсистема рендерингу, що відповідає за взаємодію з графічним API, підготовку команд, налаштування конвеєра, передавання даних у шейдери та формування кадру. Важливу роль відіграє підсистема керування сценою, яка забезпечує створення, зберігання, групування та оновлення об'єктів сцени. Необхідною також є підсистема ресурсів, що керує моделями, текстурами, матеріалами, шейдерами та іншими даними, від яких залежить процес візуалізації [2, 3].

З практичної точки зору зазначені компоненти добре ілюструє скріншот редактора сцени, на якому видно одночасно ієрархію об'єктів, область

перегляду сцени та панелі ресурсів. Така ілюстрація підкреслює, що архітектура рушія проявляється не лише на рівні програмного коду, а й у способі організації робочого простору розробника.

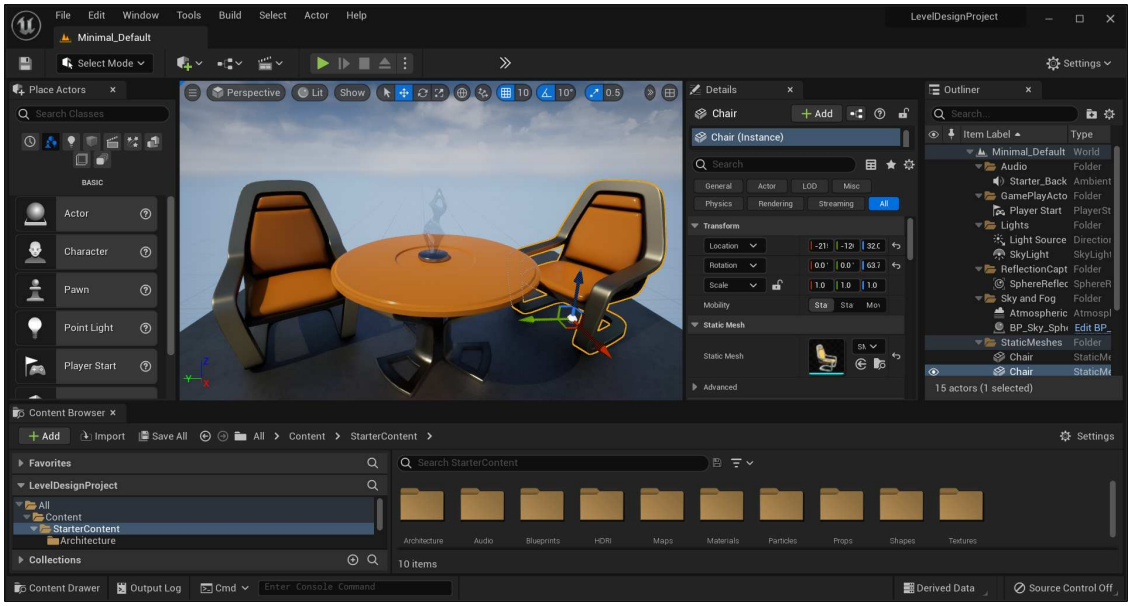


Рисунок 1.2 – Приклад організації сцени та ресурсів у редакторі 3D-рушія

На рис. 1.2 відображено взаємозв’язок між сценовими сутностями, ресурсами та візуальною областю перегляду, що допомагає пояснити структурну організацію графічного рушія.

Окрім цього, для повноцінної роботи рушія зазвичай потрібні підсистеми камер і навігації, обробки подій та вводу, логічного оновлення стану сцени, а також засоби інтеграції з користувацьким інтерфейсом. У прикладних системах важливою є також наявність редакторських можливостей, проте навіть у компактному дослідницькому рішенні необхідно забезпечити щонайменше базову взаємодію між графічною частиною та зовнішнім середовищем програми.

З позицій методології проєктування до 3D-графічного рушія висуваються такі загальні вимоги: модульність, розширюваність, повторне використання компонентів, передбачуваність життєвого циклу ресурсів, ізоля-

ція платформозалежних частин і можливість поступового нарощування функціональності. Для систем, що використовують низькорівневі графічні API, до цього переліку додаються вимоги до явного керування пам’яттю, синхронізацією та порядком виконання графічних операцій [5, 6].

Вибір архітектурного підходу суттєво впливає на якість кінцевого програмного продукту. Якщо структура рушія формується без чіткого розподілу відповідальності між компонентами, ускладнюються супровід, тестування та масштабування системи. Натомість чітке виокремлення підсистем рендерингу, сцени, ресурсів і взаємодії з користувачем створює підґрунтя для послідовного розширення можливостей програми без суттєвої переробки її основи.

Отже, методологічні засади побудови 3D-графічних рушіїв зводяться до раціонального поєднання теоретичних принципів комп’ютерної графіки, сучасних можливостей графічних API та архітектурних підходів програмної інженерії. Саме ці положення становлять основу для проєктування власного рушія, орієнтованого на реальний час, модульність, керованість і можливість подальшого розвитку.

ВИСНОВКИ ДО РОЗДІЛУ

У першому розділі було сформовано теоретичну основу дослідження. Уточнено зміст основних понять, що стосуються тривимірної графіки, рендерингу в реальному часі, графічного конвеєра, графічних API та 3D-графічних рушіїв.

Також розглянуто еволюцію технологій тривимірної графіки, виявлено тенденцію переходу до низькорівневих графічних API та виконано порівняльний аналіз сучасних 3D-рушіїв. На підставі цього обґрунтовано доцільність розробки власного спеціалізованого рішення та визначено методологічні принципи, які мають бути покладені в основу проєктування архітектури програмного рушія у наступному розділі.

РОЗДІЛ 2
МЕТОДИ, АЛГОРИТМИ ТА ТЕХНОЛОГІЇ ПОБУДОВИ ПРОГРАМНОГО 3D-ГРАФІЧНОГО РУШІЯ

У другому розділі розглядаються методи, алгоритми та технологічні рішення, що були покладені в основу побудови програмного 3D-графічного рушія. На відміну від першого розділу, де увага була зосереджена на теоретичних засадах галузі, у цьому розділі акцент зроблено на тих підходах, які безпосередньо визначають спосіб побудови системи: математичному апараті тривимірних перетворень, алгоритмах організації сцени і формування кадру, методах завантаження ресурсів, а також обґрунтуванні вибору технологій і параметрів налаштування програмних засобів [1–3].

Визначальною вимогою до розроблюваного програмного засобу є поєднання низькорівневого контролю над рендерингом засобами Vulkan з модульною архітектурою та можливістю роботи в різних режимах подання результату. Саме тому в цьому розділі методичний опис будується від математичного фундаменту до загальної архітектурної схеми, алгоритмів формування зображення та параметрів конфігурації.

2.1 Математичний фундамент 3D-графічного рушія

Побудова будь-якого 3D-графічного рушія спирається на математичний апарат лінійної алгебри, аналітичної геометрії та проєктивних перетворень. Для розроблюваного програмного засобу цей фундамент є визначальним, оскільки саме через нього описуються просторове положення об’єктів, орієнтація камери, ієрархічні перетворення, формування проєкції та узгодження даних між центральним і графічним процесорами [1, 2].

Базовими математичними об’єктами є вектори, матриці та кватерніо-

ни [15, 16]. Якщо положення точки в тривимірному просторі визначається вектором $\mathbf{p} = (x, y, z)^T$, то довжина довільного вектора $\mathbf{v} = (x, y, z)^T$ визначається виразом

$$\|\mathbf{v}\| = \sqrt{x^2 + y^2 + z^2}. \tag{1}$$

Нормований вектор обчислюється за формулою

$$\widehat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}. \tag{2}$$

Скалярний добуток двох векторів використовується для оцінювання кута між ними та визначається як

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z, \tag{3}$$

тоді як векторний добуток задає ортогональний напрямок і має вигляд

$$\mathbf{a} \times \mathbf{b} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}. \tag{4}$$

Саме ці операції використовуються при побудові систем координат камери, визначенні осей руху, розрахунку напрямків спостереження та формуванні ортонормованого базису для перетворень у просторі.

Для поєднання лінійних перетворень із перенесенням у 3D-графіці застосовуються однорідні координати. У цьому випадку точка подається у вигляді

$$\widetilde{\mathbf{p}} = (x, y, z, 1)^T, \tag{5}$$

що дає змогу описувати перенесення, масштабування та обертання уніфіковано через матричне множення. Наприклад, матриця перенесення має вигляд

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6}$$

а матриця масштабування -

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7}$$

У 3D-графічному рушії обертання об’єктів доцільно описувати кватерніонами, оскільки вони дають змогу уникати явища ґімбальної сингулярності та забезпечують стабільну композицію поворотів. Кватерніон повороту навколо нормованої осі $\widehat{\mathbf{u}}$ на кут φ можна подати як

$$q = \left(\cos\frac{\varphi}{2},\ \widehat{\mathbf{u}}\sin\frac{\varphi}{2}\right). \tag{8}$$

Локальна матриця перетворення об’єкта формується композицією перенесення, обертання і масштабування:

$$\mathbf{M}_{local} = \mathbf{T}\,\mathbf{R}(q)\,\mathbf{S}. \tag{9}$$

Якщо об’єкт належить ієрархії сцени, то його світова матриця визначається рекурсивно:

$$\mathbf{M}_{world}^{(i)} = \mathbf{M}_{world}^{(parent)} \cdot \mathbf{M}_{local}^{(i)}. \tag{10}$$

Це співвідношення є принципово важливим для опису ієрархічних сцен, оскільки дає змогу узгоджувати просторові перетворення між вузлами імпортованих моделей формату glTF.

Камера у 3D-рушії задає перехід від світового простору до простору спостереження. Якщо матриця положення камери позначена як $\mathbf{M}_{camera}$, то видова матриця обчислюється як

$$\mathbf{V} = \mathbf{M}_{camera}^{-1}. \tag{11}$$

Після переходу до простору камери виконується проєкція у кліп-простір. Для перспективної проєкції, характерної для інтерактивного 3D-відображення, використовується матриця типу

$$\mathbf{P} = \begin{bmatrix} \frac{1}{a\tan(\alpha/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\alpha/2)} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & -\frac{fn}{f-n} & 0 \end{bmatrix}, \tag{12}$$

де α - вертикальний кут огляду, a - співвідношення сторін області візуалізації, n - ближня площина відсікання, f - дальня площина відсікання. У контексті Vulkan важливим є те, що після проєкції глибина інтерпретується у діапазоні $[0; 1]$, що необхідно враховувати при узгодженні математичних перетворень із графічним API [5, 6].

Повне перетворення вершини від локального простору об'єкта до кліп-простору задається співвідношенням

$$\widetilde{\mathbf{p}}_{clip} = \mathbf{P}\,\mathbf{V}\,\mathbf{M}_{world}\,\widetilde{\mathbf{p}}_{local}. \tag{13}$$

Нормалі, необхідні для коректної інтерпретації орієнтації поверхні, у загальному випадку повинні перетворюватися через нормальну матрицю

$$\mathbf{n}' = \text{normalize}\left((\mathbf{M}_{world}^{-1})^T\mathbf{n}\right). \tag{14}$$

У системі навігації камери також використовується дискретно-неперервна модель руху. Якщо **f**, **r** і **u** - вектори напрямку вперед, вправо і вгору відповідно, то оновлення позиції камери за час Δt можна подати як

$$\mathbf{p}_{t+\Delta t} = \mathbf{p}_t + (\lambda_f \mathbf{f} + \lambda_r \mathbf{r} + \lambda_u \mathbf{u}) v \Delta t, \tag{15}$$

де v - базова швидкість руху, а коефіцієнти λ_f, λ_r, λ_u визначаються поточним станом вводу. Орієнтація камери оновлюється композицією малих поворотів:

$$q_{t+\Delta t} = \text{normalize}(q_{\Delta yaw} q_{\Delta pitch} q_t). \tag{16}$$

Сукупність наведених перетворень утворює математичний каркас рушія і визначає послідовність переходу між просторовими системами координат. Для наочності цей ланцюг наведено на рис. 2.1.

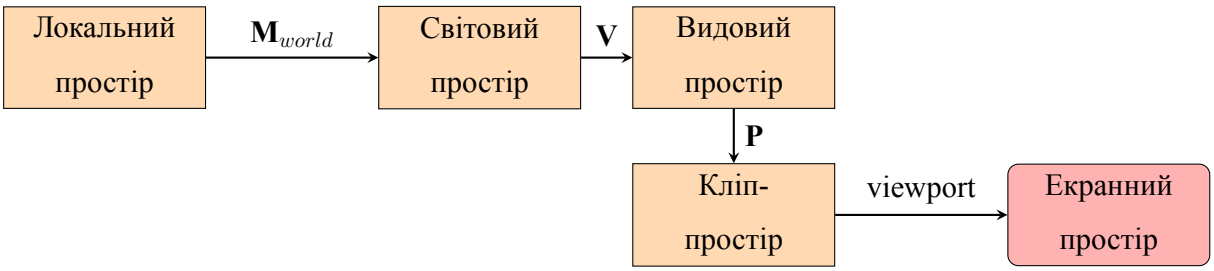


Рисунок 2.1 – Ланцюг просторових перетворень у 3D-графічному рушії

Отже, математичний фундамент розроблюваного 3D-графічного рушія охоплює векторно-матричний апарат, однорідні координати, ієрархічні перетворення, перспективну проєкцію та кватерніонне представлення орієнтації. Саме цей апарат є основою для побудови алгоритмів сцени, камери, завантаження моделей і формування кадру.

2.2 Архітектурні методи побудови рушія та організація підсистем

Для дослідницького 3D-рушія важливим є не лише вибір графічного API, а й правильна архітектурна декомпозиція системи. Доцільним є модульний підхід, за якого ядро рендерингу відокремлюється від засобів відображення результату, вводу користувача та інтерфейсу. Такий підхід зменшує зв’язність між підсистемами, спрощує розширення функціональності та дає можливість використовувати спільне ядро в різних режимах роботи [3].

Методично архітектуру системи можна подати як поєднання трьох рівнів. Перший рівень становить базове ядро рушія, що відповідає за контекст Vulkan, ресурси, сцену, камеру, пам’ять і алгоритм формування кадру. Другий рівень охоплює засоби подання кадру у зовнішнє середовище, тобто механізми роботи з цільовими поверхнями рендерингу. Третій рівень утворюють модулі взаємодії з користувачем і середовищем виконання, які визначають спосіб відображення результату та обробки подій.

Принципово важливим є відокремлення поняття «що саме рендериться» від поняття «куди саме рендериться». Завдяки цьому ядро оперує сценою, камерами, ресурсами та графічним конвеєром, тоді як конкретний механізм виведення кадру може бути реалізований або через інтегровану підсистему відображення, або через окрему віконну поверхню. Така організація є методично доцільною, оскільки робить архітектуру стабільною щодо заміни засобів представлення результату.

Узагальнену структуру рушія наведено на рис. 2.2.

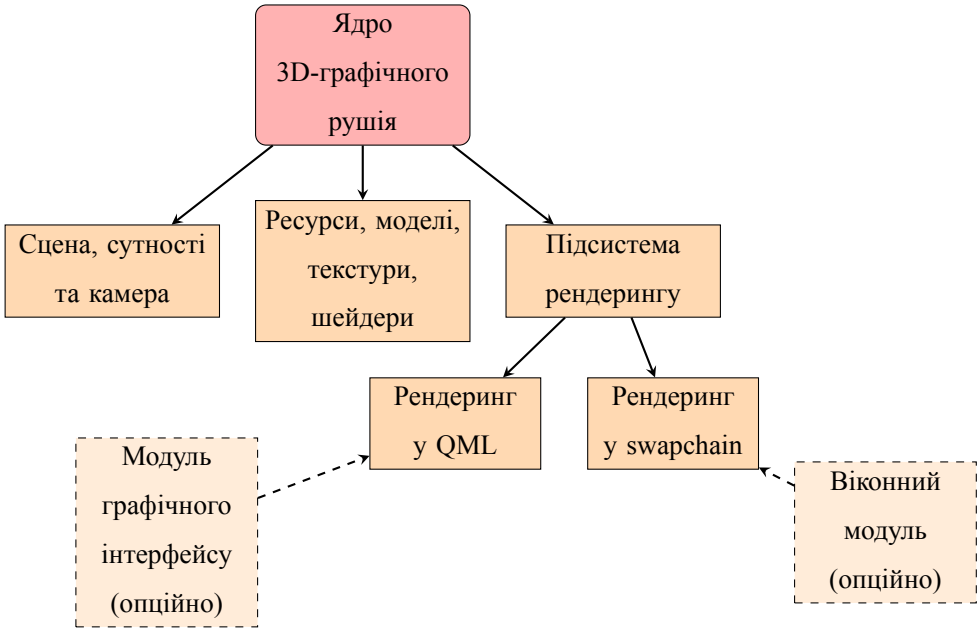


Рисунок 2.2 – Узагальнена архітектурна схема 3D-графічного рушія

У структурі сцени доцільно використовувати ієрархічне подання сутностей із компонентним наповненням. Такий підхід поєднує дві важливі властивості: з одного боку, він дозволяє виражати вкладені просторові перетворення через батьківсько-дочірні зв'язки; з іншого - забезпечує гнучке приєднання до сутностей окремих функціональних характеристик, таких як трансформація, камера, геометрія чи контролер руху. Для дослідницького рушія це є суттєвою перевагою, оскільки дозволяє розширювати систему без перебудови її основного каркаса.

Окремого обґрунтування потребує застосування кількох способів представлення кадру. У першому випадку доцільно формувати зображення у позаекранну ціль рендерингу з подальшою інтеграцією результату в інтерфейс користувача. У другому випадку доцільним є класичний спосіб відображення через віконну поверхню. Методична цінність такого розділення полягає в тому, що воно дає змогу використовувати однакове ядро для різних сценаріїв роботи без дублювання логіки побудови сцени та рендерингу.

Таким чином, архітектурна методологія розроблюваного рушія ґрун-

тується на відокремленні ядра від зовнішніх модулів взаємодії, поділі системи на самостійні підсистеми та уніфікації інтерфейсу цільового рендерингу. Це створює основу для використання спільних алгоритмів формування кадру в різних режимах роботи системи.

2.3 Методи та алгоритми формування кадру на основі Vulkan

Побудова підсистеми рендерингу базується на використанні Vulkan 1.3 як низькорівневого кросплатформного API, що забезпечує явне керування ресурсами, синхронізацією та чергами виконання [4–6, 17, 18]. Методично це означає, що алгоритм побудови кадру не приховується за високорівневими абстракціями драйвера, а визначається розробником у вигляді чіткої послідовності кроків.

Для досліджуваного рушія доцільно використовувати підхід, за якого весь цикл формування кадру розглядається як керований конвеєр із кількох етапів: підготовки даних сцени, оновлення перекадрових ресурсів, переведення зображень у потрібні стани, запуску графічного конвеєра, виконання draw-команд та завершального переведення цільового зображення в кінцевий макет. Такий підхід забезпечує передбачуваність виконання, що особливо важливо для Vulkan.

Метод формування кадру спирається на використання динамічного рендерингу [19, 20]. Це означає відмову від жорсткого попереднього опису render pass та framebuffer на користь гнучкішої схеми, коли кольорові та глибинні вкладення описуються безпосередньо під час початку рендерингу. Для дослідницького проєкту цей підхід є раціональним, оскільки зменшує кількість допоміжних структур, спрощує організацію рендерингу та робить алгоритм побудови кадру прозорішим [21, 22].

Побудову кадру доцільно описувати такою послідовністю дій:

а) вибір і підготовка поточної цілі рендерингу;

б) отримання або підготовка командного буфера поточного кадру;

в) оновлення буфера з матрицями камери та параметрами кадру;

г) переведення кольорового і глибинного зображень у стани, придатні для запису;

д) початок динамічного рендерингу з кольоровим та глибинним вкладеннями;

е) прив’язка графічного конвеєра, дескрипторів, області виведення і області відсікання;

ж) передавання матриці моделі та перекадрових параметрів через push-константи;

и) виконання індексованого малювання геометрії;

к) завершення рендерингу й переведення зображення у кінцевий макет;

л) передавання готового кадру до підсистеми відображення.

На рис. 2.3 наведено узагальнену блок-схему цього алгоритму.

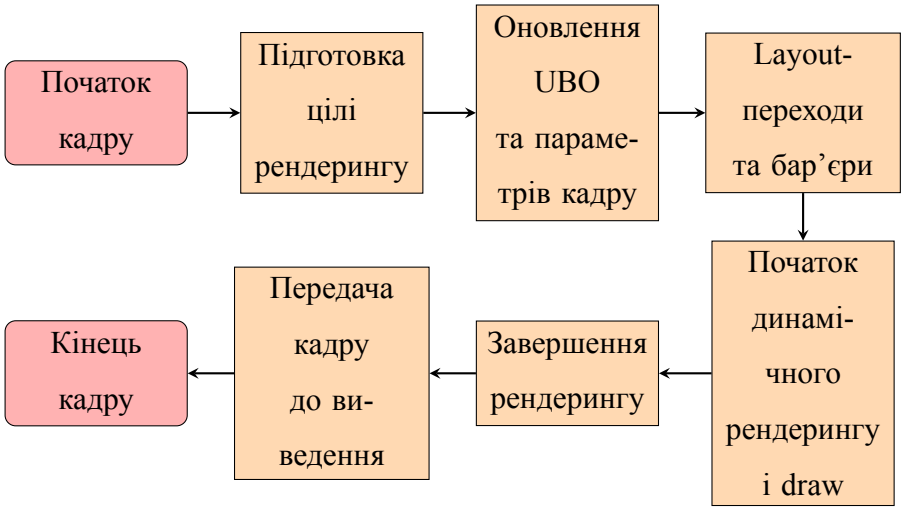


Рисунок 2.3 – Узагальнений алгоритм формування кадру на основі Vulkan

Важливим методичним рішенням є абстрагування цілі рендерингу. З позицій алгоритміки це означає, що ядро рендерингу працює не з конкретним

вікном чи конкретною поверхнею, а з абстрактною ціллю, яка забезпечує: доступ до поточного зображення, доступ до глибинного буфера, командного буфера, дескрипторів перекадрових ресурсів та механізму завершення кадру. Такий підхід дозволяє використовувати єдиний алгоритм рендерингу як для позаекранного зображення, що інтегрується в графічний інтерфейс, так і для класичного віконного відображення.

Для інтеграції результату в графічний інтерфейс доцільним є метод формування кадру в позаекранне зображення з подальшою передачею результату зовнішній підсистемі відображення. Перевага цього підходу полягає в тому, що область візуалізації стає елементом складнішого інтерфейсного середовища без відмови від повноцінного рендерингу на Vulkan. Для безпосереднього виведення у вікно, навпаки, доцільно використовувати алгоритм відображення через swapchain із підтримкою кількох кадрів у польоті, повторного створення ресурсів при зміні розміру вікна та класичної схеми acquire–submit–present.

Із практичної точки зору суттєвими є параметри конфігурації графічної підсистеми, наведені у додатку Д4 (табл. Д4.3). Саме вони визначають робочі характеристики системи та відображають обрані проектні компроміси.

Метод формування кадру передбачає також поділ даних на перекадрові та пооб’єктні. До перекадрових належать передусім матриці виду і проєкції, які залежать від активної камери. До пооб’єктних належить матриця моделі та інші локальні параметри візуалізації. Такий поділ мінімізує дублювання даних і добре узгоджується з моделлю Vulkan, у якій різні типи даних доцільно передавати через різні механізми зв’язування ресурсів.

Отже, алгоритмічна організація рендерингу базується на явному керуванні станами зображень, динамічному рендерингу, абстрагуванні цілі рендерингу та поділі даних на перекадрові й пооб’єктні. Саме такий підхід

робить систему придатною як для дослідницьких експериментів, так і для подальшого функціонального розширення.

2.4 Методи представлення сцени, керування об’єктами та навігації камерою

Побудова сцени в 3D-рушії вимагає вибору такого способу представлення даних, який одночасно забезпечував би просторову узгодженість об’єктів, можливість ієрархічного групування та гнучке розширення властивостей сутностей. Доцільним є поєднання ієрархії сутностей із компонентним підходом до опису поведінки та стану. Це означає, що сцена трактується як сукупність сутностей, кожна з яких має ідентичність, місце у дереві сцени та набір незалежних компонентів.

Такий підхід є методично виправданим з кількох причин. По-перше, він дозволяє природно описувати складені об’єкти, для яких локальна трансформація однієї сутності залежить від батьківської. По-друге, він уможливлює приєднання до сутності лише тих властивостей, які їй потрібні: наприклад, трансформації, геометрії, камери або контролера навігації. По-третє, компонентний підхід полегшує подальше розширення рушія без змін у базовій моделі сцени.

Центральним компонентом просторової організації є трансформація, яка визначає позицію, орієнтацію та масштаб об’єкта. Для камер, джерел геометрії та інших сценових сутностей саме трансформація є основою обчислення світової матриці та переходу між системами координат. Активна камера в поточній сцені задає матриці виду і проєкції, які використовуються всією підсистемою рендерингу.

У розроблюваному програмному засобі доцільно застосовувати метод «літаючої камери», за якого користувач може вільно переміщатися у сцені, змінюючи положення та напрям спостереження. Алгоритм навігації спирає-

ться на три основні ідеї:

а) окреме накопичення стану натиснутих клавіш і вказівникових подій;

б) розкладання руху на компоненти вздовж локальних осей камери;

в) оновлення орієнтації через композицію малих поворотів, а не через пряме накопичення кутів Ейлера.

Саме така схема добре узгоджується з використанням кватерніонів і дозволяє забезпечити плавну навігацію без геометричних артефактів, пов’язаних із виродженням осей обертання.

Для узгодженого функціонування сцени та користувацької взаємодії доцільно використовувати подієву модель, за якої введення не змінює стан візуалізації безпосередньо, а породжує події, що обробляються у відповідних підсистемах. Методична перевага подієвої моделі полягає в послабленні зв’язку між джерелом події та отримувачем, що є особливо важливим у модульних системах. У результаті система вводу, камера, сцена та графічний інтерфейс можуть взаємодіяти без жорсткої залежності один від одного.

Основні параметри конфігурації навігації та камери, доцільні для інтерактивного дослідницького 3D-застосунку, наведено у додатку Д4 (табл. Д4.4).

З точки зору методів оновлення стану сцени важливо, що система розрізняє логічне оновлення, оновлення просторового стану та етап візуалізації. Це дозволяє відокремити зміни внутрішнього стану від власне побудови кадру й створює умови для подальшого впровадження більш складних систем моделювання, анімації чи фізики.

Таким чином, представлення сцени базується на поєднанні ієрархічної моделі сутностей, компонентного опису властивостей, подієвого опрацювання взаємодії та кватерніонної навігації камери. Сукупність цих методів забезпечує достатню гнучкість і керованість системи при відносно компактній

архітектурі.

2.5 Методи завантаження моделей, текстур і керування графічними ресурсами

Ефективність 3D-рушія значною мірою залежить від того, наскільки раціонально організовано роботу з ресурсами. Для дослідницького рушія важливо не лише отримати можливість відобразити модель, а й забезпечити узгоджене перетворення зовнішніх даних у внутрішнє подання, придатне для рендерингу. Доцільно використовувати модель роботи з ресурсами, що включає імпорт геометрії, інтерпретацію матеріалів, підготовку текстур і завантаження шейдерних програм.

Для опису 3D-даних обрано формат glTF 2.0, який є сучасним відкритим стандартом передачі тривимірних сцен і моделей [23]. Його перевагами є чітка структура вузлів, можливість опису геометрії, матеріалів і текстур у єдиній моделі даних, а також придатність до передачі вже підготовлених для GPU ресурсів. Для задач побудови рушія це є суттєвою перевагою, оскільки glTF дозволяє зберегти ієрархію об’єктів, локальні перетворення вузлів та зв’язок мешів із матеріалами.

Метод імпорту моделі доцільно описувати як послідовність таких кроків:

а) зчитування файлу glTF або GLB та побудова внутрішнього дерева вузлів;

б) обчислення локальної матриці кожного вузла на основі перенесення, обертання та масштабування або безпосередньо заданої матриці;

в) рекурсивне накопичення світових матриць уздовж дерева сцени;

г) виділення геометрії примітивів у вигляді списків вершин і індексів;

д) зчитування атрибутів вершин: позицій, нормалей, текстурних коор-

динат;

е) витягнення параметрів матеріалу та пов’язаної текстури базового кольору;

ж) створення внутрішнього подання вузлів моделі, кожен із яких містить геометрію, світову матрицю і матеріал.

Особливу роль відіграє рекурсивний обхід дерева вузлів, оскільки саме він дозволяє узгодити просторову структуру імпортованої сцени з внутрішньою сценою рушія. При цьому кожний вузол можна розглядати як незалежну рендеровану сутність із власною геометрією та матеріалом, але з успадкованою від предків просторовою трансформацією.

Метод роботи з матеріалами повинен враховувати як наявність текстури, так і можливість її відсутності. З практичної точки зору доцільним є використання запасного сценарію, коли за відсутності текстури формується однопіксельне зображення на основі базового кольору матеріалу. Такий підхід підвищує стійкість системи до неповних або спрощених моделей і дозволяє уникати збоїв у ланцюгу візуалізації.

Після імпорту даних на стороні центрального процесора виконується етап підготовки ресурсів для графічного процесора. На цьому етапі застосовуються такі методи:

– формування буферів вершин та індексів для геометрії;
– створення текстурних зображень і відповідних представлень образів;
– розподіл даних за дескрипторними наборами відповідно до частоти їх оновлення;
– розмежування перекадрових параметрів, параметрів матеріалу та пооб’єктних трансформацій.

Важливим є і метод організації шейдерних ресурсів. Доцільно використовувати один уніфікований шейдерний модуль, який містить окремі точки

входу для вершинного та фрагментного етапів. Це спрощує супровід шейдерів, зменшує ризик розбіжності між стадіями та полегшує подальше розширення матеріальної моделі.

Послідовність підготовки ресурсів узагальнено на схемі, наведеній на рис. 2.4.

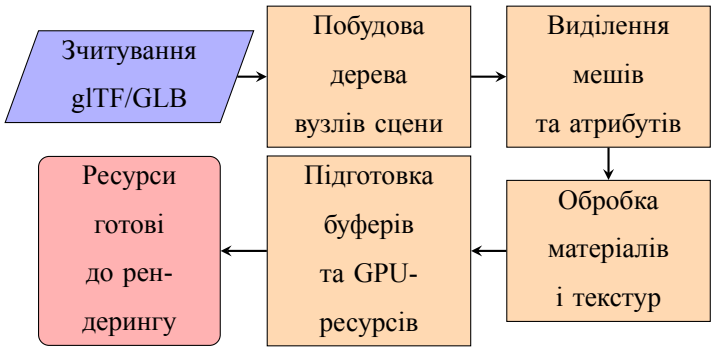


Рисунок 2.4 – Схема підготовки графічних ресурсів до рендерингу

Отже, методи роботи з ресурсами базуються на використанні формату glTF 2.0, рекурсивному обході вузлів сцени, уніфікованому поданні геометрії та матеріалів, а також поділі даних за характером їх оновлення. Це створює основу для керованого та відтворюваного процесу підготовки ресурсів до рендерингу.

2.6 Обґрунтування вибору технологій і параметри налаштування програмних засобів

Вибір технологій повинен оцінюватися не лише з позицій популярності чи зручності розробки, а насамперед з позицій відповідності цільовій задачі: побудові компактного, але архітектурно прозорого 3D-рушія з явним контролем над рендерингом. Саме тому доцільно поєднати низькорівневі засоби доступу до графічного процесора з бібліотеками, що забезпечують математичну підтримку, завантаження ресурсів, інтеграцію з інтерфейсом та організацію збірки [23–29].

Мова C++ є доцільним вибором для такого проєкту завдяки високій продуктивності, підтримці ресурсного керування через RAII, можливості виразно описувати системне програмування та ефективно взаємодіяти з Vulkan. Використання сучасного стандарту C++23 додатково покращує засоби модульної організації коду, типобезпечності та узагальненого програмування.

Vulkan 1.3 обрано як базовий графічний API, оскільки він відповідає дослідницькій меті розробки: надає явний контроль над пам'яттю, синхронізацією, командами та графічним конвеєром. Саме це дозволяє будувати рушій як керовану систему, а не як набір непрозорих драйверних операцій [4–6].

Для побудови графічного інтерфейсу раціональним є використання Qt 6 і QML, оскільки вони забезпечують сучасні засоби створення інтерфейсу, роботу з віконною системою та зручну інтеграцію області візуалізації у складене прикладне середовище [24]. Для організації окремого віконного режиму доцільним є GLFW, який забезпечує компактний і добре контрольований механізм створення вікна, обробки подій та взаємодії з платформними поверхнями Vulkan [25].

Математична бібліотека GLM є методично обґрунтованим вибором для роботи з векторами, матрицями та кватерніонами, оскільки її інтерфейс наближений до звичних для комп'ютерної графіки математичних конструкцій, а сама бібліотека добре узгоджується з практикою використання у Vulkan-проєктах [26].

Для роботи з 3D-моделями доцільним є використання TinyGLTF як компактного інструмента читання формату glTF 2.0. Такий вибір відповідає завданню створення дослідницького рушія, оскільки дозволяє зосередитися на власних алгоритмах підготовки та відображення ресурсів, не перевантажуючи проєкт зайвими зовнішніми абстракціями [23, 29].

Для шейдерів доцільно використовувати Slang, оскільки цей інстру-

ментарій дозволяє підтримувати сучасний стиль організації шейдерного коду, зберігати узгодженість між різними точками входу та компілювати результати у SPIR-V, придатний для використання у Vulkan [27, 30, 31]. Для збірки проєкту природним вибором є CMake, який дає змогу описувати багатомодульну структуру системи, незалежно керувати цілями збірки та автоматизувати компіляцію шейдерів і копіювання ресурсів [28].

Узагальнене обґрунтування вибору технологій наведено у додатку Д4 (табл. Д4.5).

Окрім вибору технологій, суттєвими є й параметри конфігурації програмних засобів. До таких параметрів належать, зокрема, версія Vulkan, наявність валідаційних шарів під час налагодження, компіляція шейдерів у SPIR-V, вибір формату кадру, параметри камери, конфігурація графічного інтерфейсу або віконного режиму та налаштування кількості кадрів у польоті. Саме ці параметри визначають не лише функціональність, а й зручність налагодження, відтворюваність експериментів та стабільність роботи системи [32–34].

Таким чином, вибір технологій є не випадковим набором інструментів, а результатом послідовного узгодження вимог до графічного API, інтерфейсної інтеграції, математичної підтримки, завантаження ресурсів і шейдерного інструментарію. Це забезпечує відповідність програмного засобу його дослідницькому призначенню та створює основу для подальшого практичного розвитку системи.

ВИСНОВКИ ДО РОЗДІЛУ

У другому розділі було розглянуто методи, алгоритми та технологічні рішення, що становлять основу побудови програмного 3D-графічного рушія. Насамперед сформовано математичний фундамент системи, який охоплює векторно-матричний апарат, однорідні координати, перспективну проєкцію, ієрархічні перетворення та кватерніонне подання орієнтації.

Також було обґрунтовано архітектурний підхід до побудови рушія як модульної системи з відокремленим ядром рендерингу, описано алгоритм формування кадру на основі Vulkan, методи представлення сцени та навігації камерою, а також підходи до імпорту й підготовки графічних ресурсів. Окремо виконано обґрунтування вибору технологій, що використовуються у проєкті, та визначено основні параметри конфігурації програмних засобів.

Отже, у цьому розділі сформовано методологічну основу практичної побудови 3D-рушія, яка поєднує математичну коректність, архітектурну впорядкованість і технологічну доцільність. Саме ці положення створюють підґрунтя для подальшого опису результатів проєктування, реалізації та апробації програмного засобу у наступних частинах дипломної роботи.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ 3D-ГРАФІЧНОГО РУШІЯ НА ОСНОВІ VULKAN

У третьому розділі подано опис програмної реалізації розробленого 3D-графічного рушія, побудованого на основі Vulkan API. Якщо у попередньому розділі увага була зосереджена на методах, алгоритмах і технологічних принципах побудови системи, то в цьому розділі акцент переноситься на практичне втілення обраних рішень у структурі програмного продукту, організації основних модулів, моделі представлення даних і способі взаємодії між підсистемами.

Особливістю розробленого програмного засобу є поєднання низькорівневого графічного ядра з модульною архітектурою, що допускає використання одного й того самого рушія в різних режимах відображення результату. Практична реалізація охоплює ядро рендерингу, систему представлення сцени, засоби завантаження графічних ресурсів, обробку вводу користувача, подієву взаємодію між частинами системи, а також два режими виконання: автономний віконний режим і режим інтеграції у графічний інтерфейс, побудований на базі Qt та QML.

Зміст цього розділу побудовано від загальної архітектури до конкретних підсистем. Спочатку розглядається структура програмного продукту та принципи організації його модулів, далі описуються ядро рушія та життєвий цикл застосунку, модель сцени і внутрішніх даних, підсистема рендерингу на основі Vulkan, механізми роботи з ресурсами, вводом і подіями, а також особливості реалізації двох режимів використання розробленого програмного засобу. Завершується розділ стислим аналізом результатів реалізації та оцінкою поточного функціонального стану системи.

3.1 Загальна архітектура програмної системи

Програмна реалізація 3D-графічного рушія побудована як багатомодульна система, у якій ядро рендерингу відокремлено від платформозалежних механізмів запуску та відображення результату. Такий підхід відповідає методологічним положенням, сформульованим у другому розділі, і дає змогу використовувати спільну обчислювальну та графічну основу як у самостійному віконному застосунку, так і в редакторському режимі з інтеграцією у графічний інтерфейс.

На практичному рівні архітектура системи організована навколо кількох основних модулів. Центральне місце посідає модуль `engine`, який містить ядро рушія, сценову модель, підсистему рендерингу, засоби роботи з ресурсами, камерами, вводом і подіями. Модуль `runtime` забезпечує автономний режим запуску на базі GLFW та класичної схеми виведення кадру через swapchain. Модуль `editor` відповідає за інтеграцію рушія з Qt/QML-інтерфейсом і реалізує режим, у якому кадр формується поза екраном, а потім передається у графічне середовище редактора. Окрім цього, у структурі програмного продукту наявні модулі `logging` і `utils`, що виконують допоміжні функції журналювання, читання файлів і супровідних сервісних операцій.

З погляду програмної інженерії таке розділення є доцільним з кількох причин. По-перше, воно зменшує зв’язність між ядром та інтерфейсним оточенням. По-друге, воно дозволяє локалізувати платформозалежний код у відносно невеликих модулях. По-третє, воно створює умови для подальшого розширення системи, наприклад, додавання інших способів представлення кадру або інших типів застосунків, що використовують те саме графічне ядро.

Узагальнену структуру програмного продукту наведено на рис. 3.1, де окремо відображено ядро рушія, модулі запуску та допоміжні підсистеми.

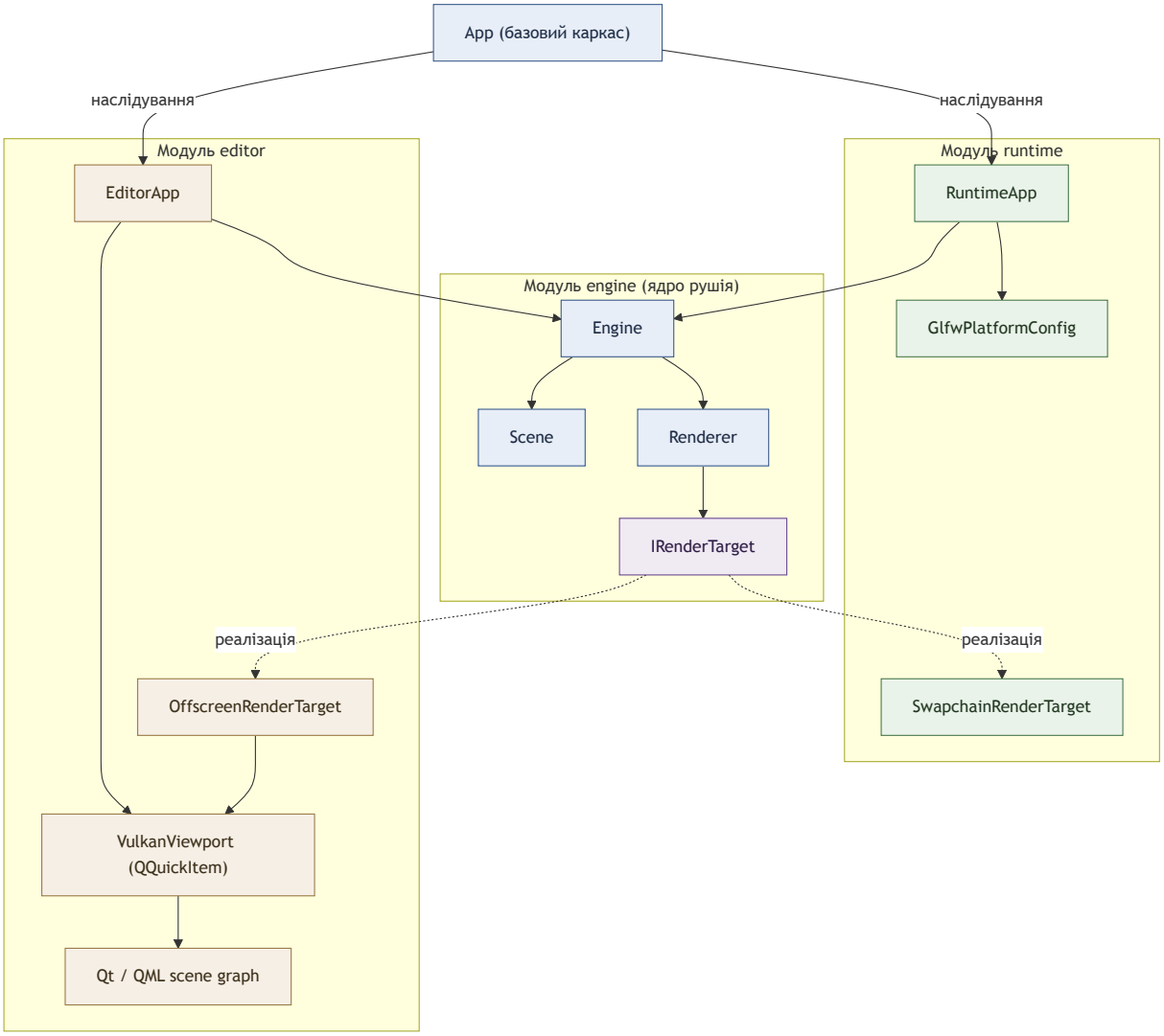


Рисунок 3.1 – Структурна схема програмного 3D-графічного рушія

Роль основних модулів у загальній архітектурі узагальнено в табл. 3.1.

Таблиця 3.1 – Основні модулі програмного продукту та їх призначення

Модуль	Основне призначення	Характерні складові
`engine`	Ядро рушія та внутрішня логіка системи	`Engine`, `Scene`, `Renderer`, `InputSystem`, `EventBus`, `GltfLoader`
`runtime`	Автономний режим виконання у вікні	`RuntimeApp`, GLFW-ініціалізація, swapchain-ціль рендерингу
`editor`	Інтеграція рушія з Qt/QML-інтерфейсом	`EditorApp`, `VulkanViewport`, offscreen-рендеринг, імпорт зображення у Qt
`logging`	Журналювання та діагностика	Налаштування кольорового логування та категорій повідомлень
`utils`	Допоміжні операції	Робота з файлами, шейдерними даними та технічними сервісами

Важливим архітектурним рішенням є використання базового прикладного класу `App`, від якого наслідуються конкретні режими виконання. Така побудова забезпечує спільний каркас ініціалізації, оновлення та завершення роботи, але водночас не обмежує систему одним способом взаємодії з користувачем. У результаті ядро не залежить безпосередньо ні від GLFW, ні від Qt/QML, а специфіка конкретного середовища локалізується у класах `RuntimeApp`, `EditorApp` та відповідних конфігураційних об’єктах.

Не менш суттєвим є відокремлення логіки рендерингу від логіки представлення кадру. У коді ця ідея реалізується через інтерфейс `IRenderTarget`, який задає єдину модель роботи з поточним кадром: початок кадру, завершення кадру, доступ до кольорового та глибинного зображень, командного буфера, дескрипторів і перекадрових ресурсів. Завдяки цьому клас `Renderer` не прив’язується до конкретної моделі виведення результату і може використовувати однакову процедуру побудови кадру і для swapchain-виведення, і

для позаекранного рендерингу.

Архітектурну логіку системи можна охарактеризувати як поєднання трьох рівнів. Перший рівень утворює ядро рушія, у якому зосереджено сцену, контекст Vulkan, рендерер, систему ресурсів, ввід і події. Другий рівень становлять механізми подання кадру, тобто різні реалізації цілі рендерингу. Третій рівень охоплює зовнішні модулі запуску та користувацької взаємодії. Саме така побудова дозволяє поєднати низькорівневий характер Vulkan із вимогою архітектурної впорядкованості та можливістю еволюційного розвитку системи.

3.2 Програмна реалізація ядра рушія та життєвого циклу застосунку

Центральним елементом програмної реалізації є клас `Engine`, який акумулює основні сервіси рушія та координує їхню спільну роботу. Саме через цей клас виконується ініціалізація контексту Vulkan, створення цілі рендерингу, керування сценами, оновлення внутрішнього стану та передавання підготовлених даних у підсистему рендерингу. На відміну від суто платформозалежного коду, `Engine` орієнтований на незалежність від конкретного середовища виконання.

Зовнішній прикладний каркас представлено класом `App`. У його відповідальність входять базові операції життєвого циклу: запуск ініціалізації, обчислення часу між кадрами, виклик оновлення рушія, обробка подій зміни розміру області рендерингу та завершення роботи. Практично це означає, що `App` задає загальний шаблон виконання, а конкретні режими запуску лише постачають платформну конфігурацію та, за потреби, власну організацію циклу подій.

Узагальнена схема запуску системи має такий вигляд. На першому етапі прикладний модуль створює відповідний об’єкт конфігурації платфор-

ми. На другому етапі клас `Engine` використовує цю конфігурацію для побудови контексту Vulkan та створення конкретної реалізації `IRenderTarget`. На третьому етапі створюється керівник пам'яті, ініціалізується рендерер і виконується початкове завантаження демонстраційної сцени. Після цього система переходить у режим регулярного оновлення, у межах якого сцена оновлюється відповідно до поточного часу кадру, а рендерер формує і відправляє новий кадр.

Суттєвим конструктивним рішенням є використання абстрактного класу `PlatformConfig`. Цей клас виконує роль інтерфейсу між універсальним ядром рушія та конкретним режимом його використання. Через нього задається спосіб побудови об'єкта `CoreCtx`, спосіб створення `IRenderTarget`, а також платформа-залежна реакція на повторне створення ресурсів, наприклад, після зміни розміру вікна. З інженерного погляду це дозволяє уникнути проникнення логіки GLFW або QML безпосередньо в ядро.

Після успішної ініціалізації рушія виконується завантаження початкової сцени. Для цього створюється об'єкт сцени, формується стандартна камера, налаштовуються її параметри проєкції, а за наявності шляху до моделі виконується імпорт зовнішніх ресурсів. Такий сценарій є достатнім для запуску системи в демонстраційному режимі й водночас забезпечує базове тестове наповнення середовища.

Цикл обробки кадру організований через метод `tick()`. На кожній ітерації обчислюється часовий крок Δt, після чого оновлюється внутрішній часовий стан рушія: зберігається поточний номер кадру, накопичується загальний час виконання і, за потреби, виконується логічне оновлення активної сцени. Після цього формується представлення кадру у вигляді структури `FrameView`, яка містить матриці камери, час і перелік об'єктів, що мають бути відрендерені. Саме ця структура надалі передається в `Renderer`.

У системі передбачено також механізм тимчасового призупинення ло-

гічного оновлення сцени. Для цього в класі `Engine` реалізовано керування ознакою паузи та режим покадрового кроку. Таке рішення не є критичним для базового рендерингу, але є корисним для редакторського режиму, налагодження і подальшого розвитку рушія як дослідницької платформи.

Завершення роботи рушія побудовано з урахуванням вимог Vulkan до коректного звільнення ресурсів. Перед очищенням внутрішніх структур здійснюється очікування завершення роботи пристрою, після чого послідовно знищуються ресурси рендерера, сцени, дескрипторні набори матеріалів, текстури, меші, ціль рендерингу, керівник пам’яті та сам контекст. Така послідовність зменшує ризик доступу до вже звільнених об’єктів і забезпечує впорядкований життєвий цикл системи.

Життєвий цикл застосунку наведено на рис. 3.2.

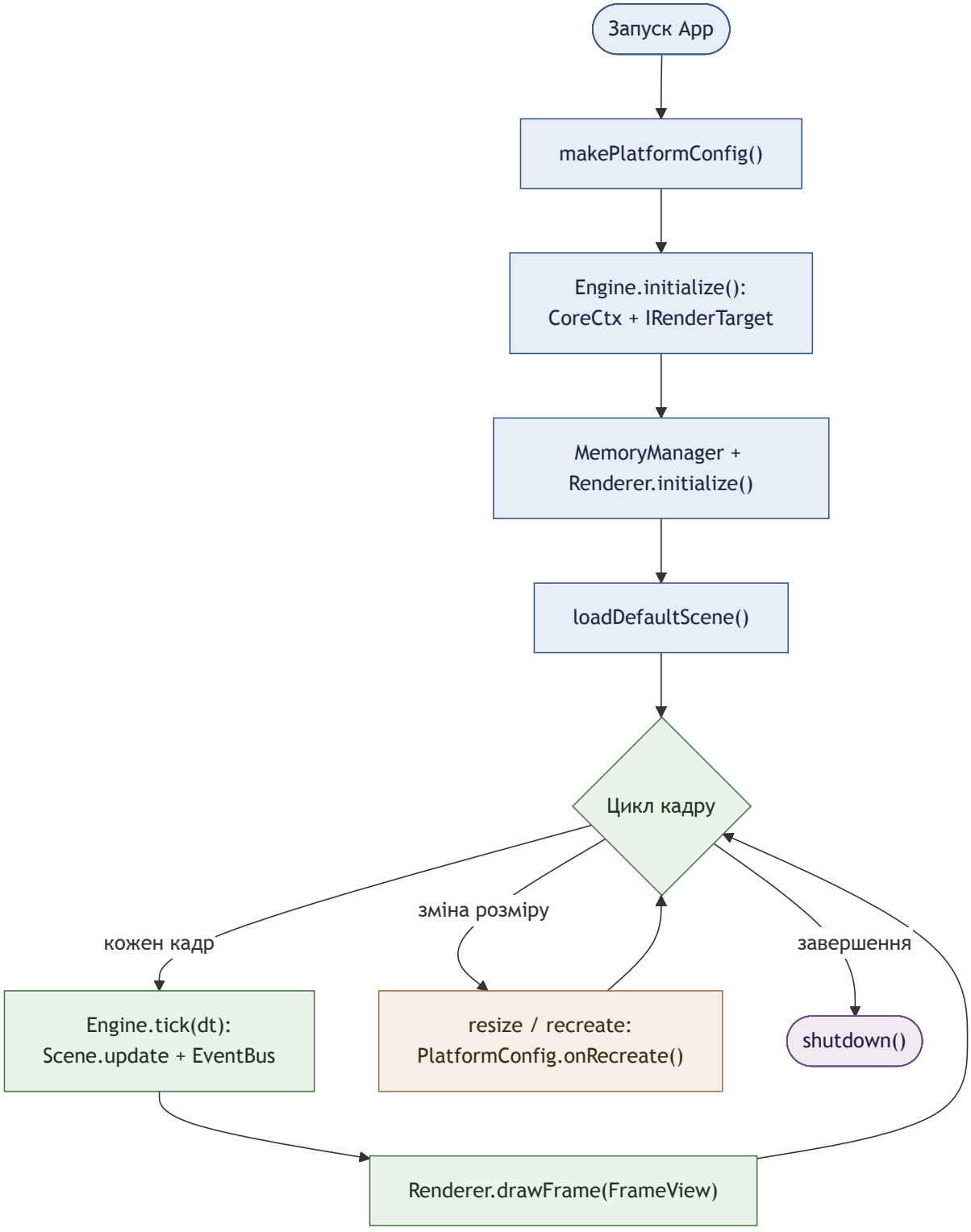


Рисунок 3.2 – Блок-схема життєвого циклу застосунку

Отже, ядро рушія реалізовано як окремий програмний шар із власним життєвим циклом, який не залежить безпосередньо від конкретної платфор-

ми відображення. Такий підхід забезпечує повторне використання спільної логіки в різних прикладних режимах і робить систему придатною для подальшого функціонального розширення.

3.3 Модель представлення сцени та внутрішніх даних системи

Важливим аспектом програмної реалізації є внутрішня модель даних, на основі якої в рушії подається сцена, просторові зв'язки між об'єктами та характеристики тих сутностей, що підлягають відображенню або оновленню. Реалізація цієї моделі безпосередньо спирається на методичні засади, розглянуті у попередньому розділі, зокрема на поєднання ієрархічної організації сцени та компонентного опису властивостей об'єктів.

Центральною структурою сцени є клас `Scene`. Він зберігає назву сцени, перелік створених сутностей, відображення ідентифікаторів на об'єкти, посилання на активну камеру та власну шину подій. Практично `Scene` виконує роль контейнера верхнього рівня, у межах якого створюються, шукаються, оновлюються, перепідпорядковуються та знищуються сутності. Така організація є достатньо простою для дослідницького проєкту, але водночас вона вже створює основу для більш складного редакторського керування сценою.

Окрема сценова сутність подається класом `Entity`. Кожна сутність має унікальний ідентифікатор, текстове ім'я, ознаку активності, посилання на сцену, посилання на батьківську сутність, перелік дочірніх сутностей і набір компонентів. Така будова відображає природний для 3D-графіки спосіб моделювання складених об'єктів, коли просторове положення та поведінка окремого вузла можуть залежати від його місця в дереві сцени.

Особливу роль у цій моделі відіграють батьківсько-дочірні зв'язки. У класі `Entity` реалізовано механізм зміни батька, який унеможливлює ство-

рення циклів у дереві сцени та підтримує впорядкований список дочірніх вузлів. Це важливо не лише для логічної організації сутностей, а й для коректного обчислення світових матриць трансформації, оскільки положення дочірнього об'єкта залежить від світового перетворення його предка.

Функціональне наповнення сутностей будується за компонентним принципом. Базовим інтерфейсом для всіх компонентів є клас `IComponent`, що визначає уніфікований набір операцій ініціалізації, оновлення та візуалізації, а також містить посилання на власника. Завдяки цьому сутність виступає насамперед контейнером ідентичності та зв'язків, а конкретні властивості або поведінка приєднуються у формі незалежних компонентів.

У поточній реалізації ключовими є чотири типи компонентів. Компонент `TransformComponent` відповідає за позицію, орієнтацію та масштаб сутності. Компонент `MeshComponent` зв'язує сутність із геометрією та матеріальним дескриптором, який використовується під час рендерингу. Компонент `CameraComponent` задає параметри перспективної проєкції та надає матриці виду й проєкції. Компонент `FlyCameraController` реалізує поведінку вільної навігації камерою на основі вводу користувача.

Трансформація є центральною характеристикою просторового стану сутності. У реалізації `TransformComponent` зберігаються вектор позиції, кватерніон орієнтації та вектор масштабу. На їх основі обчислюється локальна матриця перетворення, а світова матриця формується рекурсивно з урахуванням світового перетворення батьківської сутності. Такий спосіб узгоджується з математичними положеннями попереднього розділу і дає змогу коректно описувати складені об'єкти та імпортовані ієрархії вузлів.

Компонент камери спирається на трансформацію тієї самої сутності. Видова матриця формується як обернена до світової матриці трансформації, а матриця проєкції створюється засобами `GLM` на основі поля зору, співвідношення сторін і площин відсікання. Додатково в реалізації враховано спе-

цифіку кліп-простору Vulkan через інверсію осі Y у проєкційній матриці. Це є характерним прикладом того, як математичні положення перетворюються на конкретні програмні рішення [15].

Для навігації камерою використовується компонент `FlyCameraController`. Його логіка побудована на накопиченні стану клавіш та зміщень покажчика миші, після чого на кожному кадрі обчислюється нове положення та орієнтація камери. Реалізація підтримує рух уздовж локальних напрямків камери, переміщення по вертикалі, а також прискорений режим при натисканні клавіші модифікатора. З погляду архітектури це є вдалим прикладом компонентного розширення сцени без внесення додаткової логіки в сам клас `Entity`.

Зв'язок між сценою і рендерингом здійснюється через клас `RenderSystem`. Його завдання полягає в обході активної сцени та побудові проміжного подання кадру у вигляді структури `FrameView`. До цієї структури входять матриці камери, часовий параметр кадру і перелік об'єктів `DrawItem`, де для кожного елемента зберігаються посилання на меш, дескриптор матеріалу та модельна матриця. Таким чином, на межі між сценою і рендерером формується компактне та однозначне представлення тієї інформації, яка потрібна для побудови конкретного кадру.

Узагальнену модель представлення сцени ілюструє діаграма класів на рис. 3.3, яка фіксує основні сутності та зв'язки між ними.

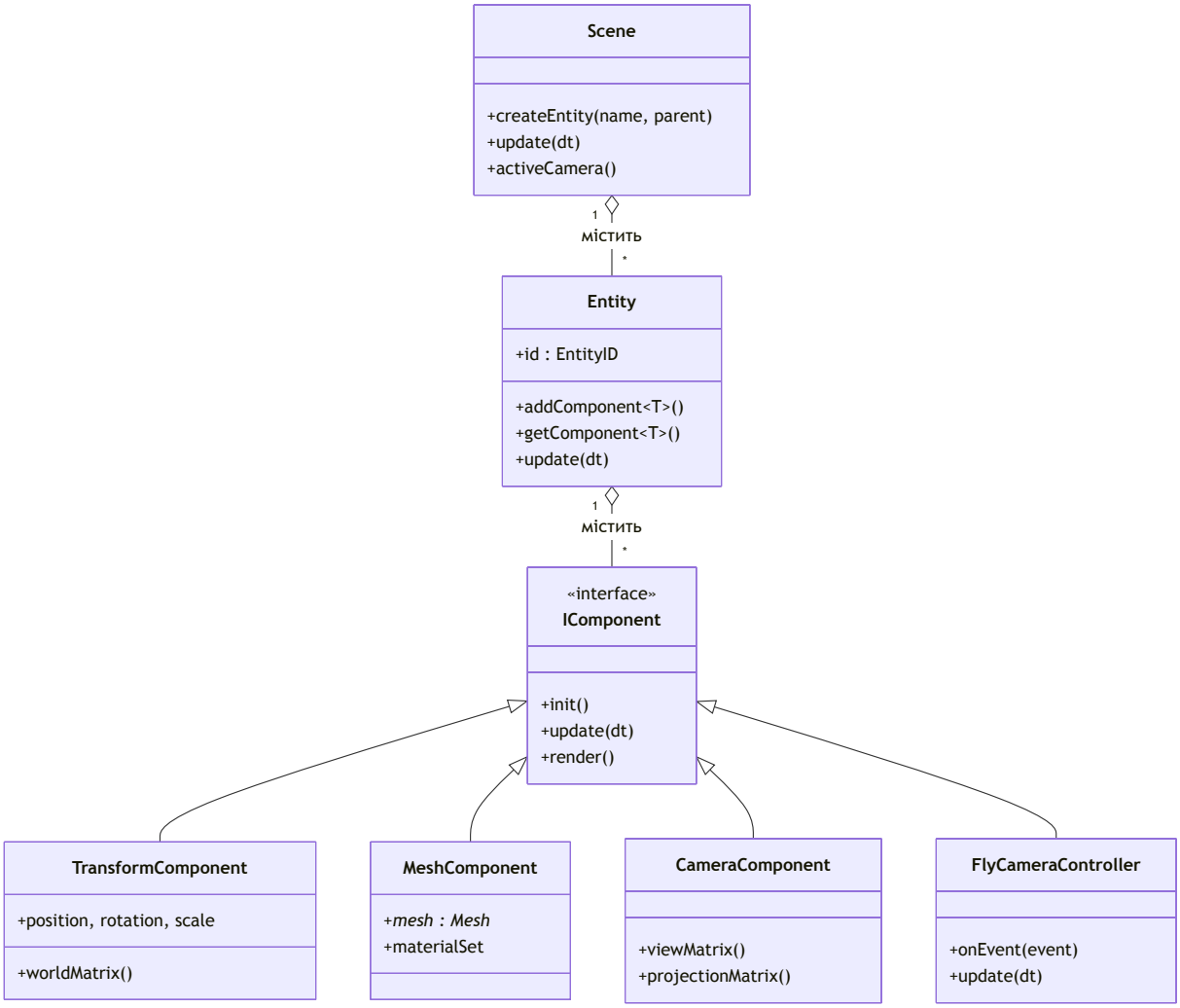


Рисунок 3.3 – UML-діаграма класів сценової підсистеми

Потоки даних між сценою, компонентами і рендерером наведено на рис. 3.4.

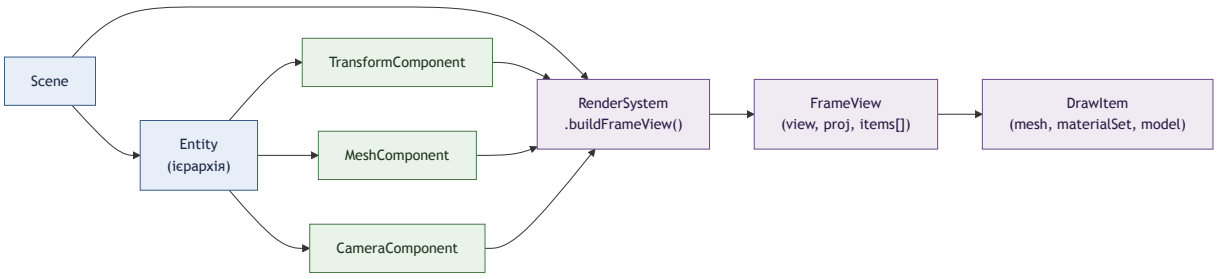


Рисунок 3.4 – Схема внутрішньої моделі даних сцени

Для формалізації відповідності між внутрішніми структурами й їх призначенням зручно використати таблицю.

Таблиця 3.2 – Основні структури даних сценової підсистеми

Структура або клас	Рівень представлення	Призначення
`Scene`	Рівень сцени	Зберігання набору сутностей, активної камери та подій сцени
`Entity`	Рівень об’єкта	Представлення окремої сутності з ідентичністю, ієрархією та компонентами
`TransformComponent`	Просторовий стан	Позиція, орієнтація, масштаб, локальна та світова матриці
`MeshComponent`	Візуальні дані	Посилання на геометрію та матеріальний дескриптор
`CameraComponent`	Візуальний контекст	Обчислення матриць виду і проєкції
`FrameView`	Проміжне подання кадру	Компактна передача даних від сцени до рендерера

Таким чином, внутрішня модель даних рушія побудована на поєднанні ієрархії сутностей і компонентного наповнення. Така організація робить систему достатньо гнучкою для розширення, але водночас зберігає прозорість і керованість, що є особливо важливим для дослідницького програмного засобу.

3.4 Програмна реалізація підсистеми рендерингу на основі Vulkan

Підсистема рендерингу є тією частиною рушія, у якій обрані алгоритмічні та архітектурні рішення отримують безпосереднє практичне втілення. У розробленій системі вона охоплює побудову й зберігання контексту Vulkan, конфігурування графічного конвеєра, організацію перекадрових ресурсів, запис команд у командні буфери та взаємодію з конкретною ціллю рендерингу.

Базовою структурою цього рівня є `CoreCtx`. Він містить RAII-об'єкти контексту Vulkan: екземпляр, фізичний пристрій і логічний пристрій, а також набір додаткових контекстних компонентів. Серед цих компонентів зберігаються графічна черга, дескрипторні макети, графічний pipeline, параметри конфігурації pipeline, інформація про підтримку present-черги, swapchain-структури тощо. Такий спосіб організації дає змогу розділити окремі частини стану Vulkan за їхнім функціональним призначенням, не перевантажуючи один монолітний клас надмірною кількістю полів.

Ініціалізація контексту побудована через клас `CoreInitializer` та механізм `BuilderChain`. По суті, це ланцюжок спеціалізованих побудовників, кожен з яких відповідає за окремий етап формування графічного середовища. У базовому сценарії використовуються побудовники створення екземпляра, вибору та налаштування пристрою, а також конфігурування графічного pipeline. У разі потреби ланцюжок доповнюється іншими builder-ами, наприклад для створення поверхні, swapchain або підключення підтримки зовнішньої пам'яті.

З погляду архітектури такий підхід має дві переваги. По-перше, він локалізує складні технічні аспекти ініціалізації Vulkan у невеликих спеціалізованих класах. По-друге, він дозволяє змінювати або розширювати порядок конфігурації залежно від режиму роботи системи. Саме цим пояснюється на-

явність окремих ініціалізаторів для GLFW-режиму та QML-режиму: перший додає логіку створення поверхні та swapchain, тоді як другий орієнтується на вибір конкретного фізичного пристрою та вмикає підтримку обміну зовнішньою пам’яттю.

Практична реалізація графічного pipeline зосереджена в класі `PipelineBuilder`. У ньому відбувається завантаження шейдерних модулів, опис формату вершини, конфігурування динамічних станів, растеризації, перевірки глибини, змішування кольору, а також створення макетів дескрипторних наборів і pipeline layout. Окремо реалізовано поділ ресурсів на два набори: перший для перекадрових даних, передусім матриць камери, і другий для матеріальних ресурсів, зокрема текстурних дескрипторів.

Важливим практичним рішенням є використання dynamic rendering замість жорстко описаних render pass та framebuffer. У результаті опис вкладень переноситься безпосередньо у фазу запису команд, а побудова кадру стає гнучкішою і менш перевантаженою допоміжними структурами. Для дослідницького рушія це є доцільним, оскільки дозволяє зосередити увагу на фактичній логіці рендерингу, а не на супровідному boilerplate-коді.

Безпосереднє формування кадру виконується класом `Renderer`. На етапі ініціалізації він отримує доступ до контексту та поточної цілі рендерингу, після чого створює перекадрові дескрипторні ресурси потрібного розміру. Під час виклику `drawFrame()` рендерер запитує початок кадру в `IRenderTarget`, записує всі необхідні графічні команди й передає керування назад цілі рендерингу для завершення кадру та, за потреби, представлення результату.

Алгоритм запису команд у поточній реалізації охоплює кілька послідовних етапів. Спочатку виконується перехід кольорового зображення у стан, придатний для запису як у color attachment, і переведення глибинного зображення у стан depth attachment. Далі починається dynamic rendering,

прив’язується графічний pipeline, задаються viewport та scissor, після чого в mapped-пам’ять поточного перекадрового буфера копіюються матриці виду і проєкції. Потім прив’язується перший дескрипторний набір із даними кадру, а для кожного `DrawItem` передаються push-константи, прив’язується матеріальний набір і виконується індексоване малювання меша [17, 19, 20].

Поділ даних на перекадрові та пооб’єктні реалізовано у вигляді двох структур. Структура `UniformBufferObject` містить матриці `view` і `proj`, які є спільними для всіх об’єктів поточного кадру. Структура `ObjectPushConstantData` містить матрицю моделі та часовий параметр, тобто ті дані, які можуть змінюватися для кожного окремого об’єкта. Такий розподіл не лише добре узгоджується з програмною моделлю Vulkan, а й зменшує обсяг зайвого дублювання даних.

Підтримка перекадрових ресурсів винесена до окремого класу `FrameResources`. У ньому для кожного кадру створюється окремий uniform buffer із host-visible пам’яттю, виконується його відображення в адресний простір процесора, а також виділяється дескрипторний набір, що зв’язує буфер з шейдером. Така схема особливо корисна у swapchain-режимі, де одночасно можуть існувати кілька кадрів у польоті, кожен із власним набором перекадрових даних.

Принципово важливим архітектурним рішенням є інтерфейс `IRenderTarget`. Він абстрагує всі деталі конкретної моделі представлення кадру й надає рендереру лише ті операції, які потрібні для універсальної побудови зображення. Завдяки цьому одна й та сама логіка рендерингу може працювати з різними реалізаціями цілі: віконною swapchain-ціллю та позаекранною ціллю для інтеграції у QML. Саме ця абстракція є одним із ключових структурних рішень усього проєкту.

Рух даних у підсистемі рендерингу наведено на рис. 3.5.

Рисунок 3.5 – Схема проходження кадру в підсистемі рендерингу

Узагальнення основних структур підсистеми рендерингу наведено в табл. 3.3.

Таблиця 3.3 – Основні структури та класи підсистеми рендерингу

Клас або структура	Рівень використання	Призначення
`CoreCtx`	Контекст Vulkan	Зберігання базових об’єктів Vulkan і контекстних компонентів
`CoreInitializer` / `BuilderChain`	Ініціалізація	Покрокова побудова та конфігурація графічного середовища
`PipelineBuilder`	Конвеєр	Створення шейдерних модулів, descriptor set layout та графічного pipeline
`Renderer`	Формування кадру	Запис графічних команд і взаємодія з поточною ціллю рендерингу
`FrameResources`	Перекадрові ресурси	Організація uniform buffer-ів і дескрипторів для поточних кадрів
`IRenderTarget`	Представлення кадру	Абстракція початку, завершення та ресурсів конкретної цілі рендерингу

Отже, підсистема рендерингу реалізує той самий алгоритмічний підхід, що був обґрунтований у другому розділі, однак у практичній формі кон-

кретних класів, структур даних і послідовності дій. Вона поєднує явне керування станом зображень, поділ ресурсів за частотою оновлення, динамічний рендеринг і платформно-незалежну абстракцію цілі відображення.

3.5 Реалізація підсистем ресурсів, пам’яті, вводу та подій

Повноцінна робота 3D-графічного рушія неможлива без тих підсистем, які забезпечують підготовку зовнішніх ресурсів, розміщення даних у пам’яті графічного пристрою, обробку дій користувача та координацію взаємодії між окремими частинами системи. Хоча ці підсистеми не формують кадр безпосередньо, саме вони створюють умови, за яких рендеринг стає змістовним і керованим.

Завантаження зовнішніх 3D-ресурсів реалізовано через клас `GltfLoader`. Його завданням є зчитування файлів формату `.gltf` або `.glb`, інтерпретація вузлової структури моделі, побудова внутрішнього подання геометрії та витягування матеріальних параметрів і текстур. У результаті роботи завантажувача формується структура `LoadedModel`, яка містить перелік вузлів `LoadedNode`. Для кожного вузла зберігаються геометрія, світова матриця трансформації та матеріальні дані.

Практично алгоритм імпорту включає кілька етапів. Спочатку визначається спосіб читання файлу залежно від того, чи є він двійковим `GLB`, чи текстовим `glTF`. Далі з моделі витягуються масиви вершинних атрибутів: позиції, нормалі та текстурні координати. Окремо зчитуються індекси примітивів. Після цього формується внутрішня структура `MeshData`, що містить вектор вершин і вектор індексів. Додатково обробляються матеріали, зокрема базовий колір, параметри PBR і посилання на текстурні зображення.

Суттєвою перевагою такого підходу є те, що імпортована модель одразу трансформується у внутрішній формат рушія. Це означає, що завантажувач не просто копіює зовнішні дані, а перетворює їх у вигляд, придатний для

подальшого розміщення на графічному пристрої та включення в сценову ієрархію. Після завершення імпорту вузли моделі використовуються для створення відповідних сутностей у сцені, яким приєднуються компоненти трансформації та геометрії.

Для подання геометрії на рівні ресурсів використовуються структури `MeshData` та `Mesh`. Перша з них є CPU-рівневим контейнером вершин і індексів, тоді як друга є GPU-орієнтованою абстракцією, яка містить буфери вершин і індексів та вміє прив'язувати їх до командного буфера. Під час завантаження даних застосовується staging-підхід: спочатку вершини та індекси копіюються у host-visible буфер, а потім переносяться у device-local пам'ять. Це є типовим, але вкрай важливим прийомом раціональної організації ресурсів у Vulkan [35].

Текстурні дані в поточній реалізації подано класом `TextureImage`. Він забезпечує створення зображення та view-об'єкта, налаштування sampler-а, а також формування дескрипторної інформації, що надалі використовується у матеріальних наборах. Текстура може бути завантажена як із зовнішнього файлу, так і безпосередньо з масиву RGBA-пікселів, який формується під час імпорту glTF-моделі. Це робить підсистему текстур достатньо універсальною для різних джерел даних.

Проміжною ланкою між CPU-рівневим поданням ресурсів і графічним пристроєм є клас `MemoryManager`. У ньому інкапсульовано типові операції роботи з пам'яттю Vulkan: створення буферів, створення зображень, пошук відповідного типу пам'яті, копіювання буферів, переведення зображень між макетами та копіювання буфера в зображення. Особливе значення має підтримка короткоживучих командних буферів для разових операцій підготовки ресурсів, оскільки саме такі буфери використовуються для копіювання та layout transitions.

Підсистема вводу реалізована у вигляді класу `InputSystem`. Її логі-

ка побудована на двох взаємопов’язаних рівнях. На першому рівні система запам’ятовує поточний стан клавіш і кнопок миші, а також останні координати покажчика. На другому рівні кожна подія вводу одночасно надсилається у внутрішню шину подій, що дає змогу іншим підсистемам реагувати на неї через підписки. Такий підхід поєднує переваги станового представлення вводу та подієвої взаємодії.

Подієву основу системи становить клас `EventBus`. У ньому реалізовано механізм підписки слухачів на події певного типу, буферизацію опублікованих повідомлень і подальшу диспетчеризацію до відповідних обробників. Для дослідницького рушія така модель є цілком виправданою, оскільки дозволяє послабити прямі залежності між підсистемами й спростити приєднання нової поведінки без жорсткого вбудовування її в наявні класи.

На практиці подієва модель використовується, зокрема, у компоненті `FlyCameraController`. Він підписується на події клавіатури та миші, накопичує інформацію про стан вводу, а в методі `update()` перетворює її на зміну положення або орієнтації камери. Це є показовим прикладом того, як компонент сцени взаємодіє не безпосередньо з віконною системою, а з узагальненим внутрішнім інтерфейсом вводу. У результаті один і той самий контролер може працювати і в runtime-режимі, і в редакторському режимі, незважаючи на відмінності між GLFW і Qt.

Варто відзначити, що саме в цих підсистемах найбільш виразно проявляється інженерна цінність модульного підходу. Завантажувач ресурсів не залежить від способу представлення кадру. Ввід не залежить від конкретного рендерера. Контролер камери не залежить від типу віконної системи. Керування пам’яттю не залежить від структури сцени. У сукупності це робить систему не лише функціональною, а й достатньо впорядкованою з погляду супроводу.

Процес завантаження моделі та її включення до сцени наведено на

рис. 3.6.

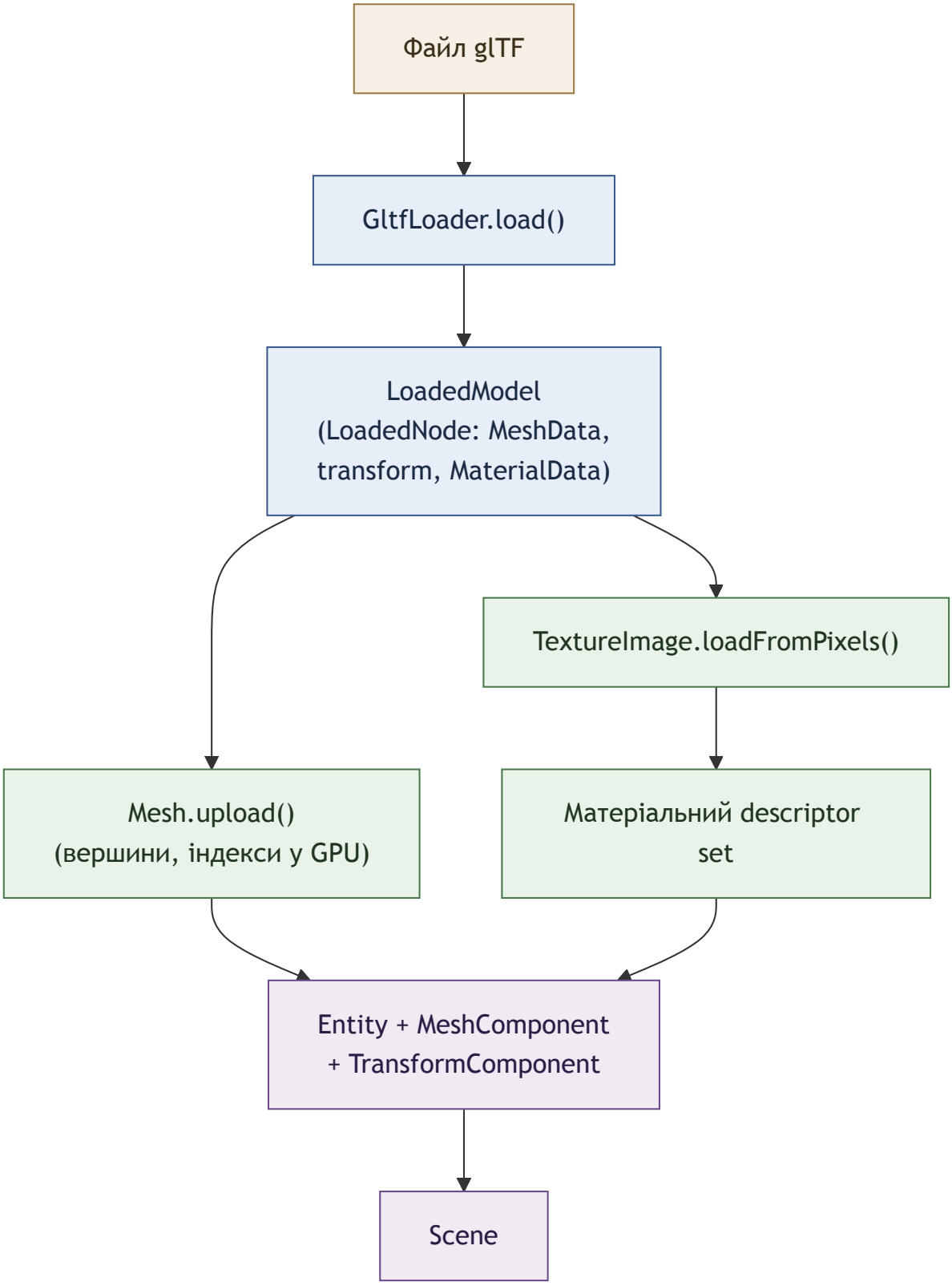


Рисунок 3.6 – Схема завантаження glTF-моделі та включення її до сцени

Схему обробки вводу та керування камерою наведено на рис. 3.7.

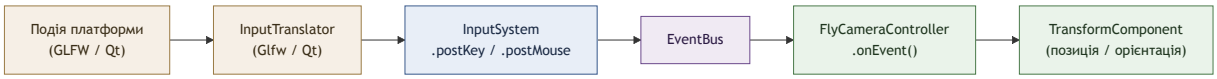


Рисунок 3.7 – Схема обробки вводу та керування камерою

Отже, підсистеми ресурсів, пам’яті, вводу та подій утворюють необхідну інфраструктурну основу для роботи всього рушія. Саме вони роблять можливим завантаження реальних моделей, інтерактивну навігацію сценою та впорядковану взаємодію між різними частинами системи.

3.6 Реалізація режимів роботи системи: runtime та editor

Однією з найбільш характерних рис розробленого програмного засобу є підтримка двох режимів використання спільного графічного ядра. Перший режим являє собою автономний віконний застосунок, у якому кадр безпосередньо виводиться у вікно через swapchain. Другий режим інтегрує рушій у графічний інтерфейс, побудований на базі Qt і QML, де кадр формується у позаекранне зображення, а потім використовується як текстура у viewport-елементі. З погляду архітектури це рішення є принципово важливим, оскільки демонструє реальну повторну використовуваність ядра.

Автономний режим реалізовано через клас `RuntimeApp`. У ньому організовано створення GLFW-вікна, підключення callback-функцій для клавіатури й миші, запуск базової ініціалізації через `App::initialize()` та власний цикл подій. Після старту застосунок регулярно викликає `tick()`, а значить, запускає універсальний цикл оновлення рушія та формування кадру. Саме в цьому режимі найпростіше тестувати базову працездатність рендерингу в класичному середовищі окремого вікна.

Платформозалежну конфігурацію автономного режиму забезпечує

клас `GlfwPlatformConfig`. Він будує спеціалізований ініціалізатор, який отримує від GLFW необхідні розширення Vulkan, створює поверхню відображення, задає початкові розміри області рендерингу та вмикає підтримку swapchain. Таким чином, логіка GLFW не проникає в `Engine`, а локалізується в конфігураційному шарі.

Відображення кадру у runtime-режимі здійснюється через клас `SwapchainRenderTarget`. У ньому реалізовано класичну схему acquire–submit–present, підтримку кількох кадрів у польоті, зберігання семафорів і fence-об'єктів синхронізації, а також допоміжні ресурси, потрібні для глибинного тестування. Коли вікно змінює розмір або swapchain стає неактуальним, система повторно створює необхідні ресурси, не змінюючи при цьому самого алгоритму рендерингу в `Renderer`.

Редакторський режим реалізовано через клас `EditorApp`. На відміну від автономного запуску, він не створює власного традиційного вікна рендерингу, а готує рушій до роботи всередині Qt/QML-середовища. Для цього застосунок зберігає ідентифікатори виробника та пристрою, розміри viewport-області, а також створює конфігурацію, яка орієнтується на той самий фізичний графічний пристрій, що використовується Qt scene graph. Саме це забезпечує сумісність при подальшому обміні графічною пам'яттю.

Ключовою частиною редакторського режиму є `OffscreenRenderTarget`. На відміну від swapchain-цілі, він створює власне кольорове зображення, глибинне зображення, view-об'єкти, командний пул і fence, після чого дозволяє рендереру використовувати це зображення як звичайну ціль формування кадру. Після завершення рендерингу кадр залишається не у макеті представлення на екран, а в макеті `shader read only`, що дозволяє далі використовувати його як текстуру.

Особливо важливою є підтримка зовнішньої пам'яті у редакторському режимі. Для цього в ланцюжок ініціалізації додається

побудовник `ExternalMemoryDeviceBuilder`, який вмикає розширення `VK_KHR_external_memory_fd` [36]. Додатково використовується `PhysicalDeviceMatchBuilder`, що орієнтує вибір фізичного пристрою на той самий GPU, який вже використовується в Qt. У результаті позаекранне зображення може бути експортоване як дескриптор зовнішньої пам'яті і повторно імпортоване в графічний стек Qt.

Інтеграція з QML здійснюється через клас `VulkanViewport`, успадкований від `QQuickItem`. У момент готовності Qt scene graph цей компонент отримує дескриптори фізичного пристрою та логічного пристрою Qt, після чого ініціалізує рушій у відповідному режимі. Далі на кожному кадрі викликається `renderFrame()`, а сформоване позаекранне зображення імпортується в оточення Qt як нативна Vulkan-текстура. Отриманий ресурс використовується для побудови `QSGSimpleTextureNode`, який і відображається у viewport-області.

З точки зору програмної інженерії такий механізм є значущим результатом дипломного проєкту, оскільки він показує, що низькорівневий рушій на основі Vulkan може бути інтегрований не лише в окремий віконний застосунок, а й у складніше інтерфейсне середовище без відмови від власного графічного циклу. У цьому сенсі редакторський режим є не просто альтернативним способом запуску, а підтвердженням архітектурної правильності відокремлення ядра від засобів представлення кадру.

Водночас слід підкреслити, що поточна реалізація редакторського режиму ще не є завершеним повнофункціональним редактором сцен у розумінні промислових рушіїв. На цьому етапі реалізовано насамперед сам механізм інтеграції viewport-а з QML-середовищем, базову взаємодію з кадром і передачу вводу користувача в рушій. Однак саме ця частина є найскладнішою з інженерного погляду і створює основу для подальшого нарощування функціональності редактора.

Порівняння двох режимів використання рушія наведено на рис. 3.8.

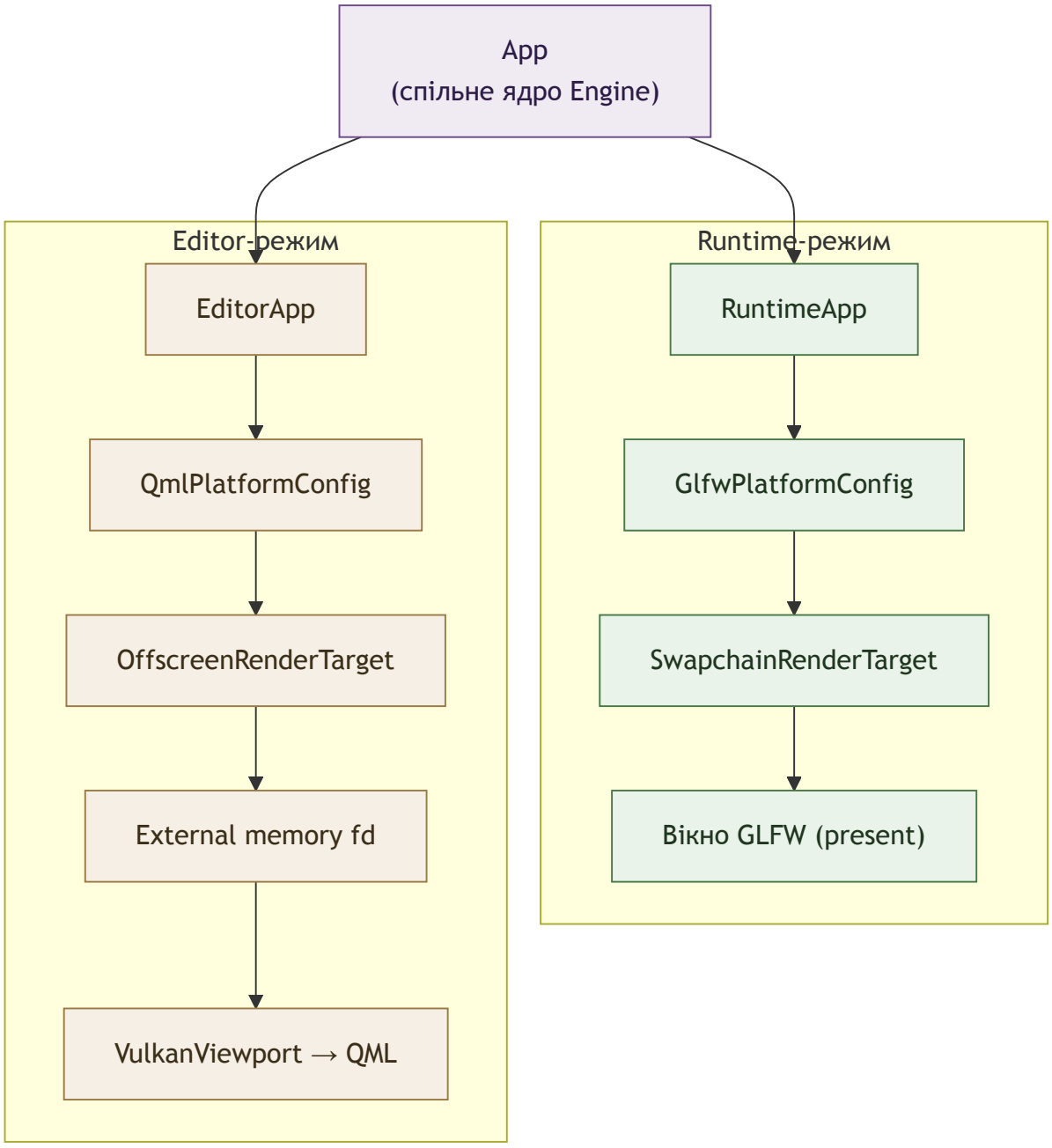


Рисунок 3.8 – Схема режимів роботи runtime та editor

Ключові відмінності між режимами запуску узагальнено в табл. 3.4.

Таблиця 3.4 – Порівняння двох режимів використання графічного ядра

Характеристика	Runtime-режим	Editor-режим
Зовнішнє середовище	GLFW-вікно	Qt/QML-інтерфейс
Ціль рендерингу	`SwapchainRenderTarget`	`OffscreenRenderTarget`
Фінальний стан кадру	`present src`	`shader read only`
Представлення результату	Безпосередньо у вікні	Як текстура у `QQuickItem`
Основне призначення	Автономне тестування та запуск	Інтеграція з інтерфейсом редактора

Отже, підтримка двох режимів роботи є не допоміжною, а системоутворювальною властивістю розробленого рушія. Саме вона підтверджує, що архітектура системи побудована на справді повторно використовуваному ядрі, а не на єдиному жорстко прив'язаному сценарії відображення.

3.7 Апробація розробленого програмного засобу та результати реалізації

Завершальним етапом опису програмної реалізації є аналіз того, які функціональні можливості були фактично реалізовані в межах дипломного проєкту та яким чином вони підтверджують працездатність розробленого програмного засобу. У контексті цієї роботи апробація не зводиться до формального запуску застосунку, а розглядається як перевірка узгодженої роботи основних підсистем: ініціалізації Vulkan, завантаження ресурсів, формування сцени, побудови кадру, обробки вводу та інтеграції з інтерфейсним середовищем.

Базовий сценарій використання системи передбачає складання проє-

кту за допомогою CMake, запуск одного з двох цільових застосунків і автоматичне завантаження демонстраційної моделі сцени. В автономному режимі користувач отримує окреме вікно рендерингу, у якому може спостерігати сцену та переміщуватися в ній за допомогою клавіатури й миші. У редакторському режимі користувач працює з QML-вікном, у якому область візуалізації відображається як частина складнішого графічного інтерфейсу.

У результаті реалізації було забезпечено коректну ініціалізацію графічного середовища Vulkan, створення графічного pipeline, підтримку глибинного тестування, формування кадру з використанням dynamic rendering, а також передачу матриць камери й пооб'єктних параметрів у шейдери. Практично це означає, що розроблений рушій виконує повний цикл побудови зображення тривимірної сцени на основі власної внутрішньої структури даних.

Не менш суттєвим результатом є реалізація засобів імпорту моделей формату glTF. Система не лише зчитує геометричні дані, а й підтримує імпорт текстурних зображень, матеріальних параметрів та ієрархічних трансформацій вузлів. Після завантаження ці дані інтегруються у внутрішню сценову модель через створення сутностей та відповідних компонентів. Це підтверджує, що рушій здатний працювати не лише з тестовою процедурною геометрією, а й із реальними зовнішніми 3D-ресурсами.

Окремого підкреслення заслуговує працездатність системи камер і навігації сценою. У ході реалізації вдалося побудувати базовий механізм вільного переміщення в просторі з керуванням орієнтацією через рух миші та переміщенням уздовж локальних осей камери. Хоча це ще не вичерпує всіх можливих режимів навігації, для дослідницького рушія така функціональність є достатньою і практично значущою.

Найбільш показовим результатом, з інженерного погляду, є підтримка двох різних способів використання спільного графічного ядра. Автоном-

ний режим підтверджує здатність системи працювати як самостійний рендерінговий застосунок. Редакторський режим, своєю чергою, підтверджує можливість інтеграції низькорівневого Vulkan-рушія в GUI-оточення на базі Qt/QML. Така апробація свідчить не лише про працездатність окремих модулів, а й про коректність їхньої архітектурної композиції.

Відповідність реалізованих функцій основним вимогам до програмного продукту наведено в табл. 3.5.

Таблиця 3.5 – Відповідність реалізованих функцій основним вимогам до програмного продукту

Функціональна вимога	Спосіб реалізації	Результат
Ініціалізація Vulkan та відображення сцени у реальному часі	`CoreInitializer`, `Renderer`, `IRenderTarget`	Реалізовано
Ієрархічна організація сцени та керування об’єктами	`Scene`, `Entity`, `TransformComponent`	Реалізовано
Завантаження моделей glTF, текстур і матеріалів	`GltfLoader`, `Mesh`, `TextureImage`	Реалізовано
Камера, навігація та ввід користувача	`CameraComponent`, `FlyCameraController`, `InputSystem`	Реалізовано
Подієва взаємодія між підсистемами	`EventBus`	Реалізовано
Окремий runtime-режим і editor-режим	`RuntimeApp`, `EditorApp`	Реалізовано
Інтеграція з Qt/QML-інтерфейсом	`VulkanViewport`, offscreen-рендеринг, зовнішня пам’ять	Реалізовано

Разом із цим поточна реалізація має й природні обмеження, що є ти-

повими для дослідницького проєкту бакалаврського рівня. Насамперед у рушії поки що відсутні розвинені засоби редагування сцени, інспектування об’єктів, управління ієрархією через GUI та повноцінна ресурсна панель редактора [37]. Крім того, система матеріалів реалізована в базовому вигляді й орієнтована передусім на підтримку базових текстур та кольорових параметрів. Не реалізовано також розвинених засобів освітлення [38–40], анімації, фізичного моделювання чи складних post-processing-ефектів [41].

Однак зазначені обмеження не знижують науково-практичної цінності виконаної роботи. Навпаки, вони окреслюють природний напрям подальшого розвитку системи. Реалізоване ядро вже містить ті структурні елементи, на які можна спиратися під час наступних етапів розробки: модульну архітектуру, компонентну сценову модель, універсальний інтерфейс цілі рендерингу, базову ресурсну систему, два режими запуску та інтеграцію з Qt/QML. Це означає, що проєкт може використовуватися як навчально-дослідницька платформа для подальших експериментів із графічною архітектурою та функціональним розширенням рушія.

Результати роботи системи в runtime- та editor-режимах наведено на рис. 3.9 і рис. 3.10.

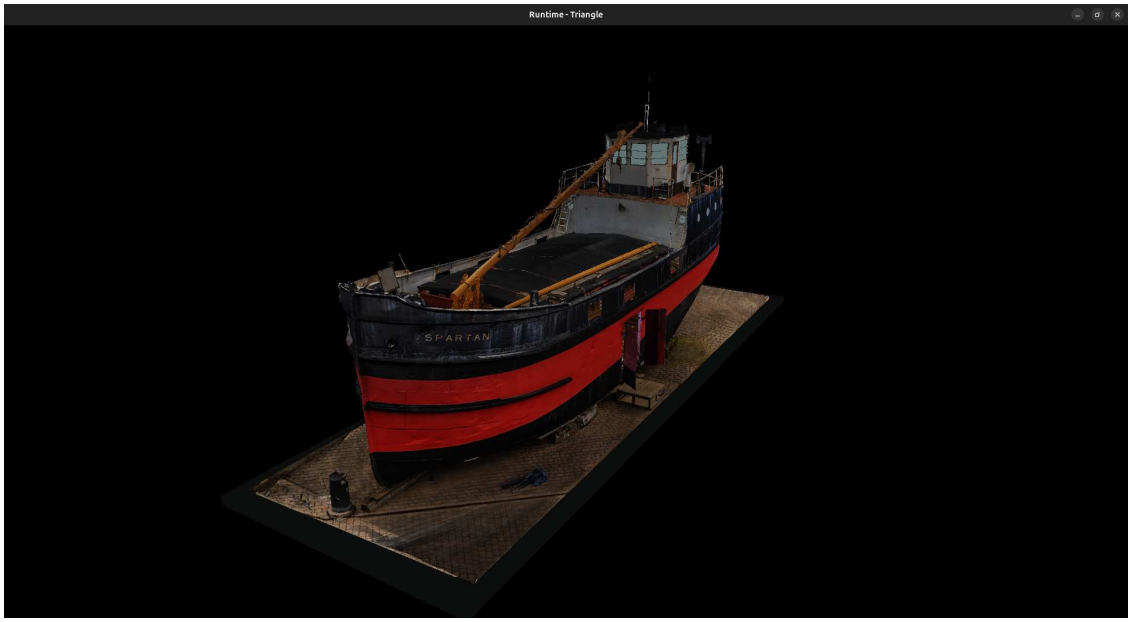


Рисунок 3.9 – Вигляд runtime-режиму із завантаженою 3D-сценою

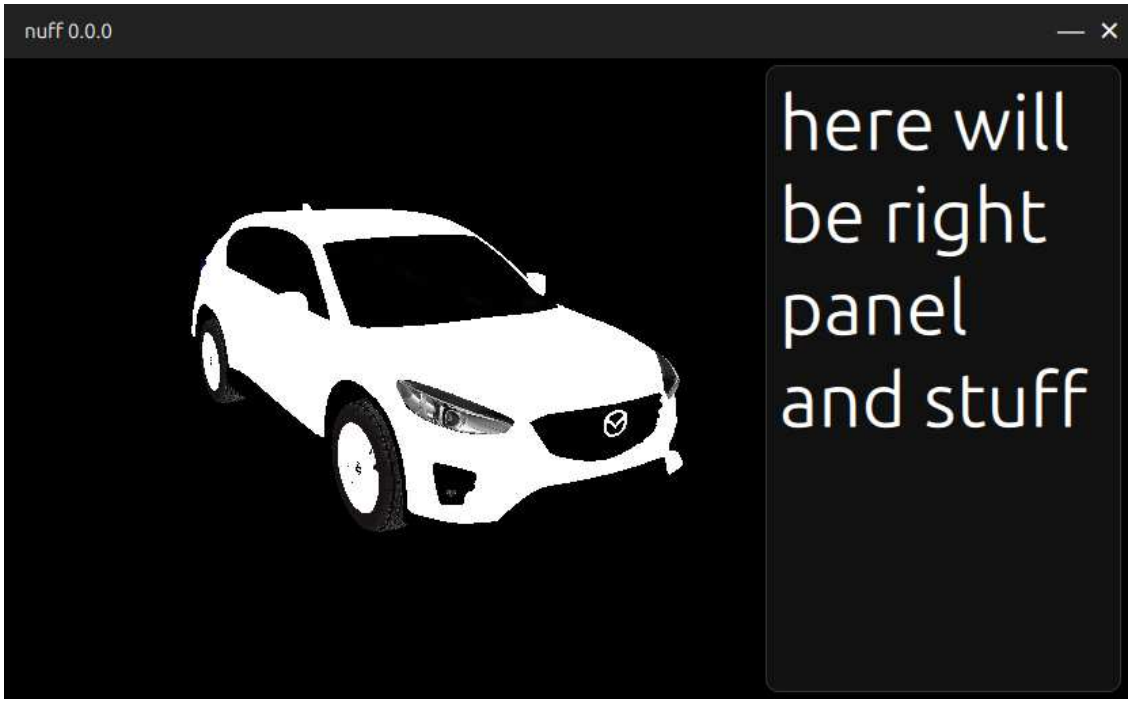


Рисунок 3.10 – Вигляд editor-режиму з відображенням сцени в QML-інтерфейсі

Отже, результати апробації підтверджують, що в межах дипломного проєкту вдалося реалізувати працездатний програмний 3D-графічний рушій

на основі Vulkan, який підтримує відображення тривимірних сцен, імпорт зовнішніх моделей, навігацію камерою, два режими представлення кадру та інтеграцію з графічним інтерфейсом. Сукупність цих результатів свідчить про досягнення поставленої мети розробки.

ВИСНОВКИ ДО РОЗДІЛУ

У третьому розділі було розглянуто програмну реалізацію 3D-графічного рушія на основі Vulkan та показано, яким чином методичні рішення, сформульовані у попередніх розділах, були втілені у конкретній структурі програмного продукту. Насамперед охарактеризовано загальну архітектуру системи, визначено призначення її основних модулів та обґрунтовано відокремлення ядра рушія від платформозалежних механізмів запуску й представлення результату.

Також описано реалізацію ядра рушія, життєвий цикл застосунку, модель сцени та компонентний спосіб представлення об’єктів. Показано, що сцена в системі подається через ієрархію сутностей із набором компонентів, а зв’язок між сценовою моделлю та підсистемою рендерингу здійснюється через проміжне представлення кадру.

Окрему увагу приділено підсистемі рендерингу на основі Vulkan, організації контексту, побудові графічного pipeline, використанню dynamic rendering, роботі з перекадровими та пооб’єктними даними, а також реалізації абстракції цілі рендерингу. Крім того, розглянуто підсистеми завантаження ресурсів, керування пам’яттю, вводу та подієвої взаємодії, які забезпечують працездатність і цілісність усього програмного засобу.

Важливим результатом є підтвердження можливості використання одного й того самого графічного ядра у двох режимах: автономному runtime-режимі та editor-режимі з інтеграцією у Qt/QML-інтерфейс. Це свідчить про архітектурну правильність прийнятих рішень і демонструє практичну придатність розробленої системи як дослідницької платформи для подальшого розвитку.

Таким чином, у третьому розділі підтверджено, що розроблений програмний засіб є цілісною багатомодульною системою, у якій реалізовано

основні функції 3D-графічного рушія: представлення сцени, імпорт ресурсів, формування кадру на основі Vulkan, керування камерою, обробку вводу та інтеграцію з інтерфейсним середовищем. Сукупність отриманих результатів засвідчує досягнення поставленої мети дипломної роботи.

ВИСНОВКИ

У дипломній роботі розв’язано актуальне науково-практичне завдання, що полягало у розробці програмного 3D-графічного рушія на основі Vulkan API для відображення тривимірних сцен у реальному часі, завантаження моделей, керування сценовими об’єктами та інтеграції з графічним інтерфейсом користувача.

На основі аналізу предметної області 3D-графіки, рендерингу в реальному часі та сучасних графічних API встановлено, що для побудови дослідницького 3D-графічного рушія доцільно використовувати низькорівневий підхід до керування графічними ресурсами, командними буферами та синхронізацією. Обґрунтовано застосування Vulkan API як кросплатформного інструменту з високим рівнем контролю над процесом формування кадру, а також доцільність модульного підходу, ієрархічного подання сцени та компонентної організації об’єктів.

Порівняльний аналіз наявних рішень показав, що промислові рушії на кшталт Unity та Unreal Engine мають значні функціональні переваги, однак є надмірно складними для задачі детального вивчення низькорівневого рендерингу та архітектури графічної системи. З огляду на це обґрунтовано доцільність створення власного спеціалізованого рішення, орієнтованого на прозору архітектуру, контрольовану сценову модель, відокремлення ядра рендерингу від платформного оточення та можливість подальшого дослідницького розширення. Виходячи зі сформованих вимог, обрано програмні засоби та технології `C++23`, `Vulkan`, `Qt 6/QML`, `GLFW`, `GLM`, формат `glTF 2.0`, бібліотеку `TinyGLTF`, шейдерний інструментарій `Slang` та систему збирання `CMake`; сформульовано архітектурні принципи побудови системи, серед яких модульність, відокремлення логіки рендерингу від способу представлення кадру, використання абстракції `IRenderTarget`, компонентна модель сцени та

розмежування перекадрових і пооб’єктних даних.

Розроблено програмну систему *nuff* – програмний 3D-графічний рушій на основі Vulkan, у якому реалізовано основні підсистеми, необхідні для роботи з тривимірною сценою: ядро рушія, підсистема рендерингу на основі dynamic rendering, сценова модель типу `Scene--Entity--Component`, засоби завантаження моделей формату `glTF`, керування графічними ресурсами та пам’яттю, підсистема вводу і подієвої взаємодії, а також два режими використання спільного графічного ядра – автономний runtime-режим на базі GLFW і editor-режим з інтеграцією у Qt/QML-інтерфейс через позаекранний рендеринг та обмін зовнішньою пам’яттю.

На основі апробації та тестування доведено коректність роботи розробленого програмного забезпечення: підтверджено успішну ініціалізацію Vulkan-контексту, побудову графічного pipeline, формування кадру для тривимірної сцени, завантаження зовнішніх моделей, функціонування камери та засобів навігації, а також працездатність двох режимів відображення результату – у віконному середовищі та в інтеграції з QML-інтерфейсом. Перспективи подальшого вдосконалення програмної системи полягають у розвитку редакторських засобів, розширенні системи матеріалів і освітлення, підтримці анімації, додаванні нових засобів інспектування сцени та поглибленні інструментів налагодження і профілювання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Computer Graphics: Principles and Practice / J. F. Hughes [та ін.]. – 3-е вид. – Boston : Addison-Wesley, 2014.
2. Real-Time Rendering / T. Akenine-Moller [та ін.]. – 4-е вид. – Boca Raton : CRC Press, 2018.
3. *Gregory J.* Game Engine Architecture / J. Gregory. – 3-е вид. – Boca Raton : CRC Press, 2018.
4. *Khronos Group*. Vulkan | Cross platform 3D Graphics [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://www.vulkan.org/ (дата зверн. 21.05.2026).
5. *Khronos Group*. What is Vulkan? :: Vulkan Documentation Project [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://docs.vulkan.org/guide/latest/what_is_vulkan.html (дата зверн. 21.05.2026).
6. *Sellers G.* Vulkan Programming Guide: The Official Guide to Learning Vulkan / G. Sellers, J. Kessenich, D. Shreiner. – Boston : Addison-Wesley, 2017.
7. *Pharr M.* Physically Based Rendering: From Theory to Implementation / M. Pharr, W. Jakob, G. Humphreys. – 3-е вид. – Cambridge : Morgan Kaufmann, 2016.
8. *Khronos Group*. OpenGL – The Industry Standard for High Performance Graphics [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://www.opengl.org/ (дата зверн. 21.05.2026).
9. *Microsoft*. Direct3D 12 programming environment setup - Win32 apps [Електронний ресурс] / Microsoft. – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-environment-set-up (дата зверн. 21.05.2026).

10. *Epic Games*. Unreal Engine [Електронний ресурс] / Epic Games. – Режим доступу до ресурсу: https://www.unrealengine.com/ (дата зверн. 21.05.2026).
11. *Epic Games*. Unreal Engine Documentation [Електронний ресурс] / Epic Games. – Режим доступу до ресурсу: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-6-documentation (дата зверн. 21.05.2026).
12. *Unity Technologies*. Unity Engine: 2D & 3D Development Platform [Електронний ресурс] / Unity Technologies. – Режим доступу до ресурсу: https://unity.com/products/unity-engine (дата зверн. 21.05.2026).
13. *Godot Foundation*. Godot Engine - Free and open source 2D and 3D game engine [Електронний ресурс] / Godot Foundation. – Режим доступу до ресурсу: https://godotengine.org/ (дата зверн. 21.05.2026).
14. *Open 3D Foundation*. Home - O3DE [Електронний ресурс] / Open 3D Foundation. – Режим доступу до ресурсу: https://o3de.org/ (дата зверн. 21.05.2026).
15. *Dunn F.* 3D Math Primer for Graphics and Game Development / F. Dunn, I. Parberry. – 2-е вид. – Boca Raton : CRC Press, 2011.
16. *Lengyel E.* Foundations of Game Engine Development, Volume 1: Mathematics / E. Lengyel. – Lincoln, CA : Terathon Software LLC, 2016.
17. *Khronos Vulkan Working Group*. Vulkan 1.3.296 – A Specification [Електронний ресурс] / Khronos Vulkan Working Group. – Режим доступу до ресурсу: https://registry.khronos.org/vulkan/specs/1.3/html/vkspec.html (дата зверн. 25.05.2026).
18. *Blanco V.* Vulkan Guide [Електронний ресурс] / V. Blanco. – Режим доступу до ресурсу: https://vkguide.dev/ (дата зверн. 25.05.2026).
19. *Khronos Group*. VK_KHR_dynamic_rendering Extension Specification [Електронний ресурс] / Khronos Group. – Режим

доступу до ресурсу: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_dynamic_rendering.html (дата зверн. 25.05.2026).

20. *Khronos Group*. VK_KHR_synchronization2 Extension Specification [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_synchronization2.html (дата зверн. 25.05.2026).
21. *NVIDIA Corporation*. NVIDIA Vulkan Developer Best Practices [Електронний ресурс] / NVIDIA Corporation. – Режим доступу до ресурсу: https://developer.nvidia.com/blog/vulkan-dos-donts/ (дата зверн. 25.05.2026).
22. *Arm Limited*. Arm Best Practices for Vulkan Developers [Електронний ресурс] / Arm Limited. – Режим доступу до ресурсу: https://developer.arm.com/documentation/101897/latest/ (дата зверн. 25.05.2026).
23. *Khronos Group*. glTF 2.0 Specification [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html (дата зверн. 24.05.2026).
24. *Qt Company*. Qt 6 Documentation [Електронний ресурс] / Qt Company. – Режим доступу до ресурсу: https://doc.qt.io/qt-6/ (дата зверн. 24.05.2026).
25. *GLFW Authors*. GLFW Documentation [Електронний ресурс] / GLFW Authors. – Режим доступу до ресурсу: https://www.glfw.org/documentation.html (дата зверн. 24.05.2026).
26. *G-Truc Creation*. OpenGL Mathematics (GLM) [Електронний ресурс] / G-Truc Creation. – Режим доступу до ресурсу: https://glm.g-truc.net/0.9.9/index.html (дата зверн. 24.05.2026).

27. *Shader Slang Project*. Slang Documentation [Електронний ресурс] / Shader Slang Project. – Режим доступу до ресурсу: https://shader-slang.org/ (дата зверн. 24.05.2026).
28. *Kitware*. CMake Documentation [Електронний ресурс] / Kitware. – Режим доступу до ресурсу: https://cmake.org/documentation/ (дата зверн. 24.05.2026).
29. *Fujita S.* TinyGLTF [Електронний ресурс] / S. Fujita, T. contributors. – Режим доступу до ресурсу: https://github.com/syoyo/tinygltf (дата зверн. 24.05.2026).
30. *Khronos Group*. SPIR-V Specification, Version 1.6 [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html (дата зверн. 25.05.2026).
31. *Khronos Group*. glslang – Khronos-reference front end for GLSL/ESSL [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://github.com/KhronosGroup/glslang (дата зверн. 25.05.2026).
32. *Khronos Group*. Vulkan Samples [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://github.com/KhronosGroup/Vulkan-Samples (дата зверн. 25.05.2026).
33. *Willems S.* Vulkan C++ examples and demos [Електронний ресурс] / S. Willems. – Режим доступу до ресурсу: https://github.com/SaschaWillems/Vulkan (дата зверн. 25.05.2026).
34. *ISO/IEC JTC1/SC22/WG21*. ISO/IEC 14882:2024 – Programming languages – C++ [Електронний ресурс] / ISO/IEC JTC1/SC22/WG21. – Режим доступу до ресурсу: https://www.iso.org/standard/83626.html (дата зверн. 25.05.2026).

35. *Sawicki A.* Vulkan Memory Allocator [Електронний ресурс] / A. Sawicki, AMD. – Режим доступу до ресурсу: https://gpuopen.com/vulkan-memory-allocator/ (дата зверн. 25.05.2026).
36. *Khronos Group*. VK_KHR_external_memory_fd Extension Specification [Електронний ресурс] / Khronos Group. – Режим доступу до ресурсу: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_external_memory_fd.html (дата зверн. 25.05.2026).
37. *Cornut O.* Dear ImGui – Bloat-free Graphical User Interface for C++ [Електронний ресурс] / O. Cornut. – Режим доступу до ресурсу: https://github.com/ocornut/imgui (дата зверн. 25.05.2026).
38. *Phong B. T.* Illumination for Computer Generated Pictures / B. T. Phong // Communications of the ACM. – 1975. – Т. 18, № 6. – С. 311–317.
39. *Cook R. L.* A Reflectance Model for Computer Graphics / R. L. Cook, K. E. Torrance // ACM Transactions on Graphics. – 1982. – Т. 1, № 1. – С. 7–24.
40. *Lengyel E.* Foundations of Game Engine Development, Volume 2: Rendering / E. Lengyel. – Lincoln, CA : Terathon Software LLC, 2019.
41. *Ericson C.* Real-Time Collision Detection / C. Ericson. – San Francisco : Morgan Kaufmann, 2005.

ДОДАТОК А

Програмний двигун на основі Vulkan API

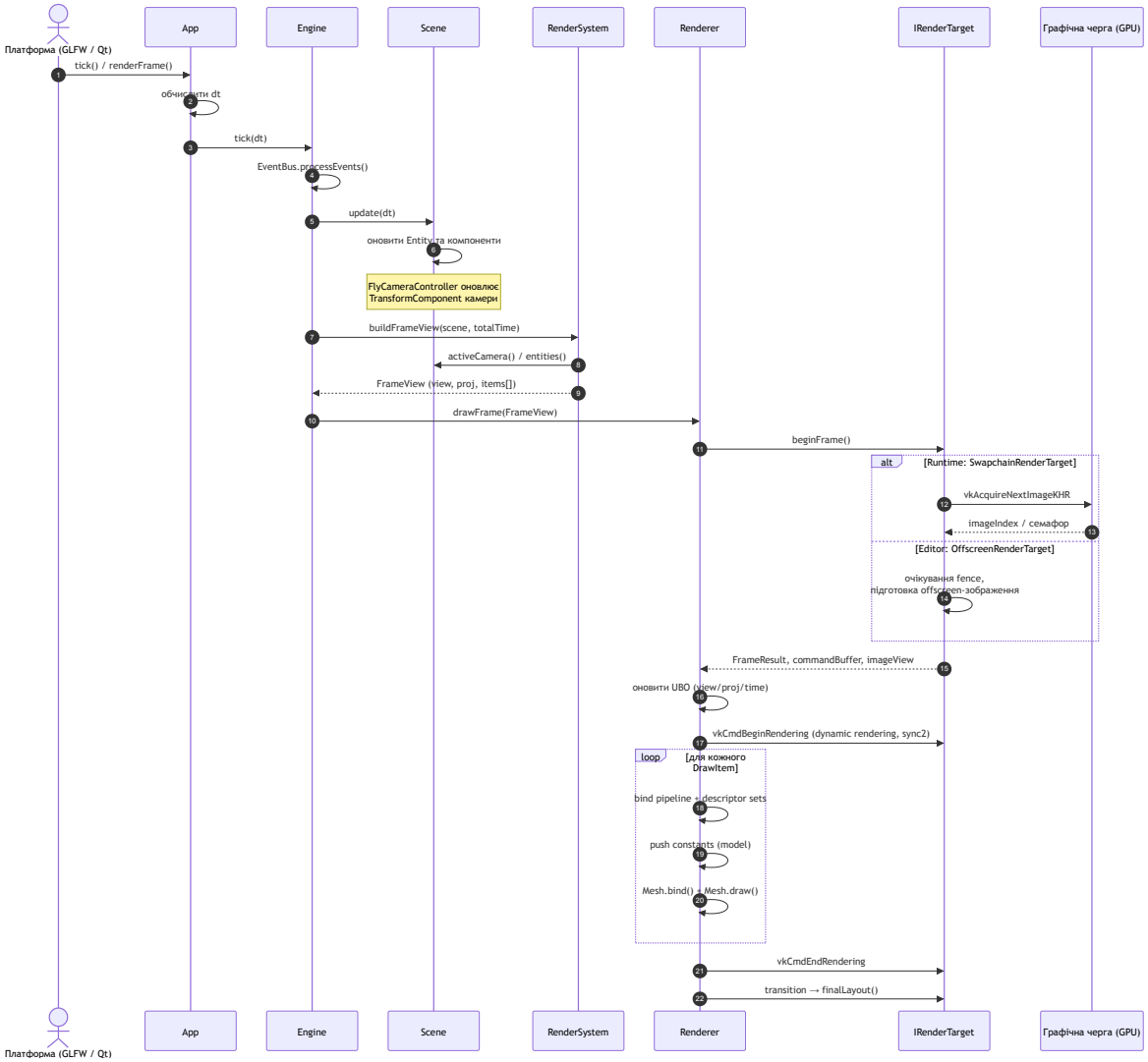
Принципова схема формування кадру

ІАЛЦ.467200.004 Д1

Аркушів 2

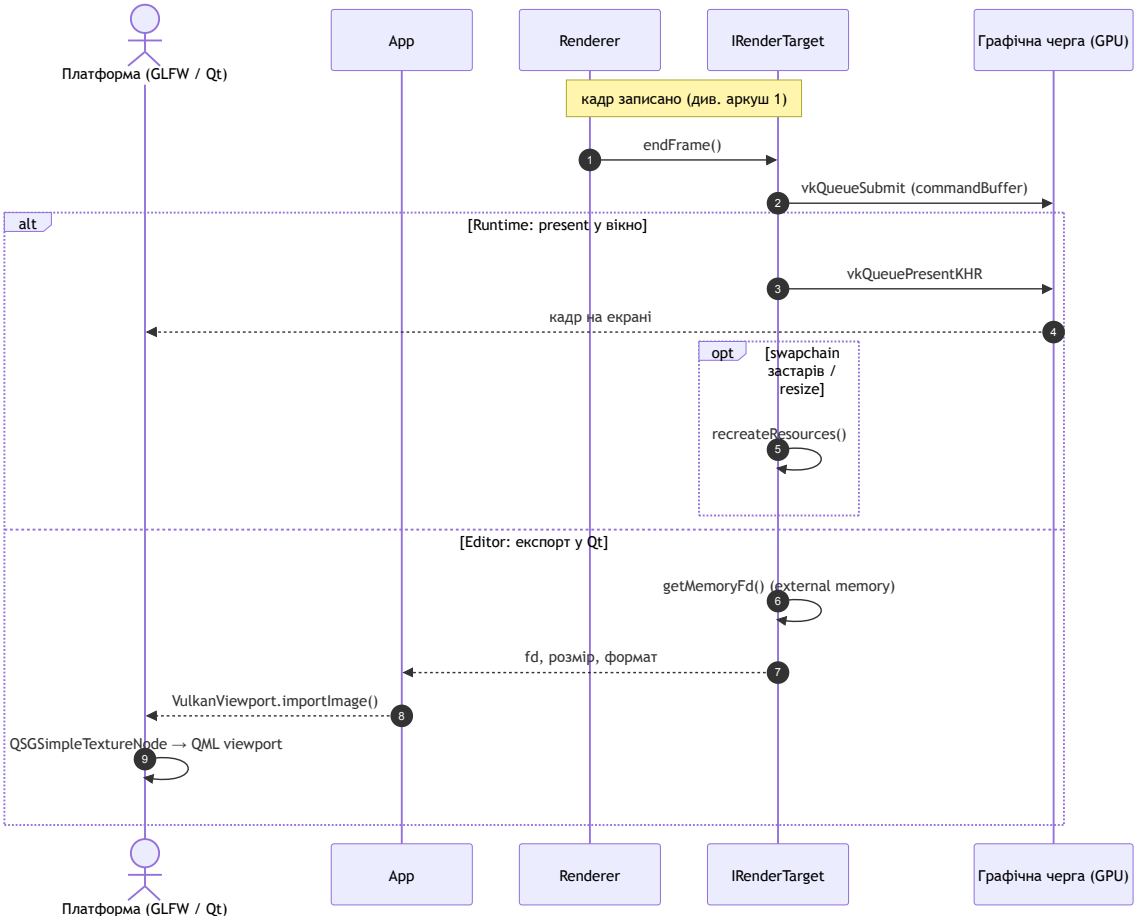
Київ 2026 р.

Принципова схема формування кадру: оновлення сцени та запис команд (аркуш 1)



					ІАЛЦ.467200.004 Д1			
Зм.	Лист	№ докум.	Підп.	Дата				
Розробив		Ярошенко О. С.			Програмний двигун на основі Vulkan API Принципова схема	Лит.	Аркуш	Аркушів
Перевірив		Порєв В. М.					1	2
						НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ ІМ-21		
Н. контр.		Нікольський С. С.						
Затвердив		Волокита А. М.						

Принципова схема формування кадру: подання у вікно та експорт у Qt (аркуш 2)



ДОДАТОК Б

Програмний двигун на основі Vulkan API

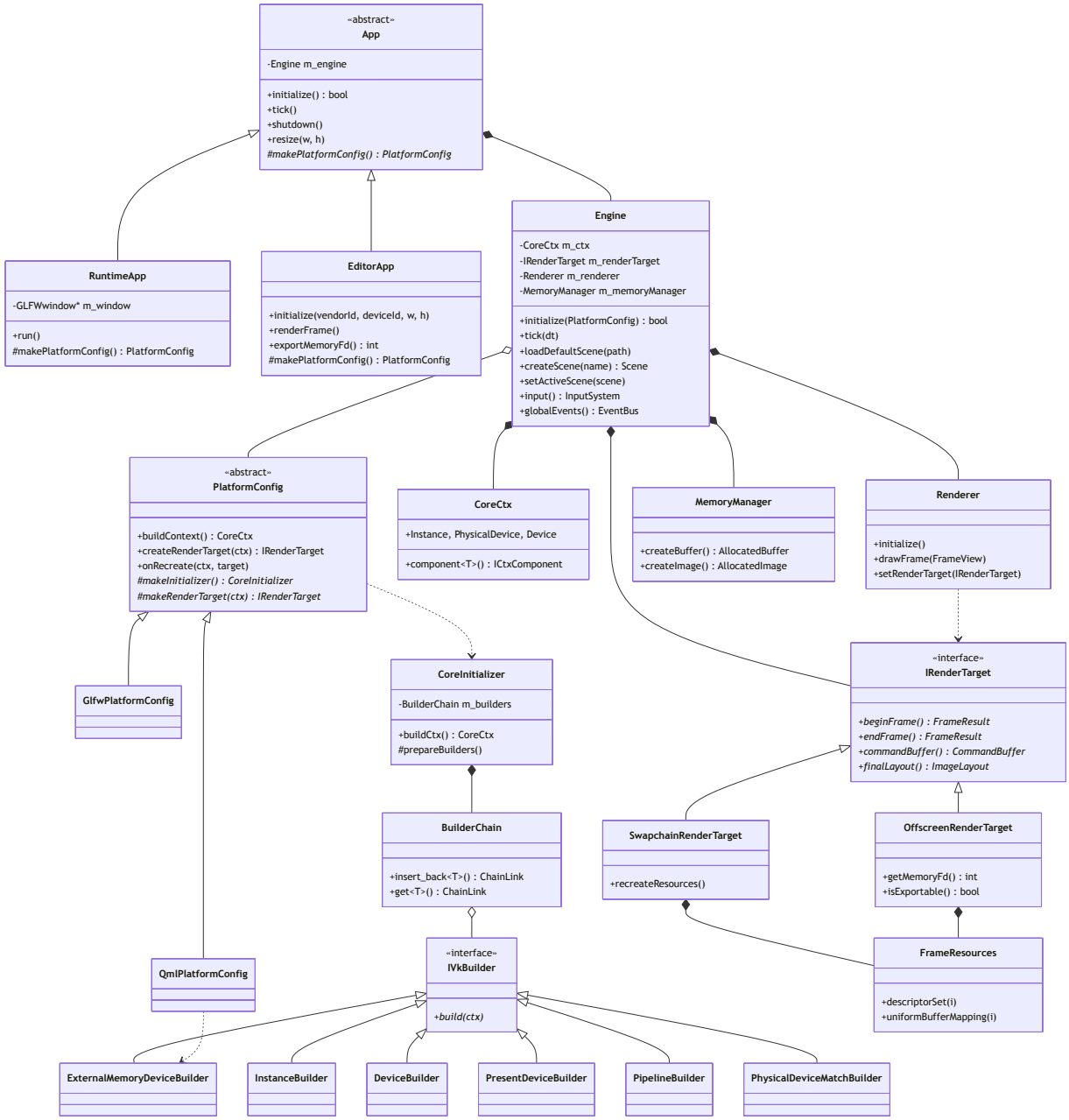
UML-діаграма компонентів програмного комплексу

ІА/ЛЦ.467200.005 Д2

Аркушів 2

Київ 2026 р.

Функціональна схема (діаграма класів): ядро застосунку, ініціалізація та конвеєр рендерингу (аркуш 1)

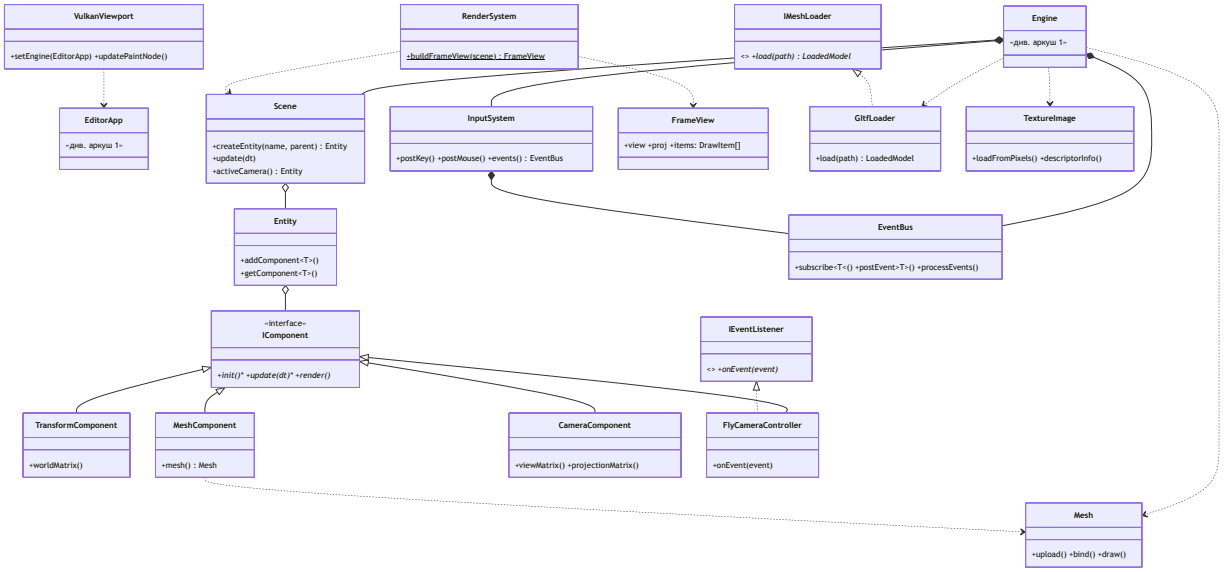


					ІАЛЦ.467200.005 Д2
Зм.	Лист	№ докум.	Підп.	Дата	
Розробив		Ярошенко О. С.			Програмний двигун на основі Vulkan API Функціональна схема
Перевірив		Порєв В. М.			
Н. контр.		Нікольський С. С.			
Затвердив		Волокита А. М.			

Лит.	Аркуш	Аркушів
	1	2

НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ ІМ-21

Функціональна схема (діаграма класів): сцена, компоненти та ресурси (аркуш 2)



ДОДАТОК В

Програмний двигун на основі Vulkan API

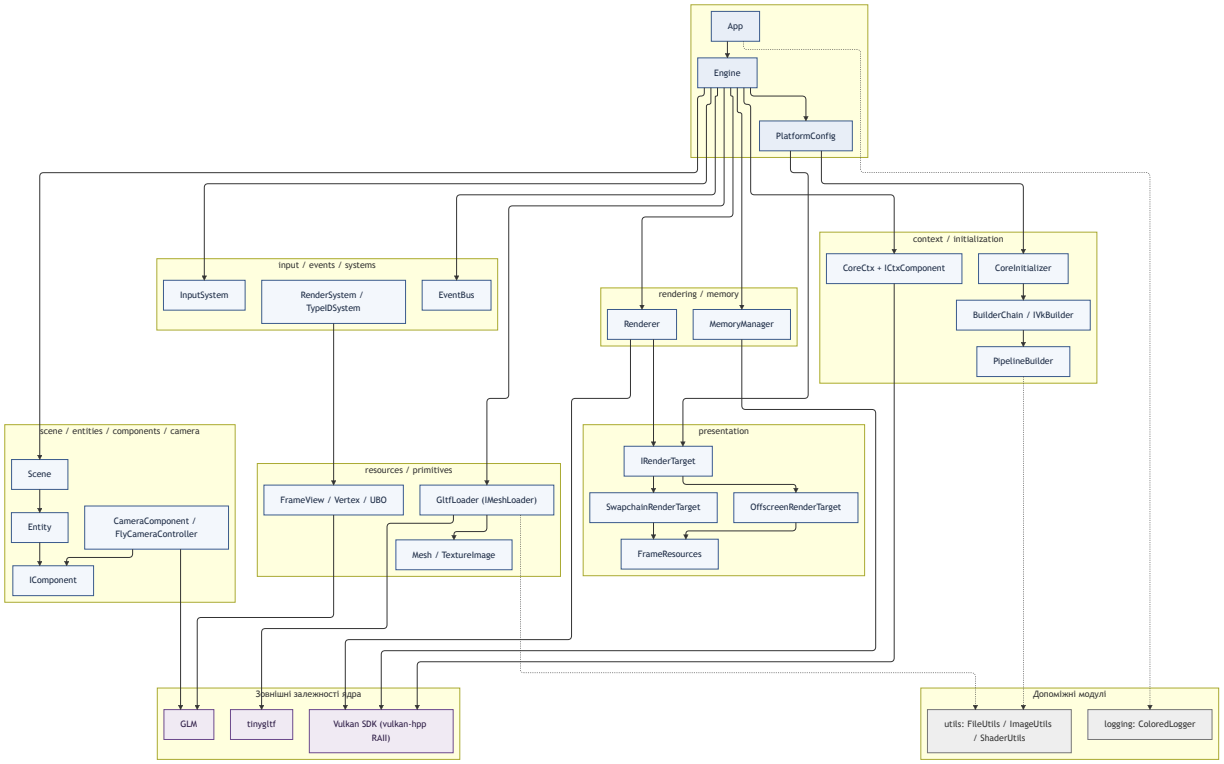
Структурна схема програмного комплексу

ІАЛЦ.467200.006 ДЗ

Аркушів 2

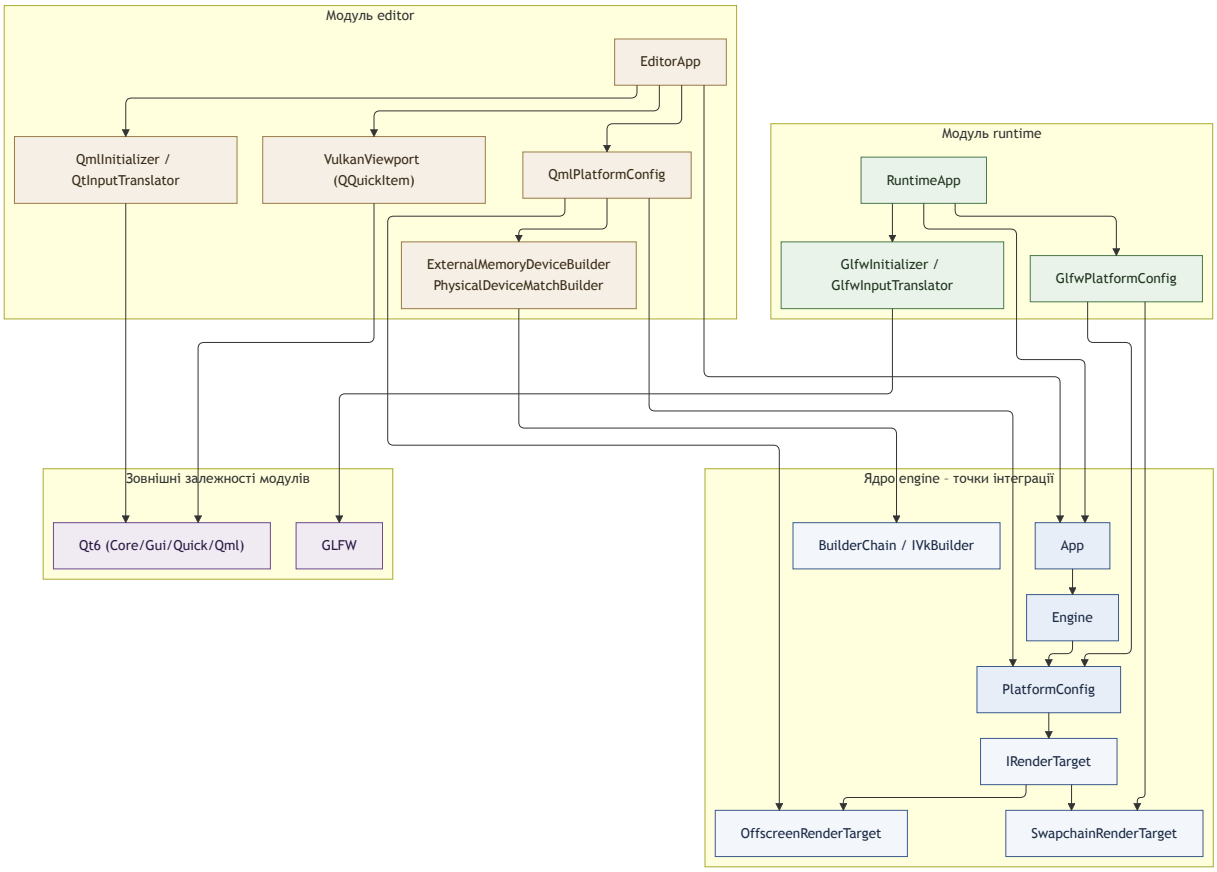
Київ 2026 р.

Структурна схема: внутрішня будова ядра engine (аркуш 1)



					ІАЛЦ.467200.006 ДЗ	
Зм.	Лист	№ докум.	Підп.	Дата		
Розробив		Ярошенко О. С.			Програмний двигун на основі Vulkan API Структурна схема	Лит. / Аркуш 1 / Аркушів 2
Перевірив		Порєв В. М.				
Н. контр.		Нікольський С. С.				НТУУ "КПІ ім. Ігоря Сікорського", ФІОТ ІМ-21
Затвердив		Волокита А. М.				

Структурна схема: прикладні модулі та інтеграція з платформою (аркуш 2)



ДОДАТОК Г

Програмний двигун на основі Vulkan API

Допоміжні таблиці до пояснювальної записки

ІАЛЦ.467200.007 Д4

Аркушів 6

Київ 2026 р.

У цьому додатку наведено допоміжні довідкові таблиці, які доповнюють виклад у розділах 1 і 2 пояснювальної записки.

Таблиця Г.1 – Порівняння сучасних графічних API

API	Основні переваги	Основні обмеження
OpenGL	Відносна простота освоєння, кросплатформність, зручність для навчальних і прототипних рішень	Менший контроль над ресурсами і синхронізацією, вища залежність від драйверної реалізації
Direct3D 12	Висока продуктивність, явне керування пам’яттю та командами, тісна інтеграція з платформою Windows	Висока складність реалізації, платформна обмеженість екосистемою Microsoft
Vulkan	Кросплатформність, низькі накладні витрати, багатопотокова підготовка команд, точний контроль над ресурсами	Значна складність архітектури та реалізації, підвищені вимоги до коректності коду

Таблиця Г.2 – Порівняння сучасних 3D-графічних рушіїв

Рушій	Основні переваги	Основні обмеження	Типові сфери застосування
Unreal Engine	Висока візуальна якість, розвинені засоби створення контенту, Blueprint, підтримка складних виробничих процесів	Висока складність, значні вимоги до ресурсів, надлишковість для компактних дослідницьких проєктів	AAA-ігри, візуалізація, віртуальне виробництво
Unity	Багатоплатформність, зручний інструментарій, велика екосистема, швидкий старт розробки	Менший акцент на дослідженні низькорівневих аспектів рендерингу	Ігри, мобільні застосунки, XR, інтерактивні 3D-системи
Godot	Відкритий вихідний код, доступність, гнучка організація сцен, зручність для навчання	Менша орієнтація на вузькоспеціалізовані низькорівневі експерименти	Незалежна розробка, навчальні та експериментальні проєкти
O3DE	Модульність, відкритість, кросплатформність, орієнтація на високоякісну графіку	Значний масштаб системи, висока складність інтеграції та освоєння	Ігрові, промислові та дослідницькі 3D-рішення

Таблиця Г.3 – Основні параметри конфігурації підсистеми рендерингу

Параметр	Типове значення	Призначення
Формат кольорового буфера	BGRA8 UNORM	Базове представлення кольорового кадру для подальшого відображення
Формат буфера глибини	D32_SFLOAT	Забезпечення точного тестування глибини
Режим рендерингу	dynamic rendering	Спрощення конфігурації вкладень і гнучке налаштування кадру
Порівняння глибини	less	Коректне відсікання фрагментів за глибиною
Відсікання граней	back-face culling	Зменшення кількості надлишкових фрагментів
Кількість кадрів у польоті	2	Баланс між продуктивністю та складністю синхронізації
Область виведення і відсікання	dynamic state	Адаптація до зміни розмірів області візуалізації

Таблиця Г.4 – Основні параметри камери і навігації

Параметр	Типове значення	Призначення
Поле зору камери	45°	Забезпечення природного перспективного сприйняття сцени
Ближня площина	0.1	Відсікання надто близьких об’єктів і стабілізація глибини
Дальня площина	1000.0	Збереження великого діапазону видимості сцени
Базова швидкість руху	5 од./с	Кероване переміщення в просторі сцени
Чутливість огляду	0.003 рад/піксель	Плавне керування поворотом камери
Прискорення руху	множник 4	Швидке переміщення на великих відстанях

Таблиця Г.5 – Обґрунтування вибору технологій для побудови 3D-графічного рушія

Технологія	Призначення	Обґрунтування вибору
C++23	Реалізація ядра рушія	Висока продуктивність, контроль над ресурсами, зручність системного програмування
Vulkan 1.3	Підсистема рендерингу	Низькорівневий контроль, керованість синхронізацією, придатність до дослідницької розробки
Qt 6 / QML	Графічний інтерфейс	Гнучке створення інтерфейсу і можливість інтеграції області візуалізації у складене середовище
GLFW	Віконний режим запуску	Просте створення вікна і платформи для незалежного тестування рендерингу
GLM	Математичний апарат	Підтримка векторів, матриць і кватерніонів у стилі, звичному для графічних застосунків
glTF 2.0	Формат 3D-ресурсів	Відкритий стандарт для моделей, матеріалів, текстур та ієрархії вузлів
TinyGLTF	Завантаження моделей	Компактна інтеграція імпорту glTF без надлишкової складності
Slang	Шейдерний інструментарій	Зручне описання шейдерних програм і компіляція у SPIR-V
CMake	Організація збірки	Підтримка багатомодульної структури, автоматизація збірки та конфігурації

ДОДАТОК Д

Програмний двигун на основі Vulkan API

Вихідний код програмного комплексу

ІАЛЦ.467200.008 Д5

Аркушів 34

Київ 2026 р.

У цьому додатку наведено вихідний код ключових модулів 3D рушія, які визначають його архітектуру: ядра `engine` (контекст Vulkan, рендерер, презентаційний шар, керування пам'яттю, завантаження glTF), редакторського застосунку `editor` з QML-viewport-ом та автономного застосунку `runtime`. Повний вихідний код проєкту, включно з допоміжними модулями, білд-скриптами та шейдерами, доступний у відкритому репозиторії:

https://github.com/notnuff/svrog

Модуль engine (ядро рушія)

engine/core/app.h

```
#pragma once

#include "engine.h"
#include "platform_config.h"

#include <QtCore/qtclasshelpermacros.h>

#include <chrono>
#include <cstdint>
#include <memory>
#include <string>

namespace nuff::engine::input { class InputSystem; }

namespace nuff::app {

class App {
public:
```

```
19      App();
20      virtual ~App();
21
22      Q_DISABLE_COPY(App)
23
24      bool initialize();
25      void shutdown();
26      void tick();
27
28      virtual void resize(uint32_t width, uint32_t height);
29      void notifyFramebufferResized();
30
31      bool isInitialized() const { return m_initialized; }
32
33      engine::Engine&       engineSystem()       { return m_engine; }
34      const engine::Engine& engineSystem() const { return m_engine; }
35
36      engine::input::InputSystem&       input()       { return m_engine.
        input(); }
37      const engine::input::InputSystem& input() const { return m_engine.
        input(); }
38
39  protected:
40      virtual std::unique_ptr<engine::PlatformConfig> makePlatformConfig() =
         0;
41      virtual std::string                             defaultScenePath()
        const;
42
43  private:
44      engine::Engine                          m_engine;
45      std::chrono::steady_clock::time_point m_lastFrameTime;
46      bool                                    m_hasLastFrameTime = false;
47      bool                                    m_initialized = false;
48  };
```

```
49
50 } // namespace nuff::app
```

engine/core/engine.h

```
1  #pragma once
2
3  #include "../events/i_event.h"
4  #include "../scene/scene.h"
5  #include "platform_config.h"
6
7  #include "context/ctx.h"
8  #include "memory/memory_manager.h"
9  #include "rendering/renderer.h"
10 #include "presentation/i_render_target.h"
11 #include "primitives/mesh.h"
12 #include "primitives/texture_image.h"
13
14 #include <QtCore/qtclasshelpermacros.h>
15
16 #include <cstdint>
17 #include <memory>
18 #include <string>
19 #include <vector>
20
21 namespace nuff::engine {
22
23 namespace events { class EventBus; }
24 namespace input  { class InputSystem; }
25
26 struct ActiveSceneChangedEvent : public events::Event<
       ActiveSceneChangedEvent> {
27     Scene* previous;
28     Scene* current;
```

```
29      ActiveSceneChangedEvent(Scene* previous, Scene* current) : previous(
        previous), current(current) {}
30  };
31
32  class Engine {
33  public:
34      Engine();
35      ~Engine();
36
37      Q_DISABLE_COPY(Engine)
38
39      bool initialize(std::unique_ptr<PlatformConfig> config);
40      void shutdown();
41
42      void notifyFramebufferResized();
43      void onTargetResized(uint32_t width, uint32_t height);
44
45      void loadDefaultScene(const std::string& modelPath);
46
47      input::InputSystem&       input();
48      const input::InputSystem& input() const;
49
50      void tick(float dt);
51
52      Scene* createScene(std::string name);
53      void   destroyScene(Scene* scene);
54
55      void   setActiveScene(Scene* scene);
56      Scene* activeScene() const;
57
58      const std::vector<std::unique_ptr<Scene>>& scenes() const;
59
60      void setSimulationPaused(bool paused);
61      bool isSimulationPaused() const;
```

```
62      void stepOneFrame();
63
64      events::EventBus& globalEvents();
65
66      uint64_t frameIndex() const;
67      float    totalTime() const;
68      float    deltaTime() const;
69
70      renderer::CoreCtx*       context()      const { return m_ctx.get(); }
71      renderer::IRenderTarget* renderTarget() const { return m_renderTarget.
    get(); }
72      template <class T> T*    renderTargetAs() const {
73          return static_cast<T*>(m_renderTarget.get());
74      }
75
76  private:
77      void uploadModelToScene(Scene& scene, const std::string& modelPath);
78      void buildMaterialDescriptorPool(uint32_t materialCount);
79
80      bool m_initialized = false;
81      std::vector<std::unique_ptr<Scene>> m_scenes;
82      Scene*   m_activeScene = nullptr;
83      std::unique_ptr<events::EventBus>   m_events;
84      std::unique_ptr<input::InputSystem> m_input;
85
86      bool     m_paused = false;
87      bool     m_stepRequested = false;
88
89      uint64_t m_frameIndex = 0;
90      float    m_totalTime = 0.0f;
91      float    m_deltaTime = 0.0f;
92
93      std::unique_ptr<PlatformConfig>          m_platform;
94      std::unique_ptr<renderer::CoreCtx>       m_ctx;
```

```
    std::unique_ptr<renderer::IRenderTarget> m_renderTarget;
    renderer::Renderer                       m_renderer;
    std::unique_ptr<renderer::MemoryManager> m_memoryManager;

    std::vector<std::unique_ptr<renderer::Mesh>>         m_meshes;
    std::vector<std::unique_ptr<renderer::TextureImage>> m_textures;
    vk::raii::DescriptorPool                             m_materialPool{
   nullptr};
    std::vector<vk::DescriptorSet>                       m_materialSets;
};

} // namespace nuff::engine
```

engine/context/ctx.h

```
#pragma once

#include "common/vk_common.h"

#include <optional>
#include <unordered_map>
#include <vector>

namespace nuff::renderer {

// TODO use separate queue for TRANSFER.

struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
    std::optional<uint32_t> presentFamily;

    [[nodiscard]] bool isComplete() const {
        return graphicsFamily.has_value();
    }
```

```
20
21      [[nodiscard]] bool hasPresentSupport() const {
22          return graphicsFamily.has_value() && presentFamily.has_value();
23      }
24  };
25
26  struct ICtxComponent {
27      ICtxComponent() = default;
28      virtual ~ICtxComponent() = default;
29  };
30
31  class CtxComponentTypeIDSystem {
32  private:
33      static size_t nextTypeID;
34
35  public:
36      template<typename T>
37      static size_t getTypeID() {
38          static size_t typeID = nextTypeID++;
39          return typeID;
40      }
41  };
42
43  struct CoreCtx {
44      // RAII context must be initialized first and destroyed last
45      vk::raii::Context context;
46
47      vk::raii::Instance instance{nullptr};
48      vk::raii::PhysicalDevice physicalDevice{nullptr};
49      vk::raii::Device device{nullptr};
50
51      template <typename CtxComponentT>
52      CtxComponentT& component() {
53          static_assert(std::is_base_of_v<ICtxComponent, CtxComponentT>,
```

```
                "extension must derive from ICtxExtension");

        auto id = CtxComponentTypeIDSystem::getTypeID<CtxComponentT>();
        if (m_ctxComponentsMap.contains(id)) {
            return *(static_cast<CtxComponentT*>(m_ctxComponentsMap[id]));
        }

        auto ptr = std::make_unique<CtxComponentT>();
        CtxComponentT& ref = *ptr;

        m_ctxComponents.emplace_back(std::move(ptr));
        m_ctxComponentsMap[id] = &ref;

        return ref;
    }

private:
    std::vector<std::unique_ptr<ICtxComponent>> m_ctxComponents;
    std::unordered_map<size_t, ICtxComponent*> m_ctxComponentsMap;
};

struct GraphicsCtxComponent : ICtxComponent {
    vk::Queue graphicsQueue;
    QueueFamilyIndices queueFamilyIndices;
};

struct InstanceExtensionsComponent : ICtxComponent {
    std::vector<const char*> instanceExtensions;
};

struct InstanceLayersComponent : ICtxComponent {
    std::vector<const char*> instanceLayers;
};

```

```
88  struct DebugMessengerCtxComponent : ICtxComponent {
89      vk::raii::DebugUtilsMessengerEXT debugMessenger{nullptr};
90  };
91
92  struct PipelineCtxComponent : ICtxComponent {
93      vk::raii::DescriptorSetLayout descriptorSetLayout{nullptr};
94      vk::raii::DescriptorSetLayout materialSetLayout{nullptr};
95      vk::raii::PipelineLayout pipelineLayout{nullptr};
96      vk::raii::Pipeline graphicsPipeline{nullptr};
97  };
98
99  struct PipelineConfigComponent : ICtxComponent {
100     vk::Format colorAttachmentFormat = vk::Format::eB8G8R8A8Unorm;
101     vk::Format depthAttachmentFormat = vk::Format::eD32Sfloat;
102 };
103
104 struct DeviceRequirementsComponent : ICtxComponent {
105     bool requirePresent = false;
106     std::vector<const char*> additionalDeviceExtensions;
107 };
108
109 struct PhysicalDevicePreferenceComponent : ICtxComponent {
110     std::optional<uint32_t> preferredVendorId;
111     std::optional<uint32_t> preferredDeviceId;
112 };
113
114 struct PresentQueueComponent : ICtxComponent {
115     vk::Queue presentQueue;
116 };
117
118 struct SwapchainCtxComponent : ICtxComponent {
119     vk::raii::SurfaceKHR surface{nullptr};
120     vk::raii::SwapchainKHR swapchain{nullptr};
121     std::vector<vk::Image> swapchainImages;
```

```
    std::vector<vk::raii::ImageView> swapchainImageViews;
    vk::Format swapchainImageFormat{};
    vk::Extent2D swapchainExtent{};
};

struct SwapchainSupportDetails {
    vk::SurfaceCapabilitiesKHR capabilities;
    std::vector<vk::SurfaceFormatKHR> formats;
    std::vector<vk::PresentModeKHR> presentModes;
};

struct RenderPassCtxComponent : ICtxComponent {
    vk::raii::RenderPass renderPass{nullptr};
    std::vector<vk::raii::Framebuffer> framebuffers;
};

} // namespace nuff::renderer
```

engine/context/builders/device_builder.h

```
#pragma once

#include ”context/builders/i_vk_builder.h”

#include <vector>

namespace nuff::renderer {

// Surface/present support is optional, configured via
   DeviceRequirementsMixin
class DeviceBuilder : public IVkBuilder {
public:
    void build(CoreCtx& ctx) override;

```

```
private:
    static QueueFamilyIndices findQueueFamilies(vk::PhysicalDevice device,
                                                vk::SurfaceKHR surface =
    nullptr);
    static bool isDeviceSuitable(vk::PhysicalDevice device, vk::SurfaceKHR
     surface,
                                 const std::vector<const char*>&
    extensions,
                                 bool requirePresent);

    template <typename... Features>
    static bool deviceHasRequiredFeatures(vk::PhysicalDevice device, const
     vk::StructureChain<Features...>& req);

    static bool checkDeviceExtensionSupport(vk::PhysicalDevice device,
                                            const std::vector<const char
    *>& reqExtensions);
};

} // namespace nuff::renderer

#include "context/builders/device_builder.inl"
```

engine/context/builders/pipeline_builder.h

```
#pragma once

#include "context/builders/i_vk_builder.h"

#include <string>
#include <vector>

namespace nuff::renderer {

```

```
// PipelineBuilder - Creates graphics pipeline
class PipelineBuilder : public IVkBuilder {
public:
    PipelineBuilder& setVertexShaderPath(const std::string& path);
    PipelineBuilder& setFragmentShaderPath(const std::string& path);
    PipelineBuilder& setVertexShaderCode(const std::vector<char>& code);
    PipelineBuilder& setFragmentShaderCode(const std::vector<char>& code);

    void build(CoreCtx& ctx) override;

private:
    std::string m_vertexShaderPath;
    std::string m_fragmentShaderPath;
    std::vector<char> m_vertexShaderCode;
    std::vector<char> m_fragmentShaderCode;
};

} // namespace nuff::renderer
```

engine/rendering/renderer.h

```
#pragma once

#include <functional>

#include "context/ctx.h"
#include "presentation/i_render_target.h"
#include "primitives/frame_view.h"

namespace nuff::renderer {

class Renderer {
public:
    using RecreateCallback = std::function<void()>;
```

```
14
15      void setContext(CoreCtx* ctx);
16      void setRenderTarget(IRenderTarget* renderTarget);
17      void setRecreateCallback(RecreateCallback callback);
18
19      void notifyFramebufferResized();
20
21      void initialize();
22      void cleanup();
23
24      void drawFrame(const FrameView& view);
25
26  private:
27      void recordRendering(const FrameView& view);
28
29      CoreCtx* m_ctx = nullptr;
30      IRenderTarget* m_renderTarget = nullptr;
31      RecreateCallback m_recreateCallback;
32      bool m_framebufferResized = false;
33  };
34
35  } // namespace nuff::renderer
```

engine/rendering/renderer.cpp

```
1   #include ”renderer.h”
2
3   #include ”primitives/push_constants.h”
4   #include ”primitives/uniform_buffer_object.h”
5   #include ”utils/image_utils.h”
6
7   #include <cstring>
8   #include <utility>
9
```

```
namespace nuff::renderer {

void Renderer::setContext(CoreCtx* ctx) {
    m_ctx = ctx;
}

void Renderer::setRenderTarget(IRenderTarget* renderTarget) {
    m_renderTarget = renderTarget;
}

void Renderer::setRecreateCallback(RecreateCallback callback) {
    m_recreateCallback = std::move(callback);
}

void Renderer::notifyFramebufferResized() {
    m_framebufferResized = true;
}

void Renderer::initialize() {
    auto& pipeline = m_ctx->component<PipelineCtxComponent>();
    m_renderTarget->initFrameResources(pipeline.descriptorSetLayout,
                                       sizeof(UniformBufferObject));
}

void Renderer::cleanup() {
    if (!m_ctx) return;
    m_ctx->device.waitIdle();
    if (m_renderTarget) m_renderTarget->cleanupFrameResources();
}

void Renderer::drawFrame(const FrameView& view) {
    auto beginResult = m_renderTarget->beginFrame();
    if (beginResult == IRenderTarget::FrameResult::Recreate) {
        if (m_recreateCallback) m_recreateCallback();
```

```
44          return;
45      }
46
47      recordRendering(view);
48
49      auto endResult = m_renderTarget->endFrame();
50      if (endResult == IRenderTarget::FrameResult::Recreate ||
        m_framebufferResized) {
51          m_framebufferResized = false;
52          if (m_recreateCallback) m_recreateCallback();
53      }
54  }
55
56  void Renderer::recordRendering(const FrameView& view) {
57      auto& pipeline = m_ctx->component<PipelineCtxComponent>();
58      auto& cmd = m_renderTarget->commandBuffer();
59      auto targetImage = m_renderTarget->image();
60      auto targetExtent = m_renderTarget->extent();
61
62      vk::CommandBufferBeginInfo beginInfo{};
63      cmd.begin(beginInfo);
64
65      auto preRenderBarrier = utils::createImageTransitionInfo(
66          targetImage,
67          vk::ImageLayout::eUndefined,
68          vk::ImageLayout::eColorAttachmentOptimal,
69          {},
70          vk::AccessFlagBits2::eColorAttachmentWrite,
71          vk::PipelineStageFlagBits2::eColorAttachmentOutput,
72          vk::PipelineStageFlagBits2::eColorAttachmentOutput
73      );
74      cmd.pipelineBarrier2(preRenderBarrier.dependencyInfo);
75
76      vk::ImageMemoryBarrier2 depthBarrier{
```

```
        .srcStageMask = vk::PipelineStageFlagBits2::eEarlyFragmentTests
            | vk::PipelineStageFlagBits2::eLateFragmentTests,
        .srcAccessMask = {},
        .dstStageMask = vk::PipelineStageFlagBits2::eEarlyFragmentTests
            | vk::PipelineStageFlagBits2::eLateFragmentTests,
        .dstAccessMask = vk::AccessFlagBits2::eDepthStencilAttachmentWrite
,
        .oldLayout = vk::ImageLayout::eUndefined,
        .newLayout = vk::ImageLayout::eDepthAttachmentOptimal,
        .srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
        .dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
        .image = m_renderTarget->depthImage(),
        .subresourceRange = {
            .aspectMask = vk::ImageAspectFlagBits::eDepth,
            .baseMipLevel = 0,
            .levelCount = 1,
            .baseArrayLayer = 0,
            .layerCount = 1
        }
    };
    vk::DependencyInfo depthDepInfo{
        .imageMemoryBarrierCount = 1,
        .pImageMemoryBarriers = &depthBarrier
    };
    cmd.pipelineBarrier2(depthDepInfo);

    vk::ClearValue clearColor{vk::ClearColorValue{std::array<float, 4>{0.0
f, 0.0f, 0.0f, 0.9f}}};
    vk::ClearValue clearDepth{vk::ClearDepthStencilValue{.depth = 1.0f, .
stencil = 0}};

    vk::RenderingAttachmentInfo attachmentInfo = {
        .imageView = m_renderTarget->imageView(),
        .imageLayout = vk::ImageLayout::eColorAttachmentOptimal,
```

```
        .loadOp = vk::AttachmentLoadOp::eClear,
        .storeOp = vk::AttachmentStoreOp::eStore,
        .clearValue = clearColor
    };

    vk::RenderingAttachmentInfo depthAttachmentInfo = {
        .imageView = m_renderTarget->depthImageView(),
        .imageLayout = vk::ImageLayout::eDepthAttachmentOptimal,
        .loadOp = vk::AttachmentLoadOp::eClear,
        .storeOp = vk::AttachmentStoreOp::eDontCare,
        .clearValue = clearDepth
    };

    vk::RenderingInfo renderingInfo = {
        .renderArea = { .offset = {0, 0}, .extent = targetExtent },
        .layerCount = 1,
        .colorAttachmentCount = 1,
        .pColorAttachments = &attachmentInfo,
        .pDepthAttachment = &depthAttachmentInfo
    };

    cmd.beginRendering(renderingInfo);
    cmd.bindPipeline(vk::PipelineBindPoint::eGraphics, *pipeline.
graphicsPipeline);

    cmd.setViewport(0, vk::Viewport{
        .width = static_cast<float>(targetExtent.width),
        .height = static_cast<float>(targetExtent.height),
        .maxDepth = 1.0f
    });
    cmd.setScissor(0, vk::Rect2D{.extent = targetExtent});

    UniformBufferObject ubo{};
    ubo.view = view.view;
```

```
141     ubo.proj = view.proj;
142     std::memcpy(m_renderTarget->currentUniformBufferMapping(), &ubo,
    sizeof(ubo));
143
144     cmd.bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
145                            *pipeline.pipelineLayout, 0,
146                            m_renderTarget->currentDescriptorSet(), nullptr
    );
147
148     for (const auto& item : view.items) {
149         if (!item.mesh) continue;
150         ObjectPushConstantData pc{ .model = item.model, .time = view.time
    };
151         cmd.pushConstants<ObjectPushConstantData>(*pipeline.pipelineLayout
    ,
152             vk::ShaderStageFlagBits::eVertex | vk::ShaderStageFlagBits::
    eFragment,
153             0, pc);
154         cmd.bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
155                                *pipeline.pipelineLayout, 1,
156                                item.materialSet, nullptr);
157         item.mesh->bind(cmd);
158         item.mesh->draw(cmd);
159     }
160
161     cmd.endRendering();
162
163     auto postRenderBarrier = utils::createImageTransitionInfo(
164         targetImage,
165         vk::ImageLayout::eColorAttachmentOptimal,
166         m_renderTarget->finalLayout(),
167         vk::AccessFlagBits2::eColorAttachmentWrite,
168         {},
169         vk::PipelineStageFlagBits2::eColorAttachmentOutput,
```

```
        vk::PipelineStageFlagBits2::eBottomOfPipe
    );
    cmd.pipelineBarrier2(postRenderBarrier.dependencyInfo);

    cmd.end();
}

} // namespace nuff::renderer
```

engine/presentation/i_render_target.h

```
#pragma once

#include <common/vk_common.h>

namespace nuff::renderer {

class IRenderTarget {
public:
    virtual ~IRenderTarget() = default;

    enum class FrameResult { Success, Recreate, Error };

    // Frame lifecycle
    virtual FrameResult beginFrame() = 0;
    virtual FrameResult endFrame() = 0;

    // Per-frame resources (valid between beginFrame/endFrame)
    virtual const vk::raii::CommandBuffer& commandBuffer() const = 0;
    virtual const vk::raii::ImageView& imageView() const = 0;
    virtual vk::Image image() const = 0;
    virtual vk::Extent2D extent() const = 0;
    virtual vk::Format format() const = 0;

```

```
    // Depth resources
    virtual const vk::raii::ImageView& depthImageView() const = 0;
    virtual vk::Image depthImage() const = 0;
    virtual vk::Format depthFormat() const = 0;

    // The layout the image should be transitioned to after rendering
    virtual vk::ImageLayout finalLayout() const = 0;

    // Frame indexing
    virtual uint32_t currentFrameIndex() const = 0;
    virtual uint32_t framesInFlight() const = 0;

    // Per-frame descriptor resources
    virtual void initFrameResources(const vk::raii::DescriptorSetLayout&
    layout,
                                    vk::DeviceSize uboSize) = 0;
    virtual void cleanupFrameResources() = 0;
    virtual vk::DescriptorSet currentDescriptorSet() const = 0;
    virtual void* currentUniformBufferMapping() const = 0;
};

} // namespace nuff::renderer
```

engine/presentation/offscreen_render_target.h

```
#pragma once

#include "i_render_target.h"
#include "frame_resources.h"
#include "context/ctx.h"

namespace nuff::renderer {

class OffscreenRenderTarget : public IRenderTarget {
```

```
public:
    OffscreenRenderTarget(CoreCtx* ctx, uint32_t width, uint32_t height,
                          vk::Format format = vk::Format::eB8G8R8A8Unorm,
                          bool exportable = false);
    ~OffscreenRenderTarget() override = default;

    void resize(uint32_t width, uint32_t height);

    FrameResult beginFrame() override;
    FrameResult endFrame() override;

    const vk::raii::CommandBuffer& commandBuffer() const override;
    const vk::raii::ImageView& imageView() const override;
    vk::Image image() const override;
    vk::Extent2D extent() const override;
    vk::Format format() const override;
    const vk::raii::ImageView& depthImageView() const override;
    vk::Image depthImage() const override;
    vk::Format depthFormat() const override;
    vk::ImageLayout finalLayout() const override;

    uint32_t currentFrameIndex() const override;
    uint32_t framesInFlight() const override;

    void initFrameResources(const vk::raii::DescriptorSetLayout& layout,
                            vk::DeviceSize uboSize) override;
    void cleanupFrameResources() override;
    vk::DescriptorSet currentDescriptorSet() const override;
    void* currentUniformBufferMapping() const override;

    VkImage nativeImage() const { return *m_image; }
    bool isExportable() const { return m_exportable; }

    int getMemoryFd() const;
```

```
    VkDeviceSize memorySize() const { return m_memorySize; }

private:
    void createResources();

    CoreCtx* m_ctx;
    uint32_t m_width;
    uint32_t m_height;
    vk::Format m_format;
    bool m_exportable;

    vk::raii::Image m_image{nullptr};
    vk::raii::DeviceMemory m_imageMemory{nullptr};
    vk::raii::ImageView m_imageView{nullptr};
    VkDeviceSize m_memorySize = 0;

    vk::Format m_depthFormat = vk::Format::eD32Sfloat;
    vk::raii::Image m_depthImage{nullptr};
    vk::raii::DeviceMemory m_depthImageMemory{nullptr};
    vk::raii::ImageView m_depthImageView{nullptr};

    vk::raii::CommandPool m_commandPool{nullptr};
    vk::raii::CommandBuffers m_commandBuffers{nullptr};
    vk::raii::Fence m_fence{nullptr};

    FrameResources m_frameResources;
};

} // namespace nuff::renderer
```

engine/presentation/swapchain/swapchain_render_target.h

```
#pragma once

```

```cpp
#include <presentation/i_render_target.h>
#include <presentation/frame_resources.h>
#include <context/ctx.h>

namespace nuff::renderer {

class SwapchainRenderTarget : public IRenderTarget {
public:
    explicit SwapchainRenderTarget(CoreCtx* ctx, uint32_t
   maxFramesInFlight = 2);
    ~SwapchainRenderTarget() override = default;

    void recreateResources();

    FrameResult beginFrame() override;
    FrameResult endFrame() override;

    const vk::raii::CommandBuffer& commandBuffer() const override;
    const vk::raii::ImageView& imageView() const override;
    vk::Image image() const override;
    vk::Extent2D extent() const override;
    vk::Format format() const override;
    const vk::raii::ImageView& depthImageView() const override;
    vk::Image depthImage() const override;
    vk::Format depthFormat() const override;
    vk::ImageLayout finalLayout() const override;

    uint32_t currentFrameIndex() const override;
    uint32_t framesInFlight() const override;

    void initFrameResources(const vk::raii::DescriptorSetLayout& layout,
                            vk::DeviceSize uboSize) override;
    void cleanupFrameResources() override;
    vk::DescriptorSet currentDescriptorSet() const override;
```

```
    void* currentUniformBufferMapping() const override;

private:
    void createResources();

    CoreCtx* m_ctx;
    uint32_t m_maxFramesInFlight;
    uint32_t m_currentFrame = 0;
    uint32_t m_imageIndex = 0;

    vk::raii::CommandPool m_commandPool{nullptr};
    vk::raii::CommandBuffers m_commandBuffers{nullptr};
    std::vector<vk::raii::Semaphore> m_imageAvailableSemaphores;
    std::vector<vk::raii::Semaphore> m_renderFinishedSemaphores;
    std::vector<vk::raii::Fence> m_inFlightFences;

    vk::Format m_depthFormat = vk::Format::eD32Sfloat;
    vk::raii::Image m_depthImage{nullptr};
    vk::raii::DeviceMemory m_depthImageMemory{nullptr};
    vk::raii::ImageView m_depthImageView{nullptr};

    FrameResources m_frameResources;
};

} // namespace nuff::renderer
```

engine/memory/memory_manager.h

```
#pragma once

#include ”context/ctx.h”

namespace nuff::renderer {

```

```
struct AllocatedBuffer {
    vk::raii::Buffer buffer{nullptr};
    vk::raii::DeviceMemory memory{nullptr};
};

struct AllocatedImage {
    vk::raii::Image image{nullptr};
    vk::raii::DeviceMemory memory{nullptr};
};

struct SingleTimeCommandContext {
    vk::raii::CommandPool commandPool{nullptr};
    vk::raii::CommandBuffers commandBuffers{nullptr};

    [[nodiscard]] const vk::raii::CommandBuffer& cmd() const { return
   commandBuffers[0]; }
};

class MemoryManager {
public:
    explicit MemoryManager(CoreCtx& ctx);

    AllocatedBuffer createBuffer(vk::DeviceSize size,
                                 vk::BufferUsageFlags usage,
                                 vk::MemoryPropertyFlags memoryProperties)
    const;

    AllocatedImage createImage(uint32_t width, uint32_t height,
                               vk::Format format,
                               vk::ImageTiling tiling,
                               vk::ImageUsageFlags usage,
                               vk::MemoryPropertyFlags memoryProperties)
   const;

```

```
    void copyBuffer(vk::Buffer src, vk::Buffer dst, vk::DeviceSize size)
    const;

    void transitionImageLayout(vk::Image image,
                               vk::ImageLayout oldLayout,
                               vk::ImageLayout newLayout) const;

    void copyBufferToImage(vk::Buffer buffer, vk::Image image,
                           uint32_t width, uint32_t height) const;

    [[nodiscard]] SingleTimeCommandContext beginSingleTimeCommands() const
    ;
    void endSingleTimeCommands(SingleTimeCommandContext& cmdCtx) const;

private:
    uint32_t findMemoryType(uint32_t typeFilter,
                            vk::MemoryPropertyFlags properties) const;

    CoreCtx& m_ctx;
};

} // namespace nuff::renderer
```

engine/scene/scene.h

```
#pragma once

#include "../entities/entity.h"
#include "scene_events.h"

#include <QtCore/qtclasshelpermacros.h>

#include <memory>
#include <string>
```

```
#include <unordered_map>
#include <vector>

namespace nuff::engine {

namespace events { class EventBus; }

class Scene {
public:
    explicit Scene(std::string name);
    ~Scene();

    Q_DISABLE_COPY(Scene)

    const std::string& name() const;
    void setName(std::string name);

    Entity* createEntity(std::string name, Entity* parent = nullptr);
    void    destroyEntity(EntityID id);
    Entity* find(EntityID id) const;

    void reparent(EntityID id, EntityID newParent);

    void setActiveCamera(EntityID id);
    Entity* activeCamera() const;

    void init();
    void update(float dt);

    std::vector<Entity*> rootEntities() const;
    const std::vector<std::unique_ptr<Entity>>& entities() const;

    events::EventBus& events();

```

```
private:
    void destroyRecursive(Entity* entity);

    std::string m_name;
    std::vector<std::unique_ptr<Entity>> m_entities;
    std::unordered_map<EntityID, Entity*> m_byId;
    EntityID m_nextId = 1;
    EntityID m_activeCameraId = 0;
    std::unique_ptr<events::EventBus> m_events;
};

} // namespace nuff::engine
```

Модуль editor (редакторський модуль)

editor/editor_app.h

```
#pragma once

#include <QObject>

#include "engine/core/app.h"
#include "presentation/offscreen_render_target.h"

namespace nuff::editor {

class EditorApp : public QObject, public app::App {
    Q_OBJECT

public:
    explicit EditorApp(QObject* parent = nullptr);
    ~EditorApp() override;

    void initialize(uint32_t vendorId, uint32_t deviceId,
```

```
                    uint32_t width, uint32_t height);

    void renderFrame();
    void resize(uint32_t width, uint32_t height) override;

    int           exportMemoryFd();
    VkDeviceSize memorySize() const;
    vk::Format   imageFormat() const;
    vk::Extent2D imageExtent() const;
    uint32_t     imageWidth() const;
    uint32_t     imageHeight() const;

protected:
    std::unique_ptr<engine::PlatformConfig> makePlatformConfig() override;
    std::string                             defaultScenePath() const
   override;

private:
    renderer::OffscreenRenderTarget* offscreenTarget() const;

    uint32_t m_vendorId = 0;
    uint32_t m_deviceId = 0;
    uint32_t m_width = 0;
    uint32_t m_height = 0;
};

} // namespace nuff::editor
```

editor/ui/VulkanViewport/vulkanviewport.h

```
#ifndef VULKANVIEWPORT_H
#define VULKANVIEWPORT_H

#include <QtQuick/QQuickItem>
```

```
#include <QtQuick/QQuickWindow>
#include <vulkan/vulkan.h>

#include ”editor/editor_app.h”

class VulkanViewport : public QQuickItem
{
    Q_OBJECT
    QML_ELEMENT
    Q_PROPERTY(nuff::editor::EditorApp* engine READ engine WRITE setEngine
    NOTIFY engineChanged)

public:
    VulkanViewport();

    nuff::editor::EditorApp* engine() const { return m_engine; }
    void setEngine(nuff::editor::EditorApp* engine);

signals:
    void engineChanged();

protected:
    QSGNode* updatePaintNode(QSGNode* oldNode, UpdatePaintNodeData*)
   override;
    void geometryChange(const QRectF& newGeometry, const QRectF&
   oldGeometry) override;

    void keyPressEvent(QKeyEvent* event) override;
    void keyReleaseEvent(QKeyEvent* event) override;
    void mousePressEvent(QMouseEvent* event) override;
    void mouseReleaseEvent(QMouseEvent* event) override;
    void mouseMoveEvent(QMouseEvent* event) override;

private slots:
```

```
    void cleanup();
    void handleWindowChanged(QQuickWindow* win);
    void onSceneGraphInitialized();
    void tryInitializeEngine();
    void onFrameSwapped();

private:
    void importImage();
    void cleanupImportedImage();

    nuff::editor::EditorApp* m_engine = nullptr;
    bool m_initialized = false;
    bool m_needsImport = false;

    uint32_t m_vendorId = 0;
    uint32_t m_deviceId = 0;

    VkDevice m_qtDevice = VK_NULL_HANDLE;
    VkImage m_importedImage = VK_NULL_HANDLE;
    VkDeviceMemory m_importedMemory = VK_NULL_HANDLE;
    uint32_t m_importedWidth = 0;
    uint32_t m_importedHeight = 0;
};

#endif // VULKANVIEWPORT_H
```

Модуль runtime (виконуваний модуль перегляду)

runtime/runtime_app.h

```
#pragma once

#ifndef GLFW_INCLUDE_VULKAN
#define GLFW_INCLUDE_VULKAN
```

```
#include <GLFW/glfw3.h>
#endif

#include "engine/core/app.h"

namespace nuff::runtime {

class RuntimeApp : public app::App {
public:
    RuntimeApp() = default;
    ~RuntimeApp() override;

    void run();

protected:
    std::unique_ptr<engine::PlatformConfig> makePlatformConfig() override;
    std::string                             defaultScenePath() const
   override;

private:
    static constexpr uint32_t DEFAULT_WIDTH = 800;
    static constexpr uint32_t DEFAULT_HEIGHT = 600;

    void initWindow();
    void mainLoop();
    void cleanup();

    static void framebufferResizeCallback(GLFWwindow* window, int width,
   int height);
    static void keyCallback(GLFWwindow* window, int key, int scancode, int
     action, int mods);
    static void mouseButtonCallback(GLFWwindow* window, int button, int
   action, int mods);
    static void cursorPosCallback(GLFWwindow* window, double xpos, double
```

```
    ypos);
35      static void scrollCallback(GLFWwindow* window, double xoffset, double
    yoffset);
36
37      GLFWwindow* m_window = nullptr;
38  };
39
40  } // namespace nuff::runtime
```