Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Системотехніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

«Проектування системи графічного представлення 3D середовищ»
(тема)

Виконав:
студент 2 курсу, групи ІТПм-22-1
Циба Є.М.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Освітня програма Інформаційні технології проектування
(повна назва освітньої програми)

Керівник Гребеннік І. В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ І.В. Гребеннік
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Системотехніки
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Інформаційні технології проектування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«»_____20____р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Цибі Єгору Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Проектування системи графічного представлення 3D середовищ
затверджена наказом університету від 20 листопада 2023 р. № 1373 Ст.
2. Термін подання студентом роботи до екзаменаційної комісії 18 листопада 2023 р.
3. Вихідні дані до роботи методи графічного представлення 3D середовищ, документація до Vulkan API, архітектури додатків для відображення 3D середовищ

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз предметної області
2. Теоретичні дослідження та проектування
3. Розробка архітектури та додатку-прототипу
4. Опис та скріншоти робочої програми

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)

процес растеризації трикутника, принцип роботи рейтрейсингу, концепція ECS на прикладі декількох сутностей, приклад архітектури з використанням ECS, модульна архітектура двигуна, принцип роботи модульної архітектури з використанням ECS

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Огляд предметної області	06.10 – 08.10	Виконано
2	Аналіз існуючих програмних застосунків для відтворення даних	08.10 – 16.10	Виконано
3	Теоретичні дослідження	16.10 – 12.11	Виконано
4	Створення модульної архітектури для додатку	12.11 – 25.11	Виконано
5	Створення додатку на основі новітньої архітектури	25.11 – 26.12	Виконано
6	Формування висновків та оформлення звіту	26.12 – 30.12	Виконано
7	Захист перед ЕК	12.01	Виконано

Дата видачі завдання 20 листопада 2023 р.

Студент _____
(підпис)

Керівник роботи _____ Гребеннік І.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Записка пояснювальна: 94 с., 27 рис., 28 джерел, 3 додатка.

ПРОЕКТУВАННЯ, РУШІЙ, МОДЕЛЮВАННЯ, ГРАФІЧНЕ ПРЕДСТАВЛЕННЯ, 3D СЕРЕДОВИЩА

Об’єкт дослідження – процес проектування систем графічного представлення 3D середовищ.

Предмет дослідження – розробка архітектури та програмного застосунку для графічного представлення 3D середовищ

Мета роботи – проектування та розробка архітектури для програмного застосунку графічного представлення 3D середовищ та створення прототипу за результатами проектування.

Методи дослідження – аналіз технічної літератури з даної предметної області, використання результатів теоретичних досліджень за заданою темою, створення архітектури додатка та практична реалізація для перевірки результатів.

Система повинна відповідати таким вимогам:

- відображення графічних середовищ різної складності;
- розширювана архітектура додатку

В роботі проведено аналіз наявних засобів та існуючих на даний момент інструментів, спроектована архітектура додатка, створений працюючий прототип.

ABSTRACT

Explanatory note: 94 pages, 27 figures, 28 references, 3 appendices.

DESIGN, ENGINE, MODELING, GRAPHICAL REPRESENTATION, 3D ENVIRONMENTS

The object of research is the process of designing a software application for graphical representation of 3D environments.

Subject of research - development of architecture and software application for graphical representation of 3D environments

Purpose - to design and develop an architecture for a software application for graphical representation of 3D environments and to create a prototype based on the design results.

Research methods - analysis of technical literature on this subject area, use of the results of theoretical research on a given topic, creation of the application architecture and practical implementation to verify the results. The system must meet the following requirements:

– display of graphical environments of varying complexity;
– extensible application architecture.

The work analyzes the available tools and existing tools, designs the application architecture, and creates a working prototype.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення

ЦП – центральний процесор

ГП – графічний процесор

CPU (Central Processor Unit) – центральний процесор

GPU (Graphical Processor Unit) – графічний процесор

3D (3-dimensional) – трьохвимірний простір

3D Rendering – це процес перетворення 3D-моделей у 2D-зображення на комп'ютері в тривимірній комп'ютерній графіці.

Rendering engine – частина рушія у програмі, що відповідає за графічне відтворення даних

ВСТУП

Сьогодні активне використання 3D технологій перетворилося на невід’ємну частину багатьох сфер життя – від графіки до медицини та архітектури. Проте, незважаючи на широке застосування, вибір ефективних систем для графічного представлення 3D середовищ залишається важливим та невирішеним завданням через величезний об'єм і складність обробки тривимірних даних.

Сфери застосування 3D технологій включають в себе:

– анімацію;
– комп'ютерні ігри;
– архітектурне моделювання та проектування;
– медицину (3D візуалізація анатомії тіла, планування хірургічних операцій);
– географічне інформаційне моделювання, виробництво (3D друк);
– наукові дослідження та освіту.

Актуальність цього дослідження обумовлена необхідністю забезпечення якісного і швидкого процесу створення, редагування та візуалізації 3D середовищ, що є ключовим для покращення розробки і реалізації 3D проектів.

Мета цієї роботи полягає в проектуванні системи графічного представлення 3D середовищ та створенні прототипу за отриманими результатами, враховуючи сучасні технічні можливості та вимоги. Дана кваліфікаційна робота буде виконана на основі аналізу існуючого досвіду програм для графічного представлення 3D середовищ. Буде проаналізовано ряд ключових існуючих систем, а їх переваги та недоліки будуть уважно враховані при проектуванні нової системи. Цей підхід допоможе уникнути поширених проблем та помилок у проектуванні, забезпечуючи високу якість,

продуктивність та зручність використання розробленої системи графічного представлення 3D середовищ.

В даному дослідженні основний акцент зроблено на процесах створення, редагування та візуалізації тривимірних об'єктів і сцен в комп'ютерних системах. Аналіз зосереджується на вивченні методів і засобів графічного представлення 3D середовищ.

З метою досягнення запланованих цілей дослідження використовуються різноманітні методи: аналіз науково-технічної літератури, комп'ютерне моделювання та експериментальні методи.

Наукова новизна розробки полягає у інтеграції сучасних технологій і методів обробки 3D даних, що дозволяє спроектувани ефективну систему для графічного представлення тривимірних об'єктів.

Практичне значення одержаних результатів полягає в можливості застосування розробленого прототипу системи в різних галузях, які активно використовують 3D графіку: в ігровій індустрії, кінематографії, архітектурі та дизайні інтер'єрів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд способів графічного представлення 3D середовищ

При розгляді способів графічного представлення 3D середовищ, слід зазначити, що вибір методу суттєво залежить від конкретних завдань і вимог. Деякі методи краще підходять для візуалізації великих ландшафтів, інші – для деталізації невеликих об'єктів. Далі будуть наведені способи графічного зображення 3D об’єктів та їх переваги і недоліки:

а) векторні методи:

- переваги: висока точність, масштабованість без втрати якості;
- недоліки: відносно великі обчислювальні затрати.

б) растрові методи:

- переваги: простота і швидкість обробки;
- недоліки: втрати якості при масштабуванні, обмежена деталізація.

в) полігональне моделювання:

- переваги: гнучкість, здатність детального відтворення об'єктів;
- недоліки: складність обробки великих об'єктів, високі вимоги до обчислювальних ресурсів. Для певних типів об’єктів можлива оптимізація використовуваних ресурсів [1], але в цілому проблема залишається суттєвою.

г) воксельне представлення:

- переваги: здатність детального відтворення об'єктів з складною геометрією.
- недоліки: великі обчислювальні затрати, вимога великих об'ємів пам'яті.

З огляду на широкий спектр існуючих методів графічного

представлення 3D середовищ, дана робота має на меті врахувати найкращі практики та техніки, щоб створити оптимальне та ефективне рішення для проектування системи графічного представлення 3D середовищ для ігор, а саме ігрових двигунів.

1.2 Аналіз існуючих програмних застосунків для відтворення даних

Області використання графічного представлення 3D даних широкі та різноманітні, відображаючи велике різноманіття сфер діяльності, де 3D моделювання відіграє ключову роль. Далі буде наведена інформація про найбільш актуальні сфери застосування та програмні застосунки, що використовуються у цих сферах.

У кіноіндустрії та анімації графічне представлення 3D даних активно використовується для створення комп'ютерної графіки в кіно та анімаційних фільмах. Це дозволяє реалістично зображати об'єкти, які було б складно або неможливо зняти за допомогою традиційного кіно. Найбільш популярним пакетом для кіноіндустрії є Houdini від компанії Side FX.

Houdini [2] — це передове програмне забезпечення для 3D анімації та візуальних ефектів, що використовується для створення складних 3D сцен, динамічних симуляцій, процедурного моделювання та іншого. Інтерфейс додатка можна побачити на рис 1.1. Архітектура Houdini:

- Houdini використовує унікальну вузлову структуру, що дозволяє артистам працювати в максимально необмеженому та гнучкому середовищі. Всі елементи сцени (геометрія, анімація, освітлення і т.д.) представлені у вигляді вузлів, які можна з’єднати та налаштувати;
- Houdini пропонує процедурний підхід до моделювання, що надає

необмежені можливості для зміни дизайну в будь-який момент проекту;

– надає потужні інструменти для симуляції рухомих об’єктів, динаміки рідин та газів, розрушення та інших.

Houdini Engine [3] дозволяє інтегрувати Houdini в інші проектні додатки. Він дозволяє використовувати звичайні асети Houdini в інших 3D програмах, реалізуючи їх потужність та гнучкість в уже звичному для розробників середовищі. Архітектура Houdini Engine:

– Houdini Engine має API [4] для інтеграції з іншими 3D додатками, що дозволяє розробникам легко інтегрувати його в свої проекти;

– завдяки відкритій архітектурі, Houdini Engine може бути адаптований до будь-якого програмного забезпечення, що підтримує API;

– Houdini Engine підтримує параметричні асети Houdini, що дозволяє розробникам створювати гнучкі та конфігуруємі рішення для своїх проектів В сучасному геймдеві 3D графіка відіграє ключову роль. Вона дозволяє створювати реалістичні, захоплюючі ігрові світи та персонажів, надаючи гравцям можливість зануритися в атмосферу гри. Графіка у 3D іграх включає в себе відтворення об'єктів, текстур, освітлення, тіней, рефлексії та інших візуальних ефектів, які допомагають створити вірогідний ігровий світ.

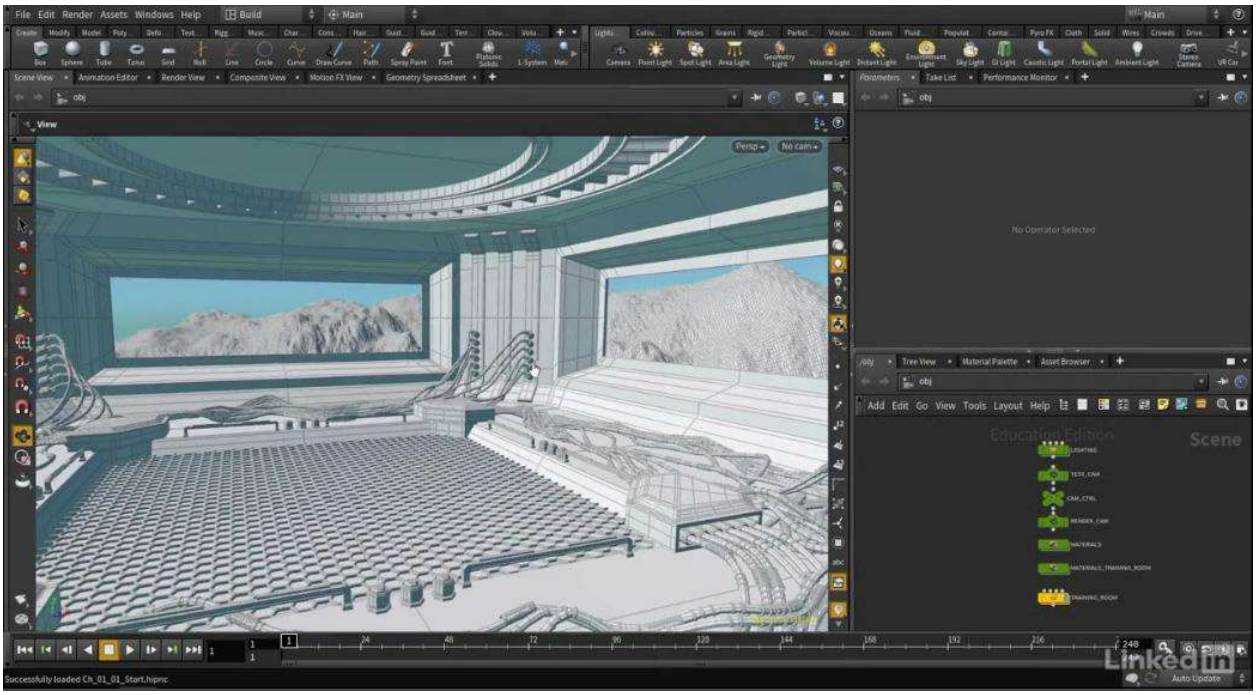

Рис 1.1 – Інтерфейс пакету для 3D моделювання Houdini

Unreal Engine [5] — це один з найпопулярніших ігрових двигунів для розробки 3D ігор, розроблений компанією Epic Games. Він використовується для створення великої кількості ігор, від інді до трипл-А проектів. Особливості Unreal Engine:

- Unreal Engine відомий своєю високою продуктивністю та здатністю обробляти великі і складні 3D сцени (рис 1.2);
- Unreal Engine пропонує систему візуального скриптингу (Blueprint), яка дозволяє розробникам створювати логіку гри без прямого написання коду;
- Unreal Engine побудований на єдиній архітектурі (monolith architecture), що забезпечує гладкість та спрощеність використання, а також полегшує процес розробки;
- розробники можуть працювати з двигуном, використовуючи як C++,

так і Blueprint, обравши зручний для себе спосіб;

– Unreal Engine сумісний з багатьма іншими платформами та інструментами розробки і легко інтегрується з іншими системами.

Рис 1.2 – 3D сцена створена з використанням Unreal Engine 4

Медицина є однією з ключових сфер використання 3D графіки для деталізованого аналізу анатомічних структур та планування хірургічних операцій. 3D Slicer [6] — це відкрите програмне забезпечення для медичної візуалізації та обчислювальної хірургії. Його основний акцент — на гнучкій і модульній архітектурі, що дозволяє швидко адаптуватися до специфічних медичних завдань. Інтерфейс програми можна побачити на рис 1.3. Особливості 3D Slicer:

– 3D Slicer відзначається здатністю ефективно аналізувати мультимодальні медичні дані та надавати інструменти для 3D

візуалізації анатомічних структур та планування хірургічних операцій;

– архітектура 3D Slicer базується на модульності, що дозволяє користувачам та розробникам легко додавати нові функціональні модулі та інструменти згідно з потребами медичних фахівців;
– завдяки відкритому коду і активній спільноті, 3D Slicer постійно розширюється та удосконалюється;
– підтримка різних операційних систем дозволяє використовувати 3D Slicer на більшості сучасних комп'ютерних платформ;
– 3D Slicer може інтегруватися з іншими медичними системами та додатками, що значно розширює його функціонал і можливості застосування в клінічній практиці.

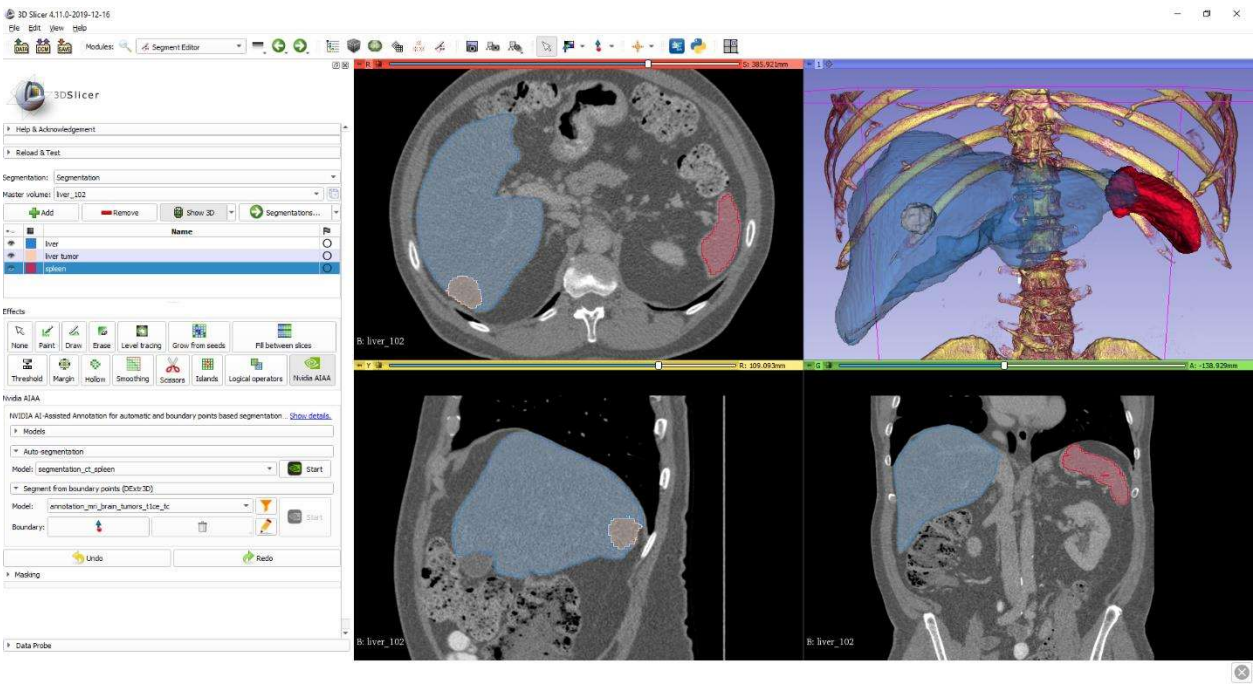

Рис 1.3 – Інтерфейс програми 3D Slicer

1.3 Огляд технічних засобів графічного зображення 3D середовищ

При проектуванні системи графічного представлення 3D середовищ, необхідно ретельно обрати технічні засоби та методи, які забезпечать оптимальну реалізацію проекту. Значимість даного вибору визначається специфікаціями та вимогами до функціональності майбутньої системи.

Розглядаючи аспекти рендерингу, первісно важливо обговорити особливості та відмінності між CPU та GPU рендерингом. Центральний процесор (CPU) характеризується великою універсальністю та сумісністю, проте має обмежені можливості щодо обробки графічної інформації. В той час як графічний процесор (GPU) спеціалізується на високоефективній обробці графіки, пропонуючи значно вищу продуктивність для завдань візуалізації.

При проектуванні системи графічного представлення 3D середовищ, значущим етапом є вибір оптимального графічного API. Основоположні характеристики та відмінності трьох поширених API подано нижче у формі списку:

- OpenGL [7]: кросплатформений API (можливість використання на різних операційних системах), має високу гнучкість та великий набір функціональності та налаштувань, але має меншу продуктивність для деяких завдань порівняно з іншими API;
- DirectX [8]: є ексклюзивним для продуктів Microsoft: оптимізований для Windows та Xbox, забезпечує відмінну продуктивність на підтримуваних платформах, але має обмежену сумісність з іншими операційними системами;
- Vulkan [9]: має високу продуктивність, оптимізацований для сучасного апаратного забезпечення, підримує багатьох платформ та операційних систем, але має вищу складність розробки через низькорівневі

можливості.

Останній пункт – вибор мови програмування для реалізації системи. Можливі такі варіанти як C++, Python або Rust. C++ надає можливість використовувати повний потенціал апаратного забезпечення, що є критично для завдань, пов’язаних з обробкою 3D-графіки. Python є зручним для швидкої розробки та експериментації, а Rust пропонує високу продуктивність та безпеку типів.

1.4 Постановка задачі

На основі проведеного огляду предметної області сформулюємо постановку задачі кваліфікаційної роботи. Спроєктувати архітектуру та здійснити програмну реалізацію додатку для відтворення ігрових 3D середовищ з використанням графічного API Vulkan та сучасних технологій проектування. Проєктована архітектура має дозволяти легко інтегрувати та розширювати двигун в майбутньому, гарантувати її адаптивність до різних ігрових проектів та платформ.

Проектований додаток буде мати модульну архітектуру, що дозволить легко розширювати його функціонал та встроювати в різні ігрові проекти, адаптуючи під специфічні потреби та вимоги. При проектуванні будуть взяті до уваги історія змін популярних двигунів для ігор, для того щоб гарантувати відсутність проблем, з якими зустрічались розробники у минулому.

Практична реалізація має ціль перевірити результати теоретичних досліджень, для неї будуть використані наступні технології:

– мова програмування C++: забезпечить високу продуктивність, гнучкість та потужний функціонал для розробки високопродуктивних додатків. Ця мова дозволяє створювати оптимізований, швидкий та надійний код,

що є критично важливим для ефективного відтворення 3D графіки;

– графічний API Vulkan: надасть повний контроль над GPU, дозволить значно оптимізація процесу графічного відтворення та забезпечить широка сумісність з сучасними відеокартами та операційними системами;
– розроблена система повинна підтримуватись на операційних системах Windows 10/11, Linux та Mac OS;
– розмір скомпільованого коду двигуна не має перевищувати 30 мегабайтів та повинен мати можливість відключити не використані модулі заради подальшого зменшення розміру.

2 ТЕОРЕТИЧНІ ДОСЛІДЖЕННЯ

2.1 Аналіз методів графічного представлення 3D середовищ

Растрове зображення даних – це процес перетворення 3D геометричних об'єктів на 2D піксельне зображення для відображення на дисплеї. Це стає можливим завдяки розбиттю 3D об'єктів на множину примітивів (зазвичай трикутників), які потім "зображуються" на 2D растр – сітку пікселів екрана.

Однією з ключових особливостей растрового зображення є його дискретизація: 3D сцена перетворюється на сітку пікселів, де кожен піксель має певний колір. В порівнянні з іншими техніками, растрове зображення вимагає менш складних математичних операцій і є одним з найшвидших методів графічного представлення, особливо з використанням сучасного апаратного обладнання.

Процес створення растрового зображення [10] можна розглядати як послідовність трьох основних дій:

– трансформація: 3D координати об'єктів сцени переводяться в координати 2D відносно вікна перегляду. Це зазвичай включає проекційні та модельно-видові трансформації;
– оклюзія (скриття об'єктів): визначається, які частини об'єктів видимі, а які ні. Для цього можуть використовуватися алгоритми, такі як Z-буферинг;
– заповнення: одержані 2D примітиви "заповнюються" колором. В цьому процесі враховуються різні фактори, такі як текстури, освітлення та інші візуальні ефекти, як можна побачити на рис. 2.1.

Цей метод широко застосовується завдяки своїй спроможності забезпечувати високу швидкість відображення в реальному часі на сучасних

графічних адаптерах [11].

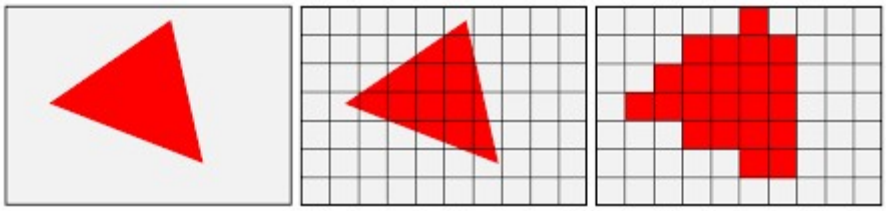

Рис. 2.1 – процес растеризації трикутника

Растрове зображення широко використовується в ігровій індустрії, де головний акцент робиться на швидкість. Ігри потребують відтворення графіки в реальному часі, і растрова графіка дозволяє досягти цього завдяки її оптимізації під сучасне апаратне обладнання. Крім ігор, растрове зображення також використовується в інтерфейсах користувача, мобільних застосунках і веб-графіці.

Переваги та недоліки методу:

– одна з найшвидших технік графічного представлення, особливо на оптимізованому обладнанні;
– сучасні графічні процесори розроблені специфічно для підтримки растрового зображення;
– з порівняння з методами, які симулюють взаємодію світла, растрова графіка менш складна у реалізації.

Недоліки:

– менший реалізм: не враховує складних взаємодій світла, які можна бачити в інших техніках;
– аліасинг: через дискретизацію можуть виникати візуальні артефакти, такі як "ламаність" країв об'єктів.

Наступна техніка для графічного представлення даних це рейтрейсинг (raytracing) [12] – це техніка графічного представлення, яка симулює взаємодію світла з об'єктами в сцені шляхом прослідковування віртуальних променів від спостерігача до джерел світла. Відрізняючись від растрового зображення, яке зосереджується на перетворенні 3D об'єктів на 2D пікселі, рейтрейсинг враховує як світло відбивається, заломлюється та поглинається об'єктами для створення фотореалістичних зображень (рис 2.2).

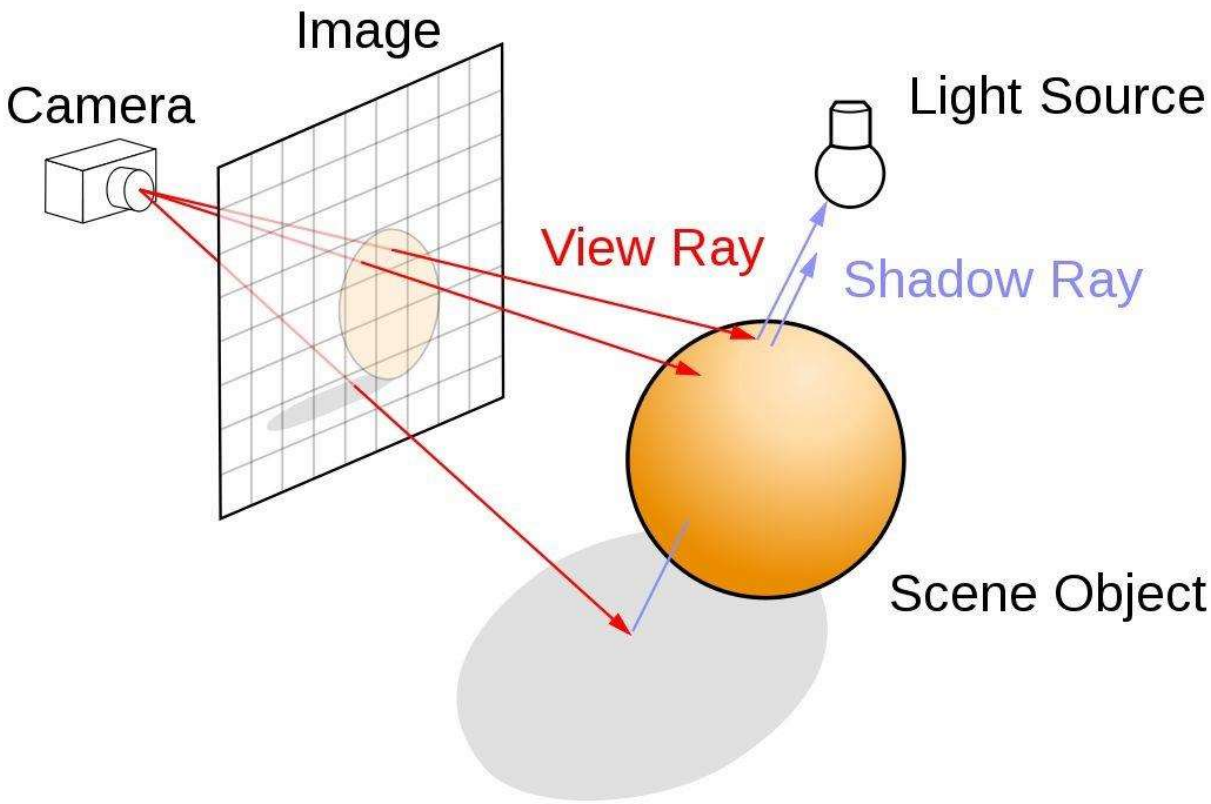

Рис 2.2 – принцип роботи рейтрейсингу

Основна відмінність рейтрейсингу полягає у його способі симуляції світла. Замість прямого відображення об'єктів на пікселі, як у растровому

зображенні, рейтрейсинг розглядає, як світло взаємодіє з матеріалами і формами в сцені, щоб створити зображення. Це дозволяє досягти високого реалізму, враховуючи такі ефекти, як відблиски, прозорість та тіні.

Переваги:

– фотореалізм: можливість досягнення високого рівня реалізму завдяки симуляції взаємодії світла з об'єктами;
– комплексні ефекти: автоматичне моделювання відблисків, прозорості, заломлення світла та інших оптичних явищ.

Недоліки:

– витрати на обчислення: техніка є обчислювально інтенсивною, що може призвести до зниження швидкості відтворення;
– складність: рейтрейсинг вимагає більш складних алгоритмів та оптимізацій для ефективної реалізації.

Пастрейсинг – це метод графічного представлення, який є розширенням рейтрейсингу і симулює взаємодію світла з об'єктами в сцені за допомогою віртуальних променів. Там, де рейтрейсинг зазвичай відслідковує один промінь для кожного пікселя і обмежується декількома відбиваннями, пастрейсинг прослідковує множинні промені, що випромінюються від кожного пікселя та інтегрує їх результати для отримання кінцевого зображення. Це дозволяє досягти глибокого і реалістичного освітлення, відображаючи такі явища, як глобальне освітлення. На рис 2.3 можна побачити порівняння вихідного зображення у складній 3D сцені для різних методів графічного представлення. Більш детально алгоритм роботи пастрейсингу було розглянуто у статті [13], зокрема, автори змогли імплементувати систему глобального освітлення у режимі реального часу.

Пастрейсинг часто використовується в областях, де потрібний найвищий рівень реалізму, таких як анімаційне кіно, архітектурна візуалізація

та дослідницька графіка.

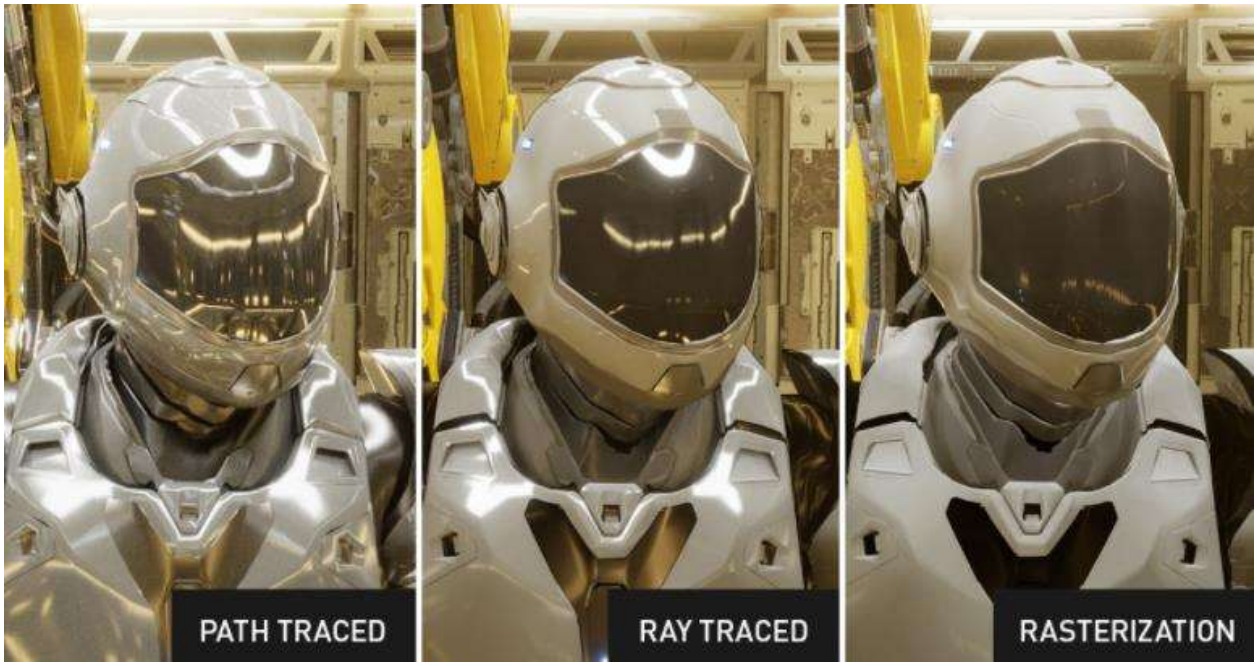

Рис 2.3 – Графічне зображення складної 3D сцени з використанням методів растеризації, рейтрейсингу та пастрейсингу

Рейтрейсинг та пастрейсинг традиційно використовувався в областях, де важливий високий реалізм зображень, наприклад, в комп'ютерній графіці, рекламі, архітектурних візуалізаціях та кіно. Однак вже існують методи, що дозволяють використовувати ці техніки у режимі реального часу [14], тому можлива також адаптація рейтрейсингу для ігрових двигунів.

Останньою технікою, яку буде розглянуто, буде метод оптичного представлення. Фізична оптика є галуззю оптики, яка зосереджується на вивченні взаємодії світла з матеріалами на молекулярному та атомному рівнях. Це дозволяє моделювати складні оптичні явища, такі як дифракція, інтерференція та поляризація. У контексті графічного представлення 3D

середовищ, фізична оптика може бути використана для створення реалістичних зображень, враховуючи складний характер світлової взаємодії.

Основні принципи та характеристики:

– дифракція: явище згинання світлових променів на границі двох середовищ, що має місце при проходженні світла через маленькі отвори або їх розсіюванні на мікроскопічних об'єктах;
– інтерференція: сумування чи віднімання амплітуд двох або більше хвиль, що перекриваються в просторі;
– поляризація: орієнтація площини коливань електромагнітної хвилі в певному напрямку.

Фізична оптика використовується в графічному представленні для створення реалістичних зображень, особливо коли необхідно відтворити ефекти, які виникають від взаємодії світла з різними матеріалами. Це може включати зображення дифракційних явищ у воді, інтерференційних малюнків на мильних бульбашках або поляризоване світло, що відбивається від поверхонь або проходить через призму (рис 2.4).

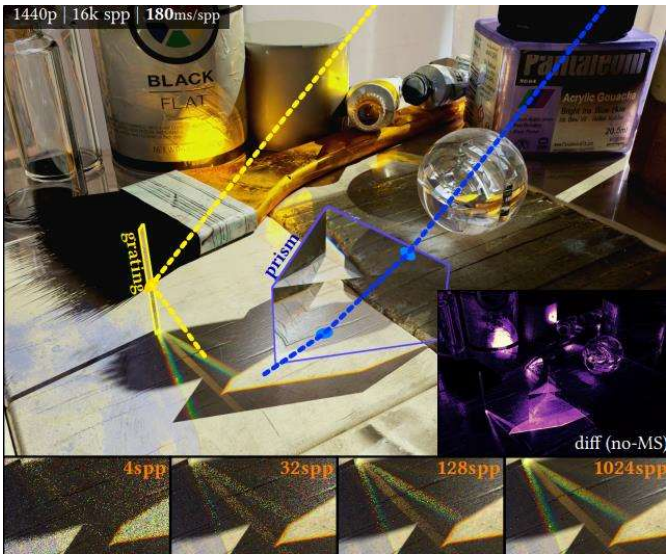

Рис 2.4 Приклад роботи графічного представлення даних на основі фізичних явищ

Переваги:

– висока реалістичність: здатність до точного моделювання світлових явищ дозволяє створювати дуже реалістичні зображення;
– інтеграція з іншими методами: фізична оптика може бути поєднана з іншими методами графічного представлення для досягнення комплексних ефектів.

Недоліки:

– високі обчислювальні витрати: точне моделювання світлових явищ може бути обчислювально вимогливим;
– складність реалізації: для коректного моделювання явищ потрібно глибоке розуміння фізики світла та математичних методів. На даний момент існують лише декілька стабільних імплементацій, одна з яких була досліджена у статті [15].

Після детального розгляду різних методів графічного представлення,

включаючи растеризацію, рейтрейсинг, пастрейсинг та фізичну оптику, можна зробити висновок щодо оптимального підходу для цієї конкретної роботи.

Оскільки головна мета цієї роботи полягає у проектуванні 3D двигуна, найбільш доцільним варіантом є растеризація. Вона забезпечує найкращу комбінацію швидкості та якості для реалізації в реальному часі, яка є критичною для 3D двигунів. Хоча інші методи, такі як рейтрейсинг або фізична оптика, можуть забезпечувати вищу якість зображення в певних сценаріях, вони часто вимагають значно більше обчислювальних ресурсів та часу на рендеринг.

Тому, враховуючи потреби та обмеження, пов'язані з проектуванням 3D двигуна, можна стверджувати, що растеризація є найбільш підходящим варіантом для досягнення бажаних результатів в рамках цієї роботи.

2.2 Структурна організація та ключові компоненти 3D двигунів для ігор

3D двигун — програмне забезпечення для створення, рендерингу та анімації 3D об'єктів. Він використовується в галузях як відеоігри, кіно та архітектура. Сучасні двигуни є високофункціональними.

Основні компоненти:

- графічний рендерер: візуалізує 3D об'єкти, враховуючи геометрію, текстури та світло;
- фізичний двигун: симулює фізичні взаємодії, такі як гравітація та зіткнення;
- звуковий двигун: управляє створенням та відтворенням звуку;
- система введення/виведення: обробляє ввод користувача;
- AI двигун: надає "інтелект" NPC в іграх.

Ці компоненти взаємодіють між собою, створюючи багатошарову систему, яка може відтворювати складні і реалістичні віртуальні середовища, тому на даний момент існують декілька архітектур для 3д двигунів.

Монолітна архітектура — це підхід до проектування програмного забезпечення, де всі компоненти системи об'єднані в одному місці і працюють як єдине ціле. У контексті ігрових двигунів це означає, що різні аспекти двигуна, такі як обробка графіки, фізична симуляція, звукова система та інтерфейс користувача, всі вбудовані безпосередньо в основний код двигуна.

Як зазначено у [16], Найважливішою перевагою монолітної архітектури є її простота - у порівнянні з розподіленими додатками, монолітні набагато легше тестувати, розгортати, налагоджувати та контролювати.

Однак така централізована структура також може призвести до деяких недоліків. Монолітні системи часто виявляються складними для модифікації або розширення, оскільки будь-які зміни в одній частині системи можуть мати вплив на інші частини. Це також може ускладнити процес тестування, оскільки потрібно враховувати взаємодію між різними компонентами.

Все вказане робить монолітну архітектуру досить потужною для проектів, де вимоги заздалегідь відомі та стабільні, але менш підходящою для динамічних проектів, де вимоги можуть часто змінюватися.

Монолітна архітектура була досить популярною у ранніх етапах розвитку ігральної індустрії, коли ресурси були обмежені, а двигуни не вимагали такої великої гнучкості, як сьогодні. Таку архітектуру мали двигуни Quake Engine [17] та GoldSrc [18].

В свою чергу, компонентна архітектура є сучасним підходом до проектування програмного забезпечення, який зосереджується на розділенні системи на окремі, незалежні компоненти. В контексті ігрових двигунів, таких як Unity, Unreal Engine 4, Godot та CryEngine, це означає, що різні аспекти

двигуна, такі як обробка графіки, фізика, звук і інше, розроблені як окремі модулі або компоненти. Ці компоненти можуть легко взаємодіяти між собою за допомогою визначених інтерфейсів, дозволяючи розробникам додавати, змінювати або видаляти окремі частини двигуна без втручання в інші частини системи.

У роботі [19] автор пояснює технічні основи компонентної архітектури – технології, яка базується на існуючих об'єктно-орієнтованих підходах до створення готових програмних компонентів для багаторазового використання та вбудовування у великі додатки.

Основною перевагою такого підходу є його гнучкість. Розробники можуть легко адаптувати систему до нових вимог, додавати нові функції або видаляти застарілі без потреби переписування великої частини коду. Однак існують і виклики. Наприклад, може виникнути додаткова накладна робота при інтеграції компонентів через потребу в узгодженні дій між ними.

На завершення, компонентна архітектура є популярним вибором для сучасних ігрових двигунів завдяки її гнучкості та модульності.

Якщо ж говорити саме про ігрові двигуни, то існує специфічна для цієї області архітектура – ECS (Entity Component System). Це архітектурний патерн у програмуванні, який став дуже популярним у розробці відеоігор завдяки його високій гнучкості та ефективності. ECS розділяє логіку та дані, що сприяє чистому коду, оптимізації та легкості розширення:

– Entity: сутність є основною одиницею об'єкта в системі. Це, зазвичай, простий ідентифікатор, який представляє об'єкт у грі (наприклад, герой, монстр, зброя тощо);
– Component: компонент представляє собою "сумку даних", яка містить конкретні атрибути або характеристики об'єкта. Наприклад, компонент "Положення" може містити координати x, y та z. Важливо зауважити, що

компоненти не містять логіку, лише дані;

– System: система відповідає за обробку логіки гри. Вона працює з компонентами, виконуючи операції на основі даних, які вони містять. Наприклад, система руху може змінювати положення об'єкта на основі його швидкості та напрямку (рис 2.5).

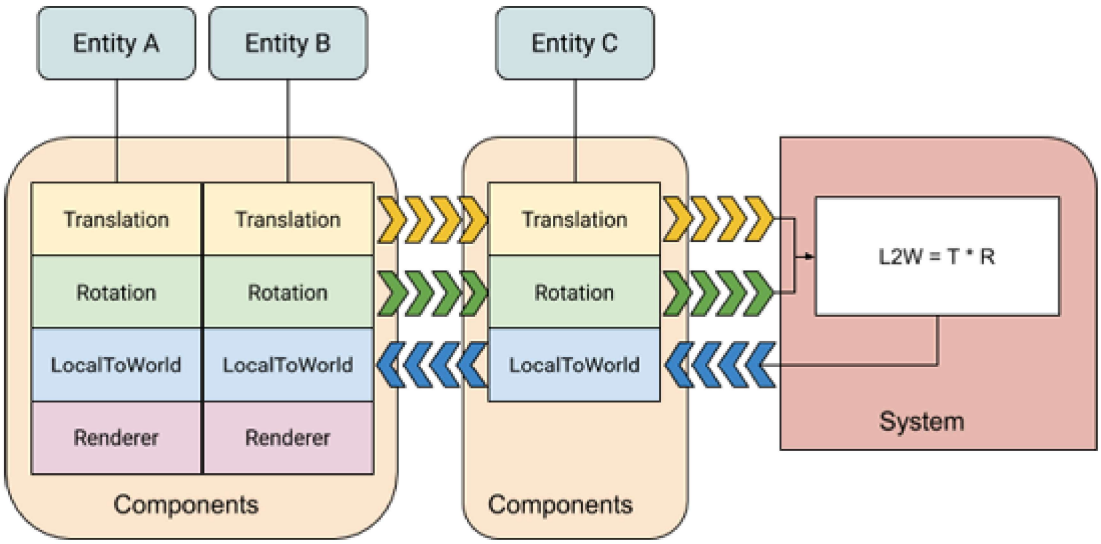

Рис 2.5 Концепція ECS на прикладі декількох сутностей

Переваги ECS:

– гнучкість: легко додавати, видаляти або змінювати компоненти об'єктів на льоту, що робить систему дуже адаптивною;
– оптимізація: завдяки чіткому розділенню даних та логіки, ECS часто дозволяє краще використовувати ресурси системи, особливо в багатопотокових додатках;
– масштабованість: завдяки своїй модульної структурі, ECS легко масштабується, дозволяючи легко додавати нові функції або оптимізувати існуючі.

Деякі сучасні ігрові двигуни, такі як Unity, вже включають [20] або рухаються в напрямку використання ECS у своїх архітектурах через його численні переваги у розробці ігор.

2.3 Архітектурні підходи до розробки рендерерів

Рендеринг є однією з найбільш критичних складових будь-якої графічної системи, особливо в контексті відеоігор, де відгук в реальному часі є ключовим. Архітектура рендерера визначає, як інформація про сцену перетворюється на пікселі на екрані, визначаючи при цьому якість зображення, продуктивність, а також гнучкість та розширюваність системи. На сьогоднішній день ця тема викликає багато дискусій, наприклад автори [21] ставлять під питання загальноприйняту практику робити великі пайплайни рендерингу, а команда [22] закликає збільшити кількість досліджень у цьому напрямку. У цьому підрозділі ми розглянемо ключові архітектурні підходи, їхні основні концепції та вплив на процес рендерингу. Це допоможе зрозуміти, як вибрати найкращий підхід для конкретних потреб і завдань, а також як оптимізувати роботу рендерера в різних умовах.

Монолітна архітектура рендерера полягає у виконанні всього процесу рендерингу в рамках одного великого блоку коду. Від завантаження та обробки моделей до фінального виводу зображення на екран, всі етапи рендерингу тісно взаємодіють і інтегровані в цю єдину структуру.

Однією з ключових переваг монолітної архітектури є її простота розуміння та реалізації. Це може спростити процес розробки, особливо для менших проектів або проектів з конкретними технічними вимогами, де можлива оптимізація під конкретні завдання.

Однак така архітектура також має свої обмеження. Внесення змін або додавання нових функцій може стати викликом, оскільки потрібно буде робити зміни в багатьох частинах коду. Крім того, монолітний підхід може бути менш гнучким у відповіді на зміни в технологіях або вимогах до продуктивності. На сьогоднішній день популярні двигуни не використовують таку архітектуру.

Подієва архітектура рендерера базується на концепції відгуку на події або сигнали, які виникають під час процесу рендерингу. Замість послідовного виконання задач або операцій, як у традиційних пайплайнах, подієва архітектура реагує на певні події, що виникають під час рендерингу.

Основна ідея полягає в тому, що різні елементи рендерера слухають або "чекають" на певні події. Коли така подія виникає, відповідний компонент або модуль рендерера активується та виконує свої задачі.

Особливості:

– відгук на зміни: якщо, наприклад, світловий джерело змінює своє положення, рендерер може відгукнутися на цю подію, перерахувавши освітлення сцени;
– модульність: компоненти можуть бути незалежними і реагувати лише на певні події, що дозволяє легше додавати або змінювати функціональність;
– оптимізація: подієва архітектура може бути ефективною, оскільки обчислення виконуються лише тоді, коли це дійсно необхідно, у відгуку на конкретні події.

Однак такий підхід також має свої виклики:

– комплексність: управління та налагодження системи, що реагує на події, може бути складнішим порівняно з традиційними пайплайнами;

– затримки: існує ризик затримок, якщо декілька подій виникає одночасно або якщо відгук на подію є дуже обчислювально інтенсивним.

На даний момент подієва архітектура рендерера не є стандартом у галузі графічних двигунів, але її принципи можуть бути використані в комбінованих системах або у специфічних випадках, де потрібна велика гнучкість або реактивність на динамічні зміни у сцені.

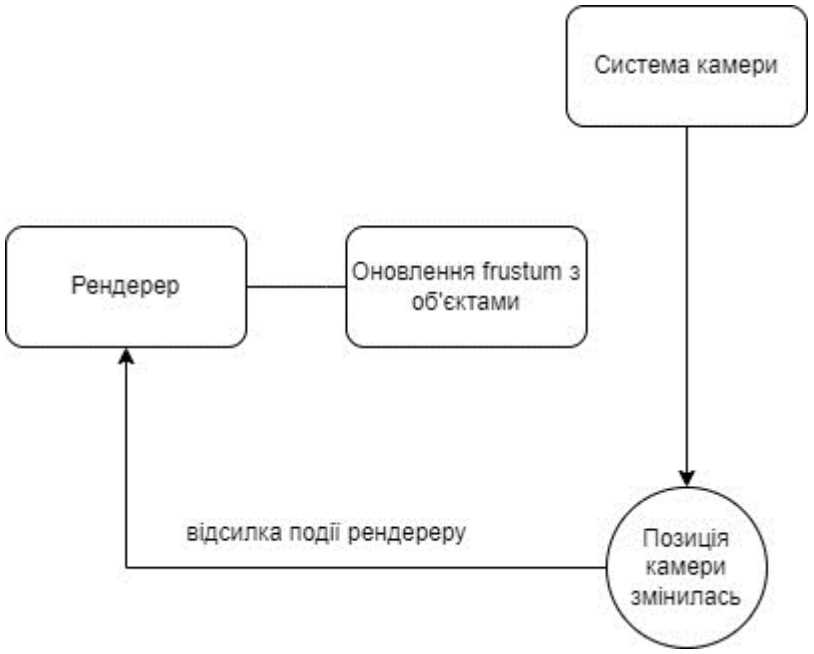

Рис 2.7 Приклад роботи подієвої архітектури рендереру

Сучасним способом організовувати двигуни для графічного представлення є рендер-графи.

Рендер-графи представляють собою концептуальний підхід до рендерингу, де весь процес рендерингу представлений як граф, з вузлами, що відображають операції рендерингу, та зв'язками, які вказують на послідовність та залежності цих операцій. Вони були вперше представлені авторами [23] і

швидко набрали популярність через високу модульність, зрозумілість та відносно просту реалізацію.

Математично, рендер-графи можна розглядати як напрямлені ациклічні графи (DAG). Вузли у такому графі представляють різні етапи рендерингу (наприклад, обробка тіней, растеризація, пост-обробка тощо), тоді як зв'язки відображають поток даних між цими етапами.

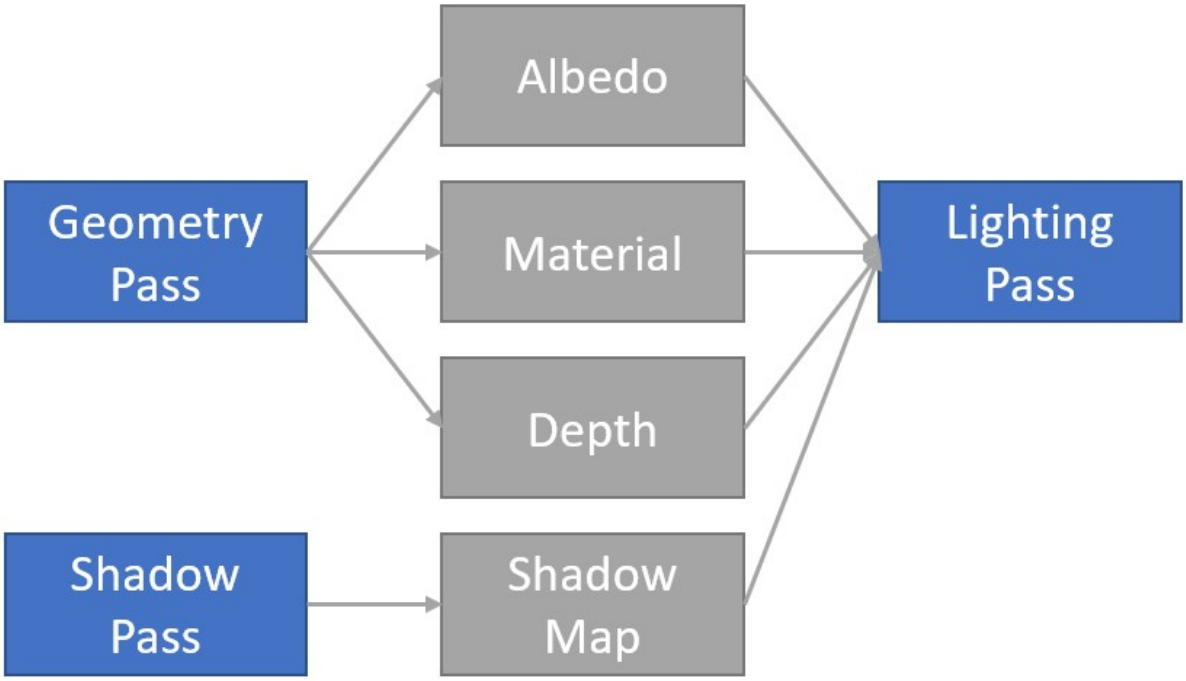

Рис 2.6 – Приклад рендер-графу

Процесс створення графу:

– на початковому етапі необхідно визначити всі операції рендерингу, які будуть представлені у графі;
– для кожної операції визначається, від яких інших операцій вона залежить. Наприклад, пост-обробка може залежати від попереднього етапу растеризації;

– за допомогою аналізу графа можливо визначити та видалити непотрібні операції або оптимізувати послідовність операцій для підвищення продуктивності.

Після створення, рендер-граф готовий до використання і для графічного відтворення даних потрібно лише обійти граф у ширину.

Представлення рендер-пайнплайну у вигляді графу надає багато переваг, зокрема:

– видалення непотрібних вузлів: така операція може здатися дивною, але під час розробки часто додають спеціальні вузли куда потрапляє інформація для подальшої відладки, і такі вузли автоматично видаляються з графу якщо включена оптимізація;
– паралелізація завдань: деякі завдання у рендер-графі можуть виконуватися паралельно. Використовуючи сучасні графічні API, такі як Vulkan або DirectX 12, можна ефективно розподілити завдання між різними ядрами процесора чи навіть різними GPU та зменшити кількість атомарних операцій між вузлами графічного процесора;
– оптимізація порядку вузлів: Правильний порядок виконання вузлів може зменшити завантаження на пам'ять, зменшуючи необхідність у проміжних буферах або дозволяючи ранньому відсіву невидимих об'єктів;
– використання менших буферів: Замість використання великих буферів для всіх вузлів, можна оптимізувати розміри буферів в залежності від конкретних потреб кожного вузла;
– злиття вузлів: у деяких випадках, замість використання декількох окремих вузлів, можна їх об'єднати в один вузол, що виконує кілька завдань одночасно. Це може зменшити накладні витрати на переключення контексту між завданнями;

– кешування результатів: якщо деякі вузли виконують однотипні розрахунки з рефреш-рейтами, які менші за частоту кадрів, результати цих розрахунків можна кешувати, щоб використовувати їх у наступних кадрах без повторного обчислення;
– адаптивний рендеринг: в залежності від складності сцени в графі можна відключати певні вузли, що дозволить значно оптимізувати загальне виконання.

На практиці рендер-графи стали популярними у сучасних графічних двигунах, таких як Frostbite від DICE, Unity та Cry Engine, оскільки вони дозволяють розробникам гнучко налаштовувати процес рендерингу, адаптуючись до конкретних вимог або технологій.

2.4 Аналіз способів організації об'єктів у 3D середовищі

Ключовим елементом у структуруванні 3D сцени є граф сцени — структура даних, що представляє ієрархічну організацію об'єктів у сцені.

Граф сцени є ключовою структурою даних у тривимірному моделюванні та анімації, який представляє собою ієрархічний відображення об'єктів та їх відносних відношень у сцені. Він допомагає організувати об'єкти за логічною структурою, забезпечуючи ефективне управління та оптимізацію об'єктів при візуалізації.

Граф сцени складається з вузлів, кожен з яких може мати дочірні вузли. Вузли можуть представляти різноманітні об'єкти, такі як геометрія (наприклад, меші), камери, джерела світла або матеріали. "Листя" графа - це вузли, які не мають дочірніх елементів. Вони часто представляють конкретні об'єкти або деталі сцени. Зв'язки між вузлами визначають відносне розташування та орієнтацію об'єктів.

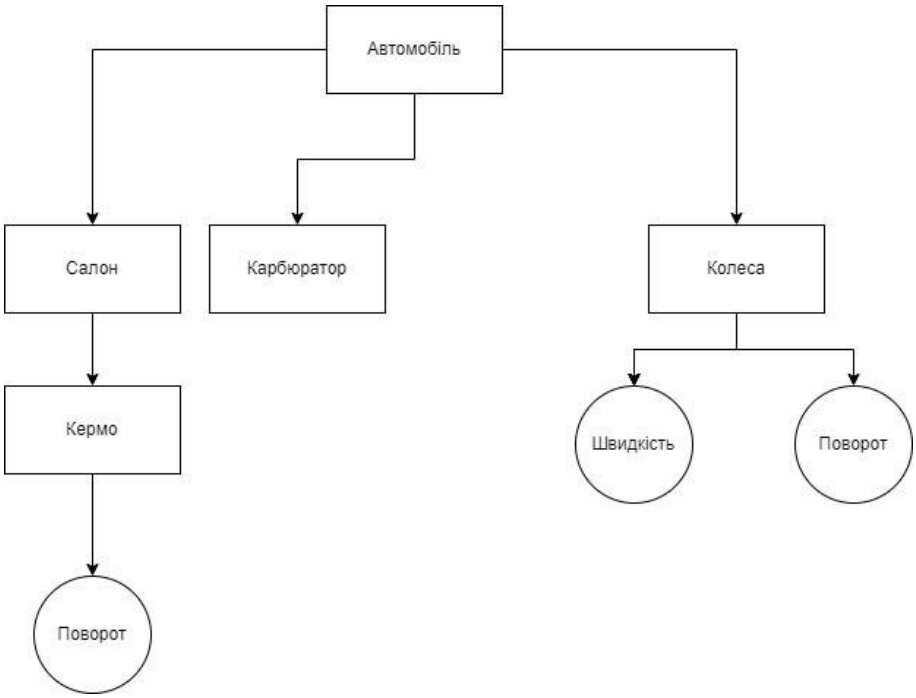

Рис 2.8 Приклад графу сцени

Кожен вузол у графі сцени має свої локальні координати. Локальні перетворення змінюють властивості об'єкта відносно його батьківського вузла. Глобальні перетворення, натомість, визначають положення та орієнтацію об'єкта відносно глобальної системи координат сцени. Коли перетворення застосовується до вузла, воно також автоматично застосовується до всіх його дочірніх вузлів.

Більшість сучасних інструментів 3D моделювання та анімації, таких як Blender, Maya або Unity, активно використовують графи сцени для організації об'єктів. Це не тільки полегшує управління великою кількістю об'єктів, але і дозволяє реалізувати такі функції, як анімація, освітлення та відображення у більш структурований та оптимізований спосіб [24].

Через велику поширеність цього формату, на сьогоднішній день існують два поширених формата для опису сцени – GLTF та USD.

GLTF, відомий як "JPEG для 3D", є відкритим стандартом, розробленим для ефективної передачі та завантаження тривимірних сцен та моделей. Ця специфікація була розроблена з метою максимізації швидкості завантаження, компактності зберігання та гарантування високої сумісності з сучасними графічними двигунами.

GLTF активно використовує граф сцени у своєму ядрі. Його основна структура складається з вузлів, що можуть містити геометрію, матеріали, анімації та інші властивості. Ресурси, такі як геометрія, зберігаються у буферах, оптимізованих для швидкої обробки (рис 2.9).

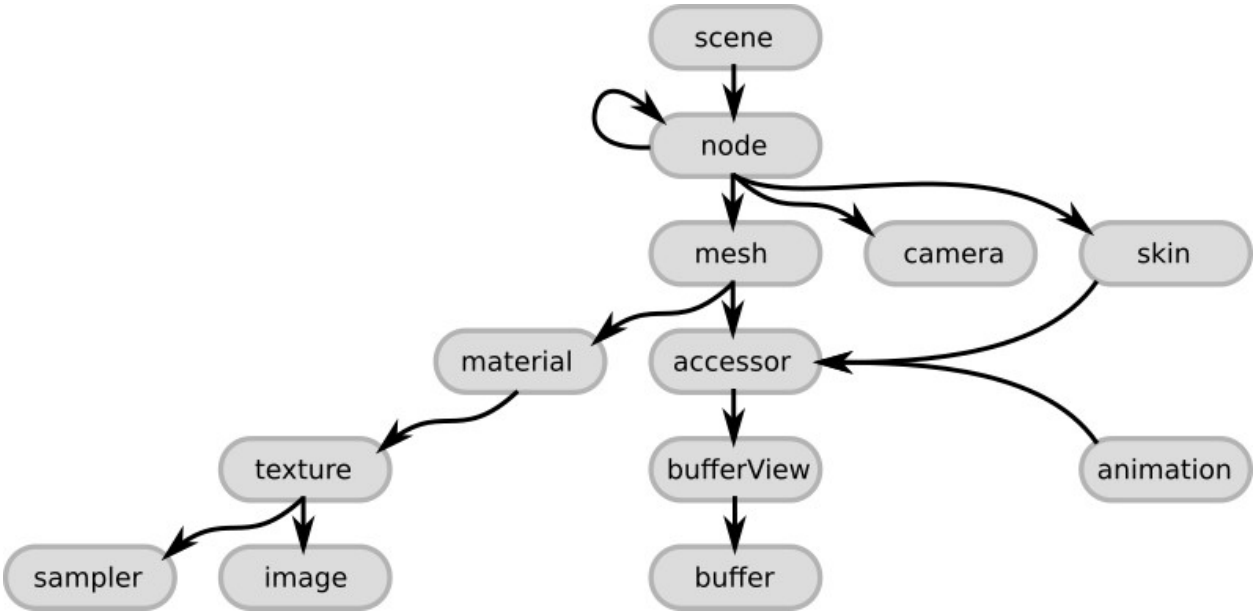

Рис 2.9 Приклад сцени у форматі GLTF

В свою чергу Universal Scene Description (USD) є відкритим форматом, розробленим Pixar. Він був створений для адресації складних потреб у виробничих пайплайнах анімації та візуальних ефектів, надаючи можливість ефективно представляти, редагувати та інтерполювати розширені 3D сцени.

USD також вбудовує концепцію графа сцени в ядро своєї специфікації. Ієрархія USD дозволяє представляти об'єкти на різних рівнях деталізації, починаючи від великих груп до індивідуальних примітивів. Особливістю USD є підтримка "варіантів", що дозволяє модифікувати і зберігати альтернативні версії сцен без необхідності дублювання основного контенту.

Обидва формати, GLTF та USD, розглядають граф сцени як центральний елемент своєї архітектури. Граф сцени використовується для представлення ієрархічної структури 3D контенту, дозволяючи цим форматам забезпечувати ефективну організацію, доступ та маніпуляцію тривимірними об'єктами. Використання графа сцени в цих форматах відображає його ключову роль у сучасній 3D графіці та допомагає забезпечити високий рівень гнучкості та оптимізації при роботі з тривимірним контентом.

USD був спеціально розроблений для роботи зі складними сценами, що включають велику кількість об'єктів, матеріалів та анімацій. Його ієрархічна структура та підтримка "варіантів" надають можливість створення та маніпулювання альтернативними версіями сцен без дублювання основного контенту.

2.5 Аналіз методів оптимізації графічного представлення об’єктів у 3D середовищі

У сучасному світі 3D графіки, де зображення стають все більш реалістичними і деталізованими, ефективність рендерингу виступає однією з найбільш критичних задач. Швидкодія, якість та реалізм є важливими складниками успішного графічного представлення, але досягнення оптимального балансу між ними може бути складним завданням. Оптимізація рендерингу допомагає не тільки покращити відгук системи та продуктивність,

але і забезпечує можливість створення високоякісних 3D сцен, які є приємними для ока користувача.

Цей розділ розглядає основні методи і підходи до оптимізації рендерингу в 3D середовищах. Ми розглянемо як класичні методи, так і сучасні підходи, що допомагають розробникам ефективно використовувати ресурси обладнання, зменшуючи невидимі обчислення та збільшуючи ефективність відображення 3D об'єктів. На прикладах різних алгоритмів та методів буде продемонстровано, як можна досягти більш швидкого та якісного рендерингу, а також знизити вимоги до обладнання, зберігаючи при цьому високу якість зображення.

Важливо розуміти, що оптимізація рендерингу є невід'ємною частиною розробки 3D додатків і ігор. Цей процес вимагає глибокого розуміння особливостей графічного процесора, алгоритмів та структур даних. Цей розділ покликаний надати читачеві цілісне уявлення про виклики та рішення в області оптимізації рендерингу в 3D середовищах.

Після того, як була висвітлена важливість оптимізації рендерингу в 3D середовищах, перейдемо до одного з ключових методів, який дозволяє ефективно зменшити обсяг обчислень під час візуалізації тривимірних об'єктів. Цей метод називається "фрустум culling".

Суть фрустум culling полягає в тому, щоб визначити, які частини сцени насправді видимі для камери, і відсікати (або ігнорувати) об'єкти або частини об'єктів, які не потрапляють у поле зору камери. "Фрустум" відноситься до піраміди видимості камери, де основа піраміди є ближнім планом, а вершина - далеким планом. Ця піраміда видимості допомагає визначити, які об'єкти потрапляють у поле зору камери.

Для реалізації фрустум culling, розробники зазвичай використовують різні методи та алгоритми для визначення, чи перетинається об'єкт з

фрустумом камери. Якщо об'єкт не перетинається, його можна безпечно ігнорувати, що призводить до зменшення обсягу рендерингу. Автори [25] детально описують способи оптимізації фрустум culling за певних обставин.

Використання фрустум culling має дві основні переваги. По-перше, воно дозволяє зменшити кількість об'єктів, які потрібно обробити, що підвищує швидкодію рендерингу. По-друге, воно дозволяє ефективніше використовувати ресурси системи, оскільки ресурси не витрачаються на об'єкти, які не будуть видимі на екрані.

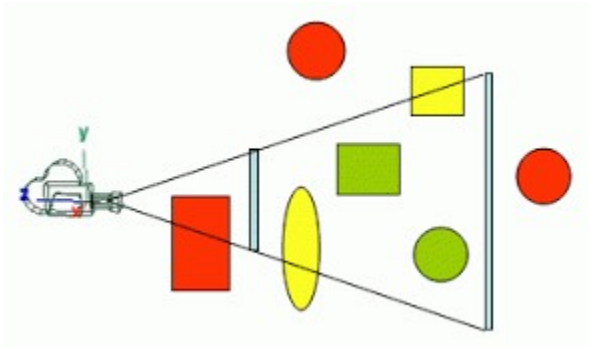

Рис 2.10 Механізм роботи оптимізації фрустум culling

Застосування методів оптимізації рендерингу, таких як фрустум culling, дозволяє зосередитися на об'єктах, які насправді видимі для користувача. Однак, коли ці об'єкти знаходяться на екрані, додаткова проблема виникає через різні рівні деталізації об'єктів в залежності від їх розміру та відстані до камери. В таких випадках використовують оптимізацію під назвою «Level of Detail» або скорочено LoD.

Основна ідея LoD полягає в тому, щоб представляти 3D об'єкти різними рівнями деталізації в залежності від їх відносної відстані до камери або їх

розміру на екрані. Це означає, що об'єкти, які знаходяться ближче до камери, будуть мати вищий рівень деталі, тоді як об'єкти, розташовані далі, будуть зображені менш деталізовано.

Переваги LoD:

- економія ресурсів: об'єкти на великій відстані зазвичай займають менше пікселів на екрані, тому їх можна зображувати менш деталізованими моделями без видимих втрат якості;
- збільшення продуктивності: використання менш деталізованих моделей для далеких об'єктів дозволяє зменшити кількість полігонів, які потрібно обробити, що підвищує швидкодію рендерингу.

Методи створення LoD моделей:

- автоматичне створення: деякі інструменти та двигуни дозволяють автоматично генерувати менш деталізовані версії моделей;
- ручне створення: Дизайнери можуть створювати окремі версії моделей для різних рівнів деталізації, контролюючи якість та оптимізуючи для конкретних вимог.

Залежно від відстані до камери або розміру об'єкта на екрані, система динамічно вибирає, яка версія моделі буде використовуватися для рендерингу (рис 2.11). Згідно з результатами роботи [26] цей метод дозволяє пришвидшити графічне відображення об’єктів майже в 8 разів та згладити артефакти на вихідному зображенні.

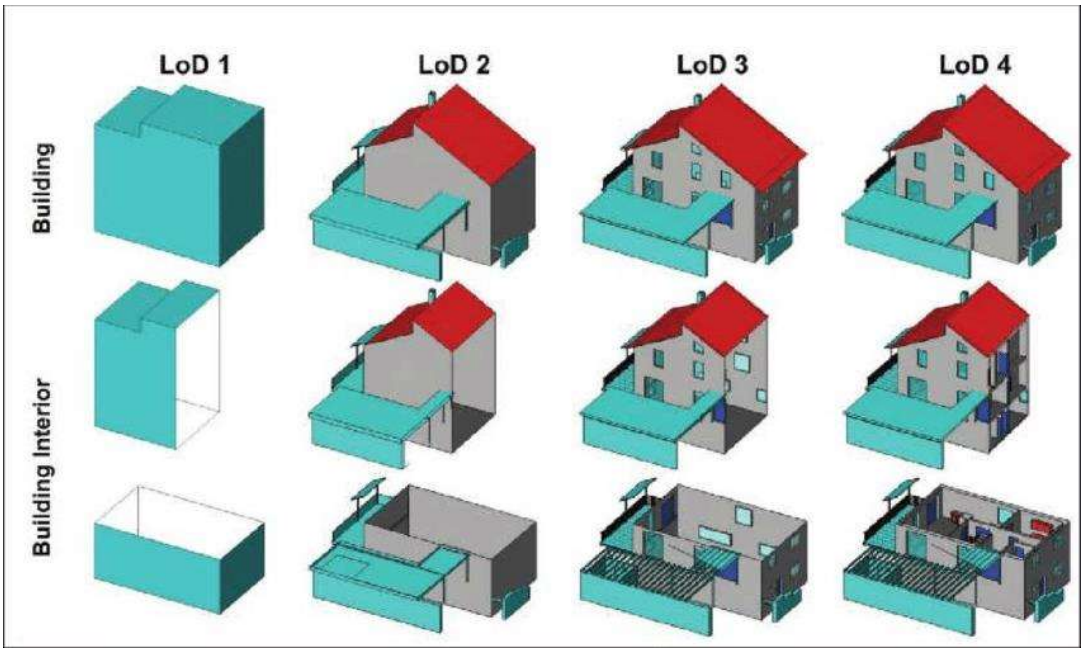

Рис 2.11 Приклад роботи оптимізації Level of Detail

Досягнення високої якості графіки при високих роздільних здатностях може бути вимогливим для обладнання, особливо у сучасних іграх і застосунках віртуальної реальності. Щоб вирішити цю проблему, було запропоновано декілька технологій, зокрема DLSS від NVIDIA та FSR від AMD [27] [28]. Обидві технології спрямовані на підвищення продуктивності та якості зображення без значного погіршення візуального досвіду.

DLSS є технологією масштабування зображення, розробленою NVIDIA, яка використовує машинне навчання та глибокі нейронні мережі для відтворення зображень високої роздільної здатності на основі зразків нижчої роздільної здатності.

NVIDIA використовує суперкомп’ютери для тренування моделі DLSS, аналізуючи тисячі пар зображень (зображення низької та високої роздільної здатності). Вхідне зображення низької роздільної здатності створюється з

використанням стохастичних вибірок, щоб додати додаткову деталь та варіативність у процес.

Попередньо навчена нейронна мережа потім використовується на кінцевому обладнанні користувача для збільшення зображення низької роздільної здатності, створеного у реальному часі, до вищої роздільної здатності.

В свою чергу FSR є технологією масштабування зображення від AMD, яка не використовує глибоке навчання, а замість цього базується на традиційних методах масштабування зображення. FSR засновано на спатіальних та темпоральних методах масштабування, використовуючи інформацію з попередніх кадрів для покращення поточного. Технологія також містить методи оптимізації для розгладжування країв об'єктів та покращення деталей текстур.

В цілому ці технології легко інтегруються і дозволяють значно підвищити швидкість графічного представлення 3D об’єктів (рис 2.12), адже системі потрібно самостійно зробити рендер лише одного з восьми кадрів, а усі інші будуть створені нейромережею. Окрім покращення продуктивності, ці технології можуть забезпечити чіткіше та деталізованіше зображення, ніж те, яке може бути отримано без їх використання при тій самій роздільній здатності, а за допомогою різних режимів масштабування користувачі можуть вибирати оптимальний баланс між якістю зображення та продуктивністю, залежно від потреби.

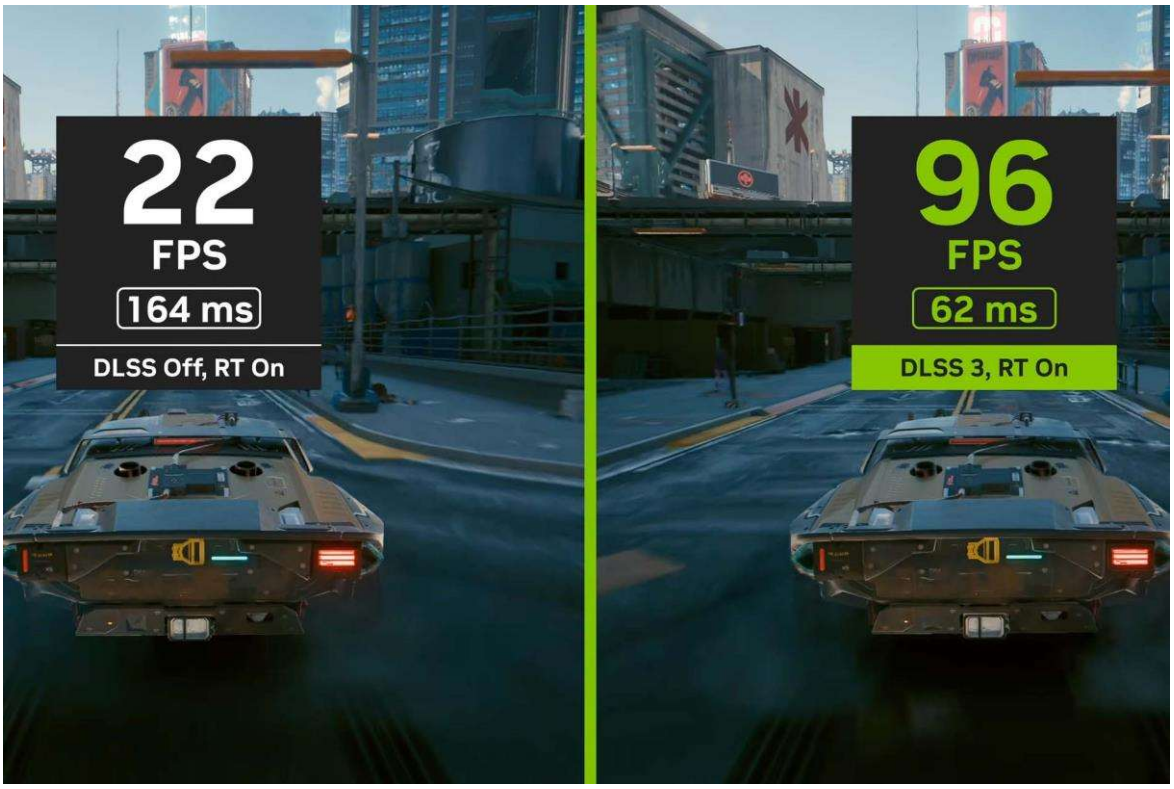

Рис 2.12 Триразове підвищення FPS за допомогою використання технології Nvidia DLSS

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

1. ECS як основа архітектури застосунка

В рамках сучасного програмування графічних застосунків, Entity Component System (ECS) відіграє ключову роль у створенні ефективних і гнучких архітектур. Цей підхід значно відрізняється від традиційних об'єктно-орієнтованих методик, пропонуючи більш модульну та високопродуктивну систему для управління даними та поведінкою в 3D середовищах.

ECS розділяє дані та логіку на незалежні компоненти, що сприяє легкості розширення та адаптації системи під конкретні потреби проекту. Такий підхід дозволяє вам ефективно управляти складними взаємодіями в 3D середовищі, забезпечуючи високу гнучкість та масштабованість. Особливо важливо це при роботі з великими та складними сценами, де кожен елемент може мати свою унікальну поведінку та характеристики. Основні концепції ECS:

- Entity (Ентіті): ентіті або сутність – це загальний, часто числовий ідентифікатор (наприклад, ID). В ECS, ентіті не має безпосередньої функціональності або даних. Воно діє як контейнер для компонентів.
- Component (Компонент): компоненти – це конкретні класи даних, які містять інформацію або стан (наприклад, положення в просторі, здоров'я, швидкість тощо). Компоненти "прикріплюються" до ентіті, але не містять логіки обробки.
- System (Система): системи відповідають за логіку та обробку даних, що знаходяться в компонентах. Кожна система зосереджена на конкретному аспекті гри або додатку (наприклад, рендеринг, обробка вводу, фізика тощо).

У ООП, об'єкт об'єднує дані та методи їх обробки. Наприклад, клас "Автомобіль" може містити дані про швидкість, колір, марку та методи для їзди або зупинки. У ECS, "Автомобіль" може бути представлений як сутність, до якої прикріплені компоненти, такі як "Швидкість" або "Колір", а системи виконуватимуть операції над цими компонентами.

Переваги ECS включають:

– модульність та гнучкість: ECS запроваджує розділення даних (компонентів) та логіки (систем), роблячи архітектуру більш модульною. Це означає, що розробники можуть легко додавати, змінювати або видаляти компоненти без необхідності переписувати існуючий код. Наприклад, якщо потрібно додати новий тип візуального ефекту до об'єктів, це можна зробити, додавши новий компонент до ентіті, без зміни інших систем;
– підвищена продуктивність: оскільки ECS розділяє дані та обробку, вона сприяє більш ефективному використанню ресурсів комп'ютера, особливо ЦП та ГП. Системи можуть бути оптимізовані для паралельної обробки, що є критично важливим для високопродуктивних 3D графічних двигунів. Це особливо актуально для обробки великої кількості об'єктів у сцені;
– краще використання пам'яті: через лінійне розміщення компонентів у пам'яті, ECS може зменшити фрагментацію та покращити кеш-продуктивність. Це особливо важливо для оптимізації графічних двигунів, де велика кількість об'єктів може бути оброблена швидше завдяки ефективній організації пам'яті.

Через це ECS буде покладений у основу архітектури застосунку, і вся подальша архітектура (рис 3.1) буде складатися з незалежних один від одного систем (наприклад система рендерингу чи симуляції фізики), які будуть

працювати з певними компонентами. Такий підхід дозволить з легкістю додавати нові системи у застосунок, і зробить самі системи більш масштабованими, оскільки код будь-якої з систем можна буде змінити за потреби.

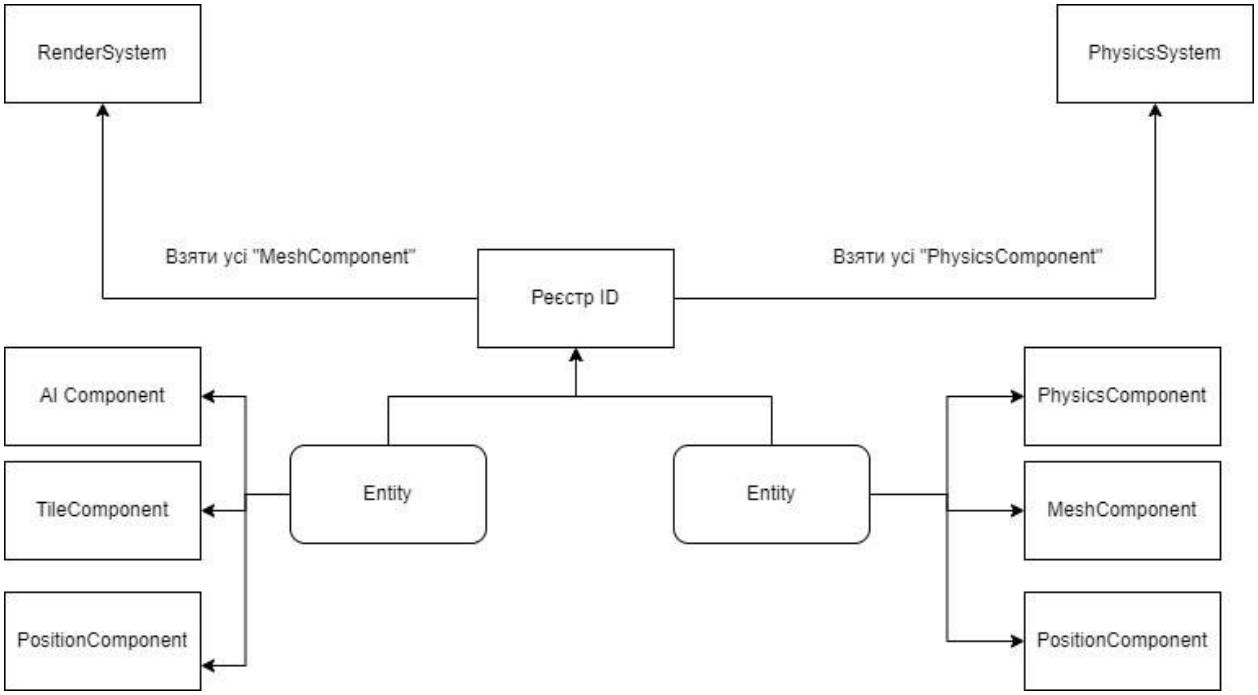

Рис 3.1 Приклад архітектури з використанням ECS

3.2 Представлення функціоналу у вигляді модулів

ООП (об'єктно-орієнтоване програмування) є домінуючим підходом до розробки программного забезпечення з моменту свого виникнення. Однак, дедалі більше виявляється, що ООП має чимало недоліків, особливо в такому проблемному полі як розробка графічних двигунів. Основні недоліки ООП у даному контексті є:

- проблема спагеті-коду: через засноване на спадкуванні структурування ООП, складність коду зростає з кількістю зв'язків між класами та об'єктами. Це призводить до важкості у розумінні, налагодження та розширення коду;
- недоліки інкапсуляції: порушення разультативності методів розділів класів може призводити до залежностей між компонентами, ускладнюючи розуміння системи;
- більш важка комбінація коду: ООП змушує розробників групувати код згідно з потребами об'єктів, а не згідно з потребами певних складових програми.

Ці та інші проблеми стають особливо актуальними при розробці складних графічних двигунів. Представлення функціоналу двигуна у вигляді модульної архітектури може допомогти подолати деякі з цих обмежень.

Однією з альтернатив ООП в організації архітектури графічного двигуна може бути модульний підхід з використанням плагінів. Плагіни дозволяють ізолювати окремі функціональні відрізки системи та розбити їх на незалежні модулі. Такий підхід дає безліч переваг порівняно з традиційним ООП. Зокрема, він забезпечує:

- зниження складності коду: відокремлення функціоналу за допомогою плагінів дає можливість розглядати код кожного модуля окремо від інших, що сприяє кращому розумінню програми;
- гнучкість та легкість розширення: система з плагінами може бути легко розширена та кастомізована за допомогою додавання нових модулів або заміни старих на нові без потреби зміни в іншому коді. Це полегшує підтримку, оновлення та адаптацію двигуна до різних вимог;
- зменшення залежностей: плагіни забезпечують ізоляцію між різними частинами системи, що менше залежні одна від одної і полегшують управління залежностями. Така відокремленість сприяє мінімізації помилок, що виникають під час впровадження нових функцій та внесення змін у існуючі компоненти.

Для успішної реалізації ідеї плагінів в графічному двигуні, важливо правильно організувати архітектуру системи на рівні модулів. Розглянемо основні аспекти організації:

- центральний компонент для керування плагінами: створення спеціального компонента (наприклад, PluginManager), який буде відповідальний за завантаження, виконання та видача доступу до плагінів. Таке централізоване керування спрощує додавання нових плагінів та розширення функціоналу системи;
- чітка декомпозиція функціоналу: для кожного плагіна має бути чітко окреслено область відповідальності та функціонал, без перекриття з іншими модулями. Це забезпечує більш ефективну розбитість архітектури та сприяє разделенню відповідальності за задачі;
- стандартизація інтерфейсів плагінів: введення стандартів взаємодії між плагінами дає можливість комбінувати їх без додаткових зусиль.

Для більшої гнучкості можуть використовуватись інтерфейси або загальні базові класи;

– забезпечення обробки залежностей між плагінами: у випадку, коли плагіни є залежними один від одного, важливо передбачити можливість обробки таких залежностей в архітектурі та гарантувати коректну роботу системи.

Також необхідно забезпечити чітке визначення області відповідальності і функціональності кожного плагіна, без перекриття з іншими модулями. Це сприяє ефективному розподілу обов'язків і розчленуваної архітектури.

Наступним кроком буде стандартизація інтерфейсів плагінів, що дає можливість комбінувати їх між собою без додаткових зусиль при інтеграції. Для забезпечення гнучкості можуть бути використані інтерфейси або загальні базові класи.

Крім того, буде передбачена обробка залежностей між плагінами, якщо такі залежності мають виникати у процесі роботи системи. Це потрібно, щоб гарантувати коректну роботу програмного забезпечення і врахувати можливі випадки, коли одні модулі залежать від інших.

З урахуванням врахування основних принципів та переваг, згаданих раніше запропонуємо короткий опис архітектури графічного двигуна, організованого з використанням плагінів. Важливим акцентом є наявність "ядра" функціоналу двигуна, яке містить базові можливості, необхідні для роботи будь-якого графічного додатка. Основні складові цієї архітектури включають:

– плагін відображення (RenderPlugin): цей компонент відповідає за створення, оптимізацію та компонування тривимірних графічних об'єктів, а також управління віртуальними камерами, освітленням та іншими параметрами візуальної частини сцени. Така архітектура

дозволить абстрагуватися від конкретних ресурсів та ефективно імплементувати рендер-граф з оптимізаціями проходів;

– плагін фізики (PhysicsPlugin): служить для симуляції руху, взаємодії та зіткнень об'єктів у віртуальному середовищі, а також управління геометрією та взаємодією сил при розрахунках;
– плагін вводу/виводу (InputOutputPlugin): дозволяє обробляти події введення та виводу, наприклад від пристроїв керування, клавіатури, миші або сенсорних екранів;
– плагін імпорту/експорту даних (ImportExportPlugin): повинен забезпечувати загрузку та зберігання моделей, сцен, текстур та інших ресурсів, що використовуються у додатках.

PluginManager має бути спільним для всіх плагінів, який забезпечує координацію їх роботи, ініціалізує їх у відповідному порядку та виконує необхідні операції при видаленні плагінів з системи.

Таким чином, архітектура графічного двигуна (рис 2.2) з орієнтацією на базовий "ядерний" функціонал стає більш модульною та гнучкою, дозволяючи розробникам легко масштабувати його функціональність та адаптувати до власних потреб.

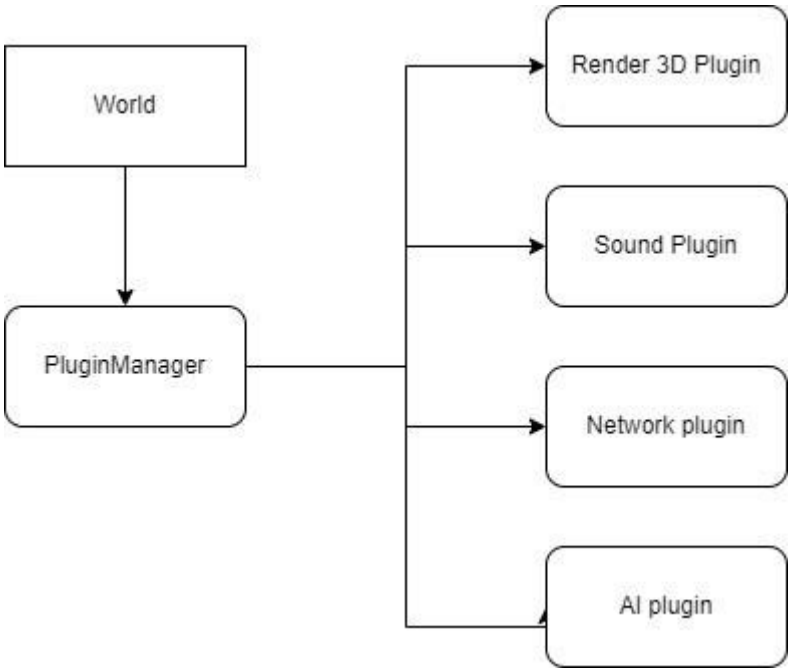

Рис 3.2 Модульна архітектура двигуна

Зв'язуючи Entity-Component-System (ECS) з поняттям плагінів, можна врахувати, що плагіни мають ролі систем (S) в ECS. Це надає ряд переваг та можливостей залучення різноманітних плагінів до створення ігрових сцен й об'єктів.

Плагіни у графічному двигуні, організованому за принципами ECS, можуть служити системами, що обробляють дії і взаємодію сутностей з відповідними компонентами. Оскільки системи в ECS-архітектурі відповідають за обробку набору компонентів, вони є відмінним способом інтеграції плагінів.

Забезпечуючи плагінам доступ до компонентів через центральний реєстр, розробники отримують можливість забезпечити максимальну гнучкість й декомпозицію коду. Центральний реєстр компонентів є спільним

сховищем даних для всіх сутностей в системі. Системи (або плагіни, в нашому контексті) можуть отримувати доступ та вносити зміни в ці компоненти без прямої залежності від сутностей.

Отже, плагіни, які є системами в ECS-архітектурі, можуть ітерувати дані компонентів, виконувати необхідні операції над ними, та відслідковувати зміни в компонентах, зберігаючи історію змін, відновлювати їх або реагувати на певні події.

Таким чином, інтегруючи плагіни як системи у ECS, забезпечується відмінна модульність і гнучкість коду, а також можливість створювати різні додатки, які відповідають конкретним потребам.

Отже, беручи до уваги все вищенаведене, ми запропоновуємо архітектуру, що складається з наступних компонентів:

а) реєстр сутностей: централізоване сховище для усіх сутностей, що відображаються на ігровій сцені. Реєстр відповідає за генерування ID сутностей, додавання та видалення сутностей та надання можливості отримувати ID сутностей для роботи з ними;

б) центральний реєстр компонентів (Entity manager): сховище для усіх компонентів, пов'язаних з сутностями. Реєстр забезпечує прозорий доступ до та повернення даних компонентів для систем та додатка в цілом;

в) PluginManager: керує життєвим циклом плагінів. Завантажує та вивантажує плагіни, ініціалізує та відключає їх відповідно до потреб системи. Плагіни реалізуються як системи у ECS;

г) системи: плагіни або модулі, що реалізують специфічні аспекти ігрового процесу або графічної реалізації. Наприклад:

- RendererPlugin: плагін відображення, що реалізує алгоритми візуалізації. Треба відокремети, що код цього плагіну буде виконуватися у окремому потоці, щоб забезпечити його

незалежність від ігрових механік та забезпечити максимальну швидкість рендерингу.

- PhysicsPlugin: плагін фізики, що реалізує моделювання фізичних взаємодій;
- InputOutputPlugin: плагін для обробки подій введення та виводу;
- ImportExportPlugin: плагін для імпорту/експорту ресурсів;
- AIPlugin: плагін, що реалізує штучний інтелект та керування NPC;
- AudioPlugin: плагін аудіо, що забезпечує обробку звукових ефектів та музики.

д) сценарії й конфігурація: файл конфігурації для налаштування графічного двигуна (відповідальності систем) та опцій реалізації, що залежать від конкретних проектів.

Розробники, які використовують цей двигун з цією архітектурою, зможуть настроювати й оптимізувати роботу проектів, обравши певні плагіни перед стартом основної гри. Саме завдяки сучасній архітектурі ECS та плагінах-системах, врахована можливість гнучкої настройки та відкритого розширення. Для підключення або відключення певного плагіну буде достатньо лише зареєструвати (або навпаки, не реєструвати) його у PluginManager.

Наприклад, при розробці проекту, що не вимагає складних фізичних взаємодій між об'єктами, розробник може не підключати PhysicsPlugin для більш легкої та швидкої роботи двигуна. У свою чергу, для ігор, що не мають необхідності в обробці введення від користувача, можна вимкнути InputOutputPlugin, зосередившись на графічному аспекті.

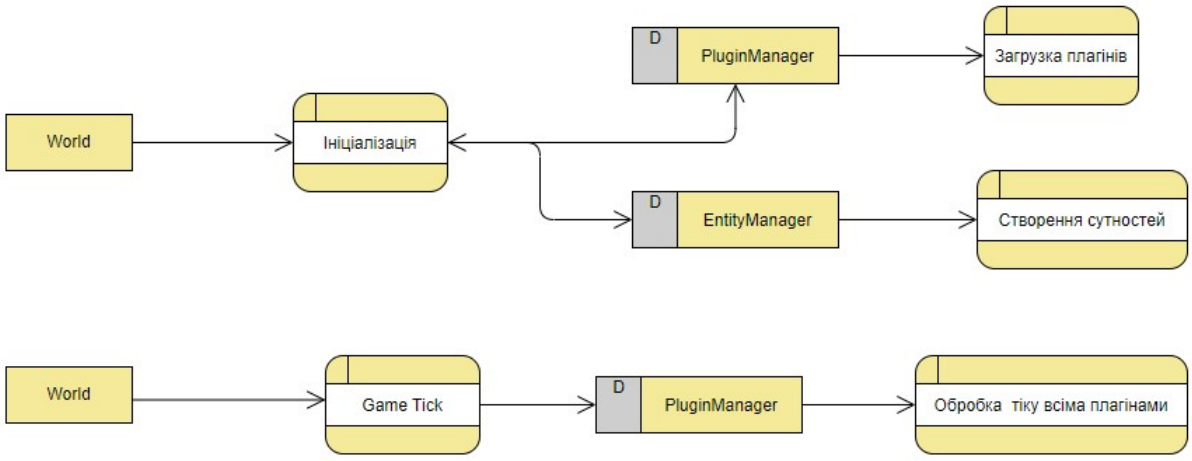

Рис 3.3 Принцип роботи модульної архітектури з використанням ECS

Як можна побачити на малюнку 2.3, об’єкт типу World нічого не знає про такі аспекти як графіка, звук, або симуляції, а є лише управляючим об’єктом. Системи, що є частинами плагінів, отримують повідомлення про те, що потрібно опрацювати наступний кадр, і виконують певну логіку, яку в них заклали розробники. Таким чином забезпечується повна незалежність систем одна від одної, незалежність систем від самих об’єктів (адже системи працюють лише з конкретними компонентами), висока швидкість роботи та можливість паралеллізації всієї системи.

3.3 Реалізація прототипу

Враховуючи всю вищенаведену інформацію, можна розробити прототип застосунка за створеною раніше архітектурою, та перевірити теоретичні дослідження на практиці. Оскільки центральна ідея архітектури це модульність та винесення функціоналу у плагіни, перш за все треба створити

саме інтерфейс плагіну, що дозволить з легкістю створювати та модифікувати системи застосунка у майбутньому. Як було наведено раніше, застосунок буде написаний на мові програмування C++ з використання графічного API Vulkan.

Як було описано раніше, плагіни фактично є системами у архітектурі ECS, тому було вирішено не виокремлювати цю логіку у програмній реалізації. На рисунку 3.4 можна побачити інтерфейс ISystem.

```
#pragma once
#include "core/entity/Registry.h"

struct ISystem
{
    virtual void OnCreate(registry_t & Registry_) = 0;

    virtual void OnDestroy(registry_t & Registry_) = 0;

    virtual void OnUpdate(registry_t & Registry_, float Delta_) = 0;

    virtual ~ISystem() = default;
};
```

Рис 3.4 Інтерфейс ISystem

Він є досить простим, фактично у нього всього 3 функції які викликаються ззовні:

1. OnCreate викликається при ініціалізації даної системи;
2. OnDestroy виклакається при знищенні даної системи;
3. OnUpdate викликається кожен кадр, це основа функція у якій плагін має можливість виконувати свій код. Наприклад система симуляції фізики може зробити прорахунок колізій для об’єктів, а система анімації змінити позиції скелетів.

Як можна побачити, всі три функції приймають registry_t як аргумент. Це тип реєстру сутностей, який містить усі сутності присутні у двигуні. Таким чином кожна система має повний доступ до всього, що на даний момент наявне у застосунку, за одним виключенням – вона не має доступу до інших систем, адже це дозволило б системам мати неявні зв'язки, погіршило потенційні оптимізації та не дозволило б створювати незалежні один від одного плагіни. Замість прямого доступу, системи можуть «спілкуватися» одне з одним через систему запитів, інтерфейс якої можна побачити на рис 3.5:

```
#pragma once
#include <entt/entity/entity.hpp>

enum class ERequestStatus
{
  Created,
  Fulfilled,
  Rejected
};

struct TRequest
{
  void *         Owner{ nullptr };
  ERequestStatus Status{ERequestStatus::Created};

  bool           DestroyWhenFulfilled{ false };
  bool           DestroyWhenRejected{ false };
};

struct TOneshotRequest : TRequest
{
  TOneshotRequest()
  {
    DestroyWhenFulfilled = true;
    DestroyWhenRejected  = true;
  }
};
```

Рис 3.5 Інтерфейс класу TRequest

Як можна побачити, запит – це сутність, яка має власника і статус (виповнений або не виповнений). Але, як було сказано раніше, це сами інтерфейс, а для того щоб створити справжній запит треба створити дочірній клас який буде містити дані необхідні для конкретного запиту. Наприклад на рис 3.6 можна побачити запит до системи фізики на виконання рейкастингу (вирахування колізії з лучем проведеним з камери)

```
#pragma once
#include <optional>
#include <glm/vec3.hpp>

#include "core/entity/Entity.h"
#include "core/entity/Request.h"

// Not set values will be calculated automatically
struct TRaycastRequest : TRequest
{
  // Request data
  std::optional<glm::vec3> StartPosition;
  std::optional<glm::vec3> Direction;
  float                    Distance;

  // Request result
  entity_t  RaycastHit{ entt::null };
  glm::vec3 EndPosition;
};
```

Рис 3.6 Приклад запиту TraycastRequest

Такий підхід надає розробникам можливість не посилати запит конкретній системі, а усім системам разом, і отримати відповідь якомога раніше.

Наступним складовим архітектури є компоненти – спеціальні структури, які не містять логіки, а лише дані. Оскільки додаток реалізується на мові C++, то для створення таких класів не потрібно інтерфейсів, а лише тег – спеціальне поле, яке дозволить чітко визначити чи є клас компонентом, або ні. Наприклад, на рис 3.7 можна побачити компонент з позицією у 3д просторі:

```
#pragma once
#include "core/entity/Component.h"
#include <glm/vec3.hpp>

struct TPositionComponent
{
  static constexpr ComponentTag ComponentTag{};

  glm::vec3 Position;
};
```

Рис 3.7 Приклад компоненту TPositionComponent

Як можна побачити, він містить лише позицію (дані), та тег, за яким пізніше система зможе зрозуміти що це саме компонент і зареєструвати його тип.

Після того, як було розглянуто імплементації систем та компонентів, можна перейти до основної частини – классу World, що керує всім додатком. На рис 3.8 можна побачити функцію ініціалізації та старту додатку, у якій відбувається реєстрація плагінів і запуск двигуна.

```
int main()
{
  World World;

  World.AddStartupFunction(InitDisplay)
       .AddStartupFunction(InitCamera)
       .AddStartupFunction(InitCoreSystems)
       .AddStartupFunction(InitTerrain)
       .AddStartupFunction(InitLight)
       .AddStartupFunction(InitInventory)
       .AddStartupFunction(InitUI)
       .AddStartupFunction(InitGameplayFeatures)
       .AddStartupFunction(InitMetrics);
  World.Run();
}
```

Рис 3.8 Код ініціалізації та старту додатку

Код реєстрації конкретної системи можна побачити на малюнку 3.9

```
void InitDisplay(World & World_)
{
  World_.Spawn(TDisplayComponent
  {
    .Width  = 1280,
    .Height = 720
  })
  .Spawn(TGLFWWindowComponent
  {
    .Title  = "Тестовий проект",
    .Icon   = "res/icons/logo.png",
    .Window = nullptr,
  });
}
```

Рис. 3.9 Код для реєстрації системи вікон

Об’єкт World містить у собі контейнер для всіх плагінів, як під час обробки кожного наступного кадру послідовно викликає і повідомляє про те,

що треба оновити дані. Таким чином всі системи є незалежні одне від одного, і працюють лише з певним набором даних.

Окремо треба виділити систему рендерингу – оскільки сучасні рендер-двигуни стали дуже складними, імплементувати весь процес рендерингу в одній системі начасі не є можливим через декілька причин:

– Великий обсяг коду доведеться перенести в одну функцію, якщо мова йде навіть про простий двигун то це приблизно 2000 строк коду, але якщо розглядати сучасні двигуни – більше 100 тисяч строк коду. Такий обсяг неможливо ефективно читати та модифікувати, якщо він знаходиться в одному файлу;
– Рендер має декілька проходів – наприклад спочатку розраховується лише відстань від камери для кожного пікселя, після чого зберігається у окрему текстуру і передається у наступні етапи рендерингу (наприклад для процесу culling-а об’єктів, що не видні з даної камери). Якщо ж імплементувати всі ці проходи у одній функції, буде витрачено набагато більше пам’яті, порівняно з підходом, коли ми розділяємо етапи.

Таким чином виникає необхідність виокремлювати різні етапи процесу рендерингу у окремі системи і передавати між ними дані – і саме для цього використовуються рендер графи. Увесь проце рендерингу можна представити у вигляді направленого графу як на рис 3.10, у якому ребра це передача ресурсів між етапами, а вузли – самі етапи рендерингу.

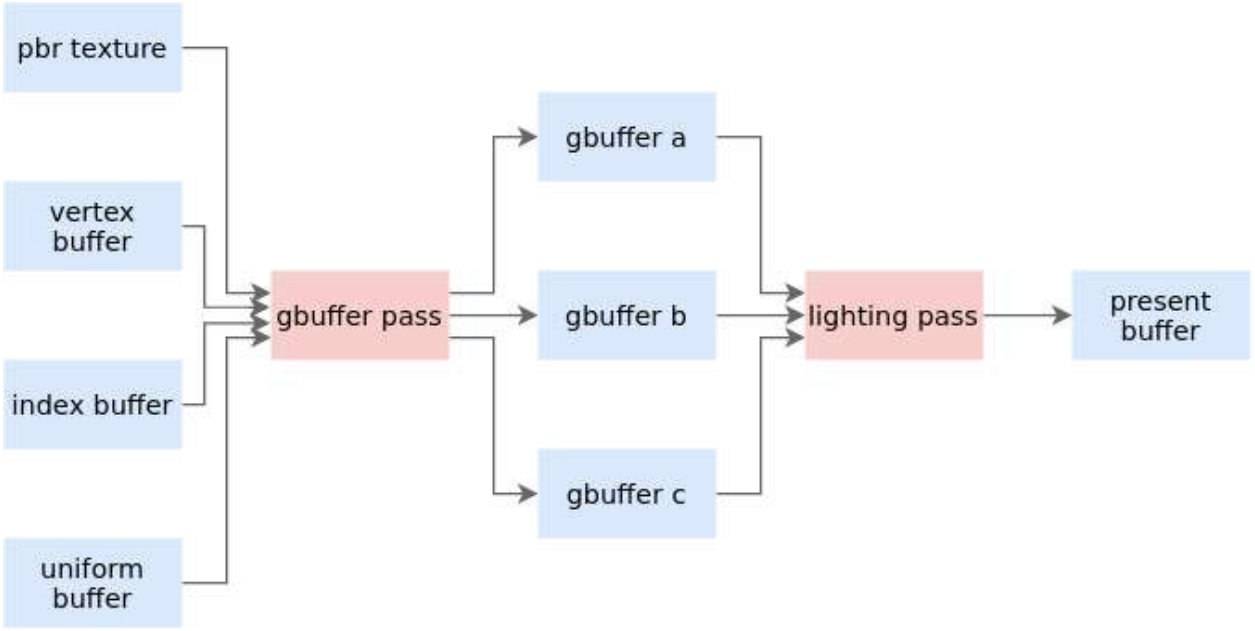

Рис 3.10 Приклад рендер графу

Архітектура у графічному API Vulkan є низькорівневим, і вже містить такі поняття як етапи рендереру та пайплайн, тому створення абстракції рендер графу є досить нескладним. На рис 3.11 можна побачити приклад імплементації рендер графу:

```
class RenderGraphBuilder
{
    struct RenderPassReference
    {
        std::string Name;
        std::unique_ptr<RenderPass> Pass;
    };

    using RenderPassName = std::string;

    template<typename ResourceType, typename TransitionType>
    struct ResourceTypeTransitions { ... };

    struct ResourceTransitions
    {
        ResourceTypeTransitions<std::string, BufferTransition> Buffers;
        ResourceTypeTransitions<std::string, ImageTransition> Images;
    };

    using AttachmentHashMap = std::unordered_map<std::string, Image>;
    using PipelineHashMap = std::unordered_map<RenderPassName, Pipeline>;
    using PipelineBarrierCallback = std::function<void(CommandBuffer&, const ResolveInfo&)>;
    using PresentCallback = std::function<void(CommandBuffer&, const Image&, const Image&)>;
    using CreateCallback = std::function<void(CommandBuffer&)>;

    std::vector<RenderPassReference> renderPassReferences;
    std::string outputName;
```

Рис 3.11 Приклад імплементації рендер графу

Можна побачити, що цей білдер містить у собі посилання на етапи рендерингу (вузли) та інформацію про передачу ресурсів (ребра), і після того як граф створений – для того щоб графічно відтворити сцену, його треба обійти пошуком в ширину. Приклад використання цього класу можна побачити на рис 3.12

```
auto CreateRenderGraph(SharedResources& resources)
{
    RenderGraphBuilder renderGraphBuilder;
    renderGraphBuilder
        .AddRenderPass("UniformSubmitPass", std::make_unique<UniformSubmitRenderPass>(resources))
        .AddRenderPass("ShadowPass", std::make_unique<ShadowRenderPass>(resources))
        .AddRenderPass("OpaquePass", std::make_unique<OpaqueRenderPass>(resources))
        .AddRenderPass("SkyboxPass", std::make_unique<SkyboxRenderPass>(resources))
        .AddRenderPass("ImGuiPass", std::make_unique<ImGuiRenderPass>("Output"))
        .SetOutputName("Output");

    return renderGraphBuilder.Build();
}
```

Рис 3.12 Приклад використання рендер графу

Використовуючи вищенаведені компоненти було створено тестовий проект, який містить наступні системи (лише основні):

- UI System – система для відображення інтерфейсу на основі Noesis GUI;
- Render System – система рендереру, що складається з 8 етапів;
- Terrain System – система для генерації 3д поверхонь;
- Physics System – система для симуляції фізики з використанням бібліотеки Bullet;
- Camera System – система для керування камерою гравця;
- Mesh System – система для створення 3д сіток з сирих даних координат.

Приклад роботи програми розробленої за допомогою двигуна на основі модульної ECS архітектури можна побачити на рис 3.13.

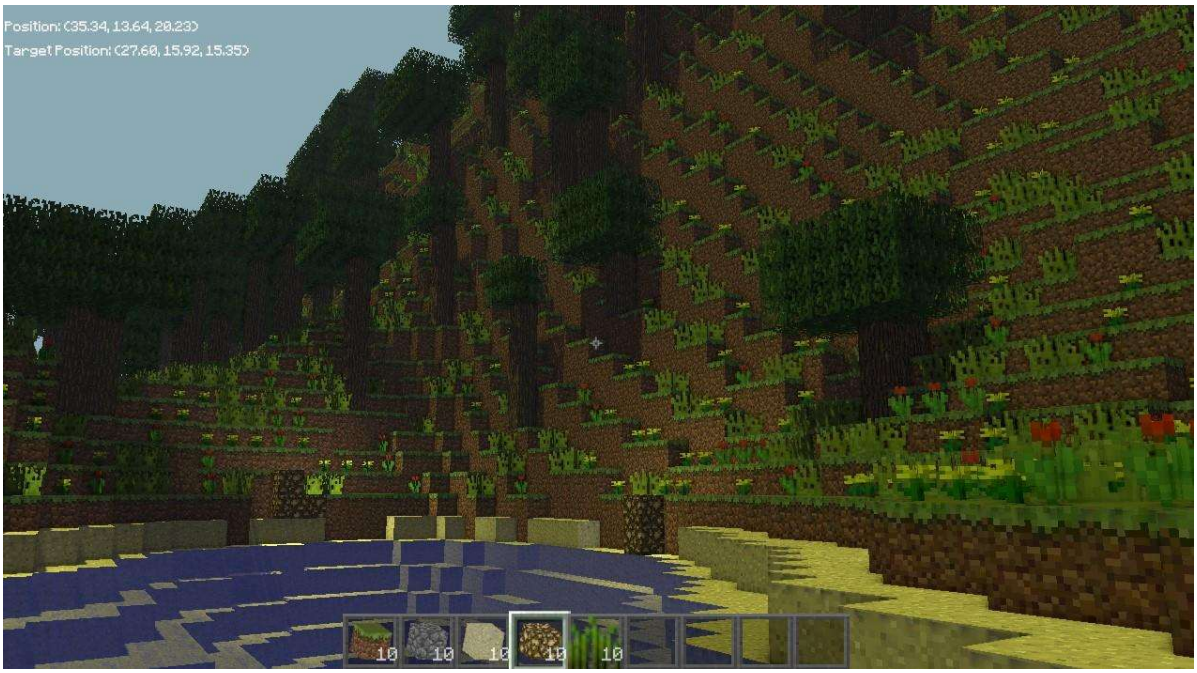

Рис 3.13 Приклад роботи проекту розробленого на основі модульної ECS архітектури

3.4 Можливості для покращення

Після практичної реалізації двигуна на основі модулів та Entity-Component-System архітектури, а також впровадження рендер-графів у системі рендерингу, настав час розглянути можливості оптимізації та покращення для подальшого зростання продуктивності та графічної якості.

У даному контексті важливим є впровадження паралельної обробки даних та взаємодії компонентів архітектури двигуна. Наявні сьогодні мікропроцесори зазвичай мають кілька ядер та можуть одночасно обробляти декілька потоків, що надає значні переваги у обробці даних. Це можна здійснити, організувавши обробку компонентів Entity в межах ECS у паралельні потоки та розбивши процес рендерингу на різні частини, що

працюють у паралельному режимі (наприклад, обробка мешів, шейдерів, текстур тощо).

Ще одним важливим аспектом оптимізації двигуна є виокремлення рендерера від ігрової логіки. Це дозволить працювати відображенню графіки на потузі, незалежному від ігрових процесів й алгоритмів. Таке розподілення завдань сприяє кращому розподілу ресурсів між компонентами та поліпшує здатність двигуна до плавного рендерингу графіки. Для цього потрібно забезпечити внутрішню буферизацію даних, необхідних для рендерингу графіки, ізоляцію засобів обробки ігрової логіки, таких як робота з тактовими частотами, від рендерингу графіки, розробку системи можливих оновлень стану графіки для правильної взаємодії між ігровою логікою та відображенням графіки, а також створення обмежень для взаємодії ігрової логіки з рендерером, щоб сприяти плавно рендерингу графіки без затримок та прогалин у процесі відображення.

Також треба розглянути додаткові можливості оптимізації застосунку за рахунок інтеграції сучасних технологій DLSS та FSR, а також оптимізації рендер-графів. Інтегруючи технології Deep Learning Super Sampling (DLSS) від Nvidia та FidelityFX Super Resolution (FSR) від AMD, можна масштабувати зображення з більш низьким вихідним роздільною здатністю з покращенням якості відображення без суттєвого впливу на продуктивність. Ці технології пропонують альтернативи до звичайного жорсткого масштабування, оптимізуючи графічні ресурси для довільного досягнення комфорту графіки та продуктивності.

Оптимізація рендер-графів націлена на поліпшення продуктивності й ефективності двігуна є ще одним методом прискорити застосунок. Деякі підходи включають кластеризацію, кешування станів ресурсів, визначення

порядку зйомки та впровадження поетапного рендерингу. Це забезпечить кращі результати за лічені мілісекунди та дозволить двигуну працювати з меншою кількістю ресурсів, що використовуються для кожного кадру.

Ці оптимізації допоможуть створити рівномірний процес рендерингу графіки і забезпечити плавну роботу графічного двигуна у застосунку, але потребують багато часу на імплементацію та тестування, і не були виконані в рамках цієї роботи, адже основним завдання було саме розробка архітектури.

ВИСНОВКИ

Результатом виконання кваліфікаційної роботи є архітектура застосунка для відображення ігрових 3д середовищ у режимі реального часу. Під час виконання роботи був створений огляд сфер використання застосунків схожого типу, проаналізовані існуючі застосунки, було виявлено їх переваги та недоліки.

На основі аналізу було запропоновано нову архітектуру, яка робить акцент на модульності всіх підсистем застосунку та представлені їх у вигляді опціональних плагінів, що дозволяє користувачу відключати функції, що не будуть використовуватись у його застосунку. Розроблена архітектура дозволяє створювати гнучку, інтуїтивну та легко налаштовувану систему, яка сприяє швидкому прототипуванню інгрових сцен. Використання рендер-графів поліпшує абстракцію, організацію рендеринга та оптимізацію використання графічних ресурсів, забезпечуючи відмінну продуктивність та графічну якість.

Розроблена архітектура має великий потенціал для обробки у декілька потоків одночасно. Паралельна обробка даних та взаємодії компонентів, а також виокремлення рендерера у окремий поток дозволяє забезпечити максимальну продуктивність та плавність рендерингу графіки.

На основі розробленої архітектури було розроблено прототип гри, який підтверджує можливість створення складних проектів, їх розширення та підтримки. Звісно на даний момент це не повноцінний двигун і не може скласти конкуренцію іншим продуктам, але він є прикладом того, що архітектура була створена загалом правильно і може використовуватись розробниками програмного забезпечення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Romanova T., Litvinchev I., Grebennik I., Kovalenko A., Urniaieva I., Shekhovtsov S. (2020) Packing Convex 3D Objects with Special Geometric and Balancing Conditions.

2. SideFX. URL: https://www.sidefx.com/ (дата звернення: 09.10.2023).

3. SideFX Houdini Engine. URL: https://www.sidefx.com/products/houdini-engine/ (дата звернення: 09.10.2023).

4. SideFX Houdini Engine Documentation. URL: https://www.sidefx.com/docs/hengine/ (дата звернення: 09.10.2023).

5. Unreal Engine. URL: https://www.unrealengine.com/ (дата звернення: 09.10.2023).

6. 3D Slicer. URL: https://www.slicer.org/ (дата звернення: 09.10.2023).

7. OpenGL. URL: https://www.opengl.org/ (дата звернення: 09.10.2023).

8. Microsoft DirectX Documentation. URL: https://learn.microsoft.com/en-us/windows/win32/directx (дата звернення: 09.10.2023).

9. Vulkan. URL: https://www.vulkan.org/ (дата звернення: 09.10.2023).

10. Halmaoui, H., Haqiq, A. (2022). Computer Graphics Rendering Survey: From Rasterization and Ray Tracing to Deep Learning. In: Abraham, A., et al. Innovations in Bio-Inspired Computing and Applications. IBICA 2021. Lecture Notes in Networks and Systems, vol 419. Springer, Cham. https://doi.org/10.1007/978-3-030-96299-9_51

11. Laine, Samuli & Karras, Tero. (2011). High-Performance Software Rasterization on GPUs. 79-88. 10.1145/2018323.2018337.

12. Ray tracing: techniques, applications and prospect. URL: https://ieeexplore.ieee.org/document/9391121 (дата звернення: 09.10.2023)

13. Ouyang, Y., Liu, S., Kettunen, M., Pharr, M., & Pantaleoni, J. (2021).

ReSTIR GI: Path Resampling for Real-Time Path Tracing. Presented on Thursday, June 24, 2021. NVIDIA.

14. Clarberg, P., Kallweit, S., Kolb, C., Kozlowski, P., He, Y., Wu, L., Liu, E., Bitterli, B., & Pharr, M. (2022). Real-Time Path Tracing and Beyond. HPG 2022 Keynote.

15. Steinberg, S., Ramamoorthi, R., Bitterli, B., d'Eon, E., Yan, L.-Q., & Pharr, M. (date not provided). A Generalized Ray Formulation For Wave-Optics Rendering.

16. G. Blinowski, A. Ojdowska and A. Przybyłek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation," in IEEE Access, vol. 10, pp. 20357-20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

17. Quake Engine. URL: https://github.com/id-Software/Quake (дата звернення: 09.10.2023)

18. GoldSrc Engine. URL: https://github.com/headcrab-junkyard/OGS (дата звернення: 09.10.2023)

19. Clemens Szyperski, Dominik Gruntz, Stephan Murer (2002). Component Software: Beyond Object-Oriented Programming. 2nd ed. ACM Press - Pearson Educational, London 2002 ISBN 0-201-74572-0

20. ECS for Unity. URL: https://unity.com/ecs (дата звернення: 09.10.2023)

21. Rendering Engine Architecture at Activison. URL: https://research.activision.com/publications/2021/09/rendering-engine-architecture-at-activision (дата звернення: 09.10.2023)

22. Anderson, Eike & Engel, Steffen & Comninos, Peter & McLoughlin, Leigh. (2008). The case for research in game engine architecture. 228-231. 10.1145/1496984.1497031.

23. Frostbite Rendering Engine Architecture. URL: https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-

Architecture-in (дата звернення: 09.10.2023)

24. Xiaojun Chang, Pengzhen Ren, Pengfei Xu, Zhihui Li, Xiaojiang Chen, Alex Hauptmann (2022). A Comprehensive Survey of Scene Graphs: Generation and Application

25. Assarsson, Ulf & Moller, Tomas. (2000). Optimized View Frustum Culling Algorithms.

26. Lindstrom, Peter & Koller, David & Ribarsky, William & Hodges, Larry & Faust, N. & Turner, Gregory. (1996). Real-Time, Continuous Level of Detail Rendering of Height Fields. Real-time, Continuous Level of Detail Rendering of Height Fields. 109-118. 10.1145/237170.237217.

27. Deep Learning Techniques for Super-Resolution in Video Games. URL: https://arxiv.org/abs/2012.09810 (дата звернення 09.10.2023)

28. AMD FidelityFX Super Resolution 2. URL https://gpuopen.com/fidelityfx-superresolution-2/ (дата звернення 09.10.2023)