Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет__Комп'ютерної інженерії та управління_____
(повна назва)
Кафедра____Автоматизації проектування обчислювальної техніки_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____другий (магістерський)_____
Модель освітлення для побудови графічних 3 D об'єктів__
_____
(тема)

Виконав:
здобувач __2025_____ року навчання,
групи ____СКСм-23-1_____
_____Стецик Д.В._____
(прізвище, ініціали)
Спеціальність___123 – Комп'ютерна інженерія_____
(код і повна назва спеціальності)
Тип програми__освітньо-професійна__
(освітньо-професійна або освітньо-наукова)
Освітня програма__Спеціалізовані комп'ютерні системи_____
_____
(повна назва освітньої програми)
Керівник _доц. Адамов О.С.
_____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ ___Чумаченко С. В._
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет___Комп'ютерної інженерії та управління_____

Кафедра_____Автоматизації проектування обчислювальної техніки_____

Рівень вищої освіти_____другий (магістерський)_____

Спеціальність__123 – Комп'ютернаінженерія_____
(код і повна назва)

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

Освітня програма___Спеціалізовані комп'ютерні системи_____
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві_____Стецику Дем'яну Васильовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи ___Модель освітлення для побудови графічних 3D об'єктів__
_____
_____
затверджена наказом університету від __08__ листопада_____2024_ р.__№ 1189___
2. Термін подання здобувачем роботи до екзаменаційної комісії __31_січня_ 2025__ р.
3. Вихідні дані до роботи _____
_____Рушій рендерінгу зображень в реальному часі_____
_____
_____
_____
_____
_____
_____
_____
_____
4. Перелік питань, що потрібно опрацювати в роботі _____
_____Огляд підходів до розрахунку локальних моделей освітлення_____
_____Огляд підходів до розрахунку глобальних моделей освітлення_____
_____Огляд підходів до організації процесу рендерінгу___
_____Розробка рушію 3Dрендерінгу_____
_____Розробка процесу розрахунку освітлення на основі рушію 3Dрендерінгу_____
_____
_____
_____

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

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	Видача теми кваліфікаційної роботи та її узгодження	02.09.2024 – 06.09.2024	
2	Аналіз предметної області	06.09.2024 – 30.09.2024	
3	Дослідження просунутих методів локального освітлення	30.09.2024 – 2.11.2024	
4	Дослідження методів реалізації ефектів глобального освітлення	3.11.2024 – 1.12.2024	
5	Реалізація рушія рендереру та моделей освітлення	2.12.2024 – 10.01.2025	
6	Оформлення пояснювальної записки	11.01.2025 – 18.01.2025	
7	Перевірка виконаного проекту керівником	18.01.2025 – 31.01.2025	

Дата видачі завдання __01___ ____09_____2024 р.

Здобувач _____
(підпис)

Керівник роботи _____ ___доц Адамов О.С._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка містить 62 сторінки, 23 рисунки,21 лістинг, 7 джерел за переліком посилань.

OPENGL, 3D ГРАФІКА, РУШІЙ РЕНДЕРІНГУ, МОДЕЛЬ ОСВІТЛЕННЯ BLINN-PHONG, FLAT-SHADING, PHYSICALLY BASED SHADING, DATA-ORIENTEDDESIGN

Метою дипломної роботи є дослідження сучасних підходів до розрахунку освітлення 3Dмоделей для рендерінгу графіки за допомогою OpenGLAPI.

Основна увага буде зосереджена на огляді та аналізі алгоритмів розрахунку локального та глобального освітлення, методів реалізації процесу рендерінгу. У ході роботи розглянуто моделі локального освітлення такі як: flat-shading, Blinn-Phongshadingта PhysicallyBasedShading, а також способи розрахунку ефектів глобального освітлення таких як тіні та відбиття.

Розроблено рушій рендерінгу що дозволяє реалізовувати багатокомпонентний процес розрахунку освітлення в реальному часі. На його основі реалізовано основні моделі освітлення і проведено аналіз алгоритмів розрахунку освітлення.

ABSTRACT

Explanatory note contains 62 pages, 23figures,21 listings,7 sources according to the list of references.

OPENGL, 3D GRAPHICS, RENDERING ENGINE, BLINN-PHONG SHADING MODEL, FLAT-SHADING, PHYSICALLY BASED SHADING, DATA-ORIENTED DESIGN

The purpose of this thesis is to research modern approaches to 3D graphics rendering using OpenGL API.

It will be focused on review and analysis of local and global illumination models and methods of rendering process implementation. In the course of the work different local illumination models, such as, flat-shading, Blinn-Phong shading and Physically Based Shading were investigated alongside with approaches to computing of global illumination effects, such as, shadows and environmental reflections.

Rendering engine has been implemented to facilitate multicompoent real-time rendering pipeline. Using this engine main shading models has been implemented and shading calculation alrorythms were analysed.

ЗМІСТ

КАЛЕНДАРНИЙ ПЛАН 2

ВСТУП 8

1 ОГЛЯД ПІДХОДІВ РОЗРАХУНКУ ОСВІТЛЕННЯ 10

1.1. Розвиток алгоритмів розрахунку освітлення 10

1.2. Локальні моделі освітлення 12

1.2.3. Фізично обґрунтоване затемнення 18

1.3. Глобальні моделі освітлення 23

1.3.1. Карти тіней 23

1.3.2. Карти оточення 24

1.3.3. Розсіяне затемнення (Ambientocclusion) 24

1.4 Мета та постановка завдання 25

2 ПІДХОДИ ДО РЕАЛІЗАЦІЇ РЕНДЕРЕРУ 26

2.1 Об’єктно орієнтований підхід 26

2.1.1 Базова архітектура рушію та рендереру 26

2.1.2 Система геометрій та матеріалів 27

2.1.3 Система менеджменту асетів 29

2.1.4 Переваги та недоліки даного підходу 29

2.2 Підхід орієнтований на дані (Data-oriented design) 30

2.2.1 Базова архітектура рушію і рендереру. 31

2.2.2 Система геометрій та матеріалів 32

2.2.3 Система завантаження асетів 33

2.2.4 Переваги та недоліки даного підходу 33

3 РЕАЛІЗАЦІЯ РЕНДЕРЕРУ І ТЕСТОВОГО ДОДАТКУ 35

3.1 Реалізація рендереру 35

3.1.1 Абстрагування об’єктів OpenGL у класи 35
3.1.2 Система геометрій, матеріалів та елементів сцени 39
3.1.3 Класи рушію та рендереру 42
3.1.4 Система загрузки ресурсів 44
3.2 Створення тестового додатку на основі розробленого рендереру 47
3.2.1 Абстрагування платформи 47
3.2.2 Додатковий рендерер користувацького інтерфейсу 48
3.2.3 Написання шейдерів 49
ВИСНОВКИ 61
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 62
ДОДАТОК А ..... **Ошибка! Закладка не определена.**

ВСТУП

3D рендерінг сьогодні є невід'ємною частиною цифрової індустрії та масової культури. Алгоритми для розрахунку освітлення 3D-об'єктів стали основою для створення реалістичних та революційних візуалізацій у різних сферах. У кіноіндустрії використання таких алгоритмів дозволяє створювати надзвичайно реалістичні спецефекти, що розширюють межі уяви глядачів. У інтерактивних розвагах, таких як відеоігри, просунуті техніки рендерінгу роблять віртуальні світи живими та захоплюючими. В інженерних задачах такі алгоритми допомагають створювати точні моделі для проектування та симуляції, а у науковій візуалізації дозволяють більше розуміти складні природні феномени. Загалом, тема 3D рендерінгу є сучасною та динамічною сферою досліджень, що знаходиться на перетині техніки, мистецтва та науки, що робить її надзвичайно цікавою та важливою.

Компанії що роблять продукти для індустрії медіа-розваг розробляють нові технології, алгоритми та підходи до рендерінгу для досягнення найбільшої якості своїх продуктів.Серед найбільш відомих слід виділити Blender, 3dMax, Мауа–програмне забезпечення що дозволяє надзвичайно гнучко працювати з графікою для широкого спектру задач, серед яких: рендерінг картинок високої якості, мультиплікація та створення 3Dмоделей для ігор. Ці програми використовуються для offlineрендерінгу, що означає що генерація результату займає порівняно значного часу. Також є програмне забезпечення що дозволяє реалізувати onlineрендерінг, при якому генерація результату займає мілісекунди. Це дозволяє візуалізувати симуляції реального часу, такі як ігри. Серед них найбільш популярними є OGREі графічні підсистеми таких ігрових рушіїв як Unityта UnrelEngine, які є найбільш провідними у сфері рендерінгу у реальному часі.

Не дивлячись на наявність таких потужних інструментів для створення медіа контенту, розробка нового програмного забезпечення для рендерінгу3Dграфіки є надзвичайно актуальною. Дуже часто компанії не можуть використовувати наявні продукти через фінансові, юридичні або технологічні обмеження. Серед найбільш відомих пропрієтарного програмного забезпечення для рендерінгу є Renderman–рушій розроблений компанією Pixarдля створення спецефектів для кіно та 3Dмультиплікації. AnvilNext, Duna, Snowdrop, Disrupt–рушії розроблені Ubisoftдля різних серій ігор. Frostbyte – рушій розроблений компанією Dice на якому розробляються більшість ігор студій що знаходяться під керівництвом видавництва EA. І цей список далекий від того щоби бути повним.

Метою цієї роботи є огляд алгоритмів і підходів до розрахунку освітлення та реалізація системи рендерінгу в реальному часі використовуючи OpenGLAPIта мову програмування C++.

1ОГЛЯД ПІДХОДІВ РОЗРАХУНКУ ОСВІТЛЕННЯ

1.1. Розвиток алгоритмів розрахунку освітлення

Техніки розрахунку освітлення в реальному часі раніше намагалися імітувати вигляд реального світу, покладаючись на прості моделі, наприклад, такі як flatshading. Цей підхід забезпечував базовий рівень реалістичності, але не враховував складних взаємодій світла з поверхнями. З розвитком технологій симуляції з'явилася можливість значно покращити якість зображення, додаючи дрібні деталі до простої геометрії за допомогою текстур. Наприклад, модель освітлення Blinn-Phong дозволяла більш точно імітувати поведінку світла, враховуючи блиски та розсіяння. Хоча це суттєво покращило кінцевий результат, такі підходи все ще не достатньо детально імітували поведінку світла в реальному світі.

З розвитком потужності графічних процесорів з'явилася можливість більш детально симулювати поведінку світла за допомогою статистичних моделей, таких як physicallybasedshading (PBS). Цей підхід базується на фізичних принципах взаємодії світла з матеріалами, що дозволяє створювати сцени з реалістичним освітленням та текстурами. Завдяки PBS стало можливим передавати нюанси відбиття, прозорості, розсіяння та інших характеристик матеріалів, що значно підвищило рівень візуального реалізму у 3D графіці.

Порівняно недавно з'явилася можливість неймовірно детально симулювати поведінку світла в реальному часі за допомогою трасування променів (ray-tracing). Ця технологія дозволяє відтворювати складні ефекти освітлення, такі як точні відбиття, заломлення, м'які тіні та глобальне освітлення. Ray-tracing значно вплинуло на якість зображення у 3D сценах, наближаючи їх вигляд до реального світу. Завдяки постійному розвитку графічних процесорів, ця технологія стала доступною для інтерактивних

застосувань, таких як відеоігри, забезпечуючи користувачам безпрецедентний рівень реалізму та занурення у віртуальні світи.

Варто зазначити, що трасування променів рідко використовується для рендерингу всієї сцени в реальному часі через його високу обчислювальну складність. Замість цього, воно часто застосовується для розрахунку відображень та непрямого освітлення, що дозволяє досягати високого рівня реалістичності в окремих аспектах зображення. Такий підхід дає змогу комбінувати ray-tracing із традиційними методами рендерингу, забезпечуючи баланс між якістю та продуктивністю.

Більш традиційні способи рендерингу часто комбінуються з іншими алгоритмами для розрахунку специфічних ефектів, а також для забезпечення більшого покриття аппаратного забезпечення для розрахунку обчислення таких ефектів. Наприклад, для відображення прозорих об'єктів використовується група алгоритмів, відома як Order-IndependentTransparency (OIT), що дозволяє коректно обчислювати прозорість незалежно від порядку об'єктів. Відображення розраховуються за допомогою алгориму відображень в просторі екрануScreen-SpaceReflections (SSR) або карт оточення (Environmentmaps). Для створення тіней застосовуються карти тіней (ShadowMaps), які забезпечують швидкий і ефективний спосіб генерування м'яких або чітких тіней. Рендеринг об'ємів (VolumetricRendering) використовується для симуляції проходження світла через середовище, що дозволяє відтворювати такі ефекти, як туман або світло, що проникає крізь воду. Ці технології у поєднанні створюють багатошаровий підхід, що забезпечує баланс між реалістичністю та ефективністю.

Не дивлячись на можливість відображати дуже деталізовані об'єкти, прості моделі все ще часто використовуються. Це особливо актуально для рендерингу графіки на мобільних пристроях, де обчислювальні ресурси обмежені, або для стилізованого зображення, наприклад, такого, що наслідує мультиплікаційний стиль. Простота таких моделей дозволяє досягти високої

продуктивності, зберігаючи при цьому естетичну привабливість у межах заданих стилістичних рамок.

Моделі освітлення поділяються на локальні і глобальні. Локальні розраховують освітленість одного об’єкту, а глобальні – всієї сцени.

1.2. Локальні моделі освітлення

Локальна модель освітлення являє собою підхід у розрахунку світлових взаємодій у тривимірній графіці, який обмежується прямою взаємодією між джерелом світла і поверхнею об'єкта в конкретній точці. Цей підхід свідомо нехтує непрямими ефектами, такими як багатократні відбиття або розсіяння світла у середовищі. Локальні моделі освітлення забезпечують основу для численних алгоритмів рендерінгу, оскільки їх обчислювальна простота і висока ефективність роблять їх особливо придатними для застосування в умовах реального часу.

Опис розрахунку освітлення варто почати з рівняння рендерінгу. Рівняння рендерінгу (renderingequation) це рівняння, що описує глобальну модель освітлення у комп'ютерній графіці. Воно враховує всі можливі взаємодії світла у сцені, включаючи прямі та непрямі ефекти, такі як багатократне відбиття, заломлення, розсіяння та навіть поглинання світла в середовищі. Це рівняння формально визначає світловий потік, що досягає точки на поверхні, як інтеграл всіх можливих напрямків світла, що надходить до цієї точки, множений на відповідний коефіцієнт відбиття або передачі матеріалу. Завдяки своїй універсальності, рівняння рендерінгу є ключовим інструментом для симуляції реалістичного освітлення, але його реалізація потребує значних обчислювальних ресурсів і складних алгоритмів.Через ці обмеження рівняння рендерінгу часто не може бути застосоване безпосередньо в реальному часі, особливо в інтерактивних додатках. Для вирішення цієї проблеми використовуються спрощення, зокрема локальні моделі освітлення, які можна розглядати як апроксимацію цього рівняння. Локальні моделі фокусуються

виключно на взаємодії світла з поверхнею у конкретній точці, ігноруючи вплив світлових взаємодій у всій сцені. Хоча це обмежує їх точність, вони дозволяють досягти прийнятного рівня реалізму, знижуючи витрати ресурсів. Таким чином, локальні моделі стають ефективним компромісом між фізичною точністю і обчислювальною ефективністю.

Визначення рівняння рендерінгу можна подати у наступному вигляді:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega f_r(\mathbf{x}, , \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n}) \mathrm{d}\omega_i$$

Тут $L_o(\mathbf{x}, \omega_o)$— вихідне світло у напрямку ω_o від точки $\mathbf{x}$; $L_e(\mathbf{x}, \omega_o)$ — світло, що випромінюється джерелом; $f_r(\mathbf{x}, , \omega_i, \omega_o)$— двопроменева функція відбивної здатності (BRDF); $L_i(\mathbf{x}, \omega_i)$ — вхідне світло з напрямку ω_i; $(\omega_i \cdot \mathbf{n})$ — косинус кута між напрямком світла ω_i та нормаллю поверхні $\mathbf{n}$; $\int_\Omega$ — інтеграл по півсфері Ω.

Двопроменева функція відбивної здатності (BRDF, BidirectionalReflectanceDistributionFunction) визначає, як світло відбивається поверхнею у заданому напрямку ω_o залежно від напрямку падаючого світла ω_i. BRDF є центральним компонентом рівняння рендерінгу, оскільки вона описує взаємодію світла з матеріалом поверхні, враховуючи його властивості, такі як гладкість, шорсткість або металічність. Математично BRDF визначається як відношення між інтенсивністю відбитого світла у напрямку ω_o та інтенсивністю падаючого світла з напрямку ω_i, нормалізоване на кут падіння. Ця функція забезпечує фізично коректний опис відбиття, дозволяючи моделювати різноманітні оптичні ефекти, включаючи дифузне та дзеркальне відбиття. Завдяки BRDF рівняння рендерінгу може враховувати складні взаємодії світла з поверхнями, що суттєво підвищує реалістичність симуляції освітлення.

Спрощуючи рівняння рендерінгу ми будемо розглядати різні моделі освітлення, що базуються на ньому. Одним із перших спрощень рівняння

рендерінгу, яке використовується для локальних моделей освітлення, є заміна інтегралу по півсфері, що визначає світло, яке надходить з усіх напрямків, на суму компонентів, що враховують лише світло від конкретних джерел. Це спрощення дозволяє зосередитися на обчисленні світла, яке надходить безпосередньо від визначених джерел, наприклад, точкових або напрямлених джерел світла. У такому підході інтеграл замінюється на дискретну суму, де кожен доданок відповідає внеску одного джерела світла. Така апроксимація значно зменшує обчислювальну складність і забезпечує швидший розрахунок освітлення, що особливо важливо для рендерінгу в реальному часі.

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \sum_i^k f_r(\mathbf{x}, , \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})$$

Де *k*– кількість джерел світла.

Це єдине спрощення що буде стосуватися усіх розглянутих моделей освітлення, інші спрощення будуть в основному відбуватися за рахунок зміни способу розрахунку функції відбивної здатності.

1.2.1 Flat-shading

Першим розглянемо примітивний випадок, у якому BRDF є константною функцією. У такій моделі BRDF не залежить від напрямків ω_i та ω_o, а рівняння рендерінгу спрощується до залежності лише від інтенсивності вхідного світла та кута між вектором нормалі поверхні і вектором від поверхні до джерела світла, а поверхня має лише параметр кольору і не може випромінювати світло. Це формулювання лежить в основі моделі освітлення, відомої як flat-shading. Flat-shading забезпечує простий спосіб розрахунку освітлення, оскільки кожна грань об'єкта має однорідний колір, який визначається орієнтацією поверхні щодо джерела світла.В такому випадку рівняння рендерінгу буде мати наступний вигляд:

$$L_o(\mathbf{x}, \omega_o) = \sum_i^k L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})$$

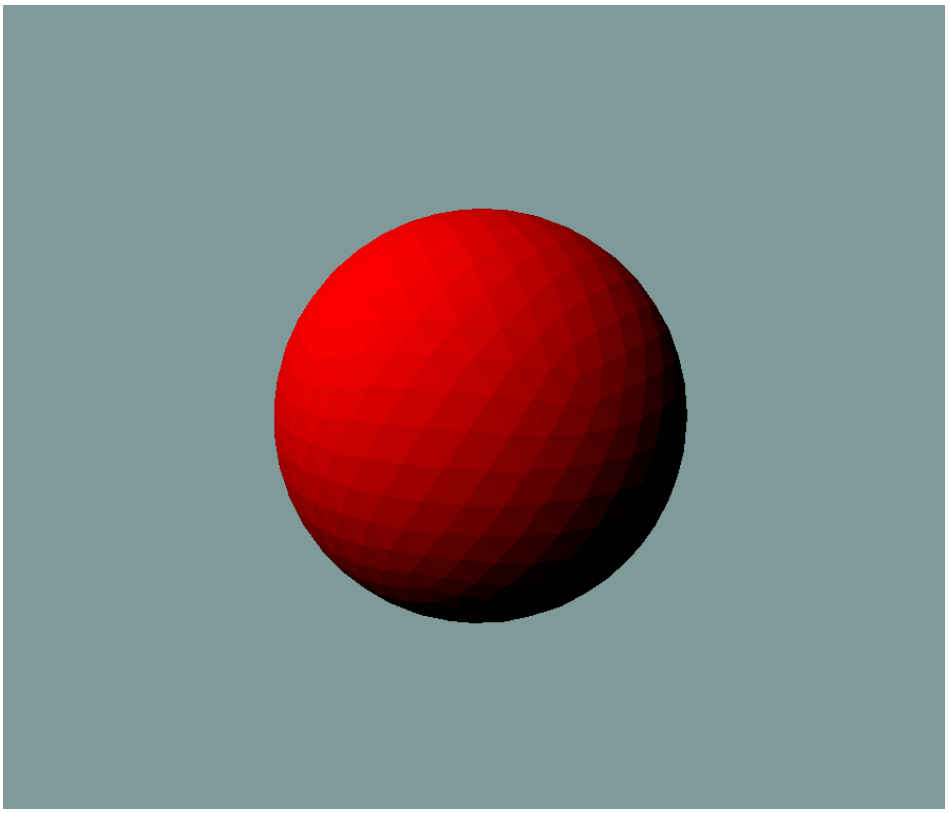

Рисунок 1.1 – Результат розрахунку освітленості поверхні сфери за допомогою методу flatshading

Незважаючи на свою простоту, модель flat-shading все ще знаходить застосування у сучасній комп'ютерній графіці. Вона активно використовується у CAD (Computer-AidedDesign) і DCC (DigitalContentCreation) застосунках, де її швидкість і ефективність є ключовими перевагами. Крім того, flat-shading іноді застосовується у стилізованих моделях освітлення, які потребують узгодженого та художньо простого вигляду, що наслідує традиційний мультиплікаційний або технічний стиль.

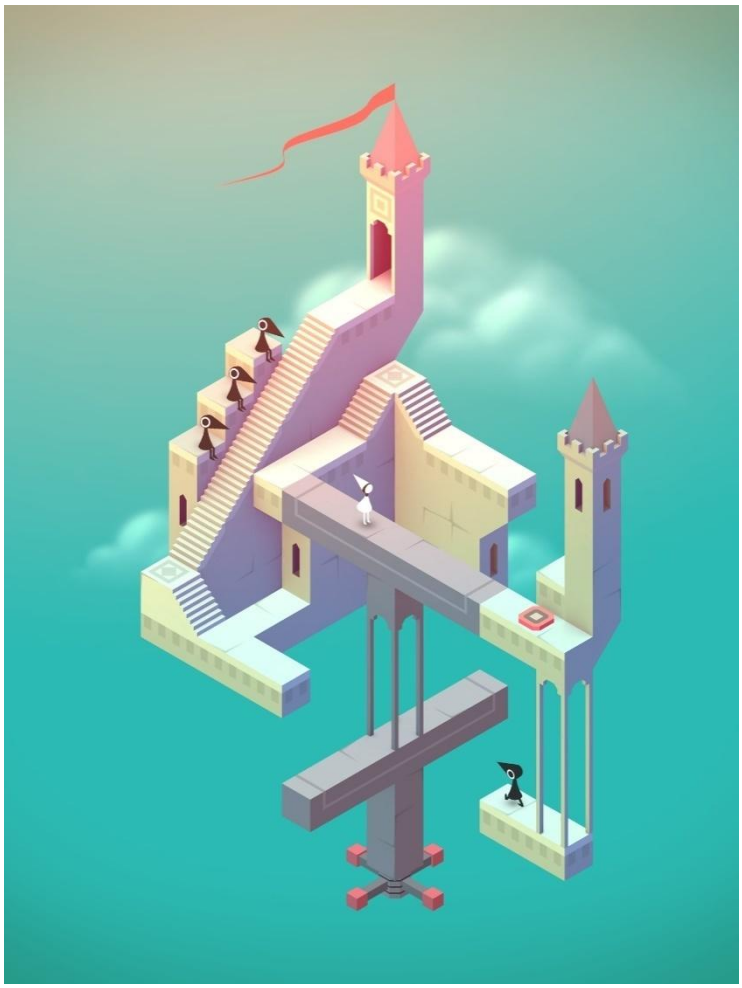

Рисунок 1.2 – Скріншот гри Monument Valley що використовує flat-shading як основний підхід до розрахунку освітлення

1.2.2. Модель освітлення Бліна-Фонга

Наступною є модель освітлення Бліна-Фонга (Blinn-Phong). Рівняння рендерінгу має три компоненти: навколишнє, розсіяне і віддзеркалене світло (ambient, diffuse та specular відповідно). Компонента навколишнього світла є константою і імітує світло, що розсіюється в середовищі. Розсіяне світло моделює освітленість поверхні джерелом світла і визначається косинусом кута між вектором нормалі поверхні та вектором від фрагменту поверхні до джерела світла. Ця компонента відображає світло, яке розсіюється рівномірно у всі сторони. Віддзеркалене світло імітує відображення світла від поверхні геометрії, яке концентрується у конусі навколо віддзеркаленого відносно нормалі поверхні вектора до джерела світла. Ширина цього конуса визначається параметром відблиску (shininess), який контролює ступінь

розмиття дзеркального відбиття.Потім всі три компоненти сумуються, що визначає фінальний колір фрагменту.При такій моделі розрахунку освітлення рівняння рендерінгу буде сформульоване таким чином:

$$L_o(\mathbf{x}, \omega_o) = L_e + \boldsymbol{S}_a + \sum_i^k \mathbf{S}_{diff}\ D(\mathbf{n}, \omega_i) + \mathbf{S}_{spec}\, S(\mathbf{n}, \omega_i, \omega_o)$$

$$D(\mathbf{n}, \omega_i) = \mathbf{n} \cdot \omega_i$$

$$S(\mathbf{n}, \omega_i, \omega_o) = (\boldsymbol{norm}(\omega_i, + \omega_o) \cdot \mathbf{n})^{\boldsymbol{S_{shininess}}}$$

$$\boldsymbol{norm}(v) = \frac{v}{|v|}$$

Де$\boldsymbol{S}_a$,$\mathbf{S}_{diff}$, $\mathbf{S}_{spec}$ та$\boldsymbol{S_{shininess}}$ – коефіцієнти навколишнього, розсіяного, віддзеркаленого світла і відблиску, $D(\mathbf{n}, \omega_i)$ – функція що розраховує освітленість фрагменту, $S(\mathbf{n}, \omega_i, \omega_o)$– функція що розраховує кількість віддзеркаленого світла.

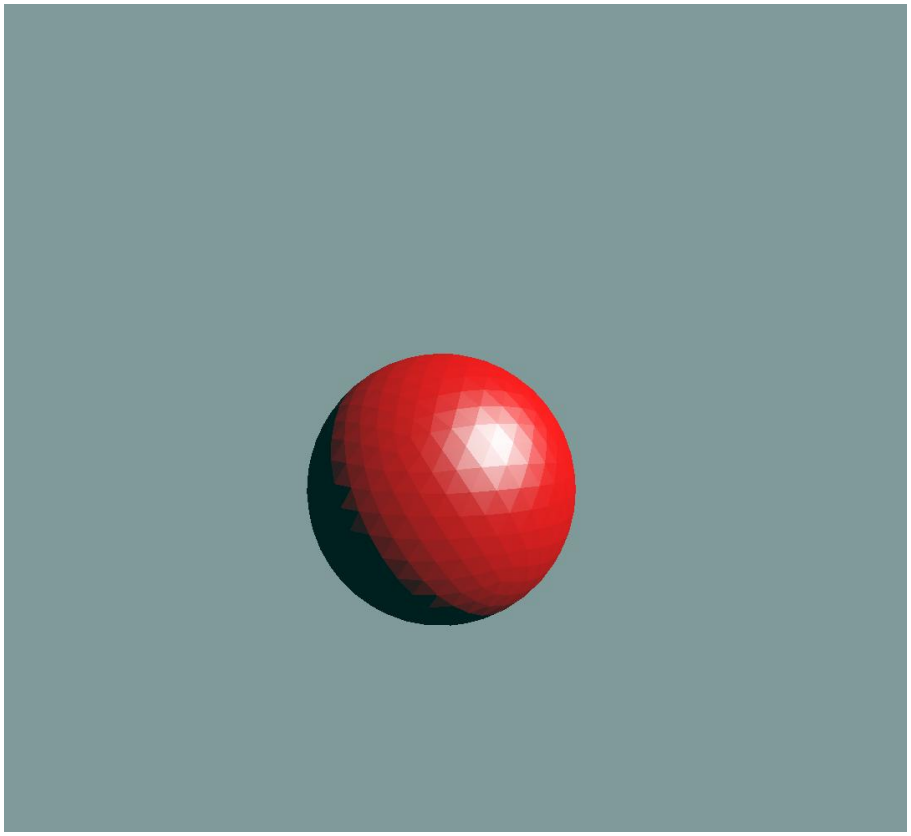

Рисунок 1.3 – Результат розрахунку освітленості поверхні сфери за допомогою методу Бліна-Фонга

Модель Бліна-Фонга забезпечує більш реалістичне освітлення, зберігаючи достатню обчислювальну ефективність для інтерактивних додатків. Вонабула надзвичайно популярною до появи можливості використовувати PhysicallyBasedShading (PBS) у реальному часі. Вона дозволяє досягти високої якості зображення завдяки своїй гнучкості та здатності інтегрувати додаткові ефекти, такі як текстуровані матеріали, м'які тіні або відблиски. Це забезпечує значну варіативність у налаштуванні освітлення для досягнення бажаної естетики, роблячи модель Бліна-Фонга важливим етапом в еволюції рендерінгу.

1.2.3. Фізично обґрунтоване затемнення

Фізично обґрунтоване затемнення (PhysicallyBasedShading, PBS) — це інноваційний підхід до моделювання освітлення в комп'ютерній графіці, який базується на строгих фізичних принципах взаємодії світла з матеріалами. Відмінною рисою PBS є його здатність враховувати матеріальні властивості, такі як мікрогеометрія поверхні, шорсткість і металічність, що дозволяє досягти фізично точного й узгодженого вигляду матеріалів незалежно від умов освітлення.

Ця модель освітлення вводить поняття мікрофасеток і визначає його як мікроскопічні повністю пласкі елементи поверхні які повністю відбивають світло що падає на них.Мікрофасетки у PBSмають розміри значно менші за піксель, колір якого розраховується під час обчислення освітлення, що дозволяє розглядати поверхню з точки зору статистичних параметрів які визначають порядок і орієнтацію мікрофасеток на поверхніа також кількість поглинаємого та відбиваємого світла.

Ключова особливість PBS — це акцент на параметрі шорсткості, який безпосередньо впливає на розподіл мікрофасеток на поверхні. Гладкі поверхні створюють чіткі, дзеркальні відбиття, тоді як шорсткі — розсіюють світло, створюючи м'які, розмиті відблиски. Крім того, PBS інтегрує параметр металічності, який дозволяє контролювати, чи поводиться матеріал як

діелектрик або метал, що значно розширює діапазон доступних матеріалів у рендерингу.

Центральним компонентом PBS є двопроменева функція відбивної здатності(BidirectionalReflectanceDistributionFunction, BRDF), яка описує взаємодію світла з поверхнею. BRDF в PBS формується на основі законів збереження енергії та враховує складні ефекти, такі як відбиття Френеля, які забезпечують правильну поведінку матеріалів під різними кутами освітлення. Такий рівень деталізації дозволяє реалістично відтворювати оптичні характеристики матеріалів — від матових до дзеркальних.

У якості BRDFу PBSвикористовують формулу Кука-Торенсаяка має наступний вигляд:

$$f_r(\mathbf{x},, \omega_i, \omega_o) = \sum_{\boldsymbol{i}}^{\boldsymbol{k}} \boldsymbol{k_d}\, f_{lambert} \; + f_{\boldsymbol{CookTorrance}}(\omega_i, \omega_o)$$

$$\boldsymbol{k_d} = 1 - \boldsymbol{S}_{metallic}$$

$$f_{lambert} = \frac{\boldsymbol{S}_{diff}}{\pi}$$

$$f_{\boldsymbol{CookTorrance}}(\omega_i, \omega_o) = \frac{DFG}{4(\omega_o \cdot \mathbf{n})(\omega_i \cdot \mathbf{n})}$$

Де $\mathbf{k_d}$– коефіцієнт світла що розсіюється в матеріалі, $\boldsymbol{S}_{metallic}$ – коефіцієнт металічності, *D*– функція нормального розподілу, що розраховує відхилення мікрофасеток від вектору ідеального відблиску,*F* – рівняння Френеля, яка визначає коефіцієнт відблиску в залежності від куту погляду на поверхню, *G* – функція геометрії яка розраховує кількість відображеного світла яке буде затінено мікрофасетками на поверхні геометрії.

Функція нормального розподілу розраховує відсоток мікрофасеток які вирівнянні з вектором ідеального відблиску на поверхні об’єкту згідно з параметром шорсткості. Вектор ідеального відблиску являє собою нормалізований вектор суми вектору від фрагменту до споглядача (*v*) і вектору падіння світла (*l*):

$$h = norm(v + l)$$

Найбільш розповсюдженою формулою для розрахунку нормального розподілу для рендерінгу в реальному часі є Trowbridge-Reitz GGX яка має наступний вигляд:

$$D(\mathbf{n}, h, \mathbf{r}) = \frac{\mathbf{r}^2}{\boldsymbol{\pi}((\mathbf{n} \cdot h)^2(\mathbf{r}^2 - 1) + 1)^{\mathbf{2}}}$$

Функція геометрії розраховує самозатінення поверхні на рівні мікрофасеток. Для цього використовуються дві функції, перша з них, яка використовується для розрахунку коефіцієнту затінення у напрямку aвідповідно до значення шорсткостіце Schlick-GGX:

$$G_{SchlickGGX}(\mathbf{n}, a, \mathbf{k}) = \frac{\mathbf{n} \cdot a}{(\mathbf{n} \cdot a)(1 - \mathbf{k}) + \mathbf{k}}$$

$$\mathbf{k} = \frac{(\boldsymbol{a} + \mathbf{1})^{\mathbf{2}}}{8}$$

Далі використовується метод Сміта для розрахунку самозатінення поверхні як у напрямку до споглядача, так і в напрямку падіння світла. В результаті функція геометрії має наступний вигляд:

$$G(\mathbf{n}, v, l, \mathbf{k}) = G_{SchlickGGX}(\mathbf{n}, v, \mathbf{k}) G_{SchlickGGX}(\mathbf{n}, l, \mathbf{k})$$

Останньою компонентоюBRDFє компонента рівняння Френеля. Для рендерінгу в реальному часі використовується апроксимація Шліка. Вона має наступний вигляд:

$$F(\mathbf{h}, v, \boldsymbol{F_0}) = \boldsymbol{F_0} + (1 - \boldsymbol{F_0})(\mathbf{1} - (\mathbf{h} \cdot v))^5$$

Де $\boldsymbol{F_0}$ – коефіцієнт віддзеркалення під прямим кутом.

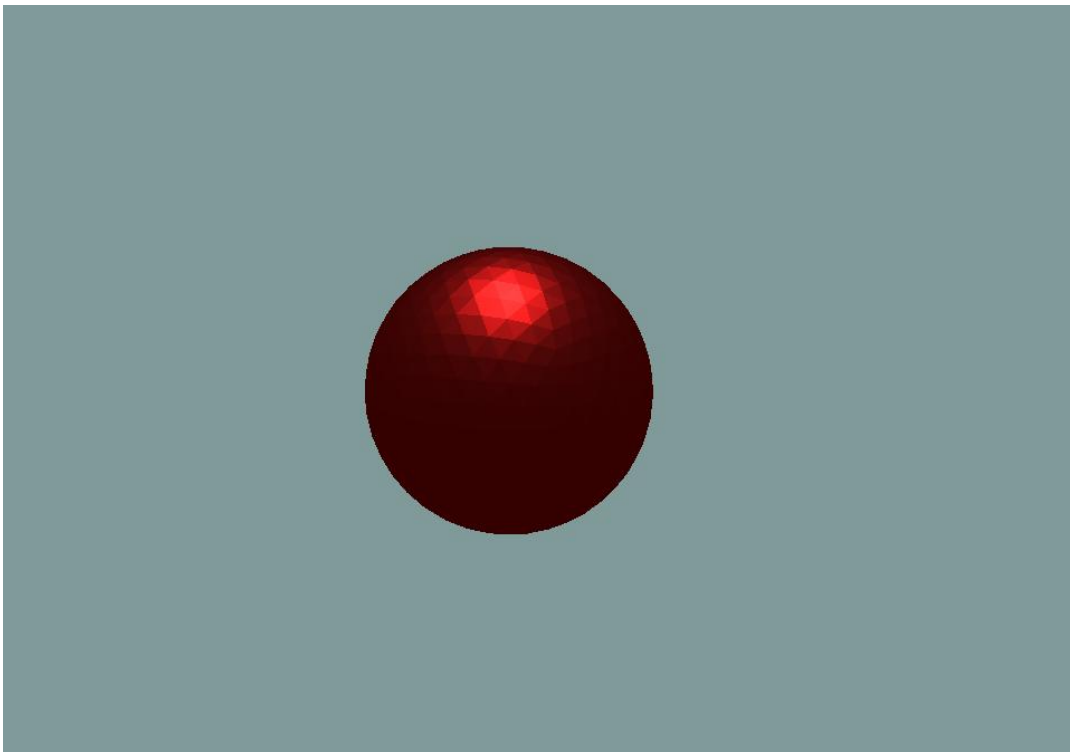

Рисунок 1.4 – Результат розрахунку освітленості поверхні сфери за допомогою методу PBS

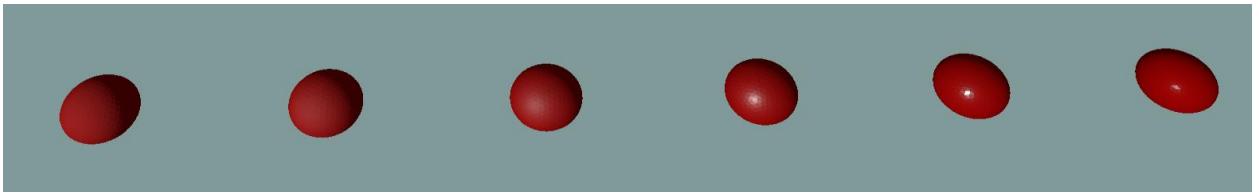

Рисунок 1.5 – Ряд сфер з різними параметрами шорсткості від 1 зліва до 0 справа

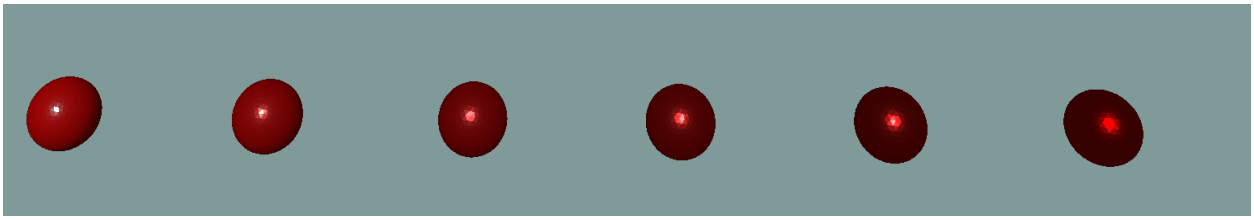

Рисунок 1.6 – Ряд сфер з різними параметрами металічності від 0 зліва до 1 справа

PBS також відмінно працює у зв'язці з освітленням на основі зображень (Image-BasedLighting, IBL), використовуючи HDR-карти середовища для моделювання складних і динамічних сценаріїв освітлення. Це забезпечує

високу реалістичність, особливо в умовах змінного освітлення, завдяки точному відтворенню світлових взаємодій між матеріалами і оточенням.

Поява PBS у рендерінгу реального часу стала можливою завдяки значному прогресу в технологіях графічних процесорів (GPU) та алгоритмах оптимізації. Методики, такі як відкладене затемнення (deferredshading) і попередньо обчислене освітлення (pre-computedlighting), дозволили ефективно реалізувати PBS навіть у великих і складних сценах. Таким чином, PBS не лише став стандартом для високоякісного візуалізування в таких галузях, як ігрова індустрія, кінематограф і віртуальна реальність, але й відкрив нові можливості для створення фотореалістичних зображень у реальному часі.

1.3. Глобальні моделі освітлення

Глобальні моделі це категорія методів освітлення які розраховують освітлення враховуючи стан всієї сцени, це дозволяє розраховувати такі ефекти як тіні, непряме освітлення та відображення. Єдиною моделлю освітлення якою можна повністю розрахувати всю сцену без використання локальних моделей є ray-tracing. Проте, як вже згадувалося вище, це все ще надзвичайно дорога операція з точки зору розрахунку. Тому зазвичай використовуються підходи які реалізують розрахунок впливу одного конкретного ефекту і поєднують це з локальними моделями освітлення. У цьому розділі ми розглянемо декілька алгоритмів розрахунку ефектів глобального освітлення.

1.3.1.Карти тіней

Карти тіней (Shadowmaps) використовуються для розрахунку тіней від інших об’єктів або від самозатінення. Для цього сцена відмальовується з точки джерела світла і у текстуру записується дистанція до найближчого фрагменту, ця текстура і є картою тіней. Після цього матриця трансформації у координатний простір джерела світла передається до шейдеру що розраховує освітлення кожного окремого об’єкту у якому позиція фрагменту конвертується у координатний простір джерела світла після чого порівнюється дистанція розрахованого фрагменту зі значенням в карті тіней і якщо значення в карті тіней менше за розраховане нами значення це означає що цей фрагмент знаходиться у тіні і на нього не впливає це джерело світла.

Такий підхід може викликати проблеми якщо кут падіння світла дуже гострий, через те що на один піксель у карті тіней буде приходитися великий сегмент геометрії, ця проблема називається shadowacne. Для вирішення цієї проблеми зазвичай використовують деякий параметр відступу, який масштабується від косинусу кута падіння світла:

$$b = b_{max}\,(l \cdot \mathbf{n})$$

Це дає доволі якісний результат і дозволяє значно підвищити якість картинки за порівняно малу ціну на обчислення.

1.3.2.Карти оточення

Карти оточення використовуються для розрахунку відображень і непрямого освітлення. Карта оточення представляє собою текстуру в яку записується оточення з усіх боків, що робить розрахунок відображень порівняно тривіальною задачею, адже маючи карту оточення можна вибирати колір відображення напряму з неї не витрачаючи час на важкі розрахунки.

Рисунок 1.7 – Приклад карти оточення

Використання карти оточення для розрахунку непрямого освітлення потребує деякої предобробки зображення, для цього потрібно розмити зображення щоб зімітувати розсіяння світла, це дозволить отримати значно стабільніші результати при розрахунку освітлення адже дозволить уникнути ситуації де при імітації променів світла які значно розсіюються ми можемо отримати різні кольори в сусідніх фрагментах.

1.3.3. Розсіяне затемнення (Ambientocclusion)

Цей підхід до глобального затемнення дозволяє розраховувати затемненість геометрії яка знаходиться у місцях куди проникає недостатньо світла, або у місцях де це світло розсіюється в протилежну від спостерігача

сторону. Найкращим прикладом можуть бути кути у приміщеннях, які завжди здаються темнішими через те що світло відбивається від однієї стіни у другу і потім знову у першу. Через це спостерігача що стоїть у центрі кімнати досягає значно менша кількість світла ніж від середини стіни. Серед підходів до розрахунку розсіяного затемнення є такі алгоритми як: SSAO, HBAO та HBAO+. В той час як вони дозволяють покращити якість зображення вони потребують не самих типових обчислень на GPUщо включають в себе вибірку псевдовипадкових векторів від фрагменту та знаходження перетину з геометрією у напрямку цього вектора. Такі операції значно знижує ефективність рендерінгу, через це розрахунок розсіяного затемнення зазвичай роблять на етапі створення геометрії. Художники створюють карту розсіяного затемнення або прораховують її на етапі експорту, що дозволяє не витрачати ресурси під час рендерінгу і просто вибирати значення з предпрорахованих карт затемнення.

1.4 Мета та постановка завдання

Метою передатестаційної практики є оглядалгоритмів та підходів до розрахунку локального та глобального освітлення 3Dсцени, реалізація рендереру що дозволить використовувати різні підходи до розрахунку освітлення і розширювати його додаванням різних ефектів.

Основнми задачами роботи є:

- Аналіз алгоритмів розрахунку освітлення;
- Реалізація процесу рендерінгу на основі OpenGLAPI;
- Написання шейдерів для розрахунку локальних моделей освітлення;
- Реалізація розрахунку тіней та відображень;

2ПІДХОДИ ДО РЕАЛІЗАЦІЇРЕНДЕРЕРУ

У даному розділі розглянутоі проаналізовано різні підходи до реалізації рушію.

2.1 Об’єктно орієнтований підхід

Найбільш поширеним способом реалізації структури рушія рендерінгу є об’єктно орієнтований підхід. Він полягає в тому щоби представити кожен елемент рушію і кожен елемент сцени за допомогою інтерфейсів та класів у програмі. Давайте розглянемо типову реалізацію кожного з компонентів рушію.

2.1.1 Базова архітектура рушію та рендереру

UMLдіаграма типового рушію показана на рисунку 2.1.

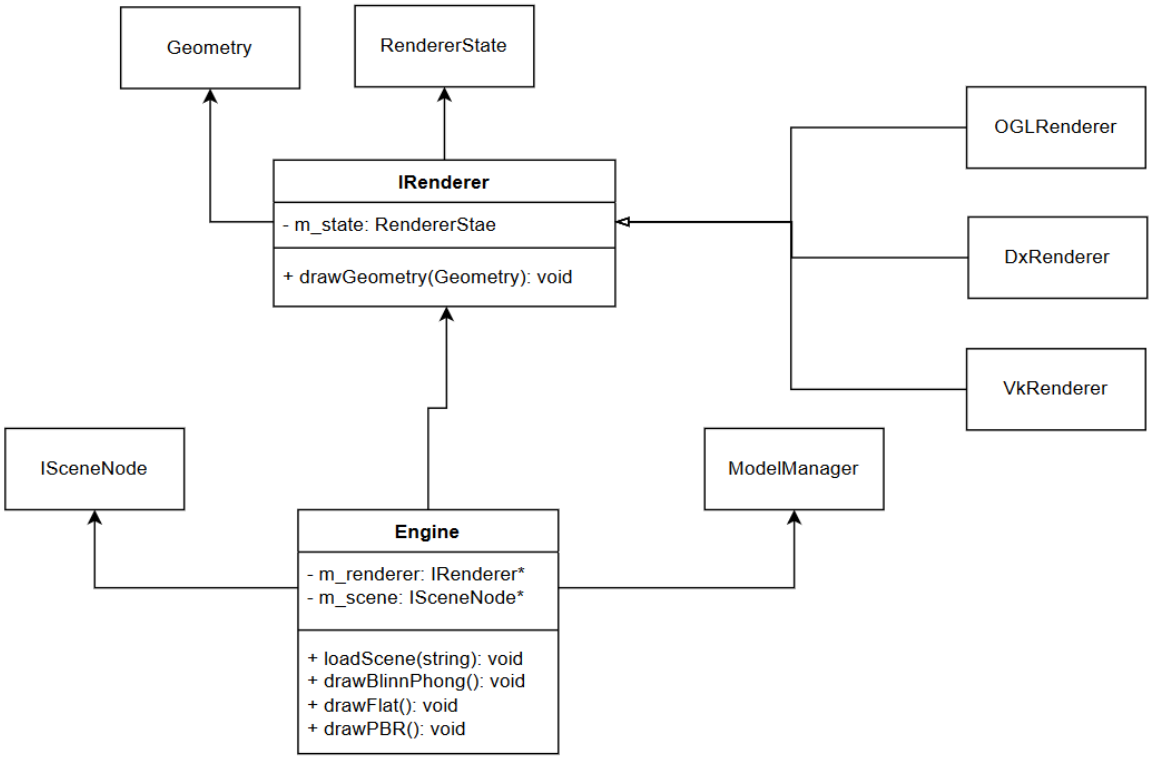

Рисунок 2.1 – UML діаграма рушію

Спочатку розглянемо інтерфейс рендереру. Серед своїх даних він містить лише поле стану рендереру. Зазвичай там зберігається значення флагів тесту глубини, відсічення задніх граней та інші флаги стану заданого API. Також цей інтерфейс відповідний за створення буферів та текстур на GPU. Від цього інтерфейсу наслідуються класи, що реалізують рендерер для конкретного API, вони можуть мати додаткові поля та методи які необхідні лише для цього API. Об’єкт що реалізує інтерфейс рендереру зберігається у об’єкті класу Engine. Engine це високорівневий клас, що означає що він не залежить від конкретної реалізації рендереру. Він зберігає сцену та відповідає за її відмалювання. Також він відповідає за увесь менеджмент ресурсів, за допомогою ModelManager. Усі виклики на відмалювання відбуваються також через нього. Тобто він являє собою основну точку для управління рендерінгом. Усі технології та підходи з допомогою яких буде відбуватися відмалювання реалізуються у цьому класі.

2.1.2 Система геометрій та матеріалів

Основою, якою представлені всі дані у рушії при цьому підході є система геометрій та матеріалів, яка абстрагується в ієрархію сцени (Scenehierarchy). UML діаграма цієї підсистеми зображена на рисунку 2.2.

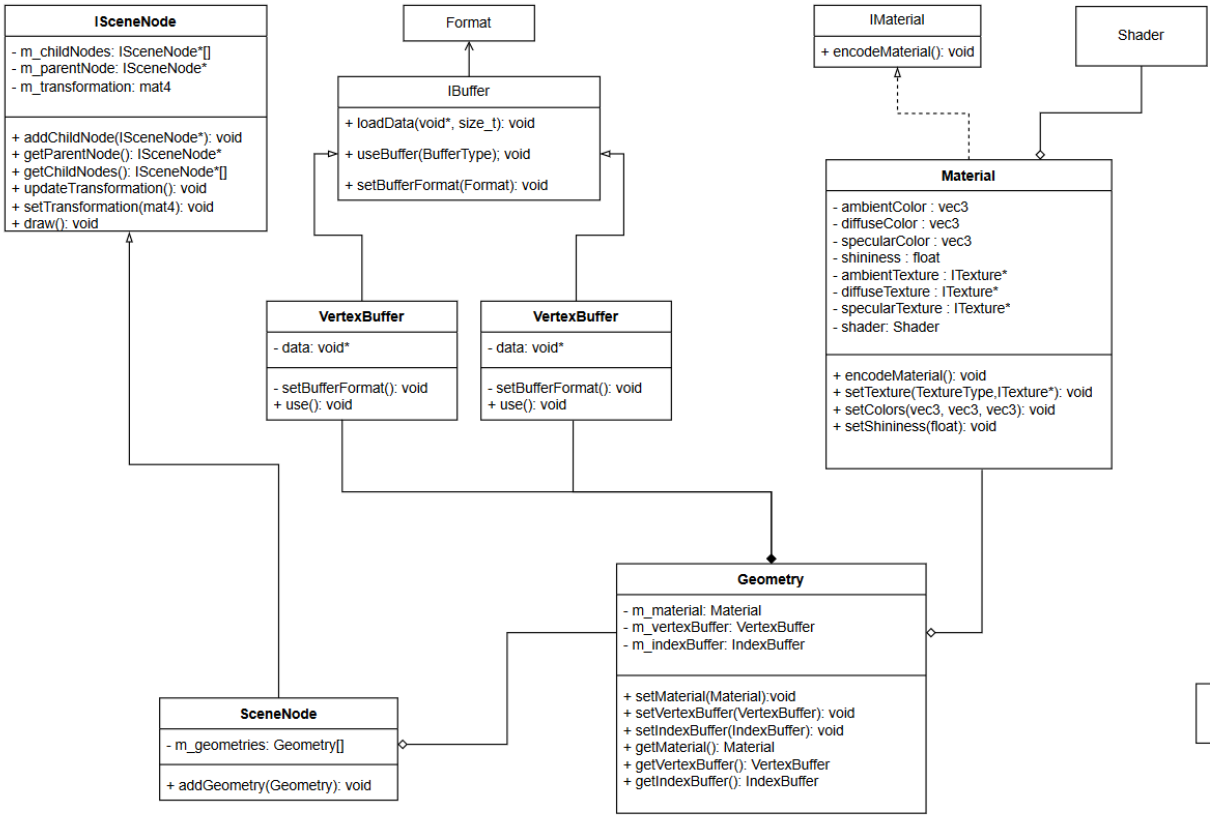

Рисунок 2.2 – Підсистема scenehierarchy

На цій діаграмі можна побачити ієрархію класів. Центральним класом в цій підсистемі є клас SceneNode. Він наслідується від інтерфейсу ISceneNode, який реалізує просторову ієрархію (spatialhierarchy), що визначає положення SceneNodeвідносно її parentnode. SceneNodeмістить масив геометрій (клас Geometry). Кожна з геометрій містить буфер вершин, буфер індексів, що наслідують інтерфейс IBuffer, і реалізують сетап власного формату. Також кожен об’єкт класу Geomtetry містить матеріал. Під матеріалом тут розуміється набор параметрів моделі освітлення що використовуються при відмалюванні цієї геометрії. У цьому прикладі я використав набір параметрів для розрахунку освітлення за моделлю Blinn-Phong.

2.1.3 Система менеджменту асетів

Система менеджменту асетів відповідає за завантаження даних необхідних рушію, та їх вивантаження, коли вони більше не потрібні. Вона реалізує доступ до файлової системи та відповідає за завантаження файлів різних форматів і створення об’єктів класу SceneNode. UML діаграма даної системи представлена на рисунку 2.3.

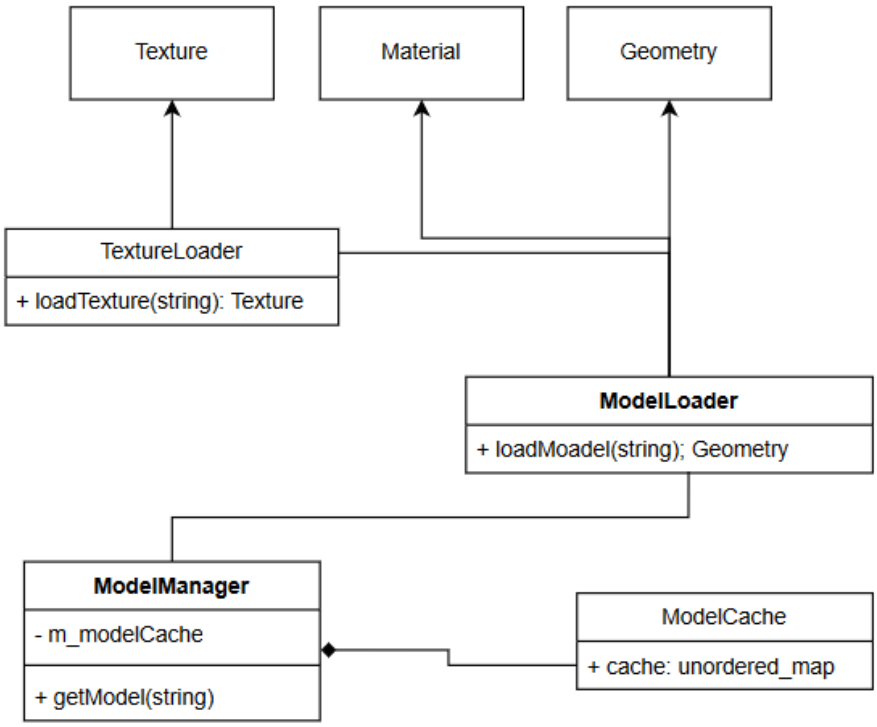

Рисунок 2.3 – Діаграма менеджеру асетів

Центральний клас цієї системи це ModelManager. За допомогою класу ModelLoaderвін завантажує об’єкти геометрій, зберігає їх у кеші, реалізованому у вигляді класу ModelCache, та конвертує в об’єкти типу SceneNode. Також реалізує функціонал звільнення місця в кеші, при недостатку пам’яті.

2.1.4 Переваги та недоліки даного підходу

Серед переваг даного підходу слід відзначити простий і прозорий інтерфейс. Він дуже мінімалістичний через те, що більшість функціоналу абстраговано саме у класі Engine. Він відповідає за завантаження і вивантаження асетів з пам’яті, реалізацію конкретних технологій рендерінгу та ефектів. Таким рушієм дуже зручно і просто користуватися.

Недоліком такого підходу є значна неповоротливість такої архітектури. Перш за все, через те що усі реалізації рендерінгу стають прихованими за інтерфейсом класу Engine, при необхідності реалізації альтернативного підходу потрібно буде вносити зміни в код рушію, що може вплинути на надійність та коректність роботи системи. Також слід відзначити, що через таку значну прихованість даних за різними класами, та інтерфейсами, можуть виникнути проблеми при спробі використати такий рушій у більшій системі (наприклад, інтегрувати фізичну симуляцію). Окрім цього, через те що за такого підходу існують класи, які агрегують велику кількість різноманітної інформації, значно уповільнюється процес, так як процесор не має змоги використовувати свій кеш у найбільш ефективний спосіб. Через те що між даними, які йому потрібні зберігається значна кількість даних, що не потрібні йому при цій задачі.

2.2 Підхід орієнтований на дані (Data-oriented design)

Підхід орієнтований на дані часто протипоставляється об’єктно орієнтованим підходам, але все ж він не заперечує реалізацію за допомогою ієрархії класів. Але наголошує на ефективній організації даних для підвищення швидкості доступу до пам’яті для CPU.

За цим підходом, одноманітні дані які будуть використовуватися одночасно повинні зберігатися в одному місці для всії об’єктів. У своєму чистому вигляді цей підхід декларує створення деякої кількості масивів, що послідовно зберігають в пам’яті конкретну властивість кінцевого об’єкту. Сам об’єкт при такому підході являє собою простий ідентифікатор, за яким відбувається відбір значень з цих масивів. Цей підхід рідко використовують в такому вигляді.

Зазвичай розробники намагаються створювати більш знайомі іншим розробникам інтерфейси для полегшення використання рушя, тому більш звичною формою є система компонентів. Коли об’єкт зберігає масив компонентів, що реалізують посилання на ці дані у інших таблицях.

Розглянемо реалізацію подібної системи з використанням підходу орієнтованого на дані.

2.2.1 Базова архітектура рушію і рендереру.

UMLдіаграма рушію та рендереру представлена на рисунку 2.4.

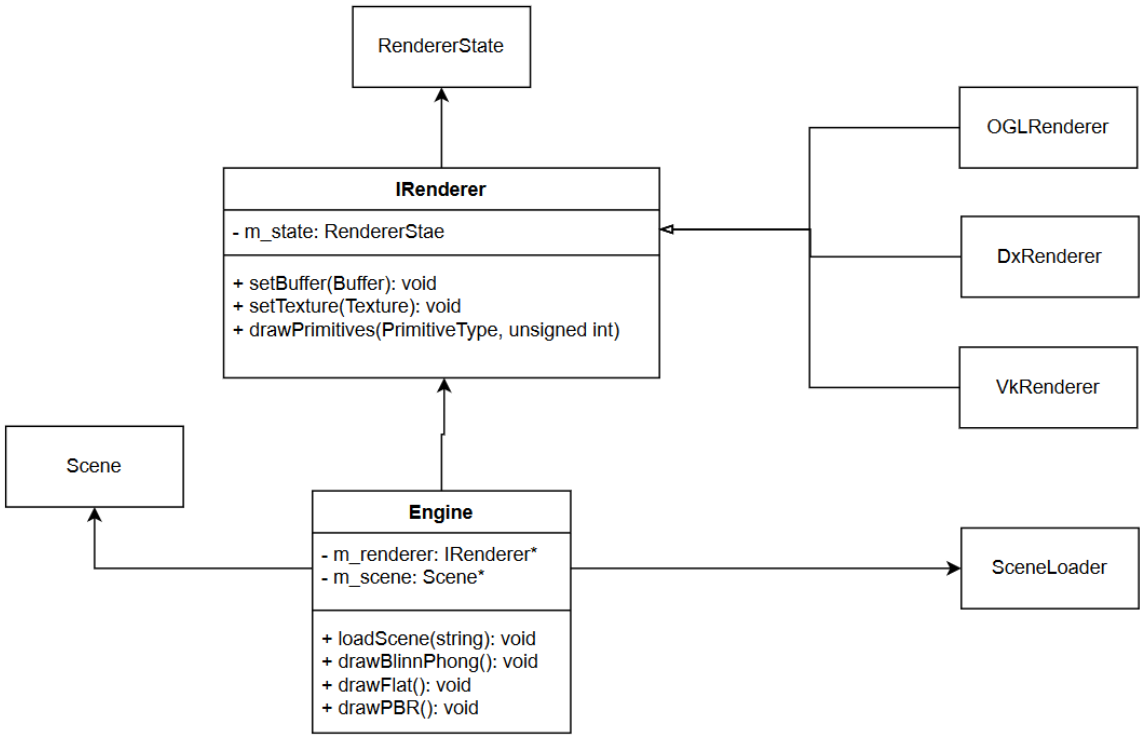

Рисунок 2.4 – UMLдіаграма рушію та рендереру

Як можна побачити, архітектура рендереру майже ідентична тій, що зображена на рисунку 2.1. Основною відмінністю є те, що рендерер тут є значно більш абстрактним і у своєму інтерфейсі має лише базові операції, тож

уся логіка відмалювання відбувається саме у класі Engine. Також сама сцена тут представлена іншим типом, про деталі організації якого йдеться далі.

2.2.2 Система геометрій та матеріалів

Для максимального збільшення ефекитвності доступу до даних система елементів сцени організована не дуже інтуітивним чином, UMLдіаграма представлена на рисунку 2.5.

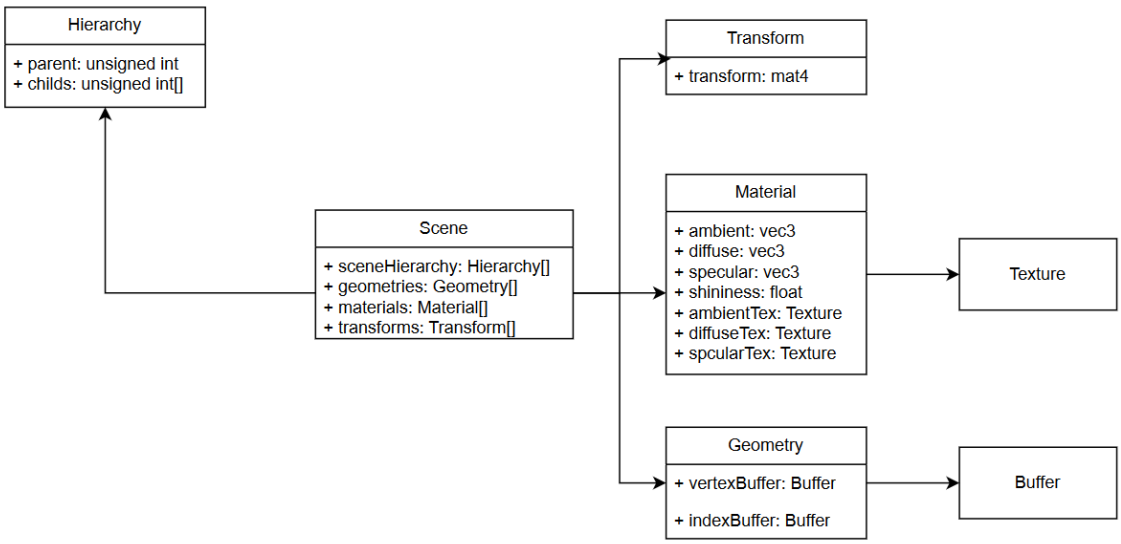

Рисунок 2.5 – UML діаграма системи геометрій та матеріалів при data-oriented підході

Основним класом у цій підсистемі є Scene. Він зберігає набір масивів, кожен з котрих містить послідовність однотипових компонентів елементів сцени. Класу, що представляє елемент сцени, на відміну від об’єктно орієнтованого підходу, тут немає, при цьому підході елемент сцени представлений сукупністю всіх своїх компонентів з різних масивів. При цьому підході, набір компонентів, що мають однакові індекси в відповіднх масивах, відносяться до одного елементу сцени (тобто перший елемент на сцені представлений набором geometries[0], materials[0], transforms[0]). А структура

цих елементів організована за допомогою класу Hierarchy, який зберігає індекс батьківського елемента (parent) і масив індексів дітей (childs).

2.2.3 Система завантаження асетів

Система завантаження асетів має такий самий вигляд як і на малюнку 2.3. Змінюється тільки тип поветраємого значення, адже при такому підході організація системи елементів сцени значно відрізняється. Варто лише зауважити, що робота по менеджменту строку зберігання асетів у пам'яті здебільшого перекладена на клас Engine, у той час як SceneLoaderвідповідає лише за завантаження даних у програму.

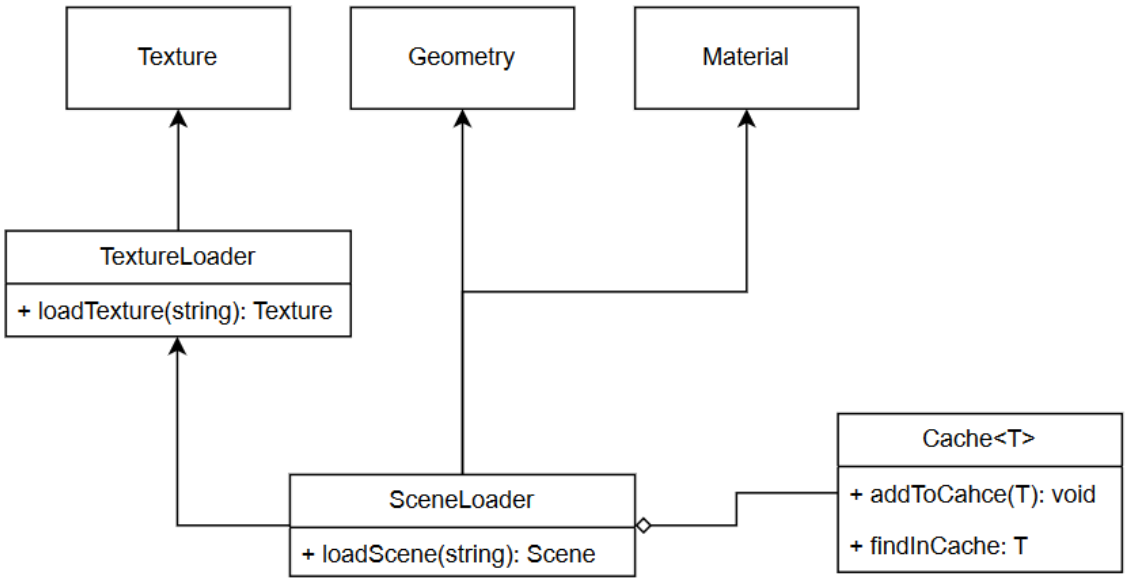

Рисунок 2.6 – UMLдіаграма системи завантаження асетів

2.2.4 Переваги та недоліки даного підходу

Data-orienteddesign був запропонований як рішення проблеми ефективності організації даних при класичному об'єктно-орієнтованому підході. Через це, його основним недоліком є те, що такий підхід не дуже знайомий більшості програмістів, через що можуть виникати труднощі з

використанням бібліотек бобудованих з використанням такого дизайну. Також цей метод пропагує мінімізацію використання вказівників, через те що непрямий доступ до пам’яті не утилізує максимально ефективно систему cahceprefetching, яка реалізована на процесорі. Це може значно погіршити гнучкість системи через обмеженість можливості використання поліморфізму.

Значною перевагою такого підходу є його надзвичайна ефективність. Для прикладу порівняємо ефективність оновлення трансформацій сцени при data-oriented і object-oriented підходах. При об’єктно орієнтованому підході клас SceneNode має розмір 176 байтів (88 байти для матеріалу, 32 байти для матриці трансформацій, 32 байти для буферів геометрії, 16 байтів для вказівників для організації ієрархії сцени, 8 байтів для vtable). При величині кеш лінії у 128 байтів можна говорити що кожен раз коли ми намагаємося отримати доступ до наступного елементу, ми звертаємось до різних кеш ліній, при цьому виростає вірогідність cache-miss, коли дані завантажені на кеш найнижчого рівня і виникає потреба завантажувати їх з кешу вищого рівня або оперативної пам’яті. За таких обставин час доступу до пам’яті виростає від 3 до 100 разів. В свою чергу при підхдоді орієнтованому на дані ми маємо масив щільно організованих матриць, що дає нам розмістити 4 матриці у кеш лінії.На моєму процесорі IntelCoreI7-7700HQє 64 кілобайти кешу першого рівня, 256 кілобайтів кешу другого рівня і 6 мегабайт кешу третього рівня.За такого рівня кеш першого рівня може містити ~360 об’єктів сцени при об’єктно орієнтованому підході і 2000 матриць трансформації при підході орієнтованому на дані. Якщо взяти приблизні показники доступу до різних рівнів кешу, де доступ до кешу першого рівня $x_1 = 3$ цикла процесора, доступ до кешу другого рівня $x_2 = 3 * x_1$, а до третього $x_3 = 5 * x_2$, то ми отримаємо що в той час коли час доступу до 2000 матриць трансформації при data-orientedпідході займає 12 000 циклів, то доступ до такої ж кількості матриць трансформації при obect-orientedпідході займе 360 * 3 + 1080 * 9 + 560 * 45 = займе 36 000 циклів.

ЗРЕАЛІЗАЦІЯ РЕНДЕРЕРУ І ТЕСТОВОГО ДОДАТКУ

У цьому розділі представлена реалізація рендереру на мові c++ з використанням OpenGLAPI.

3.1 Реалізація рендереру

За основу було взято підхід орієнтований на дані для збільшення ефективності. Було внесено значні корективи у підхід описаний вище. Як основний APIдля реалізації рушія було вибрано OpenGLчерез те що він потребує значно менший об’єм коду для отримання бажаного результату. Але за наявної організації залишається можливість інтегрувати інші API такиі як: Direct3D, Vulkan чи Metal. Для загрузки адрес функцій OpenGlAPI з dll файлу використовується бібліотека glad.

3.1.1 Абстрагування об’єктів OpenGL у класи

Для більшої зручності та гнучкості, абстрагуємо необхідні нам об’єкти у класи.

Першим розглянемо клас текстури, що представлений у файлі Texture.hy додатку А.

У ньому представлені набори значень enum, які використовуються для встановлення параметрів формату та відображення текстури. Для створення текстури достатньо завантажити масив кольорів пікселів в пам’ять та передати параметри текстури разом з масивом у функцію create. Ця функція створить об’єкт текстури з відповідними параметрами у пам’яті GPU, після чого її можна буде використовувати визвавши функцію setActivei передавши їй значення відповідного слоту для прив’язки текстури.

Далі, розглянемо буфери. В OpenGL є декілька різних типів буферів, але для нас цікаві лише два: буфер вершин (vertexbuffer) та буфер індексів (indexbuffer). Обидва типа буферів реалізовані одним классом Buffer.

```
classBuffer
{
public:
enumType
{
VERTEX,
INDEX
};

enumUsage
{
STATIC,
DYNAMIC,
};

voidcreate();
voidfree();
voidbind(Typetype)const;

voidsetData(Typetype,Usageusage,unsignedintsize,constvoid*data)const;
voidsetSubData(Typetype,unsignedintoffset,unsignedintsize,constvoid*data);

private:
unsignedintm_bufferId;
};
```

Приклад 3.1 – Класс Buffer

Для використання цього класу потрібно створити буфер визвавши функцію create, а потім завантажити в нього дані за допомогою функції setData. Після цього клас виділить пам’ять на GPUдля зберігання переданих даних. Щоб дані з буфера передавались у шейдер під час малювання його потрібно прив’язати (bind) до відповідного слоту (vertexbuffer, indexbuffer), для цього потрібно викликати функцію bind з відповідними даними. Для видалення буферу з GPU потрібно викликати функцію free.

У буфері вершин, окрім позицій, можуть зберігатися різноманітні дані, такі як: текстурні координати, вектори нормалі, колір та ін. Через це виникає необхідність якось розмічувати буфер і повідомляти OpenGL про те як інтерпретувати дані які там зберігаються. За це відповідає клас BufferState.

```
enumclassBufferDataType
```

```
{
INT,
FLOAT,
VEC2,
VEC3,
VEC4,
MAT2,
MAT3,
MAT4,
};

structVertexAttribute
{
BufferDataTypetype;
unsignedintoffset=0;
};

structVertexFormat
{
std::vector<VertexAttribute>attributes;
unsignedintsize=0;
unsignedintbaseOffset=0;
};

classBufferState
{
public:
voidcreateBufferState(constBuffer*vertexBuffer,constBuffer*indexBuffer,constVertexFormat&format);
voidfree();

voidbound()const;
voidunbound()const;

constVertexFormat&getVertexFormat()const;
private:
VertexFormatm_format;
unsignedintm_vaoId;
};
```

Приклад 3.2 – Клас BufferState

Цей клас зберігає інформацію про формат буферу. Формат представлений як набір атрибутів, відступ від початку буферу та розмір однієї вершини. Кожен атрибут представлений своїм типом, та відступом від початку вершини. Для використання BufferState необхідно створити буфери вершин та індексів, завантажити в них необхідну інформацію, потім створити об’єкт типу VertexFormat і заповнити його відповідними компонентами. Після цього необхідно викликати функцію createBufferState і передати їй буфери разом з об’єктом формату. Після цього створиться об’єкт BufferState, який дає можливість використовувати буфери з певним форматом, не розмічаючи його кожен раз при прив’язуванні буферів. Також цей клас позбавляє нас потреби прив’язувати буфери кожен раз для використання, достатньо лише визвати

функцію bound на об’єкті типу BufferState і він автоматично прив’яже усі необхідні буфери.

Останнім об’єктом який ми розглянемо буде шейдерна програма (файли ShaderProgram.h та ShaderProgram.cpp у додатку А). Шейдерна програма складається з двох елементів: вершинного (vertex) та фрагментного (fragment) шейдерів (фрагментний шейдер також називають піксельним (pixel) шейдером). Вершинний шейдер виконується для кожної вершини яка відмальовується у відповідності з вмістом вершиного та індексного буферів та рахує положення вершини на екрані у нормалізованих координатах пристрою (Normalizeddevicecoordinated або NDC). Це координати у тривимірному просторі які знаходяться у межах [-1, 1] по кожній з осей координат. Якщо якась вершина має координати що виходять за цей діапазон, то вона вважається вершиною, що знаходиться за межами поля зору. Також вершинний шейдер може передавати інші дані у фрагментний шейдер. Фрагментний шейдер викликаєтсья піля етапу растеризації для кожного пікселя геометрії яка відмальовуєтсья та розраховує колір пікселю.

Цей клас містить велику кількість метаданих для підвищення зручності його використання. Перш за все він містить інформацію про inputlayout – тобто формат вершиного буферу, який приймає цей шейдер, а також набір юніформів для вершиного і фрагментного розрахунку. Юніформи (Uniforms) – це дані, які не змінюються на протязі всього виклику відмалювання, тобто не залежать від вершин та індексів. Зазвичай через юніформи передають індекси текстур, матриці трансформацій та властивості матеріалів. Для використання цього класу вам потрібно передати код вершиного і фрагментного шейдерів до функції createShader. Після чого вам потрібно додати усі юніформи які ви збираєтесь використовувати. Після цього ви можете використовувати шейдер, викликавши функцію use. Для того щоб передати значення у потрібну вам юніформу, вам потрібно викликати метод getVertexShaderUniform або getFragmentShaderUniform і передати йому назву юніформи. Цей метод повертає об’єкт типу ShaderUniform який підтримує основні типи даних для

передачі у шейдер, для встановки потрібного значення, вам просто потрібно призначити його полю, що відповідає типу (intValue, floatValue, vec2Value і так далі). Після цього передати цей об’єкт у функцію setShaderUniform.

Це був останній клас, який абстрагував об’єкт OpenGL. Як можна побачити, ці класи не формують якусь певну ієрархію і залежність один від одного, за виключенням випадку BufferState який повинен бути залежним від об’єкту буферу. Це було зроблено спеціально для того, щоб дати максимальну гнучкість у питанні менеджменту ресурсів. Адже через те, що у програмі немає єдиного класу який відповідає за створення ресурсів, існує можливість створювати власні системи менеджменту, які будуть більш адаптовані під конкретний проект.

3.1.2 Система геометрій, матеріалів та елементів сцени

Система геометрій та матеріалів представлена простими структурами що зберігають відповідну інформацію. Така реалізація дозволяє повністю делегувати використання цих даних для рендереру.

```
structMaterial
{
TexturediffuseTex;
TexturespecularTex;
TextureemissiveTex;
TexturenormalTex;

glm::vec3ambient;
glm::vec3diffuse;
glm::vec3specular;
floatshininess;
};

structGeometry
{
BuffervertexBuffer;
BufferindexBuffer;
BufferStatebufferState;
unsignedintmaterial;
unsignedintindexSize;
};
```

Приклад 3.3 – Структури матеріалу та геометрії

Матеріал містить набір текстур та кольорів для кожної з компонентів моделі освітлення Blinn-Phong, коефіцієнт блиску, а також текстуру нормалей. Геометрія містить буфери вершин та індексів, об’єкт стану буферів, кількість вершин і ідентифікатор матеріалу.

```
structSceneStorage;

classSceneNode
{
public:
SceneNode();

conststd::vector<unsignedint>&getChilds()const
{returnm_childNodes;}

conststd::vector<unsignedint>&getGeometries()const
{returnm_geometries;}

constglm::mat4&getWorldTransform()const
{returnm_worldTransform;}

voidsetLocalTransform(constglm::mat4&mat)
{m_localTransform=mat;}

constglm::mat4&getLocalTransform()const
{returnm_localTransform;}

voidupdateWorldTransform();

private:
std::vector<unsignedint>m_childNodes;
std::vector<unsignedint>m_geometries;
glm::mat4m_localTransform;
glm::mat4m_worldTransform;
SceneStorage*m_storage;
intm_parentNode;

friendclassSceneAssembler;
};

structSceneStorage
{
std::vector<Geometry>geometries;
std::vector<Material>materials;
std::vector<SceneNode>nodes;

voidfreeStorage();
};
```

Приклад 3.4 – Система елементів сцени

Система елементів сцени складається з класу SceneNode і SceneStorage. SceneNode репрезентує елемент просторової ієрархії (spatialhierarchy). Він відповідає за поширення трансформації від батьківських елементів до елементів дітей, за це відповідають дві матриці трансформації

m_localTransformim_worldTransform. Перша – це матриця трансформації в координатному просторі батьківського елемента, а друга – в координатному просторі світу (worldspace), вона розраховується перемноженням матриць трансформації усіх батьківських елементів, починаючи від корньового. Також цей клас містить масив індексів елементів дітей і масив індексів геометрій, які будуть відмальовані з урахуванням трансформації цього елементу. Також міститься індекс батьківського елементу та вказівник на об’єкт SceneStorage який зберігає цей елемент сцени. Також слід відзначити функцію оновлення worldtransform – updateWorldTransform. Вона рекурсивно проходиться по ланцюгу батьківських елементів до корньового елемента і оновлює трансформацію у координатному просторі світу.

Об’єкт SceneStorage виконує функцію контейнеру для всіх даних сцени. Індекси які використовуються в об’єктах SceneNodeiGeometry вказують на індекси саме в масивах, що зберігаються в SceneStorage. Функція freeStorage використовується для видалення ресурсів що зберігаються у SceneStorage.

```
SceneNode::SceneNode()
:m_localTransform(glm::mat4(1.0f))
,m_worldTransform(glm::mat4(1.0f))
,m_parentNode(-1)
,m_storage(nullptr)
{}

voidSceneNode::updateWorldTransform()
{
if(m_parentNode==-1)
{
m_worldTransform=m_localTransform;
return;
}

m_worldTransform=m_storage->nodes[m_parentNode].getWorldTransform()*m_localTransform;
}

voidSceneStorage::freeStorage()
{
for(auto&geometry:geometries)
{
geometry.bufferState.free();
geometry.vertexBuffer.free();
geometry.indexBuffer.free();
}

for(auto&material:materials)
{
material.diffuseTex.free();
material.specularTex.free();
}
```

```
geometries.clear();
materials.clear();
nodes.clear();
}
```

Приклад 3.5 – Реалізація системи елементів сцени

3.1.3 Класи рушію та рендереру

Клас рушію представляє собою простий клас контейнер для об’єктів що реалізують інтерфейс рендереру. Рендерер представлений у вигляді інтерфейсу для того щоб дозволити розширення системи без модифікації вже існуючої логіки, це дає системі значну гнучкість не жертвуючи надійністю.

```
classIRenderer
{
public:
virtualvoidinit(){};

virtualvoidrender()= 0;

virtualvoidshutdown(){};

~IRenderer()=default;
};
```

Приклад 3.6 – Інтерфейс рендереру

Ітерфейс рендереру представляє собою простий абстрактний клас з трьома методами. Для реалізації цього інтерфейсу достатньо просто перегрузити функцію render.

```
classEngine
{
public:
voidaddRenderer(IRenderer*renderer);

voidrender();

voidshutdown();

private:
std::vector<IRenderer*>m_renderers;
};
```

Приклад 3.7 – Клас Engine

```
voidEngine::addRenderer(IRenderer*renderer)
{
renderer->init();
m_renderers.push_back(renderer);
}

voidEngine::render()
{
for(autorenderer:m_renderers)
{
renderer->render();
}
}

voidEngine::shutdown()
{
for(autorenderer:m_renderers)
{
renderer->shutdown();
}
}
```

Приклад 3.8 – Реалізація класу Engine

Клас Engine лише викликає методи інтерфейсу рендереру. Основна робота відбувається в конкретній реалізації рендереру.

```
constunsignedintMAX_BOUNDED_TEXTURES=16;

classRenderer:publicIRenderer
{
public:
structState
{
ShaderProgram*shaderProgram=nullptr;
BufferState*bufferState=nullptr;
unsignedintcurrentMaterial;

booldepthTestEnabled=false;
boolcullBackFace=false;
};

voidrender();

inlineconstState&getCurrentState()const
```

```
{returnm_currentState;}

voidsetState(constState&state);

voidsetScene(SceneNode*scene,SceneStorage*storage);
voiddrawScene();

private:
Statem_currentState;
SceneNode*m_rootNode;
SceneStorage*m_storage;
};
```

Приклад 3.9 – Клас рендереру

Клас рендереру містить в собі об’єкт стану, та вказівники на корньовий елемент сцени і на відповідний SceneStorage. З інтерфейсу рендереру цей рендерер перегружає лише функцію render так як не має особливої логіки для ініціалізації та завершення роботи. Функція setScene призначає сцену яку буде відображати цей рендерер. Функція drawScene потрібна, якщо потрібно відмалювати сцену в обхід виклику функції render в об’єкті Engine.

Процес рендерінгу в цьому рендерері відбувається з використанням трьох функцій encodeNode, encodeGeometryiencodeMaterial. Функція encodeNode встановлює матриці для трансформації геометрій у координатний простір світу, функція encodeMaterial встановлює юніформи що відповідають властивостям матеріалу геометрії, функція encodeGeometry встановлює стан буферів та робить виклик відмалювання.

3.1.4 Система загрузки ресурсів

Система загрузки ресурсів влаштована складніше ніж ті що, обговорювались досі. Увесь етап загрузки був розбитий на два етапи: загрузка сцени та асемблінг сцени. Таке рішання може покращити ефективність завантаження сцени, так як конструювання об’єктів OpenGLможливо лише в одному потоці, тому якщо робити це одним етапом, то можна викликати зависання рендерінгу, через затримки читання з файлу. Такий спосіб дозволяє виконати загрузку інформації у іншому потоці, а коли інформація вже буде завантажена у оперативну пам’ять виконати етап створення об’єктів. За це відповідають два класи SceneLoader та SceneAssembler.

Клас SceneLoader завантажує сцену з файлу (підтримуються різні формати, серед яких: .obj, .gltf, .fbx, .lwo) і створює об’єкт SceneData. Який містить масив даних геометрій, даних матеріалів та ієрархії елементів сцени. Завантаження відбувається за допомогою бібліотеки Assimp.

Варто відзначити те що завантаження текстур відбувається не на етапі завантаження, а на етапі асемблінгу. На цьому етапі ми зберігаємо лише назви файлів.

На етапі асемблінгу ми створюємо об’єкт SceneStorage з даних що отримуємо із SceneData. SceneAssembler дозволяє збирати сцену з файлу, тоді він сам створить SceneLoader і завантажить SceneData, і додавати до SceneStorage об’єкти створенні з уже завантаженої SceneData.

```
classSceneAssembler
{
public:
voidcreateSceneFromFile(conststd::string&filename,SceneStorage&storage);
voidaddSceneToStorage(constSceneData&data,SceneStorage&storage);
voidclearCache();
virtual~SceneAssembler()=default;

protected:
virtualvoidaddGeometryToStorage(constGeometryData&geomgetryData,SceneStorage&storage,unsignedintmaterilasOffset=
0);
virtualvoidloadMaterialToStorage(constMaterialData&materialData,SceneStorage&storage);
virtualvoidaddNodeToStorage(constNodeData&nodeData,SceneStorage&storage,unsignedintnodeOffset=
0,unsignedintgeometryOffset= 0);
TextureloadOrGetDefault(conststd::string&textureName,conststd::string&defaultTex);
};
```

Приклад 3.10 – Клас SceneAssembler

Так як зміна шейдеру це дуже дорога операція з точки зору GPU, рушій намагається рендерити як можна більше геометрії одним шейдером. Для цього, якщо матеріал не має якоїісь з текстур, клас SceneAssembler генерує текстури за замовчуванням і використовує їх.

Цей клас створює буфери вершин і індексів, задає стан буферів за допомогою класу BufferState, завантажує текстури та створює вихідний об’єкт SceneStorage.

Завантаження текстур відбувається за допомогою бібліотеки stb. Реалізація класу TextureLoaderпредставлена нижче.

```
Texture::ColorFormatgetColorFormat(intchannels)
{
switch(channels)
{
case 1:
returnTexture::R;
case 2:
returnTexture::RG;
case 3:
returnTexture::RGB;
case 4:
returnTexture::RGBA;
}
}

boolTextureLoader::load(conststd::string&path,Texture*dstTexture)
{
intwidth,height,channels;
stbi_set_flip_vertically_on_load(true);
unsignedchar*data=stbi_load(path.c_str(),&width,&height,&channels, 0);

if(!data)
{
std::cout<<"[--ERROR--]"<<" Unable to load texture: "<<path<<std::endl;
returnfalse;
}

dstTexture->create(width,height,Texture::TEXTURE2D,getColorFormat(channels),(char*)data);

stbi_image_free(data);
returntrue;
}
```

Приклад 3.11 – Реалізація класу TextureLoader

Також для кешування даних був створений шаблонний клас Cache<T>. Він представляє собою набір сінглтон об’єктів які кешують дані заданого формату.

```
template<typenameT>
classCache
{
public:
staticCache<T>&get()
{
staticCache<T>cache;
returncache;
}

voidaddToCache(conststd::string&id,Tres)
{
constauto&it=m_cache.find(id);

if(it==m_cache.end())
{
m_cache.emplace(id,res);
}
}
```

```
boolfindInCache(conststd::string&id)
{
constauto&it=m_cache.find(id);

if(it==m_cache.end())
{
returnfalse;
}

returntrue;
}

boolfindInCache(conststd::string&id,T&dst)
{
constauto&it=m_cache.find(id);

if(it==m_cache.end())
{
returnfalse;
}

dst=it->second;
returntrue;
}

voidclearCache()
{
m_cache.clear();
}
private:
Cache()=default;
std::unordered_map<std::string,T>m_cache;
};
```

Приклад 3.12 – Клас Cache<T>

3.2 Створення тестового додатку на основі розробленого рендереру

3.2.1 Абстрагування платформи

Для абстрагування платформи був створений абстрактний клас App.

```
classApp
{
public:
voidsetName(std::stringname)
{m_name=name;}

virtualvoidinit()= 0;
virtualvoidupdate()= 0;
virtualvoidshutdown()= 0;

voidexec()
{
init();
while(!m_shouldClose)
{
update();
}
```

```
shutdown();
}
protected:
std::stringm_name;
boolm_shouldClose;
};
```

Приклад 3.13 – клас App

Він представляє собою абстрактний клас, який контролює процес викоання додатку. Для реалізації додатку створимо клас OpenGlApp. Для абстрагування платформи була використана бібліотека glfw.

Цей клас являє собою просто контейнер для різних елементів додатку і реалізує обробку вводу користувача, і оновлення даних для відмалювання.

Реалізація класу міститься у файлах OpenGlApp.hiOpenGlApp.cpp вміст яких представлений у додатку А.

3.2.2 Додатковий рендерер користувацького інтерфейсу

Також був створений рендерер, для відображення користувацького інтерфейсу і маніпулювання елементами на сцені. Він дозволяє змінювати текстури в матеріалах, дивитися додаткову інформацію про сцену, та завантажувати інші сцени.

```
classResourceMenuRenderer:publicIRenderer
{
public:
structConfig
{
floatxyzRotation[3]={ .0f, .0f, .0f };
floatscale=1;
};

voidinit();
voidrender();
voidshutdown();

voidsetGLFWwindow(GLFWwindow*window);
voidsetResourcePath(std::stringpath);
voidsetSceneData(SceneData*data);

ConfiggetConfig(){returnm_globalConfig;}

boolm_reuploadScene=false;
boolm_updateShader=false;
protected:
voiddrawMaterialTab();
voiddrawGlobalTab();
```

```
voiddrawExampleFolder(conststd::filesystem::directory_entry&entry);
voiddrawExampleFile(conststd::filesystem::directory_entry&entry);

voiddrawNodeMenuItem(NodeData*nodeData);
voiddrawMaterialMenuItem(MaterialData*materialData);

voiddrawNodeEditWindow(NodeData*nodeData);
voiddrawGeometryInfo(GeometryData*geometry);

voiddrawMetadataOverlay();

private:
SceneData*m_sceneData;
GLFWwindow*m_window;
std::stringm_resourcePath;
std::stringm_currTexturesPath;
Configm_globalConfig;
std::vector<std::string>m_SupportedExtentions={".obj",".gltf",".glb",".fbx",".dae",".lwo"};
};
```

Приклад 3.14 – Класс рендереру меню

Реалізація рендерінгу інтерфейсу була зроблена за допомогою бібліотеки imGui (файл ResourceMenuRenderer.cpp у додатку А).

3.2.3 Написання шейдерів

При написанні шейдерів для розрахунку різних моделей освітлення використовувався один і той же шейдервершин.

```
#version 330 core

layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 normal;
layout(location = 2) in vec3 tangent;
layout(location = 3) in vec3 bitangent;
layout(location = 4) in vec2 texCoord;

uniform mat4 worldTransform;
uniform mat4 viewProjTransform;
uniform mat4 lightCameraTransform;

uniform vec3 eyePos;

out vec3 fsFragPos;
out vec3 fsEyePos;
out mat3 TBN;
out vec2 fsTexCoord;
out vec4 fsFragPosInLightSpace;
```

```
void main()
{
   fsFragPos = vec3(worldTransform * vec4(pos, 1.0));
   fsFragPosInLightSpace = lightCameraTransform * vec4(fsFragPos, 1.0);
   fsEyePos = eyePos;
   vec3 T = normalize(vec3(worldTransform * vec4(tangent, 0.0)));
   vec3 B = normalize(vec3(worldTransform * vec4(bitangent, 0.0)));
   vec3 N = normalize(vec3(worldTransform * vec4(normal, 0.0)));
   TBN = mat3(T, B, N);
   fsTexCoord = texCoord;
   gl_Position = viewProjTransform * worldTransform * vec4(pos, 1.0);
}
```

Приклад 3.15 –Шейдер вершин для розрахунку моделей освітлення

Дле реалізації моделі flat-shadingпотрібно знати лише три параметри. Напрямок світла, колір поверхні і вектор нормалі. Ці дані передаються у наступний шейдер.

```
#version 330 core

uniform vec3 diffuse;

in vec3 fsFragPos;
in vec3 fsEyePos;
in mat3 TBN;
in vec2 fsTexCoord;

uniform vec3 lightDir;

out vec4 FragColor;

void main()
{
   vec3 normal = normalize(TBN * vec3(0.0, 0.0, 1.0));

   vec3 color = diffuse * (dot(normal, normalize(lightDir)))
   FragColor = vec4(color, 1.0);
}
```

Приклад 3.16 –Фрагментнийшейдер для розрахунку моделі освітлення за допомогою моделі flat-shading

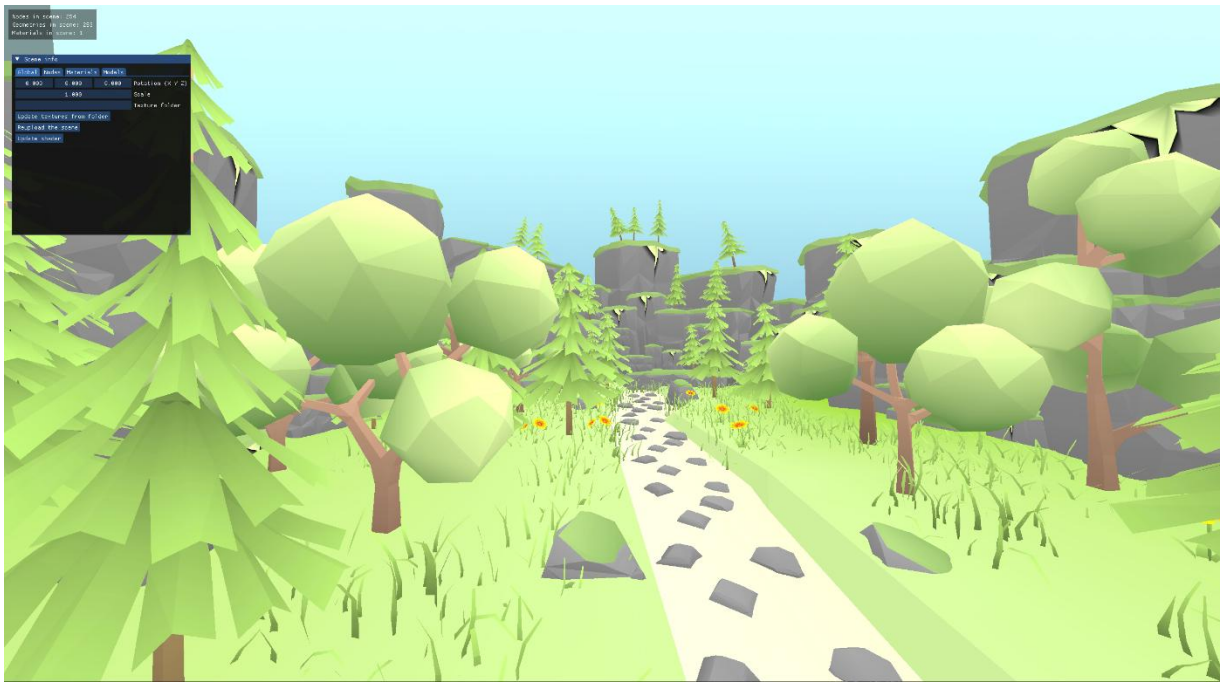

Рисунок 3.1 – Модель лісу розрахована за допомогою моделі flat-shading

Рисунок 3.2 - Модель приміщення розрахована за допомогою моделі flat-shading

Для реалізації моделі освітлення Бліна-Фонга використовується більш складний шейдер. Він приймає на вхід значення всіх компронентівмоделі освітлення Бліна-Фонга а також відповідні текстури, на додачу у цьому шейдері використовуються карти нормалей, що задають направлення вектору нормалі в

залежності від його положення на поверхні. Це дозволяє значно покращити якість вихідної картинки без необхідності використовувати складну геометрію.

```
#version 330 core

uniform vec3 ambient;
uniform vec3 diffuse;
uniform vec3 specular;
uniform float shininess;

uniform sampler2D emissiveTex;
uniform sampler2D diffuseTex;
uniform sampler2D normalTex;
uniform sampler2D specularTex;

uniform vec3 lightDir;

in vec3 fsFragPos;
in vec3 fsEyePos;
in mat3 TBN;
in vec2 fsTexCoord;

out vec4 FragColor;

void main()
{
   vec3 normal = normalize(TBN * (texture(normalTex, fsTexCoord).xyz * 2 - 1));
   vec3 lightIntensity = vec3(1.0, 1.0, 1.0);
   vec3 diffuseColor = max(dot(normal, lightDir), 0) * diffuse * texture(diffuseTex, fsTexCoord).xyz;

   vec3 eyeDir = normalize(fsEyePos - fsFragPos);
   float specularFactor = pow(max(dot(normalize(eyeDir + lightDir), normal), 0), shininess);
   vec3 specularColor = specular * specularFactor * texture(specularTex, fsTexCoord).xyz;

   vec3 linearColor = texture(emissiveTex, fsTexCoord).xyz + diffuseColor + specularColor;
   vec3 color = pow(linearColor * lightIntensity, vec3(1.0 / 2.2));
   FragColor = vec4(color, 1.0);
}
```

Приклад 3.17 – Фрагментнийшейдер для розрахунку освітлення за методом Бліна-Фонга

Рисунок 3.3 – Приклад сцени розрахованої методом Бліна-Фонга

Для розрахунку освітлення методом PBSу шейдерподаються всі компоненти цієї моделі освітлення (колір, металічність та шорсткість) і відповідні текстури. На додачу до цього подаються карти нормалей та розсіяного затінення.

```
#version 330 core

in vec3 fsFragPos;
in vec2 fsTexCoord;
in vec3 fsEyePos;
in mat3 TBN;
in vec4 fsFragPosInLightSpace;
out vec4 FragColor;

uniform sampler2D emissiveTex;
uniform sampler2D diffuseTex;
uniform sampler2D normalTex;
uniform sampler2D metallicTex;
uniform sampler2D roughnessTex;
uniform sampler2D aoTex;

uniform vec3 roughness;
uniform vec3 diffuse;
uniform float metallicFactor;

uniform vec3 lightDir;

const float PI = 3.14159265359;

// F
vec3 fresnelSchlick(float cosTheta, vec3 F0) {
```

```
    return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
}

// D
float distributionGGX(vec3 N, vec3 H, float roughness) {
    roughness = max(roughness, 0.0001);
    float a = roughness * roughness;
    float a2 = a * a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH * NdotH;

    float num = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;

    return num / denom;
}

// G_schlic
float geometrySchlickGGX(float NdotV, float roughness) {
    float r = (roughness + 1.0);
    float k = (r * r) / 8.0;

    float num = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return num / denom;
}

// G
float geometrySmith(vec3 N, vec3 V, vec3 L, float roughness) {
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx1 = geometrySchlickGGX(NdotV, roughness);
    float ggx2 = geometrySchlickGGX(NdotL, roughness);
    return ggx1 * ggx2;
}

vec3 CaclulatePBRFromLightSource()
{
    vec3 N = normalize(TBN * (texture(normalTex, fsTexCoord).xyz * 2 - 1));
    vec3 V = normalize(fsEyePos - fsFragPos);
    vec3 L = normalize(lightDir);
    vec3 H = normalize(V + L); // Half vector

    vec3 emissive = texture(emissiveTex, fsTexCoord).rgb;
    vec3 albedo = diffuse * texture(diffuseTex, fsTexCoord).rgb;
    float metallic = metallicFactor * texture(metallicTex, fsTexCoord).x;
    float roughness = roughness.x * texture(roughnessTex, fsTexCoord).x;
    float ao = texture(aoTex, fsTexCoord).x;

    vec3 F0 = vec3(0.04); // Default for non-metals
    F0 = mix(F0, albedo, metallic);

    float D = distributionGGX(N, H, roughness);
    float G = geometrySmith(N, V, L, roughness);
    vec3 F = fresnelSchlick(max(dot(H, V), 0.0), F0);

    vec3 specular = D * G * F;

    vec3 kD = vec3(1.0) - F;
    kD *= 1.0 - metallic;

    vec3 Lo = (kD * albedo / PI + specular) * max(dot(N, L), 0.0001);

    vec3 ambient = vec3(0.05) * albedo;
```

```
    vec3 color = ambient + Lo;
    return color;
}

void main() {
    vec3 color = CaclulatePBRFromLightSource();

    color = pow(color, vec3(1.0 / 2.2));

    FragColor = vec4(color, 1.0);
}
```

Приклад 3.18 – Шейдер для розрахунку освітлення за моделлю PBS

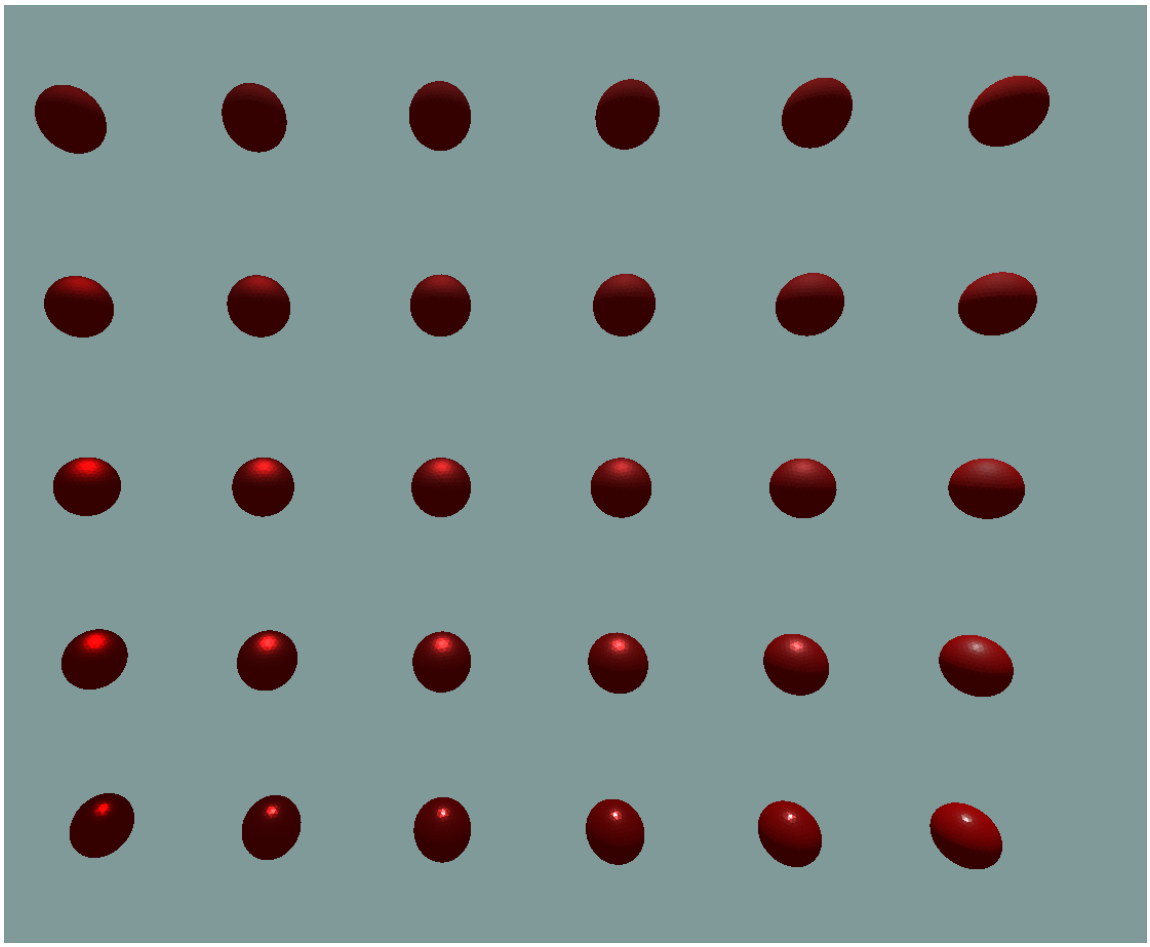

Рисунок 3.4 – Приклад тестової сцени освітлення якої розраховано за допомогою методу PBS

На рисунку 3.4 показані сфери з різними параметрами поверхні, які змінюються з фіксованим шагом від 0 до 1 зверху вниз за параметром шорсткості, а зправа на ліво за параметром металічності.

Розрахунок тіней відбувається за допомогою методу карт тіней. Для цього розрахунок освітлення треба робити у два проходи. При першому проході необхідно відмалювати всю сцену з точки зору джерела світла. Для цього використовується пустий шейдер, так як нам не потрібно розраховувати колір, а потрібна лише дистанція, яку GPUрозрахує на етапі разсеризації. Потім цю текстуру потрібно передати фрагментномушейдеру для розрахунку затінення на сцені. Для цього в шейдер потрібно додати нову текстуру як вхідний параметр, яка буди містити інформацію про відстань до фрагменту. Розрахунок тіні відбується наступною функцією.

```
uniform sampler2D shadowMap;
float calculateDirectedShadow(vec3 normal)
{
   vec3 projCoords = (fsFragPosInLightSpace.xyz / fsFragPosInLightSpace.w) * 0.5 + 0.5;
   float shadowMapDepth = texture(shadowMap, projCoords.xy).x;
   float bias = -0.1 * dot(normalize(lightDir, normal));
   shadowMapDepth -= bias;
   float shadow = projCoords.z > shadowMapDepth ? 1.0 : 0.0;

   return shadow;
}
```

Приклад 3.19 – Функція розрахунку тіней

Цю функцію можна примінити на обидва розглянуті підходи до розрахунку освітлення. Наприклад, взявши сцену з рисунку 3.3 і модифікувавши шейдердомноженнямdiffusei specularкомпонент на значення (1.0 –calculateDirectedShadow(normal)) можна отримати результат зображений на рисунку 3.5.

Рисунок 3.5 –Використання методу карт тіней з моделлю освітлення Бліна-Фонга

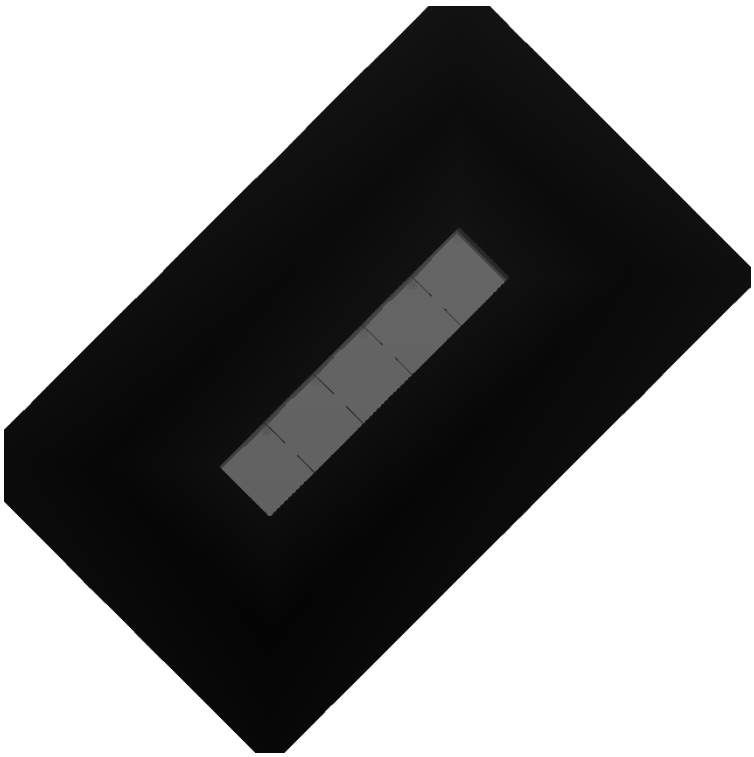

Рисунок 3.6 – Карта тіней згенерована під час розрахунку освітлення для рисунку 3.5 (чим темніший колір тим фрагмент ближчий до камери)

Метод карт тіней також можна використовувати і з flat-shadingдомножуючи колір на значення (1.0 –calculateDirectedShadow(normal)).

Рисунок 3.7 – Приклад використання карт тіней у поєднанні з моделлю flat-shading

Для розрахунку віддзеркалень ми використаємо підхід з картою оточення. Для цього потрібно додати карту оточення як вхідну текстуру для шейдеру і реалізувати функцію конвертації вектору напрямку у тестурні координати.

```
uniform sampler2D envMap;
const vec2 invAtan = vec2(0.1591, 0.3183);

vec2 getEnvMapCoords(vec3 n)
{
   vec2 uv = vec2(atan(n.z, n.x), asin(n.y));
   uv *= invAtan;
   uv += 0.5;
   uv.y *= -1;
   return uv;
}
```

Приклад 3.20 – Функція для конвертації вектора у текстурні координати

Далі для розрахунку відображень достатньо відобразити вектор направлений від камери до фрагменту і відобразивши його відповідно вектору нормалі, отримати дані з мапи оточення.

```
#version 330 core
```

```
in vec3 fsFragPos;
in vec2 fsTexCoord;
in vec3 fsEyePos;
in mat3 TBN;
in vec4 fsFragPosInLightSpace;
out vec4 FragColor;

uniform sampler2D envMap;
const vec2 invAtan = vec2(0.1591, 0.3183);

vec2 getEnvMapCoords(vec3 n)
{
   vec2 uv = vec2(atan(n.z, n.x), asin(n.y));
   uv *= invAtan;
   uv += 0.5;
   uv.y *= -1;
   return uv;
}

void main()
{
   vec3 V = normalize(fsFragPos - fsEyePos);
   vec3 color = texture(envMap, getEnvMapCoords(reflect(V, N)));

   color = pow(color, vec3(1.0 / 2.2));

   FragColor = vec4(color, 1.0);
}
```

Приклад 3.21 –Шейдер для розрахунку відображень від ідеально дзеркальних поверхонь

Рисунок 3.8 – Приклад розрахунку відображень від дзеркальної поверхні

Поєднання цихпідходіврозрахунку освітлення з підходомPBS дозволяє значно збільшити якість вихідної картинки, додавши до неї тіні і відображення.

Рисунок 3.9 – Приклад рендерінгу сцени з використанням PBSта карти тіней

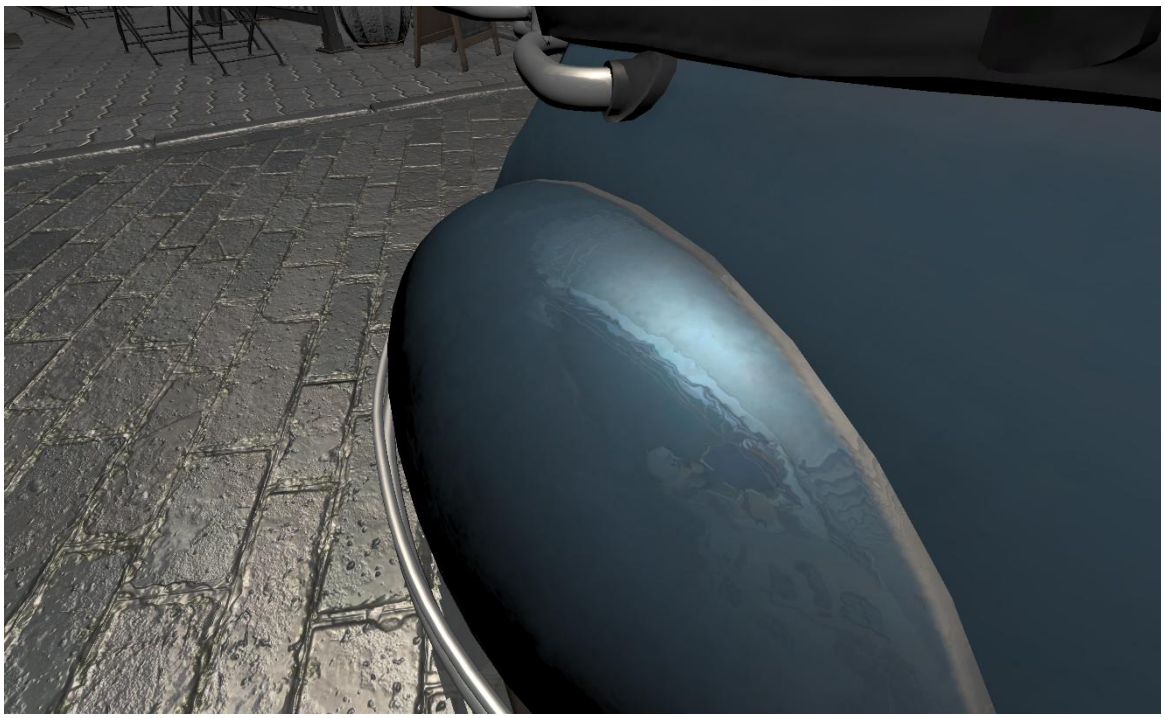

Рисунок 3.10 – Приклад відображення оточення від металічного об’єкту

ВИСНОВКИ

В результаті виконання пояснювальної записки, присвяченої дослідженню підходів до розрахунку 3Dграфіки. Проведено аналіз підходів до розрахунку 3Dграфіки, розглянуто сучасні моделі розрахунку освітлення які використовуються для рендерінгув реальному часі.

У записці також було проаналізовано підходи до організаціпроцесу рендерінгу. Реалізовано рушій 3Dрендерінгу в реальному часі на мові c++ і OpenGLAPI який дозволяє розробляти графічні додатки, завантажувати різні формати 3Dмоделей і прискорює процес реалізації алгоритмів розрахунку освітлення. За допомогою нього реалізовані основні алгоритми для розрахунку освітлення об'єктів (flat-shading, Blinn-Phong, PBS).Проаналізовано сфери застосування різних моделей освітлення, їх переваги та недоліки.Цей рушій дозволяє створювати на його основі великий спектр програмного забезпечення від простих візуалізацій для CADсистем до повноцінних комп'ютерних ігор.

Сфера розрахунків 3Dграфіки стрімко розвивається з початку поточного тисячоліття. І з кожним роком з'являються нові підходи до розрахунку різних ефектів освітлення що дозволяють значно покращувати якість вихідного зображення. Серед головних проривів в області розрахунку освітлення в реальному часі в останні роки без сумніву став підхід з трасуванням променів, який може дозволити у майбутньому розраховувати освітлення всієї сцени за допомогою однієї моделі освітлення з достатньою швидкістю для реального часу.З кожним роком сфера потребує більше спеціалістів для розробки алгоритмів та програмного забезпечення для роботи над постійно зростаючими стандартами у генерації зображень.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. OpenGLReference. URL: https://registry.khronos.org/OpenGL-Refpages/gl4/ (дата звернення: 18.01.2025).

2. R. Fabian. Data-Oriented Design. URL: https://www.dataorienteddesign.com/dodbook/(дата звернення: 18.01.2025).

3. Assimp reference. URL: https://assimp-docs.readthedocs.io/en/v5.1.0/about/index.html(дата звернення: 18.01.2025).

4. D. Eberly. 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics

5. S. Kosarevsky, V. Latypov. 3D Graphics Rendering Cookbook: A comprehensive guide to exploring rendering algorithms in modern OpenGL and Vulkan

6. M. Pharr, W. Jakob, G. Humphreys, Physically Based Rendering From Theory To Implementation 4th edition

7. Nvidia: Open Research Content Archive https://developer.nvidia.com/orca (дата звернення 18.01.2025)