ДВНЗ «ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра прикладної математики та інформатики

«До захисту допущено»
В.о. завідувача кафедри
Наталія МАСЛОВА
(підпис) (Ім'я та ПРІЗВИЩЕ)

«_____»_____2023 р.

Випускна кваліфікаційна робота

бакалавра
(освітній ступінь)

на тему "Додаток для створення і редагування світлових джерел як моделі освітлення у комп'ютерних іграх та кіноіндустрії" зі спецчастиною "Розробка і реалізація додатку, користувацького інтерфейсу, моделей освітлення на мові програмування C++, які базуються на бібліотеках OpenGL, GLFW, Glad, ImGui, GLM, Assimp, stb_image, розробка шейдерів на мові програмування GLSL"

Виконав (-ла): студент (-ка) 4 курсу, групи ІПЗ-19
(шифр групи)

спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

освітньої програми 121 Інженерія програмного забезпечення

Кощей Р. Р.
(прізвище та ініціали) (підпис)

Керівник стар. викладач каф. ПМІ Валерій КОСТІН
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Рецензент канд.техн.наук, доцент,

доцент каф. ЕТ Штепа Александр
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань
Студент _____
(підпис)

Луцьк – 2023р.

ДВНЗ «Донецький національний технічний університет»
Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра кафедра прикладної математики та інформатики
Освітній ступінь бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітня програма 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
В.о. завідувача кафедри ПМІ
_____Наталія МАСЛОВА
"____" _____2023 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТУ

Кощею Руслану Романовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Додаток для створення і редагування світлових джерел як моделі освітлення у комп'ютерних іграх та кіноіндустрії»
зі спецчастиною «Розробка і реалізація додатку, користувацького інтерфейсу, моделей освітлення на мові програмування C++, які базуються на бібліотеках OpenGL, GLFW, Glad, ImGui, GLM, Assimp, stb_image, розробка шейдерів на мові програмування GLSL»
керівник роботи Костін Валерій Іванович,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом від "05" травня 2023 року № 175
2. Строк подання студентом роботи "09" червня 2023 року
3. Вихідні дані до роботи результати науково-дослідної роботи, матеріали переддипломної практики
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
1) Аналіз предметної області
2) Розробка архітектури додатку та визначення залежностей та системних вимог
3) Детальна розробка архітектури додатку та опис моделей освітлення
4) Демонстрація результатів розробки
5) Тестування додатку
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)робота містить 59 рисунків, 4 формули, та інші матеріали, у кількості, достатній для демонстрації результатів кваліфікаційної роботи бакалавра.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Назарова І.А., доцент каф. ПМІ		10.06.2023

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області	24.04.2023-01.05.2023	
2	Вивчення роботи моделей освітлення	01.05.2023-15.05.2023	
3	Проєктування додатку	15.05.2023-22.05.2023	
4	Розробка додатку	22.05.2023-29.05.2023	
5	Тестування додатку та підготовка пояснювальної записки	29.05.2023-08.05.2023	
6	Нормоконтроль	09.06.2023	
7	Захист	22.06.2023	

Студент _____ Кощей Р. Р.
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кощей Р. Р. Додаток для створення і редагування світлових джерел як моделі освітлення у комп’ютерних іграх та кіноіндустрії / Випускна кваліфікаційна робота на здобуття освітньо-кваліфікаційного рівня «бакалавра» за спеціальністю 121 Інженерія програмного забезпечення. ДВНЗ ДонНТУ, Луцьк, 2023.

Метою цієї роботи є дослідження різних моделей освітлення в комп'ютерній графіці та їх реалізація на базі бібліотеки OpenGL. У роботі розглядаються одні з основних моделей освітлення, які використовуються для створення реалістичних або інших видів зображень.

Під час роботи було вивчено різні моделі освітлення, зокрема моделі Фонга, Блінна-Фонга, Ламберта, Гуро та інші. Було досліджено особливості кожної моделі та їхню застосовність у різних ситуаціях.

Для реалізації моделей освітлення було використано бібліотеку OpenGL та мови програмування C++, GLSL. Було розроблено програмні модулі для реалізації кожної моделі освітлення. У результаті було створено систему візуалізації, що дає змогу створювати реалістичні зображення із застосуванням різних моделей освітлення.

Під час дослідження та реалізації було виявлено переваги та недоліки кожної моделі освітлення.

Загалом, дана робота дає змогу отримати глибше розуміння принципів роботи моделей освітлення в комп'ютерній графіці та способів їх реалізації на базі бібліотеки OpenGL. Результати роботи можуть бути використані для створення більш реалістичних і якісних зображень у різних галузях.

Ключові слова: моделі освітлення, Фонг, Блінн-Фонг, Ламберт, Гуро, C++, GLSL

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

OpenGL – Open Graphics Library

GLFW – Graphics Library Framework

SDL - Simple DirectMedia Layer

Glad – Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator

GLEW - OpenGL Extension Wrangler Library

ImGui – Immediate Mode GUI

GLM – OpenGL Mathematics

Assimp – Open Asset Import Library

stb_image – Small Tool Box Image

GLSL – OpenGL Shading Language

MVS – Microsoft Visual Studio

API - Application Programming Interface

CMYK – Cyan, Magenta, Yellow, Key

RGB – Red, Green, Blue

Lab – Lightness, a, b

HSV – Hue, Saturation, Value

2D, 3D, 4D – Two, Three, Four-Dimensional

libxml2 – Extensible Markup Language Library Two

OBJ – Wavefront OBJ (Object)

FBX – Film Box

Collada – Collaborative Design Activity

GPU - Graphics Processing Unit

IntelliJ IDEA – Integrated Development Environment for Java

Xcode - eXcode Integrated Development Environment

MacOS - Macintosh Operating System

UML - Unified Modeling Language

UI - User Interface

DI - Dependency Injection

ECS - Entity Component System

STL - Standard Template Library

LIFO - Last-In-First-Out

ін - інші

ВСТУП

У сучасному світі комп'ютерні ігри та кіноіндустрія стали невід'ємною частиною нашого життя, і багато хто з нас проводить багато часу, насолоджуючись віртуальними світами та їхньою барвистою графікою. Одним із важливих компонентів, що відповідають за реалістичність або атмосферність зображення, є світлові ефекти і моделі освітлення. Саме завдяки їм ми можемо отримати реалістичні та часом дивовижні враження і заглибитися в атмосферу гри або фільму.

У рамках цього дипломного проєкту було розроблено додаток для створення і редагування світлових джерел як моделей освітлення в комп'ютерних іграх і кіноіндустрії.

У даній роботі буде розглянуто процес розробки додатка, а також деякі моделі освітлення. Зокрема буде розглянуто різні бібліотеки та технології, що використовуються для створення додатків у галузі комп'ютерної графіки. Наприклад при створенні додатку були використані бібліотека OpenGL і мова програмування GLSL для рендерингу графіки та написання шейдерів, а також бібліотека GLFW для керування вікном додатка. В якості мови програмування було застосовано мову C++ яка є де факто стандартом, який використовують під час роботи з API OpenGL, Vulkan, DirectX.

Загалом, ця робота має практичне застосування в галузі комп'ютерної графіки і може бути розширена новим функціоналом та застосована для створення кіносцен та комп’ютерних ігор.

1. ВИБІР І ЗАТВЕРДЖЕННЯ ТЕМИ ТА КЕРІВНИКА ВИПУСКНОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1.1. Важливість алгоритмів освітлення у сучасному світі

Алгоритми освітлення є найважливішим компонентом комп'ютерної графіки, оскільки дають змогу створювати реалістичні або іншого виду візуалізації, які можна використовувати в різних галузях, як-от ігрова графіка, візуалізація архітектурних проєктів, наукова візуалізація і кіноіндустрія.

Освітлення визначає, як світло поширюється і відбивається на поверхнях об'єктів у сцені, і як це впливає на колір, тіні та інші властивості об'єктів.

Деякі відомі алгоритми освітлення наприклад модель освітлення Фонга, модель освітлення Ламберта, модель освітлення Блінна-Фонга і модель освітлення Кука-Торранса. Ці алгоритми можуть використовуватися в різних додатках, таких як 3D-моделювання, комп'ютерна анімація, віртуальна реальність і багато інших.

Загалом, алгоритми освітлення є ключовим елементом для створення високоякісних, реалістичних, мультиплікаційних та ін. зображень у комп'ютерній графіці. Вони відіграють важливу роль у створенні ефектів освітлення, які роблять сцени більш привабливими, переконливими та цікавими для глядача.

1.2. Додатки які застосовують алгоритми освітлення

Blender - це безкоштовне і відкрите програмне забезпечення для тривимірного моделювання, анімації, рендерингу, композитингу, створення інтерактивних застосунків та інших завдань, пов'язаних із 3D-графікою.

На рисунку 1.1 зображено сцену в Блендер, яку буде використано для демострації різних джерел світла. На даному зображені немає джерел світла.

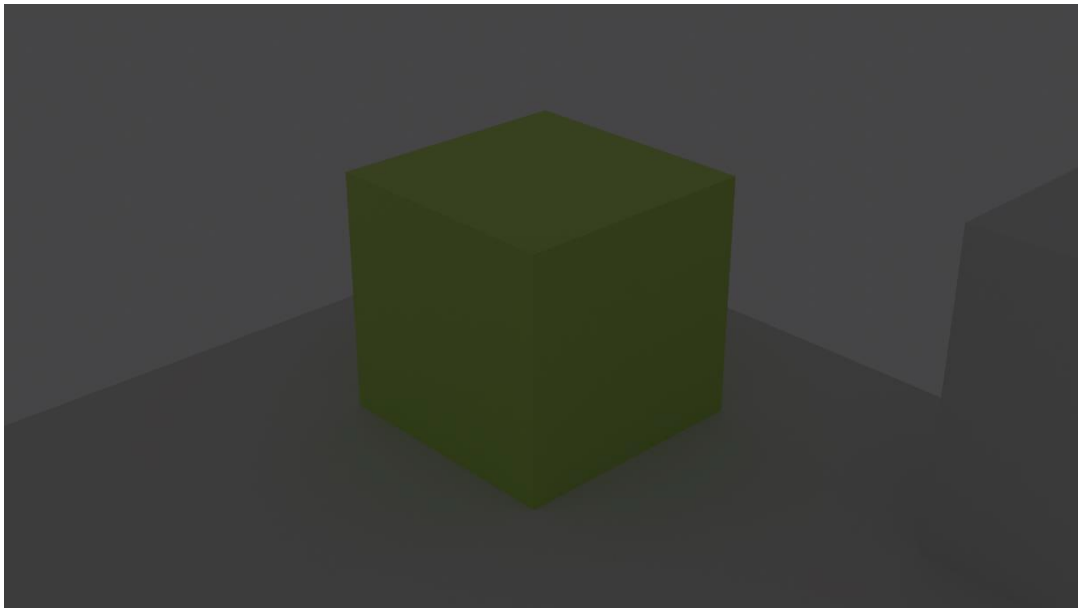

Рисунок 1.1 – Сцена без джерел світла

У Блендері є такі джерела освітлення.

Точкове освітлення (Point): джерело, в якому світло виходить від однієї точки, розташованої в просторі. Точкове освітлення може використовуватися для створення ефекту лампи, свічки або багаття.

На рисунку 1.2 зображено приклад точкового освітлення у Blender.

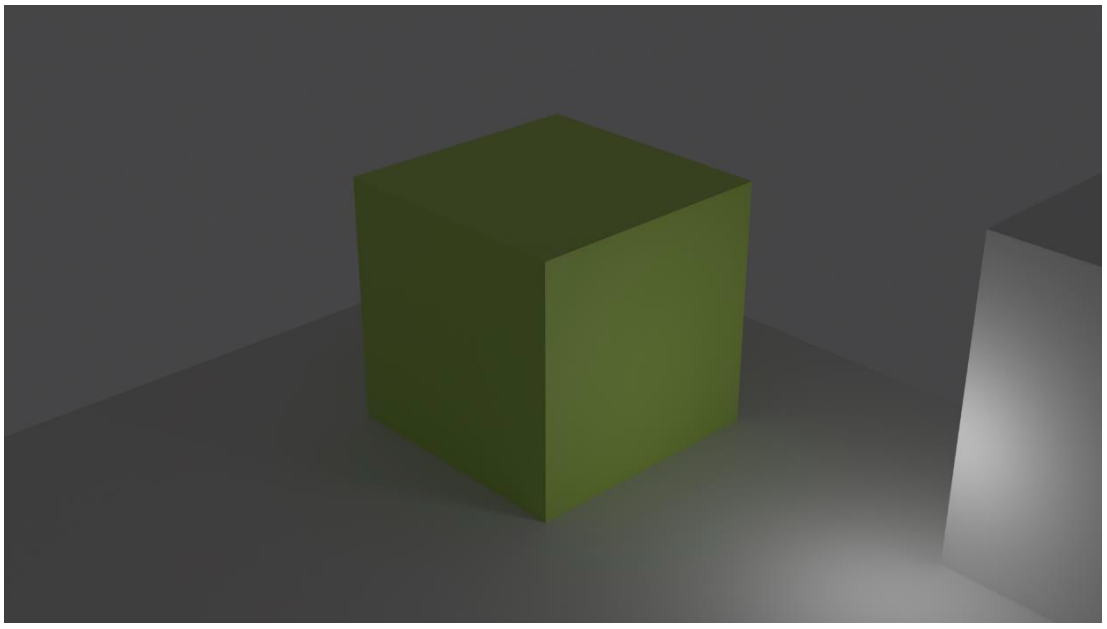

Рисунок 1.2 – Точкове освітлення

Спрямоване освітлення (Sun): джерело, яке моделює природне освітлення від Сонця. Спрямоване освітлення може використовуватися для створення денних сцен.

На рисунку 1.3 зображено приклад спрямовоного освітлення у Blender. Джерело світла знаходиться з боку.

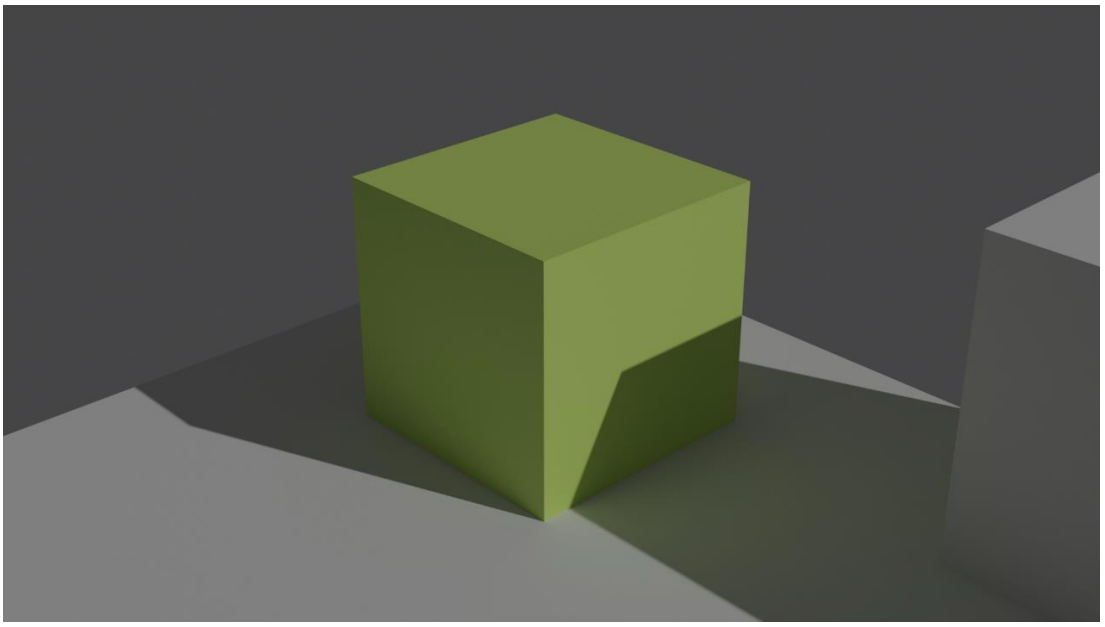

Рисунок 1.3 – Спрямоване освітлення

Освітлення області (Area): джерело, яке дає змогу створювати рівномірне і м'яке освітлення в сцені. Він імітує світло, що виходить з панелі, розташованої в певному місці в сцені, і може бути використаний для створення різних джерел світла, таких як вікна, двері, лампи та інші поверхні, що виступають в ролі джерел світла.

На рисунку 1.4 зображено приклад освітлення області у Blender. Джерело світла знаходиться зверху. Цей рисунок відрізняється від рисунку 1.1. Слід також зазначити, що кожне джерело освітлення має параметри, які можна налаштовувати за бажанням.

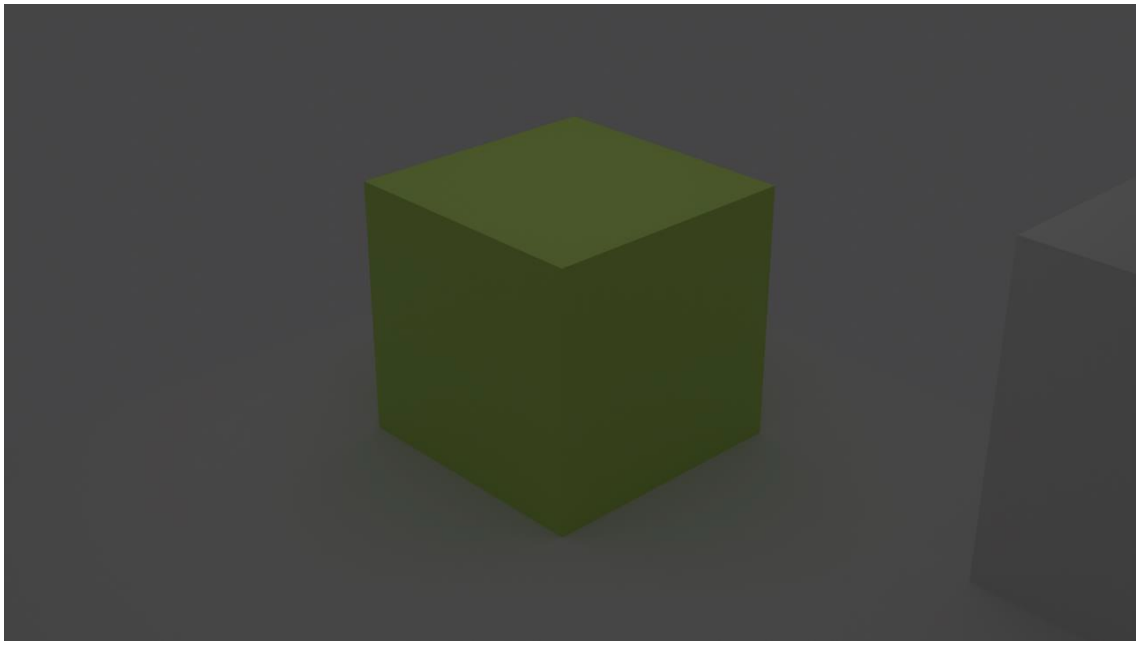

Рисунок 1.4 – Освітлення області

Нижче наведено ще декілька відомих додатків які мають у своєму функціоналі роботу з освітленням:

Autodesk Maya - професійне програмне забезпечення для створення 3D-анімації, яке містить у собі різноманітні інструменти для моделювання, анімації, рендерингу та створення ефектів освітлення.

Autodesk 3ds Max - ще одне професійне програмне забезпечення для створення 3D-графіки та анімації, що містить у собі широкий набір інструментів для моделювання, анімації, рендерингу та створення ефектів освітлення.

Cinema 4D - програмне забезпечення для тривимірного моделювання, анімації та візуалізації, що містить інструменти для створення ефектів освітлення та текстурування об'єктів.

Unity - платформа для створення інтерактивних додатків та ігор, яка містить інструменти для створення ефектів освітлення та створення реалістичних 3D-сцен.

Unreal Engine - ще одна платформа для створення ігор та інтерактивних додатків, яка містить у собі широкий набір інструментів для створення ефектів освітлення та створення реалістичних 3D-сцен.

Є ще багато додатків, які використовують алгоритми освітлення для моделювання певної середи. Нові досягнення, як і раніше, з'являються в галузі комп'ютерної графіки зокрема завдяки людям, які розробляють ці додатки. І ця галуззь має великі перспективи.

При роботі з освітленням та вцілому з компьютерною графікою треба мати розуміння деяких концепції та вміти користуватися деяким математичним апаратом.

1.3. Концепція представлення кольору в комп’ютері

Колір представлений у комп'ютері за допомогою моделей кольору, які визначають, як кольори можна представити й обробляти в цифровому форматі.

Модель кольору - це система, що використовується для опису кольорів і представлення їх у числовій формі. Ця система дає змогу визначити, яким чином кольори можуть бути представлені у формі чисел.

Існує безліч моделей кольору, які використовують у різних галузях, таких як комп'ютерна графіка, фотографія, друк і наука про колір. Кожна модель кольору має свої особливості та специфікації, які визначають її застосування в конкретній галузі.

Однією з найпоширеніших моделей кольору є RGB (червоний, зелений, синій), яка використовується в комп'ютерній графіці для створення і відображення кольорових зображень на екрані. Іншою моделлю кольору є CMYK (ціан, магента, жовтий, чорний), яка широко застосовується в друкарській промисловості для точного відтворення кольорів на папері.

Крім того, є також моделі кольору, які використовуються в науці про колір, як-от модель Lab (яскравість, зелений-червоний спектр, синій-жовтий спектр), що використовується для визначення колірного простору та відстаней між кольорами. Ще однією моделлю кольору є модель HSV (відтінок, насиченість, яскравість), яка використовується для опису кольору в термінах його відтінку,

яскравості та насиченості.

Існує ще багато кольорових моделей але для цілей цієї роботи було використано RGB-модель. У комп'ютерній графіці RGB-модель широко використовується для створення і редагування зображень, відео, анімації, ігор та інших проєктів, пов'язаних з відображенням кольорів на екрані. Це пов'язано з тим, що екрани цих пристроїв складаються з точок, що світяться, які можуть відображати тільки певний набір колірних значень, вимірюваних у RGB-значеннях.

Загалом, модель кольору є важливим інструментом для опису, представлення та відтворення кольорів у різних галузях і дисциплінах.

1.4. Числове представлення RGB – моделі [11]

Кожен канал (червоний, зелений або синій) RGB-моделі представлений у 8-бітному форматі, що означає, що кожен канал може приймати 256 різних значень (від 0 до 255). Таким чином, кожен колір можна представити у вигляді комбінації трьох чисел (R, G, B) від 0 до 255.

Наприклад, колір білого матиме максимальне значення для кожного з трьох каналів: R=255, G=255, B=255. Чорний колір матиме мінімальне значення для кожного каналу: R=0, G=0, B=0.

На рисунку 1.5 наведено графічне представлення кольорового простору RGB. В центрі зображено кольори, які отримуються комбінуючи червоний, зелений та синій. Тобто в моделі RGB при 8-и бітній глубині кольору можно представити 16 777 216 (256 у ступені 3) кольорів. Однак, слід зазначити, що людське око не може розрізняти всі 16 мільйонів кольорів, які можуть бути представлені в RGB-моделі. Найточніші оцінки говорять, що людське око може розрізняти від 2 до 7 мільйонів кольорів залежно від умов.

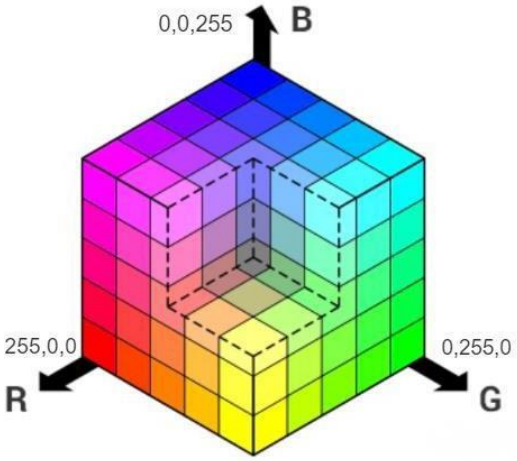

Рисунок 1.5 – Графічне представлення кольорового простору RGB

Для реалізації моделей освітлення в комп'ютерній графіці використовуються різні графічні бібліотеки, такі як OpenGL, Vulkan, DirectX і Metal. Вони надають програмісту набір функцій і класів, які дають змогу працювати з графічними об'єктами та обробляти графічні дані, включно з даними про освітлення.

1.5. Вибір графічного API для рендарингу [9]

Вибір графічного API є важливим рішенням під час розроблення графічних додатків, таких як комп'ютерні ігри або додатки віртуальної реальності чи моделювання. Існує безліч графічних API, кожен з яких має свої переваги та недоліки залежно від вимог проєкту та характеристик цільової платформи. Деякі з найбільш популярних API включають OpenGL, Vulkan, DirectX і Metal. Під час вибору графічного API необхідно враховувати різні чинники, як-от цільову платформу, вимоги до продуктивності та функціональності, доступність інструментів розробки та спільноти розробників.

Деякі відомі графічні API.

OpenGL - це кросплатформений API, який використовується для створення високопродуктивної 3D-графіки на комп'ютерах і мобільних пристроях. OpenGL стандартом і підтримується широким спектром операційних систем,

включно з Windows, MacOS, Linux, iOS і Android.

API OpenGL складається з набору функцій, які можна використовувати для створення та маніпулювання 3D-графічними об'єктами, такими як тривимірні моделі, текстури, матеріали та освітлення. Він також надає можливість створення інтерактивних 3D-додатків, у яких користувачі можуть взаємодіяти з об'єктами на екрані.

OpenGL використовує графічний процесор (GPU) для прискорення обробки графіки, що дає змогу створювати складні 3D-сцени з високою якістю та швидкістю відтворення. Він підтримує широкий спектр ефектів, включно з тінями, віддзеркаленнями, прозорістю та анімацією.

Vulkan - це низькорівневий API для створення високопродуктивної 3D-графіки. Його розробила компанія Khronos Group і випустила у 2016 році. Vulkan підтримує багатопоточність і дає змогу програмістам отримати максимальну продуктивність із сучасних графічних процесорів (GPU) і багатоядерних процесорів.

API Vulkan надає програмістам повний контроль над процесом створення і маніпулювання графічними об'єктами, такими як тривимірні моделі, текстури, матеріали та освітлення. Він також надає широкий спектр функцій для створення ефектів, включно з тінями, віддзеркаленням, прозорістю та анімацією.

DirectX - це набір технологій, створених компанією Microsoft для розроблення ігор та інших мультимедійних додатків на платформі Windows. DirectX надає високопродуктивний API для доступу до апаратних можливостей комп'ютера, таких як графічний процесор (GPU) і звукова карта, що дає змогу створювати швидкі та ефективні мультимедійні додатки.

DirectX містить у собі різні компоненти, такі як Direct3D для створення 3D- графіки, Direct2D для створення 2D-графіки, DirectSound для роботи зі звуком та інші. Кожен із цих компонентів надає свій власний API для доступу до відповідної апаратної функціональності. Він також надає широкий спектр функцій для, віддзеркаленням, прозорістю та анімацією.

Metal - це API для створення високопродуктивної графіки на платформі Apple. Він був розроблений компанією Apple і призначений для використання на Mac, iPhone, iPad та інших пристроях Apple.

Основними перевагами Metal є висока продуктивність та ефективність, що досягається за рахунок нижчого рівня абстракції та зменшення накладних витрат на обробку графічних даних. Це дає змогу Metal досягати вищої продуктивності, ніж інші API, такі як OpenGL або DirectX. Він як і попередні API надає широкий спектр функцій для створення ефектів, включно з тінями, віддзеркаленням, прозорістю та анімацією.

Вибір конкретного API для додатку залежить від платформ для яких розроблюється додаток та апаратних можливостей комп'ютерів. Не всі GPU підтримують Vulkan, так як це новий API, також, наприклад, Windows не підтримує Metal, а Mac не підтримує DirectX. Через це було зроблено рішення що додаток буде розроблено з API OpenGL, це пов'язано з часом який відведено на розробку, апаратними можливостями комп'ютерів (планується що додаток буде використовувати OpenGL версії 3.3, що дає змогу використовувати його на більшої кількості комп'ютерів) та платформами, які підтримують цей API (OpenGL підтримується Windows, Linux, Mac, саме для цих платформ буде розроблятися додаток).

1.6. Колір в OpenGL [6]

В OpenGL колір зазвичай подається у вигляді чотирикомпонентного вектора RGBA, де кожен компонент (R - червоний, G - зелений, B - синій, A – альфа або прозорість) є значенням від 0 до 1, яке визначає інтенсивність кольору. Діапазон від 0 до 1 використовується в OpenGL та інших графічних бібліотеках для представлення кольору у вигляді нормалізованих значень, що забезпечує більш просту і стандартизовану роботу з кольором у різних додатках і системах.

У цьому випадку, "нормалізований" означає, що кожна колірна компонента представлена у вигляді значення, нормалізованого до діапазону від 0 до 1. Це означає, що найтемніше значення кольору (наприклад, чорний) представлено як (0.0, 0.0, 0.0, 0.0), а найсвітліше значення (наприклад, білий) як (1.0, 1.0, 1.0, 1.0).

Для переведення значення кольору з діапазону 0-255 (восьмибітовий колір) у діапазон 0-1 (нормалізований колір) необхідно розділити значення кольору на 255.0.

1.7. Rendering pipeline [7]

OpenGL rendering pipeline (графічний конвеєр OpenGL) - це послідовність етапів і операцій, які виконує OpenGL для створення зображення на екрані. Тобто OpenGL отримує вхідні данні (позиції та кольори вершин тощо), ці дані обробляються в графічному конвеєрі поетапно, після обробки OpenGL повертає зображення в якості вихідних даних.

На рисунку 1.6 зображено спрощене графічне представлення графічного конвеєра.

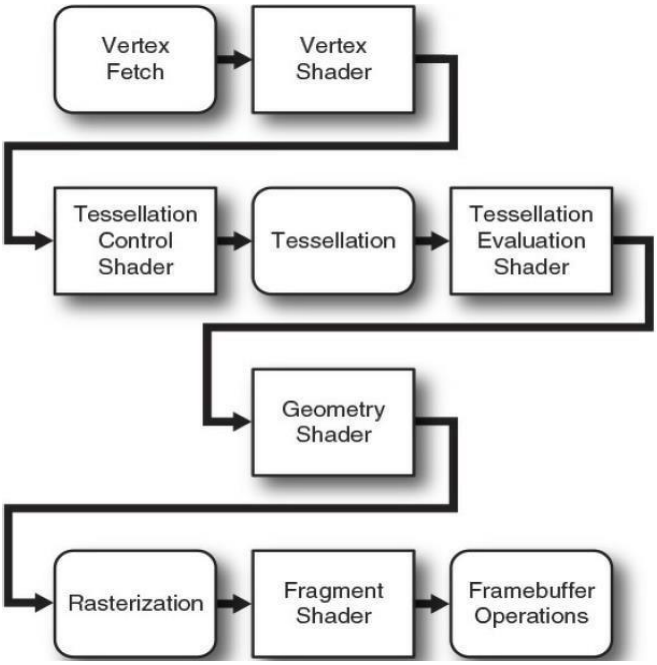

Рисунок 1.6 – Спрощене графічне представлення графічного конвеєра

Графічний конвеєр.

Vertex Fetch (витяг вершин) - це перший етап графічного конвеєра OpenGL, який відповідає за витяг вершин із буфера пам'яті на відеокарті. На цьому етапі OpenGL отримує дані про вершини (їхні координати, колір, текстурні координати та інші атрибути) і завантажує їх у пам'ять графічної карти. Потім, після вилучення вершин, OpenGL передає їх на наступний етап конвеєра - Vertex Shader.

Vertex Shader (вершинний шейдер) - це другий етап графічного конвеєра OpenGL. Він приймає на вхід набір вершин, отриманих на попередньому етапі, і виконує певні операції над ними. У вершинному шейдері відбувається трансформація вершин із локальної системи координат об'єкта в глобальну систему координат з урахуванням положення, орієнтації та масштабування об'єкта в сцені. У вершинному шейдері також можна реалізовувати такі ефекти, як анімація та деформація об'єктів. Результат роботи вершинного шейдера передається на наступний етап конвеєра - Tessellation Shader.

Теселяція (англ. tessellation) - це процес розбиття більших геометричних фігур на більш дрібні елементи, такі як трикутники або чотирикутники. Теселяція може застосовуватися в комп'ютерній графіці для створення більш детальних і реалістичних поверхонь, особливо в 3D-моделюванні та візуалізації. Теселяція може виконуватися апаратно на графічному процесорі (GPU) за допомогою відповідних шейдерів, таких як Tessellation Control Shader і Tessellation Evaluation Shader в OpenGL.

Tessellation Evaluation Shader і Tessellation Control Shader - це шейдери, які використовуються в процесі теселяції (розщеплення) поверхні в OpenGL.

Tessellation Control Shader відповідає за налаштування й керування параметрами теселяції, як-от рівень деталізації, тип теселяції (трикутна або чотирикутна), а також за передавання даних у Tessellation Evaluation Shader.

Tessellation Evaluation Shader виконує фактичну теселяцію, тобто розбиває

задану поверхню на безліч дрібніших трикутників або чотирикутників відповідно до параметрів, встановлених у Tessellation Control Shader. Він також може виконувати додаткові операції, як-от розрахунок нормалей або текстурних координат для кожного нового трикутника.

Geometry Shader - це шейдер, який дає змогу створювати нові геометричні об'єкти, як-от точки, лінії та трикутники, на основі вхідних даних, а також видаляти або змінювати наявні об'єкти. Основна відмінність між Tessellation і Geometry Shader полягає в тому, що Tessellation змінює кількість наявної геометрії, тоді як Geometry Shader створює нову геометрію на основі наявної.

Rasterization - це процес перетворення геометричних об'єктів, представлених у вигляді вершин, ребер і граней, у зображення, що відображається на екрані. Після того, як графічний процесор обробив вершини, що пройшли через вершинний, тесселяційний і геометричний шейдери, наступним етапом у графічному конвеєрі OpenGL є растеризація.

У процесі растеризації графічний процесор ділить полігони на безліч маленьких фрагментів, які будуть відображатися на екрані. Кожен фрагмент отримує інформацію про колір, координати текстури та інші властивості з відповідних вершин полігона, які були оброблені на попередніх етапах графічного конвеєра. Потім ці властивості застосовуються до фрагмента за допомогою фрагментного шейдера і формується підсумкове зображення.

Fragment Shader - це шейдер, що виконується на кожному фрагменті (також відомому як піксель) візуалізованого зображення. Кожен фрагмент проходить через фрагментний шейдер, який визначає остаточний колір та інші властивості фрагмента.

Framebuffer Operations у пайплайні OpenGL - це останній етап відтворення графіки перед виведенням на екран. На цьому етапі виконуються операції над пікселями, які були отримані в результаті проходження через усі попередні етапи пайплайна.

Framebuffer (буфер кадру) - це область пам'яті, яка містить зображення для

виведення на екран. Операції, що виконуються на цьому етапі, дають змогу змінювати вміст буфера кадру, що впливає на кінцевий результат відтворення зображення на екрані.

Наприклад, можна використовувати такі операції, як накладення текстур на поверхню об'єкта, змішування кольорів і багато іншого. Також на цьому етапі можуть бути застосовані операції post-processing для зміни остаточного зображення перед його виведенням на екран.

Для цілей проєкту застосовуються три програмованих шейдери, а саме вершиний, геометричний та фрагментний шейдери.

1.8. Математичний апарат [14]

Математичний апарат комп'ютерної графіки охоплює різні галузі математики, такі як лінійна алгебра, геометрія, топологія, чисельні методи тощо.

Лінійна алгебра використовується для роботи з векторами і матрицями, які широко застосовуються в комп'ютерній графіці для представлення геометричних об'єктів і трансформацій над ними. Вектори можуть представляти позиції, напрямки, швидкості, кольори та інші властивості об'єктів, а матриці можуть використовуватися для перетворень, таких як повороти, масштабування і перенесення.

Геометрія відіграє важливу роль у комп'ютерній графіці, оскільки вона визначає форму і розміри об'єктів. Вона охоплює такі поняття, як точки, лінії, криві, поверхні та багатогранники. У комп'ютерній графіці часто використовуються геометричні примітиви, такі як куби, сфери, циліндри та конуси.

Топологія використовується для визначення зв'язків між геометричними об'єктами та їхніми частинами. Наприклад, у комп'ютерній графіці топологія може використовуватися для визначення, які вершини належать грані багатогранника або які грані є сусідніми.

Чисельні методи використовуються для розв'язання різних завдань у

комп'ютерній графіці, таких як рендеринг, моделювання фізичних процесів і обробка зображень. Це може включати в себе методи розв'язання рівнянь, інтерполяції, апроксимації та інші.

Для цілей проєкту на прикладному рівні застосовується переважно лінійна алгебра.

Деякі з найбільш популярних операцій лінійної алгебри, що використовуються в комп'ютерній графіці, включають в себе:

– множення матриць: це одна з основних операцій лінійної алгебри, яка широко використовується в комп'ютерній графіці. множення матриць дозволяє комбінувати перетворення, такі як масштабування, поворот і трансляція, щоб отримати кінцеве перетворення об'єкта;

– векторний добуток: це операція, яка дає змогу знаходити вектор, перпендикулярний двом іншим векторам. ця операція часто використовується в комп'ютерній графіці для знаходження нормалі до поверхні;

– нормалізація вектора: нормалізація вектора дозволяє привести його до одиничної довжини, що є необхідним для багатьох операцій, таких як знаходження нормалі до поверхні.

Також може бути застосована комбінація цих операцій, загалом існує ще багато операцій які можна використовувати як по одинці так і в комбінації. Для математичних розрахунків згідно проєкту було прийнято рішення використовувати бібліотеку GLM. Що стосується розрахунків на рівні шейдерів то мова GLSL надає необхідний функціонал для маніпулювання векторами та матрицями.

2. СИСТЕМНІ ВИМОГИ ТА АРХІТЕКТУРА ДОДАТКУ

2.0 Цільові платформи

Метою проєкту є створити додаток, який можуть застосовувати люди для цілей моделювання освітлення або вивчення роботи моделей освітлення. Різні люди використовують різні операційні системи, деякі люди використовують декілька операційних систем. Через цю поширеність раціональним буде зробити додаток який підтримується на всіх цих операційних системах. Найпоширенішими операційними системами персональних комп'ютерів є: Windows, MacOS та Linux. Є ще кілька переваг, які наведено нижче.

Розробка крос-платформного додатку з підтримкою на macOS, Linux і Windows має такі переваги:

Економія часу та ресурсів: Розробка окремих додатків для кожної операційної системи потребує додаткових зусиль і ресурсів. Крос-платформна розробка може значно спростити процес розробки, оскільки використовується загальний код для всіх платформ, адаптований до специфічних вимог кожної операційної системи.

Зниження витрат на підтримку: Підтримка декількох версій додатків для різних операційних систем може бути складним і витратним завданням. Розробка крос-платформного застосунку дає змогу скоротити витрати на підтримку, оскільки використовується загальний код, а оновлення застосовуватимуться до однієї основної версії застосунку.

Зручність для користувачів: Користувачі цінують зручність використання додатків на своїх бажаних операційних системах. Крос-платформний додаток забезпечує користувачів можливістю вибору ОС без необхідності змінювати програмне забезпечення або звичний інтерфейс.

Але є і мінуси крос-платфомної розробки.

Деякі операційні системи можуть мати унікальні особливості, які не можуть бути повністю підтримані в крос-платформному контексті. Також

можуть виникнути проблеми з продуктивністю або доступом до деяких функцій операційної системи.

Але для цілей проєкту серед яких є розробка учбового додатку переваги крос-платформного додатку мають більшу перевагу ніж розробка додатку для однієї конкретної операційної системи.

Серед особливостей платформ є їхня розрядність.

Розрядність операційної системи відноситься до кількості біт, які використовуються для представлення адрес пам'яті та розміру регістрів у центральному процесорі. Вона визначає, скільки пам'яті та ресурсів може використовувати операційна система та додатки, що працюють на цій ОС.

Для цілей проєкту було вирішено розробляти додаток з підтримкою на x64 бітних операційних системах, які були представлені вище.

x64 (або 64-розрядна) операційна система підтримує обробку даних і адресацію пам'яті на 64 біти. Це означає, що в такій системі адресація пам'яті може використовувати 64-бітові числа, що дає змогу обробляти набагато більший обсяг пам'яті, ніж у 32-розрядній операційній системі.

Ось кілька причин, чому варто вибирати x64 бітну операційну систему.

Підтримка більшого обсягу пам'яті: 64-розрядна операційна система може адресувати набагато більший обсяг оперативної пам'яті порівняно з 32- розрядною системою. Це особливо важливо під час роботи з великими наборами даних, багатопотокових додатків або під час запуску кількох додатків одночасно. Покращена продуктивність. Деякі додатки та завдання можуть отримати перевагу від роботи в 64-розрядному середовищі. Наприклад, додатки, що використовують великі обсяги даних або виконують складні обчислення, можуть працювати більш ефективно на x64 системі завдяки можливості оброблення більшої кількості даних і використанню розширених інструкцій центрального процесору.

Безпека та сумісність. Деякі сучасні технології та механізми безпеки вимагають використання 64-розрядної операційної системи. Наприклад, багато

антивірусних програм і систем шифрування рекомендують використовувати 64-розрядні операційні системи для забезпечення оптимального захисту і сумісності з сучасними стандартами безпеки.

Все більше додатків розробляються для 64-розрядних операційних систем, так як 64-розрядні процесори стають все більш поширеними. А використання додатку відповідного розрядності процесору є більш раціональним рішенням.

2.1 Вимоги до апаратного забезпечення

Центральний процесор має мати розрядність x64 біти.

Графічна відео-карта має підтримувати OpenGL версії 3.3 та вище.

2.2 Сторонні залежності

Однією з цілей проєкту є створення кросплатформенного додатку але створення такого типу додатку є дуже трудомістким завданням. Тому для полегшення праці та збереження часу було вирішено застосовувати вже готові кросплатформені бібліотеки які в професійному середовищі називають залежностями.

Сторонні залежності - це зовнішні бібліотеки або компоненти, які використовуються в проєкті, але не є частиною його коду і не розробляються безпосередньо командою програмістів.

Використання сторонніх залежностей має свої переваги [20] та недоліки. З одного боку, вони дають змогу прискорити процес розроблення і поліпшити функціональність проєкту, оскільки можуть надавати готові рішення для різних завдань. З іншого боку, використання сторонніх залежностей може призвести до проблем сумісності, нестабільності роботи та вразливостей у безпеці.

Сторонні залежності проєкту.

GLFW - це кросплатформна бібліотека на мові C для створення вікон і обробки подій у додатках OpenGL. Вона надає програмістам простий інтерфейс для створення вікон і контекстів OpenGL, а також опрацювання введення з

клавіатури, миші та джойстиків.

Аналогом GLFW є одна бібліотека яка має назву SDL.

SDL - це кросплатформна бібліотека на мові C для створення вікон і обробки введення в додатках, включно з підтримкою OpenGL і OpenGL ES.

GLFW і SDL обидві є популярними кросплатформеними бібліотеками для створення вікон і обробки введення в додатках, включно з підтримкою OpenGL, але вони мають деякі відмінності.

GLFW є більш простою і легковажною бібліотекою, ніж SDL, і надає тільки базовий набір інструментів для роботи з вікнами та обробки введення. Вона може бути більш зручною для використання в невеликих і простих додатках, таких як ігри, які не вимагають багатьох розширених можливостей.

Для цілей проєкту вистачає можливостей бібліотеки GLFW. Тому щоб не перевантажувати проєкт непотрібними можливостями і файлами, було прийнято рішення використовувати саме цю бібліотеку.

Наступною залежністю є бібліотека Glad.

Glad - це бібліотека для завантаження функцій OpenGL у додатках на основі C/C++. Glad генерує завантажувальний код для обраних версій OpenGL і розширень, що дає змогу забезпечити переносимість між різними операційними системами та платформами.

Аналогом Glad є бібліотека яка має назву GLEW.

GLEW - бібліотека для завантаження функцій OpenGL у додатках на основі C/C++. Вона також надає утиліти для перевірки наявності конкретних розширень і функцій OpenGL.

Glad і GLEW є двома популярними кросплатформними бібліотеками для завантаження функцій OpenGL у додатках на основі C/C++, але у них є деякі відмінності.

Glad більш простий і легковагий, ніж GLEW. Glad має менше залежностей і простіший у використанні, оскільки він не вимагає наявності бібліотеки libxml2, яка необхідна для GLEW. Крім того, Glad надає більш надійний спосіб

завантаження функцій OpenGL з використанням макросів, в той час як GLEW використовує покажчики на функції, які можуть бути менш надійними і більш складними у використанні. Ці переваги сприяють прийняттю рішення на користь Glad.

Наступною залежністю є бібліотека ImGui.

ImGui - це бібліотека для створення користувацького інтерфейсу у додатках на основі C++, яка є швидкою, легковажною і простою у використанні. Вона розроблена для використання в додатках, пов'язаних з іграми та графікою, і дає змогу швидко та легко створювати користувацькі елементи інтерфейсу, як-от кнопки, текстові поля, панелі, списки, що випадають, тощо.

Не було знайдено гідного аналога для вибору між. Тому саме цю бібліотеку було обрано для створення користувацького інтерфейсу.

Наступною залежністю є бібліотека GLM.

GLM - це бібліотека математичних функцій для додатків, що використовують OpenGL і створені мовою C++. Вона надає інструменти для виконання математичних операцій, необхідних для роботи з 3D-графікою, такими як вектори, матриці, кватерніони, трансформації та інші.

GLM є кросплатформеною бібліотекою і підтримує різні операційні системи, такі як Windows, MacOS і Linux. Вона має простий і зрозумілий інтерфейс, який дає змогу розробникам швидко і легко виконувати математичні операції, необхідні для створення 3D-додатків.

Аналогом GLM є бібліотека яка має назву Eigen.

Eigen - це бібліотека лінійної алгебри на мові C++, яка надає інструменти для виконання математичних операцій з матрицями, векторами та кватерніонами. Вона також може використовуватися для створення додатків, що використовують OpenGL.

Однак GLM має простіший і зрозуміліший інтерфейс, що робить його зручнішим для новачків у галузі графіки та OpenGL. GLM здебільшого націлений та векторами, що використовуються в 3D-

графіці, тоді як Eigen надає ширший набір інструментів для роботи з різними математичними об'єктами, такими як матриці, вектори, кватерніони, тензори тощо. Як і у випадку з GLFW щоб не перевантажувати проєкт непотрібними файлами або функціоналом було вирішено обрати бібліотеку GLM.

Наступною залежністю є бібліотека Assimp.

Assimp – це кросплатформна бібліотека з відкритим вихідним кодом, яка призначена для імпорту 3D-моделей і сцен у різних форматах, як-от OBJ, FBX, Collada і багато інших, у додатки, що використовують OpenGL та інші графічні рушії.

Assimp може завантажувати моделі у вигляді мешів, кісткових анімацій, світла і камер. Вона надає доступ до метаданих моделі, таких як імена матеріалів і текстур, розміри моделі та інші важливі характеристики.

Не було знайдено гідного аналога для вибору між. Тому саме цю бібліотеку було обрано для завантаження 3D-моделей.

stb_image - це невелика бібліотека з відкритим вихідним кодом, яка призначена для завантаження зображень у форматах. Бібліотека написана мовою C і не вимагає сторонніх залежностей, що робить її дуже легковажною і зручною у використанні.

Основною метою stb_image є швидке і просте читання зображень у додатках, що використовують OpenGL та інші графічні движки. Бібліотека має дуже простий API і може завантажувати зображення у вигляді 8-бітних або 16-бітних компонент кольору.

Не було знайдено гідного аналога для вибору між. Тому саме цю бібліотеку було обрано для завантаження зображень або текстур.

2.3 Тип додатку

Існує декілька типів додатків: [2] додатки які використовують ститичну лінковку та додатки які використовують динамічну лінковку.

Додатки, що використовують статичні бібліотеки.

У таких додатках усі необхідні бібліотеки включаються у виконуваний файл додатка в момент компіляції. Коли додаток запускається, він самодостатній і не потребує наявності окремих бібліотек на цільовій системі. Переваги статичних бібліотек включають:

Простота встановлення. Додаток зі статичними бібліотеками може бути легко встановлено на інших комп'ютерах без необхідності встановлення додаткових залежностей.

Підвищена надійність. Такі додатки можуть бути більш надійними, оскільки вони не залежать від зовнішніх чинників, таких як наявність або версія динамічних бібліотек.

Однак, використання статичних бібліотек також має деякі недоліки:

Збільшений розмір виконуваного файлу: Оскільки всі бібліотеки включені до виконуваного файлу, його розмір може бути значно більшим, ніж у застосунків, що використовують динамічні бібліотеки.

Відсутність оновлень. Якщо оновлення бібліотеки потрібне для виправлення помилок або додавання нових функцій, застосунок потрібно перекомпілювати і повторно поширити.

Додатки, що використовують динамічні бібліотеки.

У таких додатках основний код компілюється у виконуваний файл, а залежні бібліотеки завантажуються під час виконання програми. Коли додаток запускається, він посилається на зовнішні динамічні бібліотеки, які можуть бути спільними для кількох додатків. Переваги динамічних бібліотек включають:

Економія пам'яті. Додатки, що використовують динамічні бібліотеки, можуть спільно використовувати одну копію бібліотеки, що може заощадити оперативну пам'ять на цільовій системі.

Легкість оновлення. Якщо оновлення бібліотеки потрібне, достатньо оновити її на системі, і воно автоматично буде доступне для всіх додатків, які її використовують.

Однак, використання динамічних бібліотек також має деякі недоліки.

Залежність від наявності бібліотек. Додаток вимагає наявності відповідних динамічних бібліотек на цільовій системі. Якщо бібліотека відсутня або має несумісну версію, додаток може не працювати коректно.

Додаткові кроки розгортання: Додаток має бути поставлений разом із необхідними динамічними бібліотеками або має бути встановлена їхня наявність на цільовій системі.

Слід також зазначити, що розробка динамічної бібліотки може вимагати додаткових людино-ресурсів, так як може бути більш складним завданням ніж статичний додаток. Так як команда розробників невелика а додаток є спіцифічним (спільного використання бібліотеки між додатками не буде) то було прийнято рішення розробляти статичний додаток.

2.4 Середовище розробки

Існує безліч інтегрованих середовищ розробки для різних мов програмування включаючи C++. Нижче наведено кілька популярних середовищ та їхні особливості:

Eclipse: Eclipse є потужною і розширюваною середою, яка підтримує безліч мов програмування, включно з C++. Вона пропонує широкий набір інструментів, включно з автодоповненням коду, налагоджувачем, системою керування версіями та багатьма іншими функціями. Eclipse також має активну спільноту розробників і безліч доступних плагінів.

IntelliJ IDEA: IntelliJ IDEA розроблена для мови Java, але також має хорошу підтримку C++. Вона пропонує просунуті функції, такі як інтелектуальне автодоповнення, статичний аналіз коду, відладчик і підтримку системи контролю версій. IntelliJ IDEA відома своєю високою продуктивністю і зручністю використання.

Xcode: Xcode є основним середовищем для розроблення додатків під платформу MacOS і iOS. Вона пропонує великі інструменти для розробки мовою Objective-C і Swift, але також підтримує C++. Xcode включає в себе відладчик,

візуальні інструменти інтерфейсу користувача і широкий набір бібліотек і фреймворків для розробки додатків для Apple-платформ.

Visual Studio: Visual Studio розроблена компанією Microsoft та є одним з найбільш популярних і повнофункціональних середовищ для розробки програмного забезпечення під різні платформи, на різних мовах програмування включаючи C++. Вона пропонує потужний набір інструментів для розробки, налагодження, профілювання та тестування додатків. Visual Studio має широку підтримку стандартів C++ та інтегровані інструменти розробки користувацьких інтерфейсів і роботи з базами даних. Крім того, Visual Studio має величезну спільноту розробників і велику документацію.

Крім цих особливостей слід додати, що розробка проводиться на операційній системі Windows, яку розробила компанія Microsoft. Використання саме Visual Studio буде найкращим рішенням, так як ця середа, з самого початку, була націлена на операційну систему Windows. Але Visual Studio забезпечує підтримку розробки додатків для різних платформ, включно з Windows, Linux і мобільними платформами. Це дає змогу розробникам створювати кросплатформні додатки на C++ з використанням одного середовища розробки.

2.5 Архітектурне проєктування. UML діаграмми класів

Для демонстрації архітектури за основу було прийнято рішення використовувати UML діаграму класів. Але через великий розмір діаграми було вирішено розподілити її на невеликі блоки кожен з яких має певний сенс та демонструє зв’язки між класами.

Загалом весь додаток обиртається навкого головного класу який має назву Application. Клас Application створює, розподіляє компоненти між іншими компонентами, та видаляє їх по завершенню виконання додатка.

На рисунку 2.1 зображено UML представлення класу Application. Більш детально члени класу будуть описані у розділі 3.

Рисунок 2.1 – Головний клас додатку

Для реалізації зваємозв’язків між класами планується використати паттерн DI. Більш детально патерн буде описано у розділі 3.

Нижче наведено зв’язки класу Application з іншими класами.

На рисунку 2.2 зображено зв’язки залежності класу Application. Зв'язок "залежність" у діаграмі класів означає, що один клас залежить від іншого класу для виконання своїх завдань.

Залежність може проявлятися в різних формах.

Використання методів або операцій. Клас може залежати від іншого класу, викликаючи його методи або операції для виконання певних завдань. Наприклад, клас Order може залежати від класу Customer, щоб отримати інформацію про клієнта.

Використання аргументів або параметрів. Клас може залежати від іншого класу, передаючи йому аргументи або параметри для виконання певних дій або отримання даних. Наприклад, клас PaymentProcessor (обробник платежів) може залежати від класу PaymentGateway (платіжний шлюз), щоб виконати платіж із використанням певного платіжного шлюзу.

Використання властивостей або змінних. Клас може залежати від іншого класу, отримуючи доступ до його властивостей або змінних для отримання

необхідних даних або стану. Наприклад, клас ShoppingCart може залежати від класу Product, щоб отримати інформацію про продукт, таку як його назва, ціна та опис.

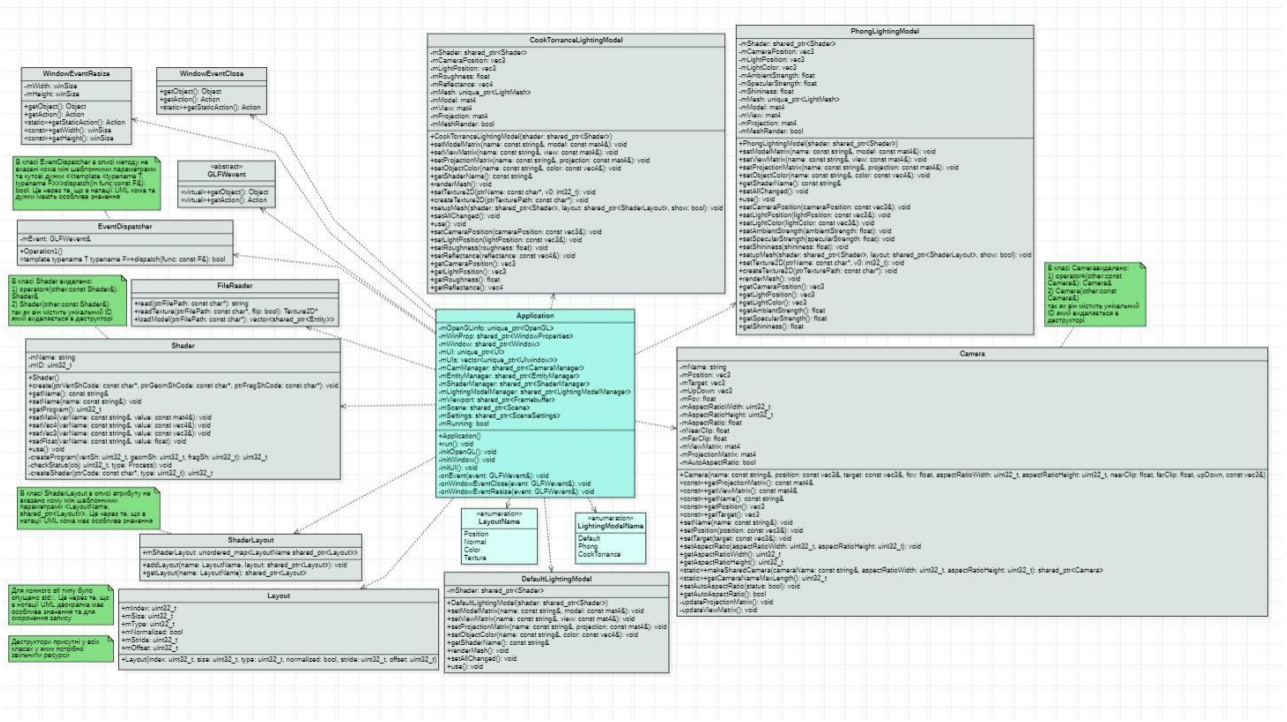

Рисунок 2.2 – Залежності класу Application

Як можна побачити клас Application залежить від класів: Shader, FileReader та ін. Також він залежить від перелічених типів: LayoutName, LightingModelName. Зелені блоки демонструють додаткові коментарі які пояснюють специфіку реалізації UML, так як синтаксис UML не надає деяких можливостей мов.

На рисунку 2.3 зображено зв’язки композиції класу Application. Зв'язок "композиція" в об'єктно-орієнтованому програмуванні позначає відношення між класами, де один клас містить екземпляри інших класів як своїх членів. Це відношення передбачає, що існування одного класу залежить від існування іншого класу і що клас-власник керує життєвим циклом об'єктів класу-члена.

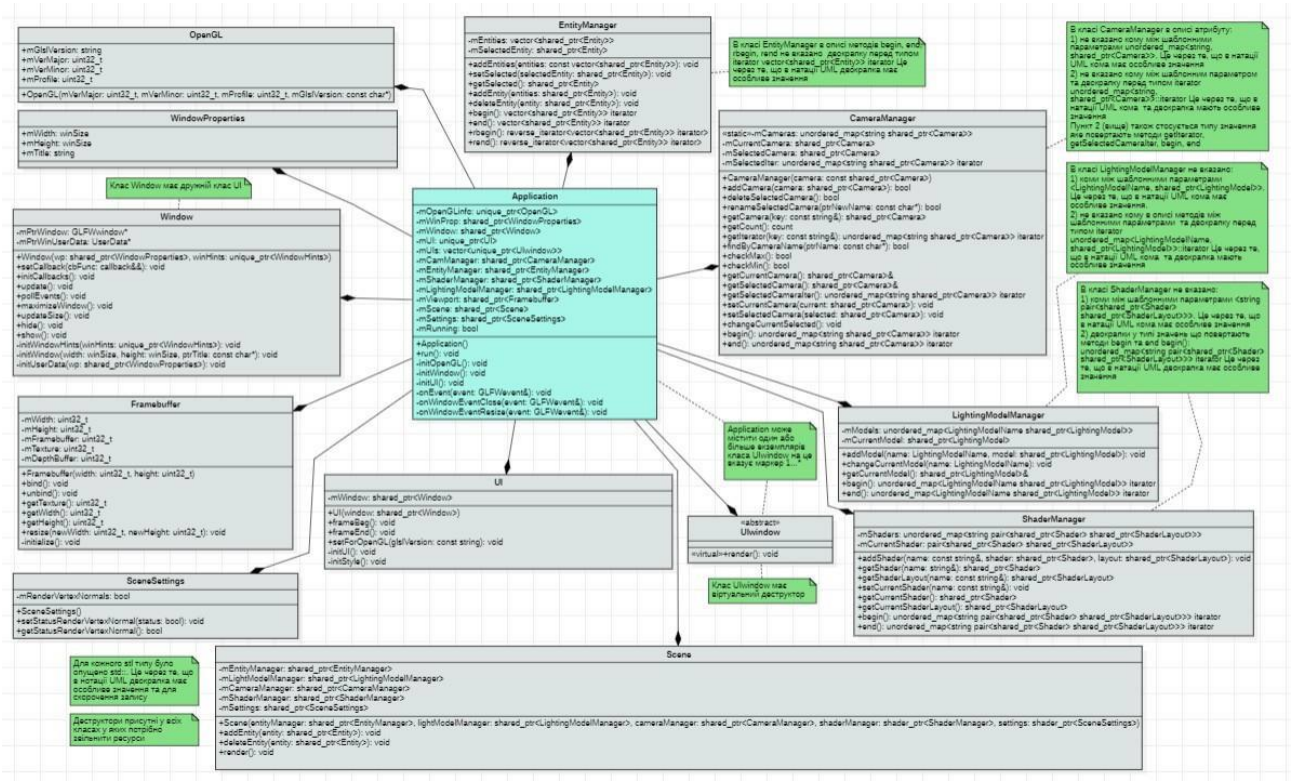

Рисунок 2.3 – Композиції класу Application

Як можна побачити клас Application керує життєвим циклом класів: UI, Scene та ін. Клас Application також керує розподілення ресурсів, тобто він ініціалізує ці класи та передає вказівник на них у інші класи додатку які потребують функціонал відповідного класу.

Клас Application має декілька екземплярів класу UIwindow та зберігає їх компактно у масиві який має тип vector, але UI елементів повинно бути декілька і кожен з них має певну функціональність. Не можливо створити декілька класів з однаковим іменем та різною функціональністю. Але об’єктно-орієнтоване парадигма надає функціонал “Успадкування” завдяки якому загальний клас батько надає можливість дочірнім класам перевизначати свої методи та може зберігати вказівник на дочірній клас. Якщо батько зберігає вказівник на дочірній клас та дочірній клас перевизначив батьківський метод то при виклику батьківського метода буде викликатися саме перевизначений метод. Це дає змогу зробити єдиний метод який буде відображати різні UI блоки, які мають свої компоненти та структуру.

На рисунку 2.4 зображено абстрактний клас [1] UIwindow та успадкування його функціоналу іншими класами UI.

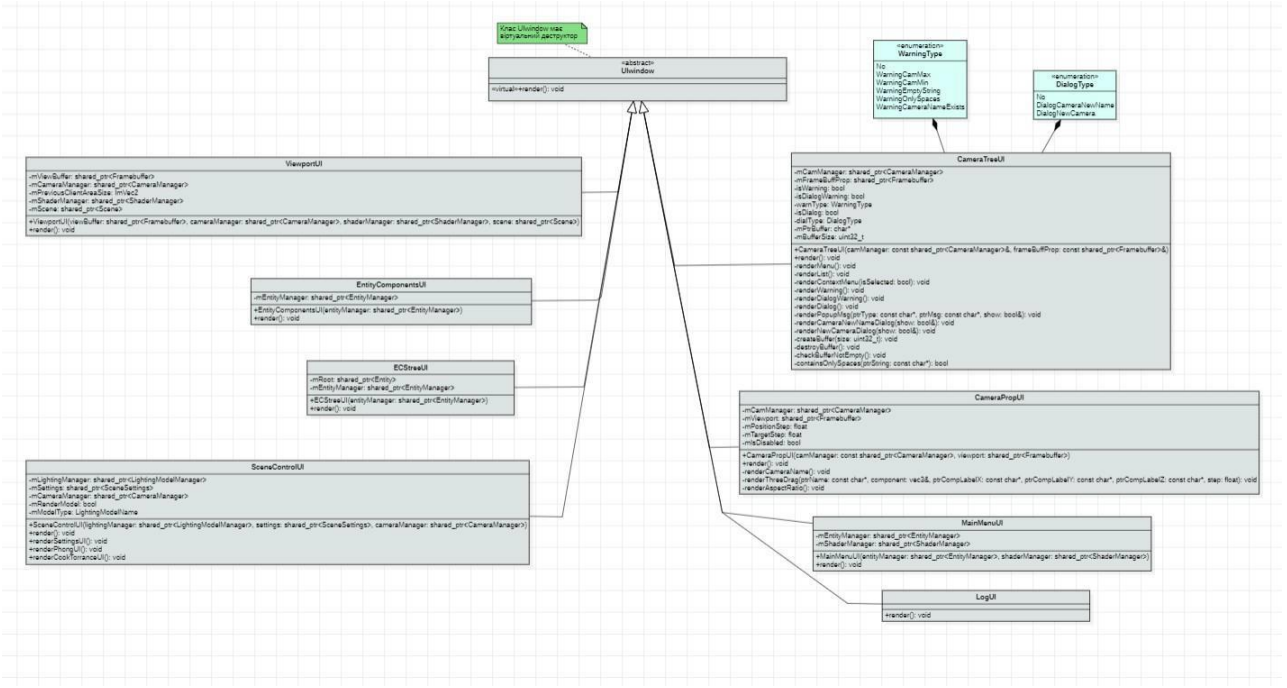

Рисунок 2.4 – Абстракція UI елементів

Клас UIwindow описує загальний інтерфейс який перевизначається у дочірніх класах. Загальним інтерфейсом на диний момент є метод render. Так як для створення користувацького інтерфейсу використовується бібліотека ImGui, метод render має розташування елементів окна інтерфейса, тип елементів та їх порядок можуть відрізнятися для кожного дочірнього класу. При виклику цього методу робиться рендаринг відповідного окна та його елементів. Також саме в цьому методі робиться обробка подій, які відбулися при взаємодії з елементами, це пов’язано з специфікою реалізації бібілотеки ImGui.

Зв’язок класу UIwindow з Application зображено на рисунку 2.3. Зв’язок інших UI класів з класом Application зображено на рисунку 2.5.

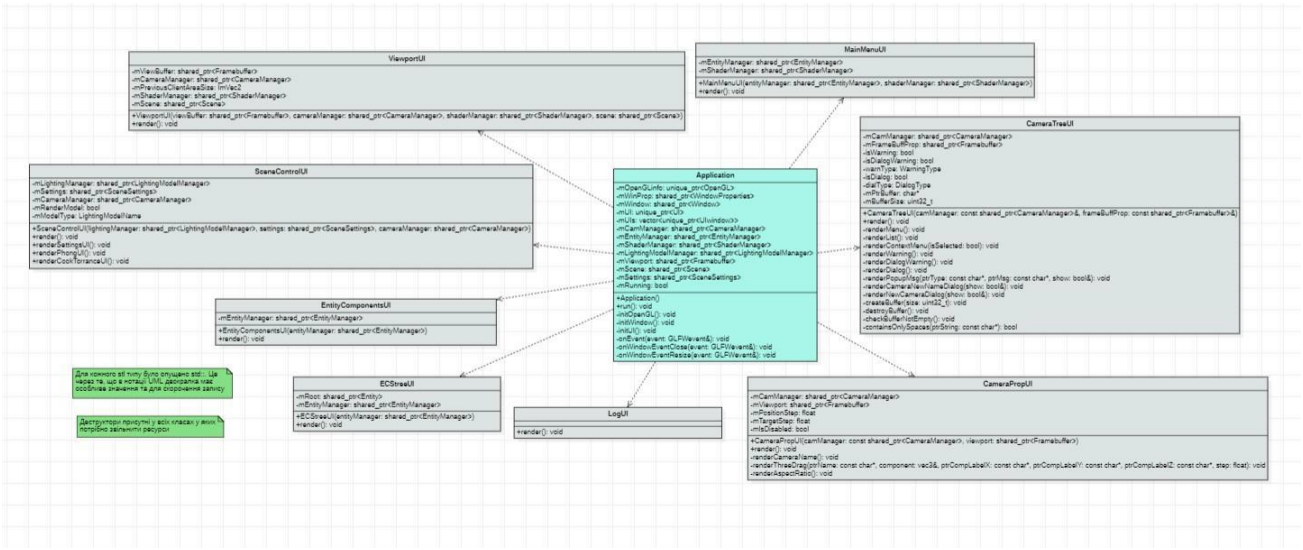

Рисунок 2.5 – Залежності класу Application

В основу додатку покладено 8 UI вікон кожне з яких має свою мету та набір елементів.

Аналогічно абстракції UIwindow була розроблена абстракція GLFWevent. Ця абстракція застосовується для отримання класом Application повідомлень різних типів від класу Window, наприклад: натискання клавіши клавіатури, зміна розміру вікна і ін. Клас Application отримує ці подій як подію GLFWevent потім завдяки статичному методу перевіряє реальний тип, та відправляє у відповідний метод обробник там він перетворює GLFWevent у відповідний тип та обробляє подію з можливістю застосування всіх специфічних методів цієї події.

На рисунку 2.6 зображено UML реалізацію цього механізму подій.

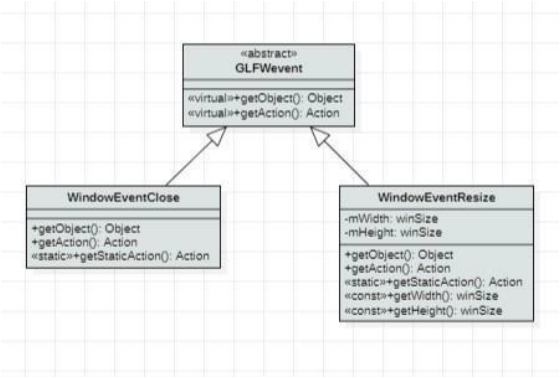

Рисунок 2.6 – Абстракція подій

Також абстрація була розроблена для моделей освітлення та компонент. На рисунку 2.7 зображено абстракцію моделей освітлення.

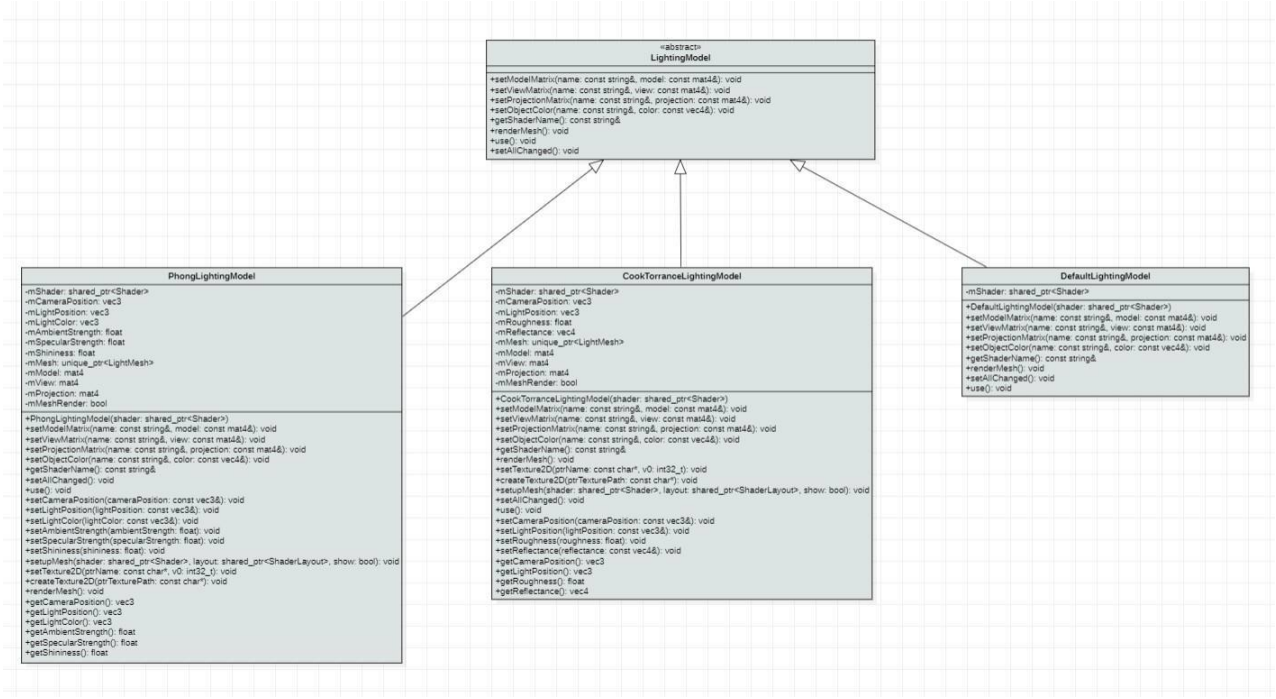

Рисунок 2.7 – Абстракція моделей освітлення

На рисунку 2.8 зображено абстракцію компонент ECS.

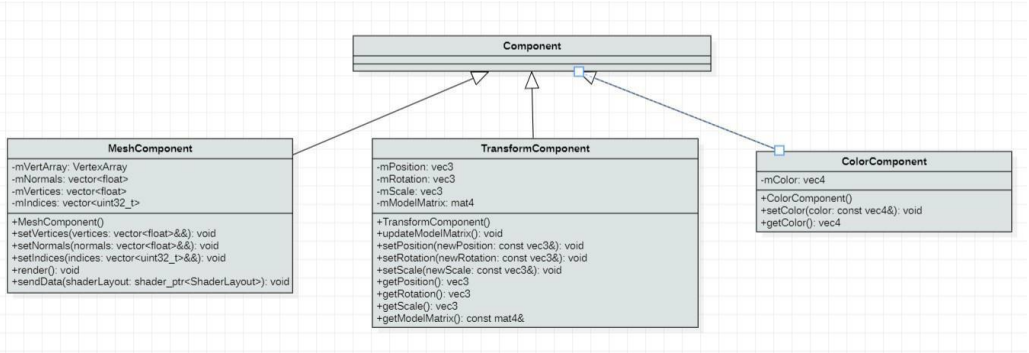

Рисунок 2.8 – Абстракція компонент ECS

Найважливішою перевагою всіх цих абстракцій є їхня здатність до маштабування. З часом можуть бути додані нові UI вікна, подій або моделі освітлення. Їх додавання не потребує перепроєктування архітектури додатку.

3. ДЕТАЛЬНЕ ПРОЄКТУВАННЯ ТА РОЗРОБКА ДОДАТКУ

3.1 Детальне проєктування. UML діаграми класів

Нижче наведено різні системи які мають сенс, вони наведені у вигляді класів та зв’язків між ними.

На рисунку 3.1 зображено віконну систему подій.

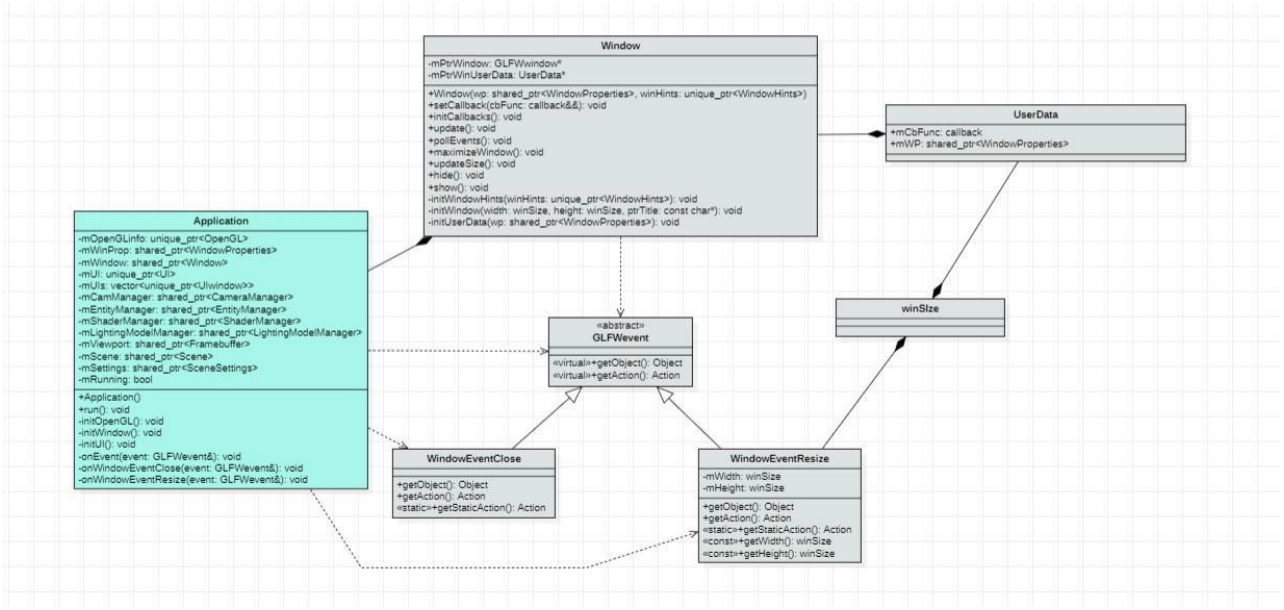

Рисунок 3.1 – Віконна система подій

Клас Application містить у якості члена клас Window, який є платформо-незалежним класом вікна, він використовує бібліотеку GLFW, яка підтримує створення вікон на платформах: Windows, Linux, MacOS. Клас Application повинен бути сповіщен про появу події у системі GLFW, щоб коректно її обробити і сповістити інші класи, які потребують оновлення відповідної інформації (розмір вікна, код натиснутої клавіши на клавіатурі та ін.). Для реалізації цієї системи застосовуються механізми: абстракція подій та зворотній виклик.

Абстракція подій та її переваги описуються в розділі 2. Зворотній виклик це функція, вказівник передається в якості аргументу іншому об’єкту

(через його метод(и)), який викликає її та передає аргументи (якщо це передбачено) у потрібний момент, наприклад якщо сталася подія.

Через специфіку бібліотеки GLFW було створено тип UserData який зберігається біля структури вікна GLFW. В цій структурі зберігається вказівник на функцію зворотного виклику. Доступ до цієї структури мають внутрішні функції (класу Window) обробки подій. Ці функції частково обробляють подію та сповіщають про подію клас Application та передають йому керування через виклик зворотньої функції.

На рисунку 3.2 зображено систему моделей освітлення.

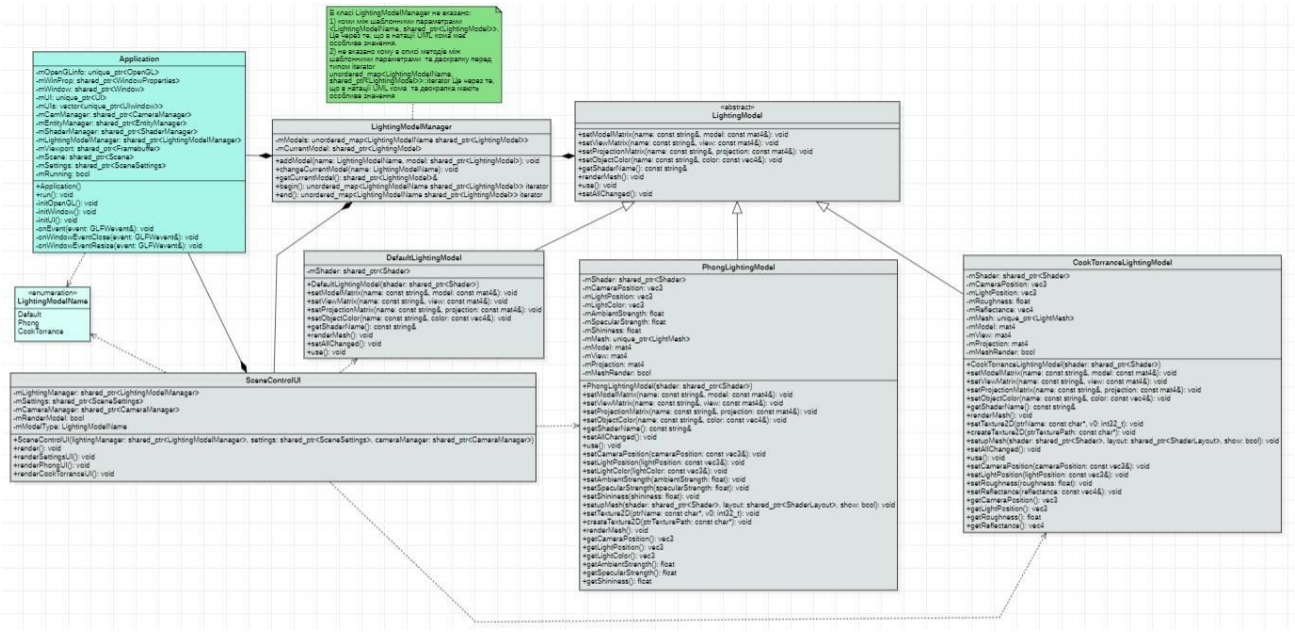

Рисунок 3.2 – Система моделей освітлення

Життєвим циклом кожної моделі освітлення керує клас LightingModelManager, а життєвим циклом LightingModelManager керує клас Application. Абстракція LightingModel та її переваги описуються в розділі 2. Суть цієї системи полягає в тому, що SceneControlUI отримує LightingModelManager та тим самим отримує контроль над усіма моделями освітлення. Користувач взаємодіє з SceneControlUI та змінює поле mCurrentModel в класі LightingModelManager. Це поле використовує інший клас який відповідає за

рендаринг. Він рендарить об’єкти використовуючи обрану модель освітлення.

На рисунку 3.3 зображено систему камер.

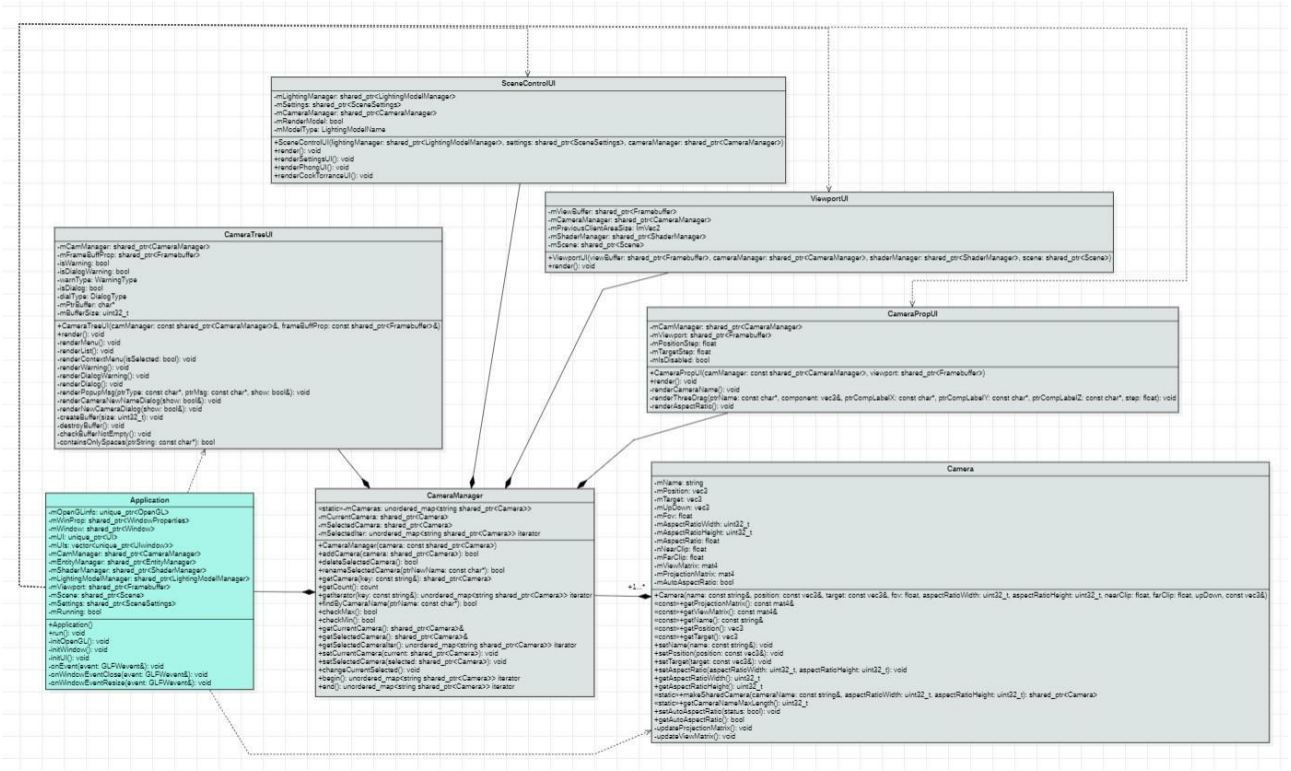

Рисунок 3.3 – Система камер

Мета цієї системи створити гнучкий доступ до всіх камер додатку включно з поточною камерою, яка застосовується для рендарингу. Життєвим циклом кожної камери керує клас CameraManager, а життєвим циклом CameraManager керує клас Application. Всі UI вікна які потребують доступ до поточної камери або інших камер отримують вказівник на тип CameraManager. Потреби можуть носити різний характер, наприклад: SceneControlUI використовує клас CameraManager для передачі інформації о поточній камері поточнії моделі освітлення для коректного рендарингу, а тип (вікно) CameraPropUI відображає характеристики вибраної камери (не обов’язково поточної), ці характеристики можна змінювати. Типи в яких зберігаються класи Camera або LightingModel та ін. подібні описані нижче у розділі 3.

На рисунку 3.4 зображено систему сутностей.

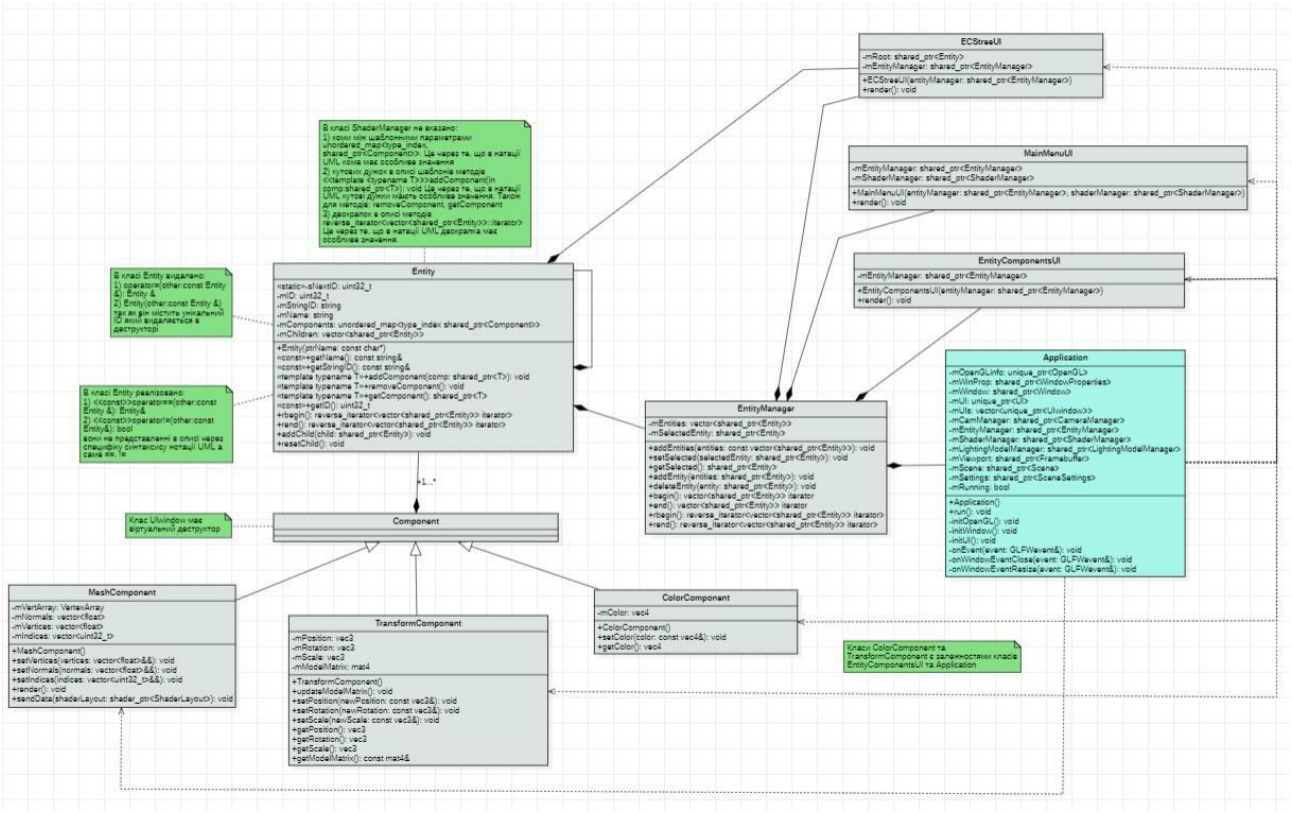

Рисунок 3.4 – Система сутностей

Мета системи сутностей зробити гнучкий механізм для редагування та рендарингу об’єктів. Сутність (Entity) це будь який об’єкт який має властивості або компоненти. Механізм абстракції компонент та його переваги наведені в розділі 2. Життєвим циклом всіх сутностей керує клас EntityManager, а життєвим циклом класу EntityManager керує клас Application. Клас Application постачає клас EntityManager всім іншим класам, які цього потребують. Класи використовують клас EntityManager та маніпулюють сутностями та їх компонентами. Наприклад: клас EntityComponentUI використовує клас EntityManager для відображення властивостей (компонент) обраної сутності та для редагування цих властивостей, а клас MainMenuUI використовує його для завантаження нових сутностей.

На рисунку 3.5 зображено систему завантаження моделей.

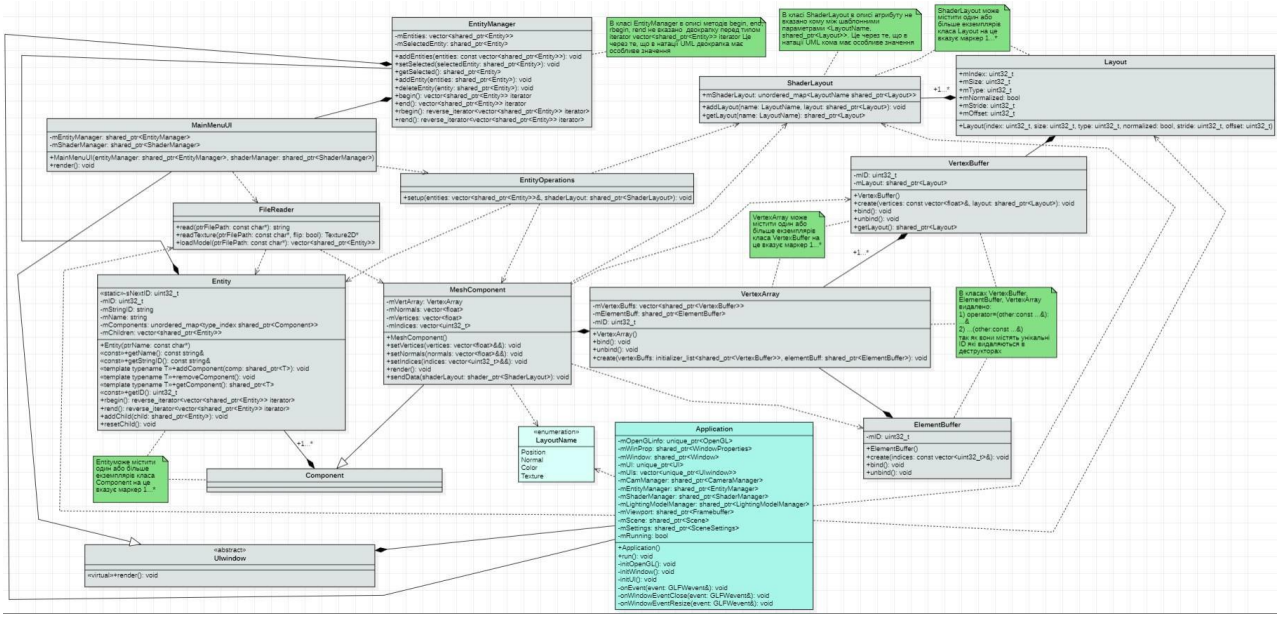

Рисунок 3.5 – Система завантаження моделей

На рисунку 3.5 показан взаємозв’язок між класами які приймають участь в завантажені нових моделей. Композиція вказує на володіння об’єктами як членами а залежність на те, що об’єкти використовуються але не є членами. Завантаження моделей виконується з використанням класу FileReader. Цей клас використовується для завантаження шейдерів, моделей та текстур та в подальшому може бути розширений для завантаження ін. типів файлів.

Цей клас не є членом жодного класу але використовується класами: Application та MainMenuUI. Клас MainMenuUI використовує його саме для завантаження моделей, через опцію меню Import Model. Для завантаження моделей FileReader використовує бібліотеку Assimp. Також для кожного завантаженого об’єкту рендарингу він створює та заповнює компоненти.

В даний момент він завантажує такі дані моделі як: вершини, індекси та нормалі. Якщо нормалей в моделі немає то бібліотека Assimp надає функціонал для розрахунку нормалей. Нормалі це дуже важлива частина додатку, так як вони використовуються при розрахунку освітлення.

Після завантаження моделей клас MainMenuUI створює об’єкт EntityOperations який (методи) для обробки сутностей.

EntityOperations ініціалізує завантажені моделі. Ініціалізація проводиться з використанням класу (компоненту) MeshComponent, який містить всі необхідні дані мешу (сітки). Після ініціалізації MainMenuUI додає завантажені сутності до загальних сутностей додатку в EntityManager.

3.2 Застосовані патерни проєктування

Патерн Dependency Injection (DI).

Dependency Injection, або впровадження залежностей, це патерн проєктування та підхід до організації коду, який допомагає керувати залежностями між класами та забезпечує більш гнучку й тестовану архітектуру. Основна ідея DI полягає в тому, що класи не повинні створювати свої залежності самостійно, а повинні отримувати їх ззовні. Це дає змогу зробити класи більш незалежними, перевикористовуваними і тестованими.

У DI залежності передаються в клас через конструктор, методи або властивості. Це може бути екземпляр іншого класу, інтерфейс, фабрика або будь-який інший об'єкт, який клас вимагає для виконання своєї роботи.

Переваги використання DI.

Зменшення зв'язаності (loose coupling). Класи залежать від абстракцій, а не від конкретних реалізацій. Це спрощує зміну і заміну залежностей без внесення змін до коду класу.

Читабельність і зрозумілість коду. Залежності явно визначені і передаються в клас, що робить код більш зрозумілим і легко читабельним.

Тестованість: Залежності можуть бути замінені моками (об'єкти, які імітують поведінку реальних об'єктів або компонентів) або підробленими об'єктами під час тестування, що полегшує написання автоматичних тестів.

Тестованість: Залежності можуть бути замінені моками (об'єкти, які імітують поведінку реальних об'єктів або компонентів) або підробленими об'єктами під час тестування, що полегшує написання автоматичних тестів.

В рамках розробленого додатку цей патерн було застосовано до класу

Application та всіх інших, яким він робить ін’єкції.

На рисунку 3.6 наведено приклад ін’єкції. CameraManager використовується в CameraPropUI та ViewportUI.

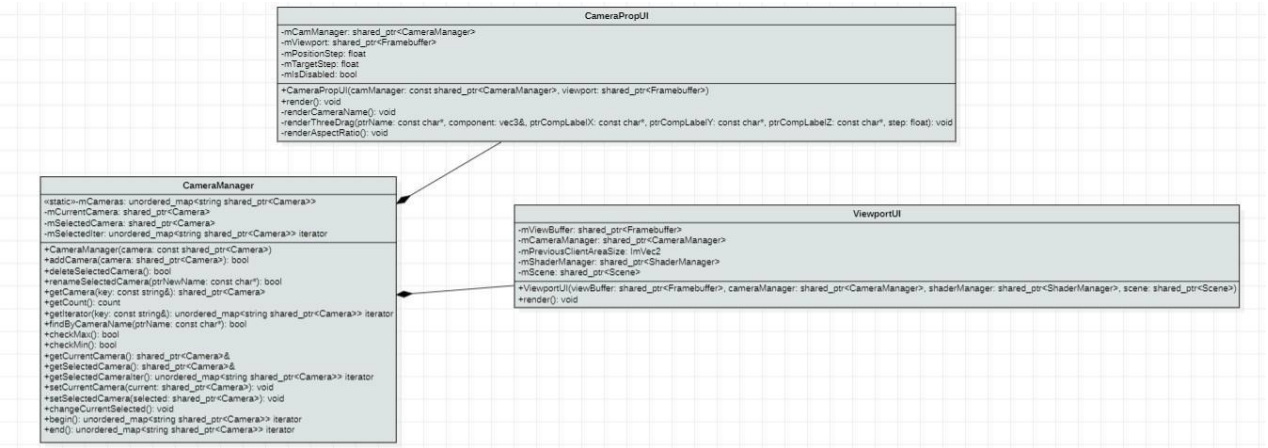

Рисунок 3.6 – Приклад DI Патерн

Monostate.

Патерн "Моностейт" (Monostate), також відомий як "Shared Static State", належить до класу патернів проєктування, що дають змогу забезпечити спільне використання стану між об'єктами без явної передачі цього стану між ними.

Основна ідея патерну Моностейт полягає в тому, що всі екземпляри класу спільно використовують один і той самий стан, збережений у статичних полях класу. При цьому кожен екземпляр класу може вносити зміни в цей стан, і всі інші екземпляри будуть бачити ці зміни.

Основні принципи та характеристики патерну Моностейт.

Спільне використання стану: Усі екземпляри класу спільно використовують один і той самий стан, що зберігається в статичних полях класу.

Відсутність прив'язки до конкретного екземпляра: Кожен екземпляр класу може вносити зміни в стан, і всі інші екземпляри будуть бачити ці зміни.

Приховування деталей реалізації. Клієнтський код працює з об'єктами класу, ніби кожен екземпляр має свій власний стан, але насправді всі екземпляри спільно використовують один і той самий стан.

Глобальність стану. Стан, який спільно використовується всіма екземплярами класу, може бути доступний з будь-якого місця програми, що може призвести до потенційних проблем із безпекою та узгодженістю даних.

Прикладом використання патерну Моностейт може бути клас, що представляє налаштування програми. У цьому разі всі екземпляри класу матимуть доступ до одного й того самого набору налаштувань, і будь-які зміни, внесені одним екземпляром, будуть відображені на всіх інших.

В рамках розробленого додатку цей патерн було застосовано для класу CameraManager, так як було прийнято рішення, що цей клас має бути єдиним у додатку. Для реалізації єдиного стану існує декілька патернів Monostate та Singelton. Але Singelton клас доступний з будь-якою точки додатку, що не є необхідною умовою так як в проєктуванні додатку використовується патерн DI. Завдяки патерну DI можно забезпечити кожен клас іншим класом (який необхідний) зробивши ін’єкцію.

На рисунку 3.7 зображено клас CameraManager. Клас має поле стан mCameras.

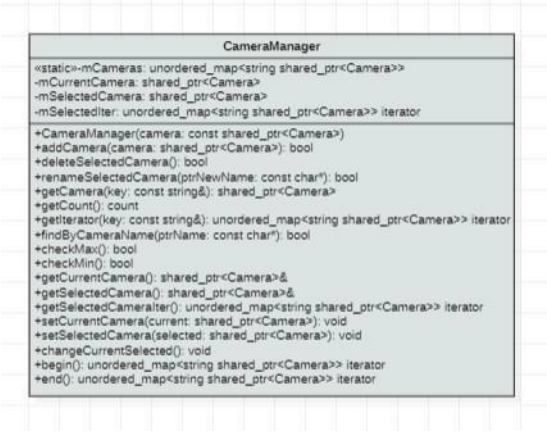

Рисунок 3.7 – Клас CameraManager

Патерн Entity Component System (ECS).

ECS [20] - це патерн проєктування та архітектурний підхід, що використовується в розробці комп'ютерних ігор та інших програм, де присутня велика кількість об'єктів та їхніх взаємодій.

В основі ECS лежить поділ даних і логіки на компоненти, сутності та системи. Ось як працює ECS.

Сутність (Entity): сутність являє собою порожній контейнер, який не містить будь-якої функціональності або даних. Вона служить лише для об'єднання компонентів.

Компонент (Component): компоненти представляють дані або функціональність, які притаманні конкретному об'єкту або сутності в грі. Наприклад, у компонента "Позиція" можуть бути дані про поточні координати об'єкта, а у компонента "Намальовування" може бути функціональність для відображення об'єкта на екрані.

Система (System): системи являють собою функціональні блоки, які працюють над набором компонентів. Вони містять логіку опрацювання даних компонентів і можуть виконувати операції, як-от оновлення позиції об'єкта, перевірка зіткнень або відтворення об'єктів на екрані. Кожна система обробляє тільки ті компоненти, які їй необхідні.

Основні переваги ECS.

Гнучкість і розширюваність: завдяки поділу даних і логіки, ECS забезпечує гнучку й розширювану архітектуру. Ви можете додавати і видаляти компоненти для сутностей .

Спрощення керування станом: ECS дає змогу зручно керувати станом об'єктів або сутностей, оскільки стан зберігається в компонентах, а системи маніпулюють цими компонентами.

Паралельна обробка: ECS незалежно один від одного. Це може

призвести до поліпшення продуктивності та ефективності.

ECS є потужним і гнучким підходом до розроблення ігор та інших програм із великою кількістю об'єктів і взаємодій. Він дає змогу легко масштабувати та змінювати системи та компоненти, що робить його популярним серед розробників.

На рисунку 3.8 зображено ECS.

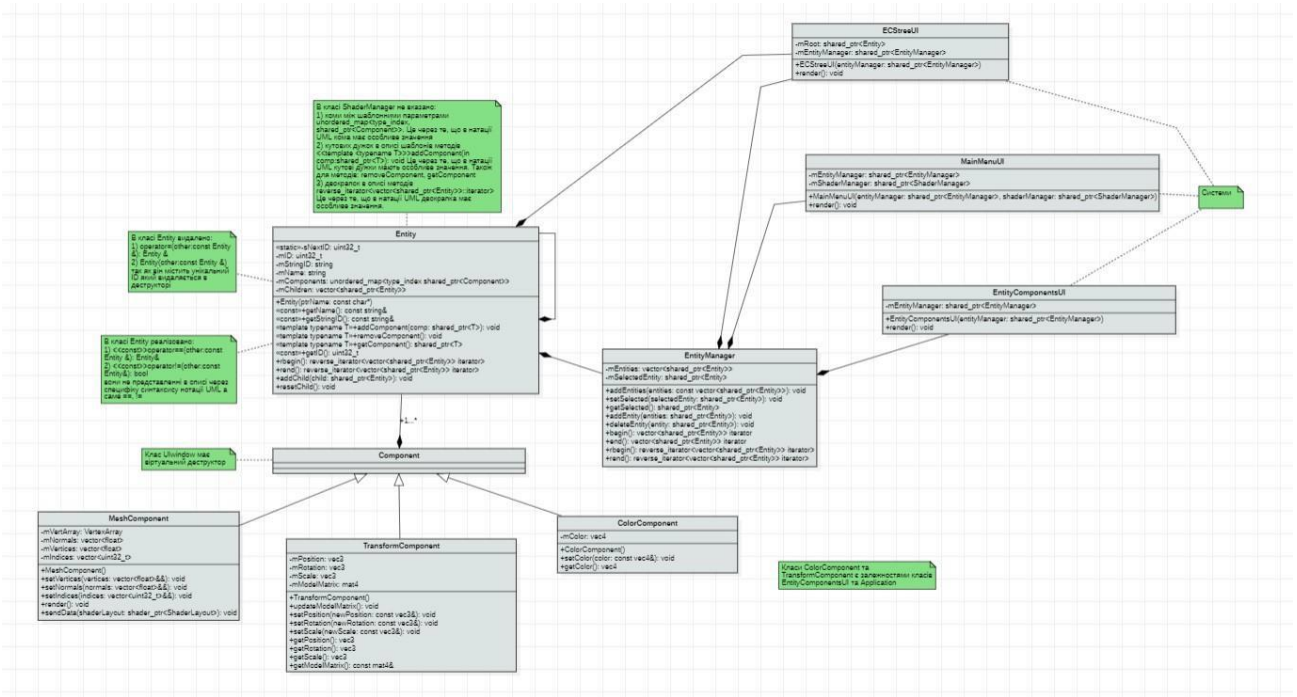

Рисунок 3.8 – ECS

3.3 Стандартні (STL) та сторонні типи даних

У цьому підрозділі наведено опис типів та контейнерів стандартної бібліотеки які були застосовані у додатку. Також наведено їх швидкість роботи.

Контейнер std::stack.

std::stack [3] в STL являє собою контейнерний адаптер, який реалізує структуру даних стек (stack). Стек являє собою колекцію об'єктів, керовану за принципом "останнім прийшов, першим вийшов".

Основні характеристики std::stack.

Обмежений доступ: std::stack надає тільки обмежений набір операцій для роботи зі стеком. Основні операції включають додавання елемента у верхівку стека (push), видалення елемента з верхівки стека (pop) і доступ до верхівки стека (top).

Внутрішній контейнер: std::stack не є самостійною структурою даних, а є адаптером, який використовує інший контейнер як внутрішню реалізацію. За замовчуванням std::deque використовується як контейнер для std::stack, але можна вказати будь-який інший контейнер, який відповідає вимогам послідовного контейнера.

Мала складність операцій: операції push, pop і top виконуються за постійний час O(1), незалежно від розміру стека.

std::stack надає простий і зручний інтерфейс для роботи зі стеком і є корисним інструментом для реалізації алгоритмів, де потрібна LIFO-поведінка. Він дає змогу ефективно додавати і видаляти елементи зі стека, забезпечуючи коректність порядку обробки даних.

Контейнер std::unordered_map.

std::unordered_map [4] в STL є реалізацією асоціативного контейнера у вигляді хеш-таблиці. Він надає функціональність зберігання пар ключ-значення, де ключі є унікальними і хешованими, а доступ до значень здійснюється за ключем.

Основні характеристики std::unordered_map.

Швидкий доступ: std::unordered_map забезпечує майже постійний час виконання для операцій пошуку, вставки та видалення елементів. У добре спроектованій хеш-таблиці складність цих операцій становить O(1), хоча в найгіршому випадку може досягати O(n), де n - кількість елементів у контейнері.

Невпорядкованість: порядок елементів у std::unordered_map не визначено і він може змінюватися під час операцій вставки та видалення.

Унікальні ключі: ключі в std::unordered_map мають бути унікальними. Якщо під час вставки виявляється дублікат ключа, нове значення замінює наявне.

Хешування: ключі в std::unordered_map мають бути хешованими. Контейнер використовує хеш-функцію для перетворення ключів в індекси внутрішньої хеш-таблиці.

Складність операцій std::unordered_map.

Вставка (insertion): у добре спроєктованій хеш-таблиці вставка елемента виконується за майже постійний час O(1), але може сягати O(n) у найгіршому випадку, якщо потрібне перехешування і перебудова таблиці.

Пошук (lookup): складність операції пошуку елемента також становить майже постійний час O(1), але в гіршому випадку може бути O(n).

Видалення (deletion): Складність операції видалення елемента також становить майже постійний час O(1), але в гіршому випадку може бути O(n).

Важливо зазначити, що продуктивність std::unordered_map залежить від обраної хеш-функції та її рівномірності розподілу значень ключів. При невдалому виборі або поганому розподілі ключів може виникнути колізія, що призведе до погіршення продуктивності.

Загалом, std::unordered_map є ефективним контейнером для швидкого пошуку елементів за ключем, але вимагає ретельного вибору хеш-функції та поводження з колізіями для досягнення найкращої продуктивності.

Контейнер std::vector.

std::vector [5] у STL є реалізацією динамічного масиву. Він надає функціональність зберігання та керування елементами в послідовному контейнері.

Основні характеристики std::vector.

Доступ до елемента за індексом здійснюється за постійний час O(1).

Динамічна зміна розміру: std::vector може змінювати свій розмір динамічно. При вставці нових елементів, контейнер автоматично збільшує свій розмір і виділяє додаткову пам'ять. При видаленні елементів, контейнер автоматично зменшує свій розмір.

Послідовне розташування елементів. Елементи в std::vector розташовуються послідовно в пам'яті. Це забезпечує швидкий доступ до елементів і дозволяє використовувати ітератори для обходу контейнера.

Доступні операції: std::vector надає операції вставки, видалення та доступу до елементів, а також можливість зміни розміру контейнера. Він також підтримує операції порівняння, сортування та інші корисні функції.

Складність операцій std::vector.

Доступ за індексом (indexing): отримання елемента за індексом виконується за постійний час O(1).

Вставка (insertion): вставка нового елемента в кінець вектора виконується за амортизований постійний час O(1). Однак, якщо вставка відбувається в середину вектора або потрібен перерозподіл пам'яті, то складність може бути O(n), де n - кількість елементів у векторі.

Видалення (deletion): видалення елемента з вектора виконується за лінійний час O(n), де n - кількість елементів у векторі. При видаленні елемента з кінця вектора операція виконується за постійний час O(1).

Зміна розміру (resizing): Зміна розміру вектора виконується за лінійний час O(n), де n - новий розмір вектора. При збільшенні розміру вектора може бути також потрібен перерозподіл пам'яті та копіювання елементів.

std::vector є одним із найпоширеніших контейнерів у STL завдяки своїй ефективності та зручності використання. Він широко застосовується в ситуаціях, де потрібно зберігати та керувати динамічними масивами елементів.

Тип std::string.

std::string у STL є класом, призначеним для роботи з рядками. Він надає функціональність зберігання, маніпулювання та обробки строкових даних.

Основні характеристики std::string.

Зручність роботи з рядками: std::string надає широкий набір функцій і операцій для роботи з рядками. Він дозволяє виконувати конкатенацію, порівняння, пошук, витяг підрядків та інші операції над рядками.

Динамічна зміна розміру: std::string може змінювати свій розмір динамічно. При додаванні або видаленні символів, розмір рядка автоматично змінюється, без необхідності вручну керувати пам'яттю.

Робота з символами: std::string дає змогу звертатися до окремих символів рядка та змінювати їхні значення. Також можна отримати доступ до символів рядка за допомогою оператора індексації або ітераторів.

Складність операцій std::string.

Доступ за індексом (indexing): отримання символу за індексом виконується за постійний час O(1).

Вставка (insertion): Вставка символу або підрядка в рядок виконується за лінійний час O(n), де n - довжина підрядка, що вставляється, або кількість символів, що вставляються.

Видалення (deletion): Видалення символів із рядка також виконується за лінійний час O(n), де n - кількість видалених символів.

Порівняння (comparison): порівняння двох рядків виконується за лінійний час O(n), де n - довжина довшого рядка.

Пошук (search): пошук підрядка в рядку виконується за лінійний час O(n), де n - довжина рядка.

std::string є основним інструментом для роботи з рядками в C++. Він надає зручний та ефективний спосіб роботи з текстовими даними і широко використовується в різних додатках та алгоритмах.

Тип std::shared_ptr.

std::shared_ptr у STL є класом, призначеним для управління спільною власністю (shared ownership) динамічно виділених об'єктів у C++. Він забезпечує автоматичне керування часом життя об'єктів або автоматичне звільнення пам'яті, коли об'єкт більше не використовується.

Основні характеристики std::shared_ptr.

Спільна власність: std::shared_ptr дозволяє кільком покажчикам одночасно володіти об'єктом. Це корисно у випадках, коли необхідно передавати об'єкт між різними частинами програми або коли об'єкти потрібно розділяти між кількома компонентами.

Лічильник посилань: std::shared_ptr використовує лічильник посилань для відстеження кількості покажчиків, які володіють об'єктом. Коли лічильник посилань стає рівним нулю, об'єкт автоматично видаляється з пам'яті.

Динамічне виділення пам'яті: std::shared_ptr використовується для управління динамічно виділеною пам'яттю, у тому числі для об'єктів, створених з використанням оператора new.

Поділ об'єктів між потоками: std::shared_ptr може бути безпечно використаний у багатопотоковому середовищі, де кілька потоків можуть одночасно звертатися до одного об'єкта.

Складність операцій std::shared_ptr.

Створення (creation): створення std::shared_ptr виконується за постійний час O(1). Створення прийнято роботи функцією std::make_shared.

Збільшення лічильника посилань (incrementing): збільшення лічильника посилань при створенні нового std::shared_ptr виконується за постійний час O(1).

Зменшення лічильника посилань (decrementing): зменшення лічильника посилань при видаленні std::shared_ptr виконується за постійний час O(1). Коли лічильник посилань стає рівним нулю, об'єкт видаляється з пам'яті.

Видалення (deletion): видалення об'єкта з пам'яті виконується при досягненні лічильником посилань значення нуль і займає постійний час O(1).

std::shared_ptr є потужним інструментом керування пам'яттю та спільною власністю в C++. Він дає змогу ефективно керувати часом життя об'єктів і уникати витоків пам'яті.

Тип std::unique_ptr.

std::unique_ptr у STL є класом, призначеним для управління унікальною власністю (unique ownership) динамічно виділених об'єктів у C++. Він забезпечує автоматичне звільнення пам'яті, коли об'єкт більше не використовується, і гарантує, що тільки один покажчик володіє об'єктом.

Основні характеристики std::unique_ptr.

Унікальна власність: std::unique_ptr дозволяє тільки одному покажчику володіти об'єктом. Це гарантує, що тільки один покажчик буде керувати часом життя об'єкта, і дозволяє уникнути проблем, пов'язаних з розділеною власністю. Управління часом життя: std::unique_ptr автоматично звільняє пам'ять, коли об'єкт більше не використовується або коли std::unique_ptr знищується. Це гарантує, що ресурси будуть коректно звільнені і не буде витоків пам'яті.

Відсутність копіювання: std::unique_ptr не підтримує копіювання об'єктів. Це означає, що його не можна копіювати, але можна переміщати (move). Це пов'язано з його унікальною власністю і гарантує, що тільки один покажчик може володіти об'єктом у будь-який момент часу.

Складність операцій std::unique_ptr.

Створення (creation): створення std::unique_ptr виконується за постійний час O(1). Створення прийнято роботи функцією std::make_unique.

Переміщення (moving): переміщення std::unique_ptr від одного покажчика до іншого виконується за постійний час O(1).

Видалення (deletion): видалення об'єкта з пам'яті виконується при знищенні std::unique_ptr і займає постійний час O(1).

std::unique_ptr є корисним інструментом для керування пам'яттю в

ситуаціях, де об'єкт має мати лише одного власника і автоматичне звільнення пам'яті є бажаним. Він допомагає уникнути помилок зі спільною власністю та витоками пам'яті, надаючи безпечний та ефективний спосіб керування динамічними об'єктами в C++.

Інші сторонні типи даних.

Також при розробці додатку були застосовані вбудовані типи мови C++: int, char та ін. Та дуже активно використовувались типи бібліотеки GLM: glm::vec3 (для зберігання вектора 1x3), glm::mat4 (для зберігання матриці 4x4) та ін. Ці типи не потребують тестування на швидкість обробки так як не мають методів обробки.

Власні користувацькі типи даних.

Було створено багато власних користувацьких типів даних серед яких: Layout (для збереження параметрів атрибуту шейдера), ShaderLayout (для збереження всіх атрибутів шейдера), Shader (представляє тип даних шейдер), VertexBuffer (представляє вершиний буфер для зберігання даних атрибуту на GPU), ElementBuffer (представляє індексний буфер для зберігання індексів вершин на GPU), VertexArray (представляє масив для збереження всіх VertexBuffer та ElementBuffer відповідної сутності) та ін. Ці типи поки не були протестовані на швидкість обробки. Код цих типів наведено у ДОДАТОК А КОД.

3.4 Проєктування графічного інтерфейсу

Для проєктування графічного інтерфейсу було застосовано Wireframe діаграми.

Нижче наведені діаграми, які демонструють базовий графічний інтерфейс

додатку. В майбутньому ці даіграми, як і додаток можуть бути розширені новими графічними елементами або вікнами.

На рисунку 3.9 зображено спроєктований загальний вигляд UI додатку.

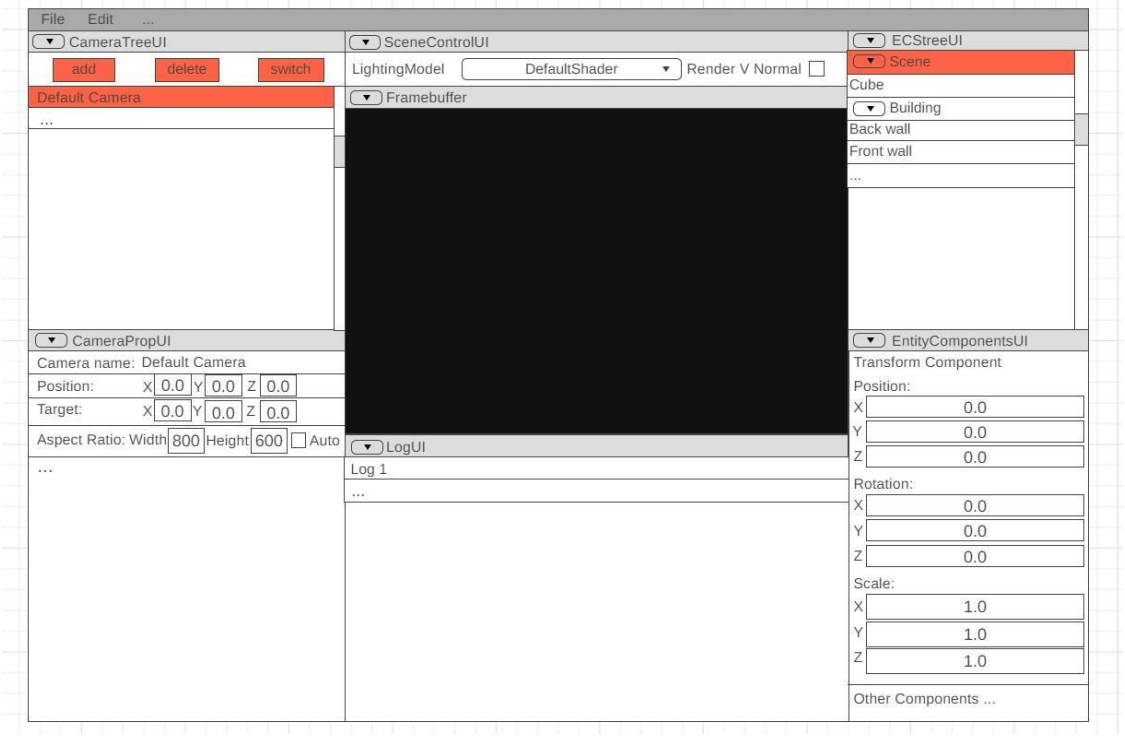

Рисунок 3.9 – Спроєктований загальний вигляд UI додатку

На рисунку 3.10 зображено спроєктований меню UI додатку.

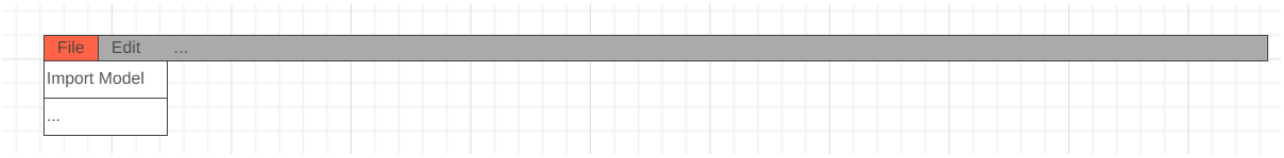

Рисунок 3.10 –Спроєктований меню UI додатку

На рисунку 3.11 зображено вигляд UI вікна для завантаження файлів.

Рисунок 3.11 – UI вікно для завантаження файлів

На рисунку 3.12 зображено спроєктоване контекстне меню та вікно переменування імені камери.

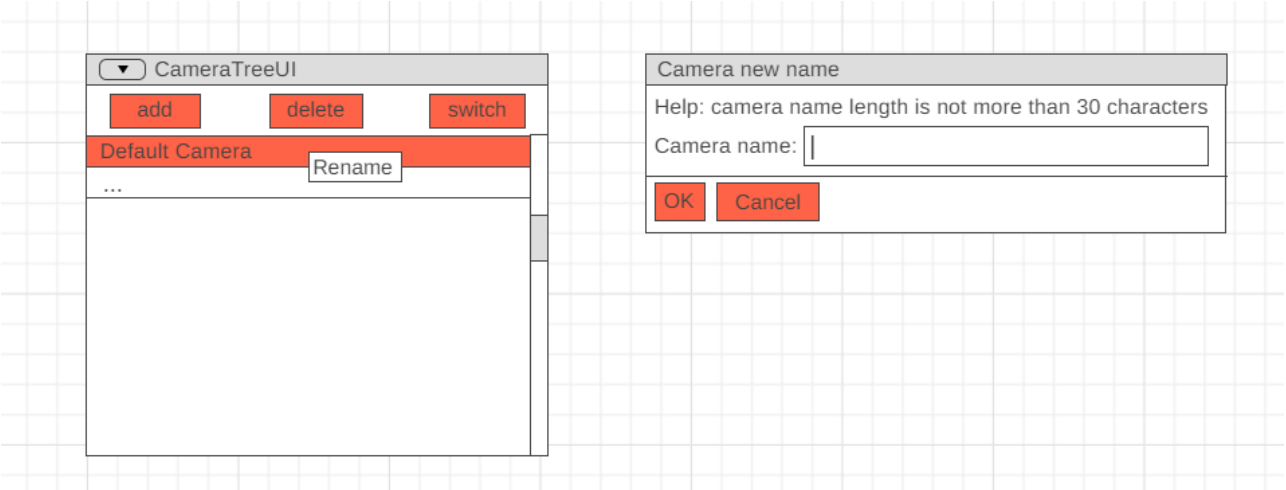

Рисунок 3.12 – Контекстне меню та вікно переменування імені камери

На рисунку 3.13 зображено спроєктоване вікно попередження.

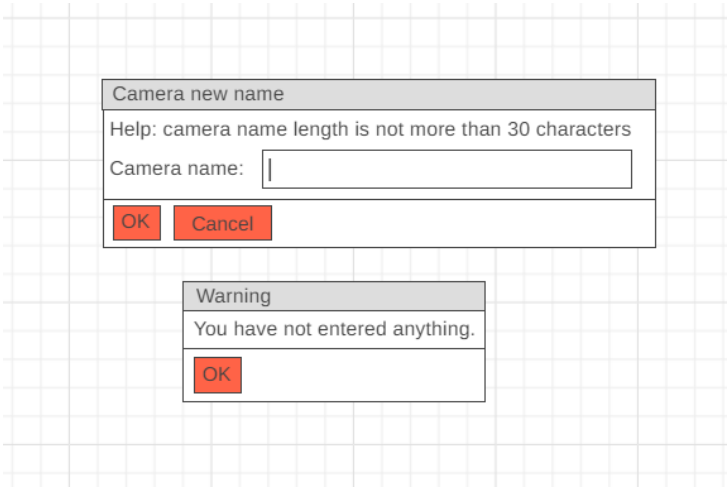

Рисунок 3.13 – Вікно попередження

На рисунку 3.14 зображено спроєктоване вікно для додавання нової камери.

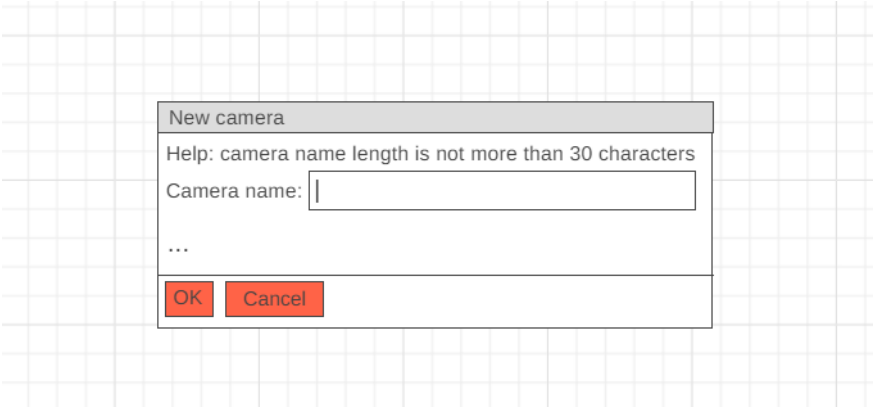

Рисунок 3.14 – Вікно для додавання нової камери

4 РЕЗУЛЬТАТИ РОЗРОБКИ ДОДАТКУ ТА ОПИС РОБОТИ МОДЕЛЕЙ ОСВІТЛЕННЯ

4.1 Реалізований графічний інтерфейс

Нижче наведено реалізований графічний інтерфейс користувача який було спроєктовано та представлено у розділі 3.

На рисунку 4.1 зображено загальний вигляд графічного інтерфейсу додатку. Слід зазначити, що графічний інтерфейс підтримує систему докінг, тобто елементи (вікна) графічного інтерфейсу можна розмістити у бажоному місці. Приклад докингу зображено на рисунку 4.2, вікно CameraTreeUI було перенесено поряд з вікном LogUI. Цей інтерфейс можна назвати адаптивним так як користувач може адаптувати його під себе.

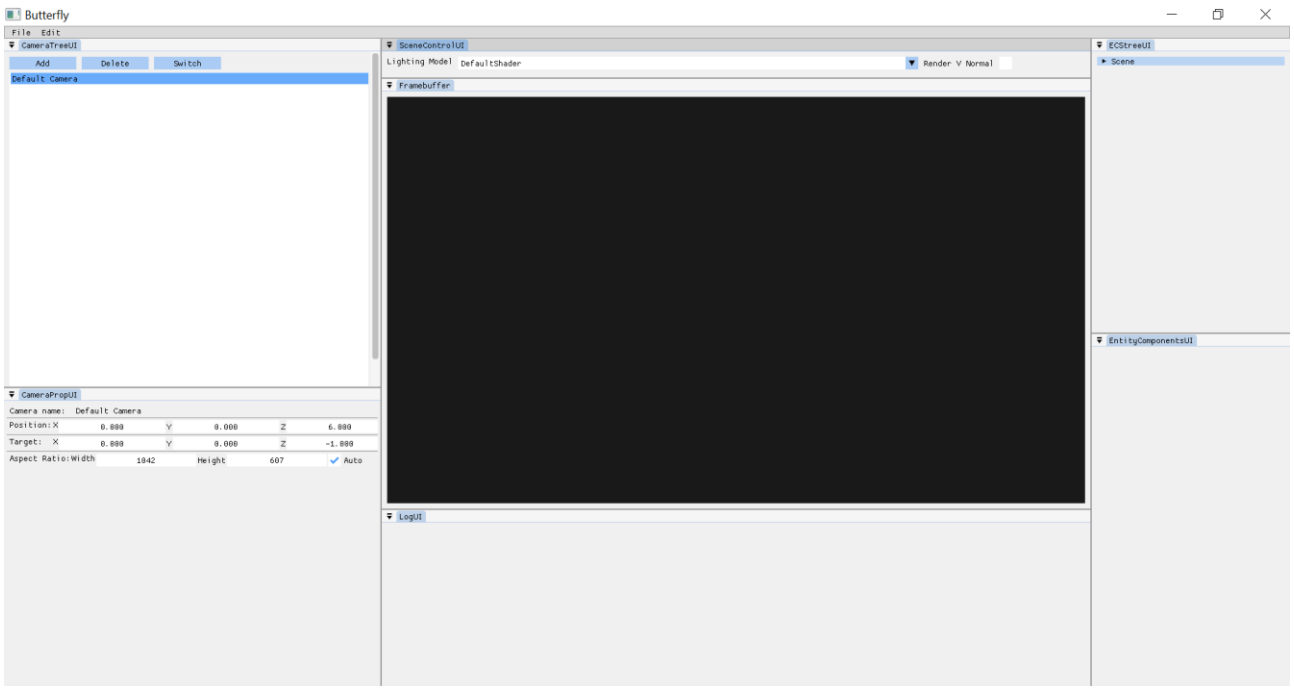

Рисунок 4.1 – Загальний вигляд графічного інтерфейсу додатку

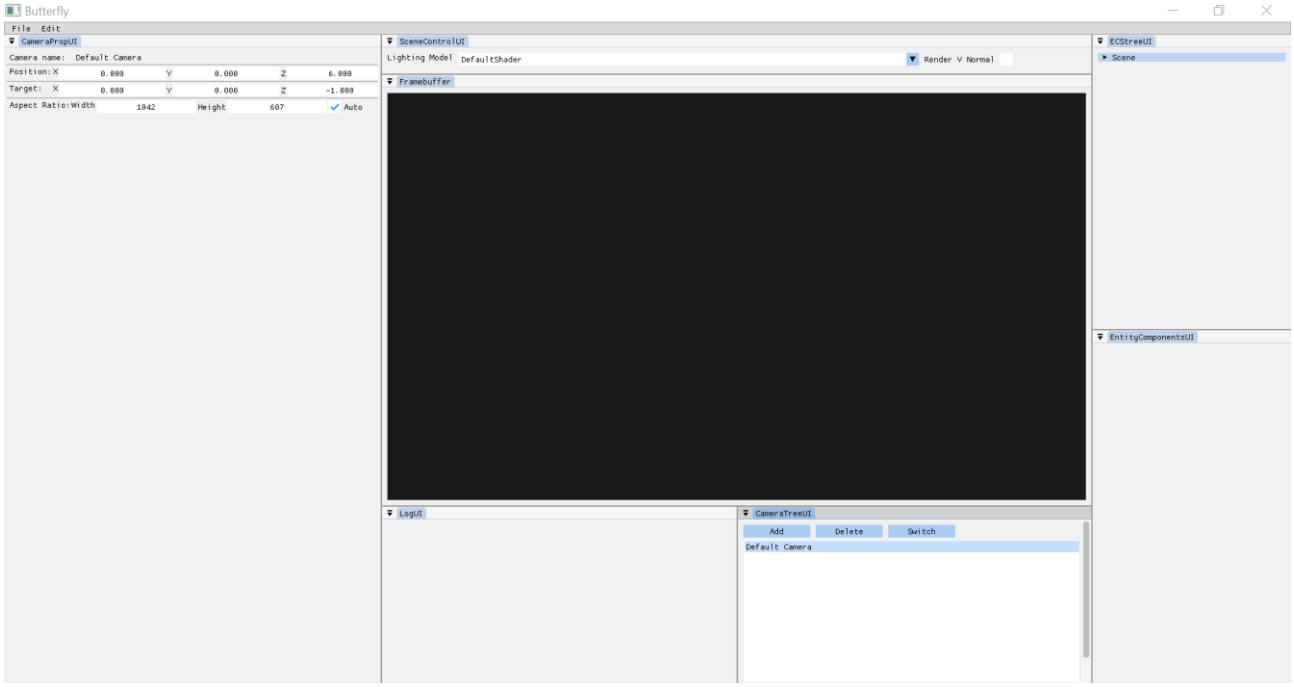

Рисунок 4.2 – Приклад докингу

На рисунку 4.3 зображено головне меню додатку.

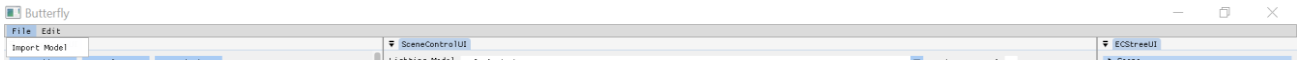

Рисунок 4.3 – Головне меню додатку

На рисунку 4.4 зображено діалогове вікно для завантаження моделей. Слід зазначити, що це діаголове вікно в подальшому може бути використано для завантаження інших типів файлів. Всі діалогові вікна та вікна попереджень, окрім вікна представленого на рисунку 4.4, блокують роботу додатку поки користувач не закриє їх будь-яким способом. Формати файлів, які можуть бути завантажені наведені в лівому нижньому куті вікна. Пошук файлів можна проводити через натискання на каталоги або через пошуковий рядок, який розміщено у верхній частині вікна, також можна комбінувати ці способи пошуку. Поточний шлях показано у верхній частині вікна починаючи з диска “D:” і до кінця.

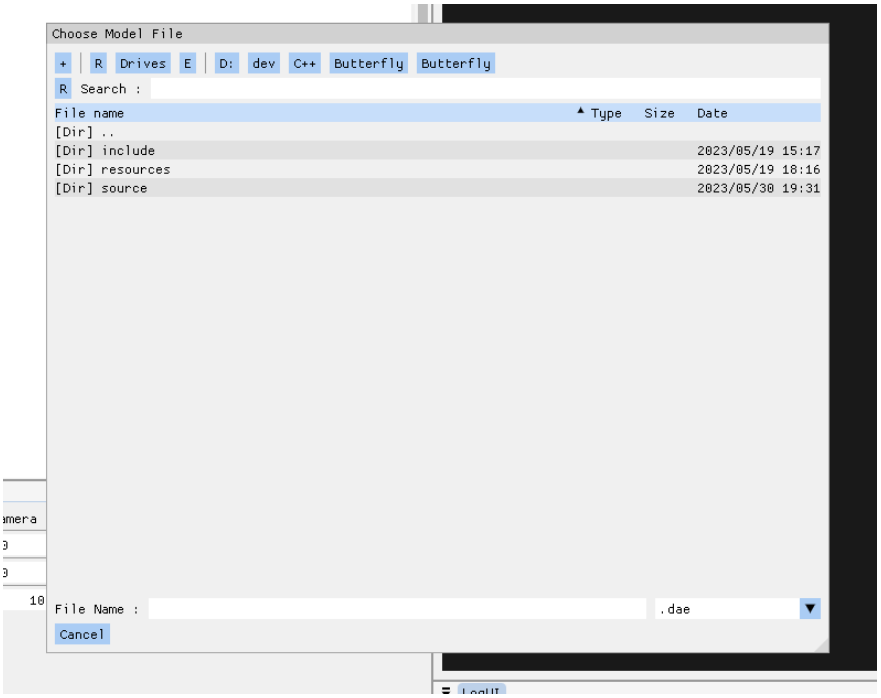

Рисунок 4.4 – Діалогове вікно для завантаження моделей

На рисунку 4.5 зображено контекстне меню вікна CameraTreeUI.

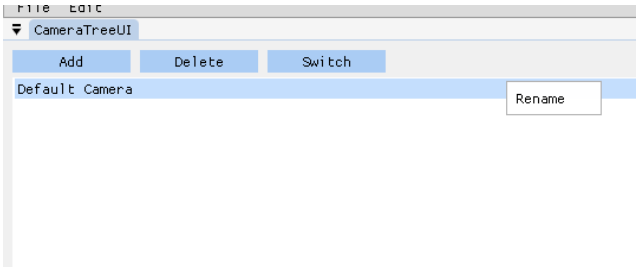

Рисунок 4.5 – Контекстне меню вікна CameraTreeUI

На рисунку 4.6 зображено діалогове вікно перейменування імені камери.

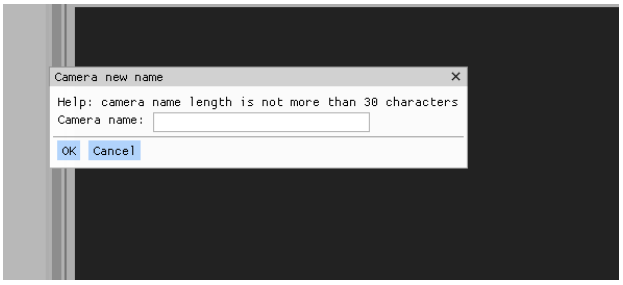

Рисунок 4.6 – Діалогове вікно перейменування імені камери

На рисунку 4.7 зображено діалогове вікно додавання нової камери. Це вікно з часом може бути розширене додатковими полями характеристик камери.

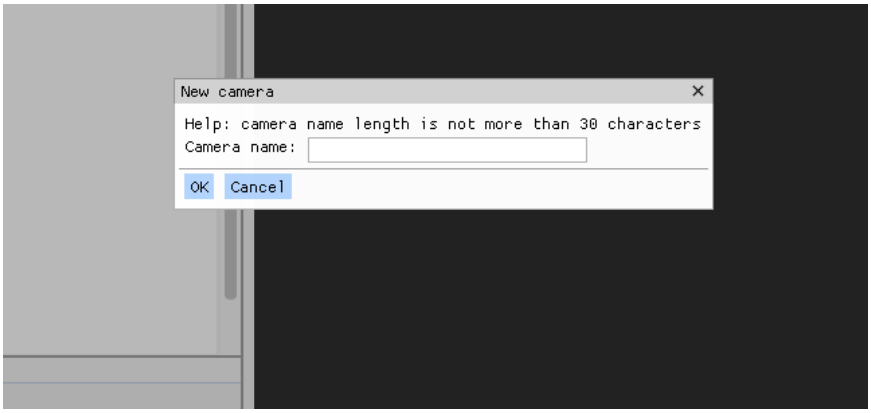

Рисунок 4.7 – Діалогове вікно додавання нової камери

На рисунку 4.8 зображено приклад діалогового вікна попередження.

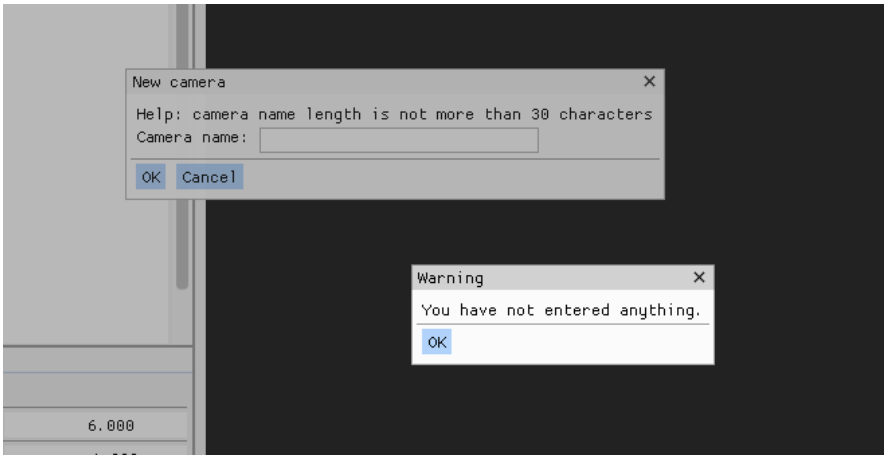

Рисунок 4.8 – Приклад діалогового вікна попередження

На прикінці розробки основи проєкту було виявлено недопрацювання при проєктуванні. А саме відсутність взаємодії з моделями освітлення для їх налаштування. Тому під час конструювання було додано ще два вікна UI, а саме вікно для налаштування моделі освітлення Фонга та вікно для налаштування моделі освітлення Кука-Торранса.

На рисунку 4.9 зображено діалогове вікно для налаштування моделі

освітлення Фонга.

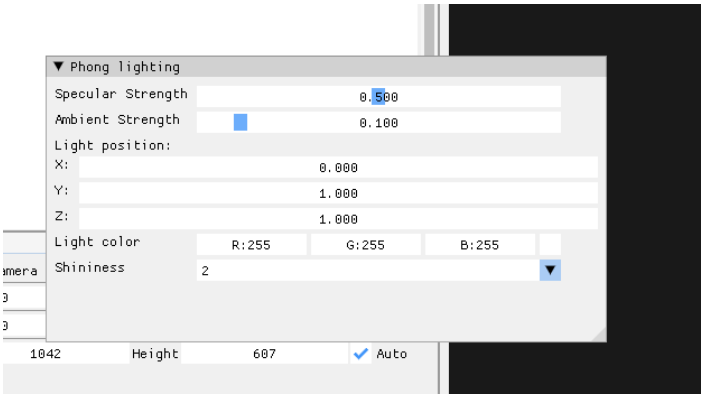

Рисунок 4.9 – Діалогове вікно для налаштування моделі освітлення Фонга

На рисунку 4.10 зображено діалогове вікно для налаштування моделі освітлення Кука-Торранса.

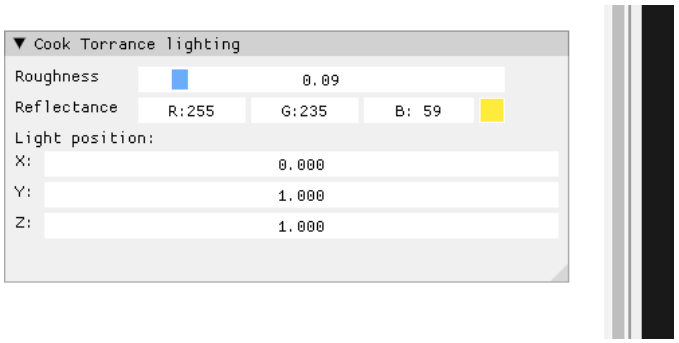

Рисунок 4.10 – Діалогове вікно для налаштування моделі освітлення Кука- Торранса

4.2 Послідовнійсть дій при взаємодії з інтерфейсом

Нижче наведені Use Case діаграми які демонструють роботу графічного інтерфейсу.

На рисунку 4.11 зображено дії входу до додатку та виходу з нього. Вход робиться при натисканні на ярлик додатку а виход при натисканні на хрестик в правому верхньому куті додатку. Також розробник успадковує всі дії користувача.

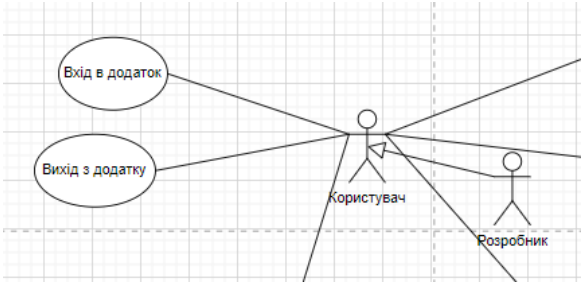

Рисунок 4.11 – Дії входу та виходу

На рисунку 4.12 зображено дії з системою сутностей. Сутності можна додавати, змінювати їх колір, положення, орієнтацію та маштаб.

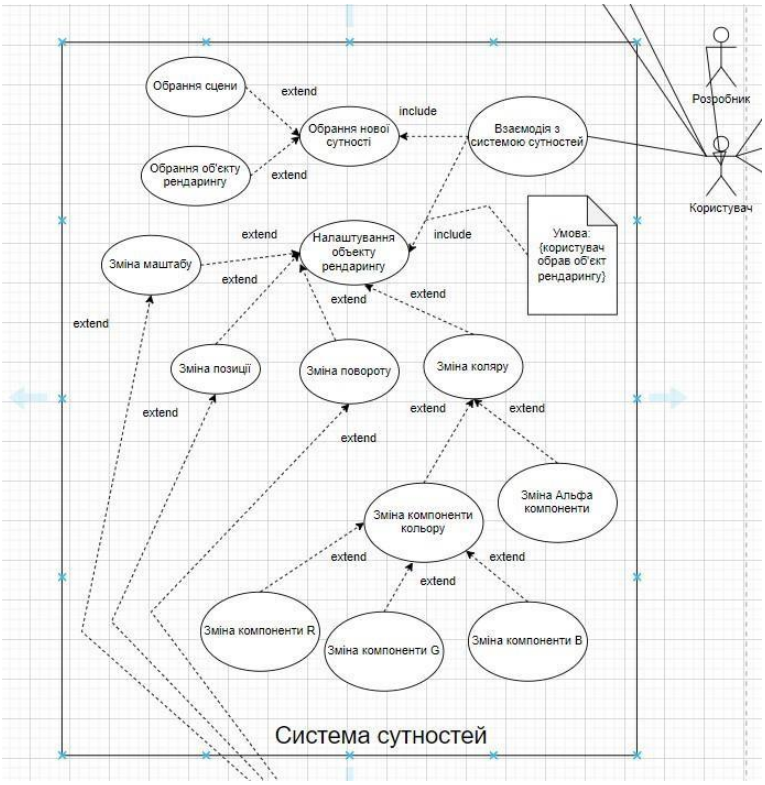

Рисунок 4.12 – Дії з системою сутностей

На рисунку 4.13 зображено дії з системою камер.

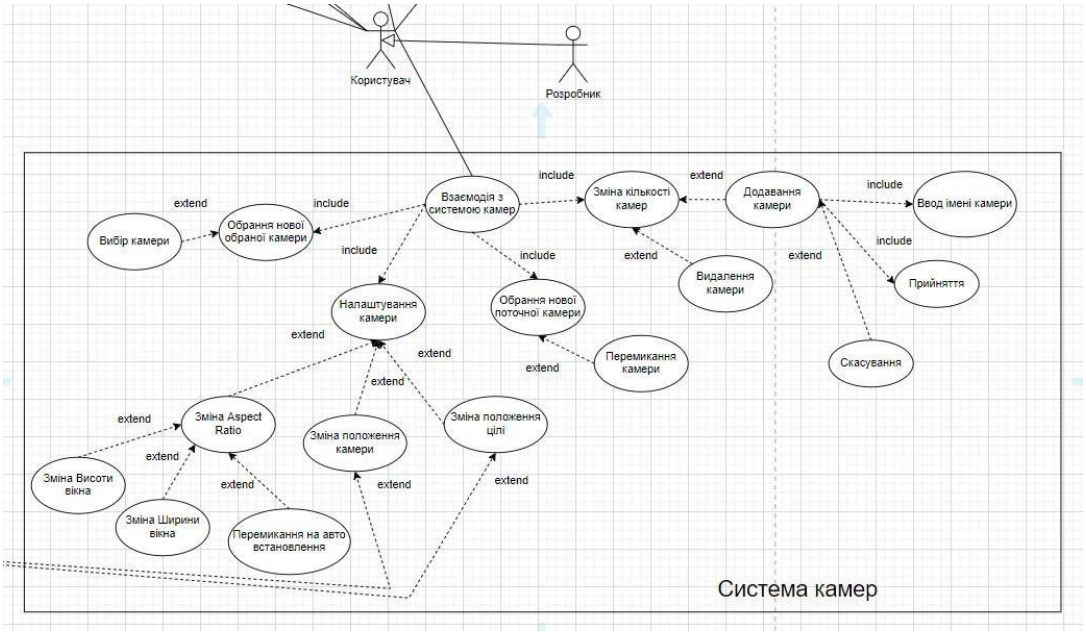

Рисунок 4.13 – Дії з системою камер

На рисунку 4.14 зображено дії з системою рендаринг.

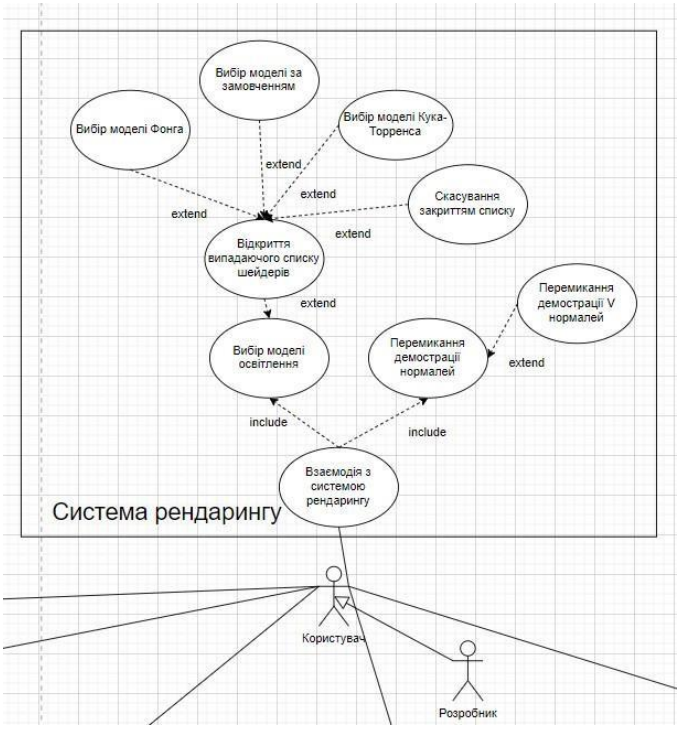

Рисунок 4.14 – Дії з системою рендаринг

На рисунку 4.15 зображено дії з меню.

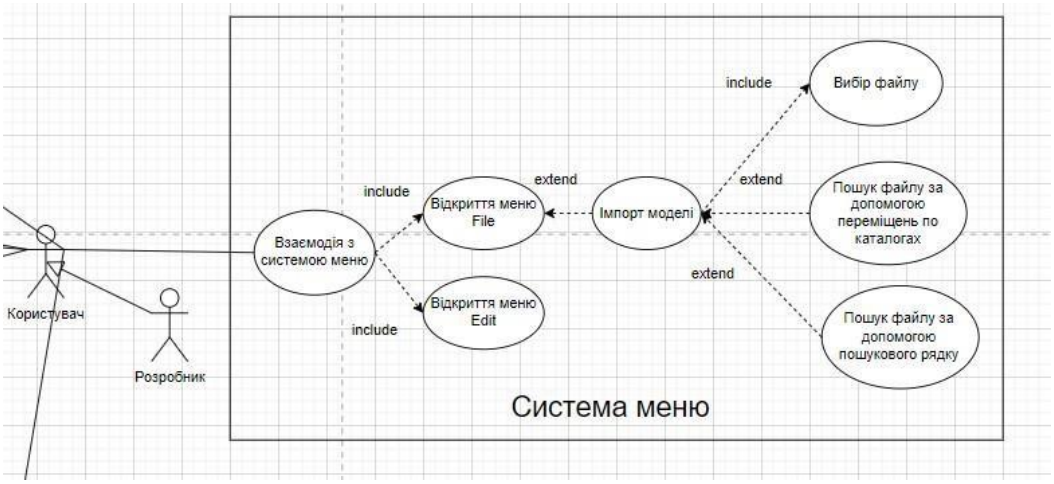

Рисунок 4.15 – Дії з меню

На рисунку 4.16 зображено абстракцію діаграми для тривимірних координат. Ця абстракція використовується системами: камер та сутностей, які наведені вище.

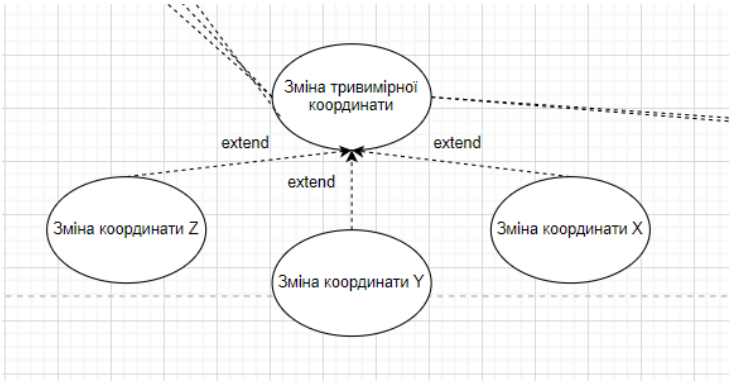

Рисунок 4.16 – Абстракція діаграми для тривимірних координат

4.3 Безпека роботи додатку

На момент розробки основи додатку було виявлено два небезпечних місця:

- Обробка виключень,
- Перевірка даних, що вводяться;

Обробка виключень не була вирішена на момент завершення розробки основи проєкту.

Перевірка даних, що вводяться була вирішена, та демонструється в розділі

4.4 Опис моделей освітлення

Існує багато моделей освітлення [9]:

- Модель Ламберта
- Модель Фонга
- Модель Бліна-Фонга
- Ізотропна модель Уорда
- Модель Міннаерта [8]
- Модель Ломмеля-Зиліджера
- Модель Страуса [13]
- Найпростіша анізотропна модель
- Анізотропна модель Ворда
- Модель Кука-Торранса [10]
- Модель Орена-Найара та ін. [10]

При розробці додатку було реалізовано дві моделі: модель Фонга та модель Кука-Торранса.

Опис моделі Фонга.

Модель освітлення Фонга [15], названа на честь Джеймса Фонга, є класичною моделлю для симуляції освітлення в комп'ютерній графіці. Вона належить до групи моделей освітлення, відомих як моделі освітлення з локальним освітленням.

Локальна модель у контексті комп'ютерної графіки відноситься до моделювання освітлення, яка враховує тільки пряму взаємодію між джерелами світла і поверхнями об'єктів. Вона припускає, що світло поширюється незалежно від інших об'єктів у сцені та не враховує глобальні освітлені ефекти, такі як відбиття, заломлення і тіні від інших об'єктів або оточення.

У локальній моделі освітлення об'єкти розглядаються ізольовано від навколишнього простору, і їхнє освітлення визначається тільки світлом, що йде безпосередньо від джерел світла. Ця модель спрощує обчислення і дає змогу досягти досить реалістичного вигляду освітлення для багатьох додатків комп'ютерної графіки, таких як ігри, анімація та візуалізація.

Модель освітлення Фонга являє собою комбінацію трьох компонент освітлення: фонового (ambient), дифузного (diffuse) і дзеркального (specular). Кожна з цих компонент визначає внесок світла в підсумковий колір об'єкта на основі його матеріалу і положення джерела світла.

Фонове освітлення (ambient): це постійне освітлення, яке присутнє скрізь і не залежить від положення джерела світла. Воно моделює розсіяне і відбите світлове випромінювання в навколишньому середовищі. Фонова складова надає об'єкту мінімальну яскравість і колір.

Дифузне освітлення (diffuse): дифузне освітлення моделює розсіяне світло, яке поширюється рівномірно в усіх напрямках від джерела світла. Воно залежить від кута падіння світла на поверхню об'єкта і від властивостей матеріалу цієї поверхні. Дифузна складова визначає колір і яскравість об'єкта залежно від кута падіння світла на його поверхню.

Дзеркальне освітлення (specular): дзеркальне освітлення моделює відбите світло, яке створює відблиск на поверхні об'єкта. Воно залежить від положення спостерігача, положення джерела світла і властивостей матеріалу поверхні. Дзеркальна складова визначає яскраві металеві відблиски на поверхні об'єкта.

На рисунку 4.17 зображено формулу [12] для розрахунку коліру об’єкту за моделю Фонга.

$$I = k_a I_a C + I_l \cdot C \cdot \max(0, (n, l)) + k_s C_s I_l \cdot \max(0, (r, l))^p$$

Рисунок 4.17 – Формула розрахунку коліру об’єкта за моделю Фонга

де:
I: загальний колір освітлення на поверхні

k_a: коефіцієнт відбиття фонового освітлення (ambient reflection coefficient)

I_a: інтенсивність фонового освітлення (ambient light)

C: колір поверхні об'єкта

I_l: інтенсивність джерела світла

C: колір поверхні об'єкта

n: нормаль поверхні

l: вектор напрямку джерела світла

k_s: коефіцієнт дзеркального відбиття (specular reflection coefficient)

C_s: колір дзеркального освітлення

r: вектор відбитого світла

p: коефіцієнт блиску (shininess factor)

Формула для розрахунку вектора відбитого світла 4.1 має такий вигляд:

$$r = 2n * (n, l) – l, \quad (4.1)$$

r: вектор відбитого світла

n: нормаль поверхні

l: вектор напрямку джерела світла

Розбиття формули:

Формула 4.2 демонструє формулу для розрахунку фонового освітлення.

Фонове освітлення (ambient lighting):

$$k_a * I_a * C \quad 4.2$$

k_a: коефіцієнт відбиття фонового освітлення

I_a: інтенсивність фонового освітлення

C: колір поверхні об'єкта

Формула 4.3 демонструє формулу для розрахунку дифузного освітлення.

Дифузне освітлення (diffuse lighting):

$$I_l * C * \max(0, (n, l)) \quad 4.3$$

I_l: інтенсивність джерела світла

C: колір поверхні об'єкта

n: нормаль поверхні

l: вектор напрямку джерела світла

Формула 4.4 демонструє формулу для розрахунку дзеркального освітлення.

Дзеркальне освітлення (specular lighting):

$$k_s * C_s * I_l - \max(0, (r, v))^p \quad 4.4$$

k_s: Коефіцієнт дзеркального відображення

C_s: Колір дзеркального освітлення

I_l: Інтенсивність джерела світла

r: Вектор відбитого світла

v: Вектор напрямку огляду (напрямок камери)

p: Коефіцієнт блиску

Слід зазначити, що деяки частини розрахунку компонент могут бути відсутні, наприклад: C_s, I_a.

Шейдер моделі Фонга наведено в ДОДАТОК Б ШЕЙДЕРИ.

Опис моделі Кука-Торранса.

Модель Кука-Торранса [16] була розроблена вченими на ім'я Брусом Куком (Bruce T. Phong) і Дороном Торрансом (Doron Torrance) в 1977 року. Ця модель є моделлю мікрофасетного освітлення і належить до групи фізично обґрунтованих моделей освітлення.

Група фізично обґрунтованих моделей освітлення в комп'ютерній графіці відноситься до моделей, які прагнуть враховувати фізичні властивості світла і матеріалів для досягнення більш реалістичної візуалізації. Вони ґрунтуються на фізичних законах і принципах, пов'язаних із поширенням світла та його взаємодією з поверхнями.

Група фізично обґрунтованих моделей освітлення охоплює моделі, які враховують такі чинники, як відбивна здатність матеріалу, розподіл інтенсивності світла, геометрію поверхні, взаємну взаємодію мікрофасет на поверхні та інші фізичні параметри. Вони прагнуть точніше відтворити реальну поведінку світла і його взаємодію з матеріалами.

Такі моделі дають змогу досягти реалістичніших ефектів освітлення, відбиття, тіней і матеріалів у комп'ютерній графіці та візуалізації. Вони відіграють важливу роль у створенні візуально привабливих і правдоподібних сцен і об'єктів у різних додатках, включно з іграми, фільмами, архітектурним моделюванням і дизайном.

На рисунку 4.18 зображено формулу [17] для розрахунку коліру об’єкту за моделю Кука-Торранса.

$$I = I_l \frac{F(\theta) D_B(\alpha) G}{4(n,l)(n,v)}$$

Рисунок 4.18 – Формула розрахунку коліру об’єкта за моделю Кука-Торранса

Компоненти моделі Кука-Торранса.

Інтенсивність джерела світла (I_l).

Визначає яскравість джерела світла, яке освітлює поверхню. Це зовнішній фактор, що впливає на інтенсивність відбитого світла.

Функція відбивної здатності (F(θ)).

Функція відбивної здатності (Fresnel Reflectance) визначає, який відсоток світла відбивається від поверхні залежно від кута падіння світла (θ). Функція Френеля широко використовується в моделі Кука-Торранса для розрахунку цього коефіцієнта відбиття.

Функція розподілу мікрофасет (D(α)).

Функція розподілу мікрофасет (Microfacet Distribution Function) визначає, як мікрофасети (мікроскопічні елементи поверхні) розподілені по поверхні матеріалу. Вона описує, який відсоток мікрофасет у певному напрямку може відбивати світло. У моделі Кука-Торранса використовуються різні функції розподілу мікрофасет, такі як GGX (Trowbridge-Reitz) і Beckmann.

Геометричний множник (G).

Геометричний множник (Geometric Factor) враховує взаємодію мікрофасет між собою і з векторами напряму світла та огляду. Він враховує геометрію поверхні та взаємне розташування мікрофасет для коректного моделювання відбитого світла. У моделі Кука-Торранса застосовуються різні моделі геометричного множника, як-от модель Шліка (Schlick) і модель Хеїдера (Heidrich).

Скалярні добутки (n, l) і (n, v).

Скалярні добутки між вектором нормалі поверхні (n) і вектором напряму світла (l) (n, l) і вектором напряму огляду (v) (n, v) беруть участь у нормалізації та обчисленні підсумкової інтенсивності відбитого світла.

Далі наведені функція розподілу мікрофасет та геометричний множник, які були застосовані в моделі додатку.

На рисунку 4.19 зображено формулу [18] для розрахунку геометричний множник.

$$G_{Cook-Torrance} = \min\left(1, \frac{2(n,h)(n,v)}{(v,h)}, \frac{2(n,h)(n,l)}{(v,h)}\right)$$

Рисунок 4.19 – Формула розрахунку геометричний множник

G: геометричний множник, який враховує геометрію поверхні та взаємне розташування векторів.

(n, h): скалярний добуток між вектором нормалі (n) і напрямком напіввектора (h). Піввектор (h) являє собою напрямок, отриманий шляхом додавання вектора світла (l) і вектора огляду (v), а потім нормалізації.

(n, v): скалярний добуток між вектором нормалі (n) і напрямком огляду (v). Вектор огляду (v) представляє напрямок від точки спостереження до поверхні.

(n, l): скалярний добуток між вектором нормалі (n) і напрямком світла (l). Вектор світла (l) представляє напрямок від поверхні до джерела світла.

(v, h): скалярний добуток між вектором огляду (v) і напрямком напіввектора (h).

На рисунку 4.20 зображено формулу [19] розподілу мікрофасет.

$$D_{Beckman}(\alpha) = \frac{1}{m^2(n,h)^4}\exp\left(-\frac{1-(n,h)^2}{m^2(n,h)^2}\right)$$

Рисунок 4.20 – Формула розрахунку розподілу мікрофасет

D(α): функція розподілу мікрофасет, що представляє ймовірність існування мікрофасети з нормою (n) і напіввектором (h).

α: кут нахилу мікрофасети щодо нормалі поверхні.

m: параметр шорсткості поверхні, також званий шорсткістю або грубістю матеріалу.

(n, h): скалярний добуток між вектором нормалі (n) і напіввектором (h). Піввектор (h) являє собою напрямок, отриманий шляхом додавання вектора світла (l) і вектора огляду (v), а потім нормалізації.

Існують ще варіанти розрахунку цих компонент але в розроблюваному додатку на диний момент були застосовані саме ці.

5 ТЕСТУВАННЯ ТА РОБОТА З ДОДАТКОМ

5.1 Тестування безпеки вводу даних

Ввод даних виконується тільки при перейменуванні імені камери, при створенні нової камери та при завантаженні моделей.

На рисунку 5.1 зображено результат тестування завантаження моделей. Всі моделі було завантажено без помилок. Однак слід зазначити, що кожна модель потребує пам’ять і при завантаженні більшої кількості моделей може статися помилка (виключення) розподілення пам’яті. Виключення поки не обробляються в додатку.

Рендаринг об’єктів відбувається без моделей освітлення та всі об’єкти за замовченням мають білий колір.

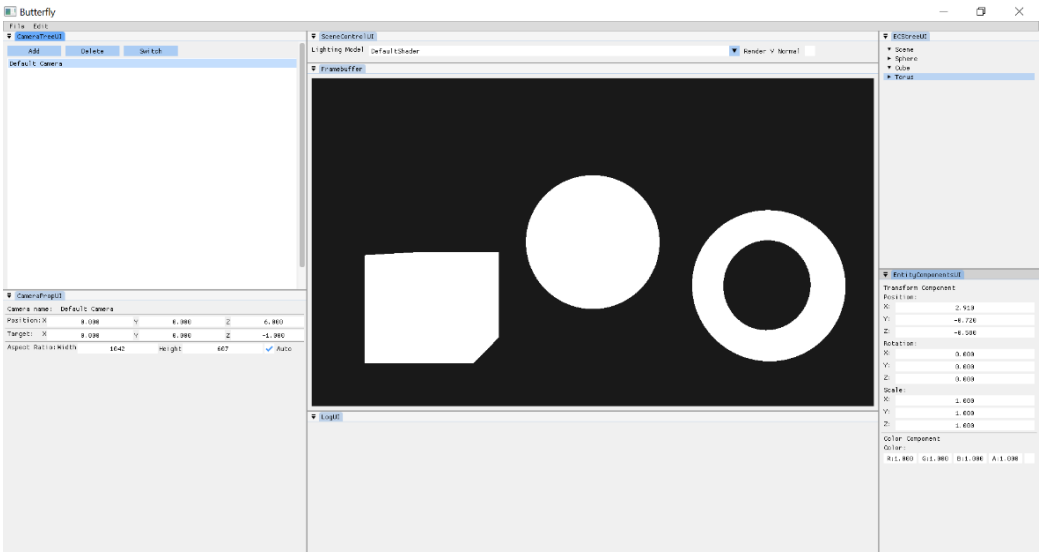

Рисунок 5.1 – Успішне завантаження моделей

На рисунку 5.2 зображено результат тестування вводу імені камери при її створені. Було виведено попередження, що поле порожнє.

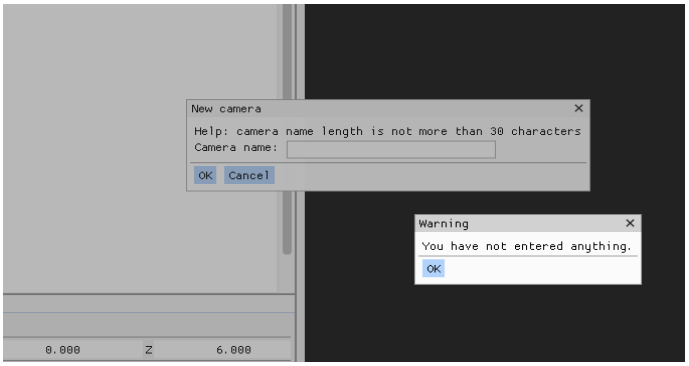

Рисунок 5.2 – Результат тестування вводу порожнього поля

На рисунку 5.3 зображено результат тестування вводу імені камери при її створені. Було виведено попередження, що поле складається тільки з пробілів чи табуляцій.

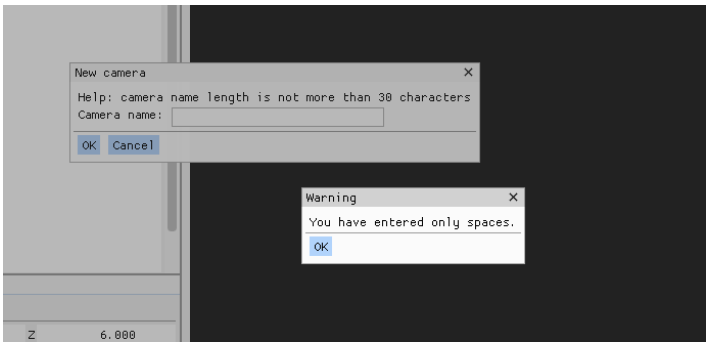

Рисунок 5.3 – Результат тестування вводу тільки пробілів чи табуляцій

На рисунку 5.4 зображено результат тестування вводу імені яке вже існує. Було виведено попередження, що задане ім’я вже використовується.

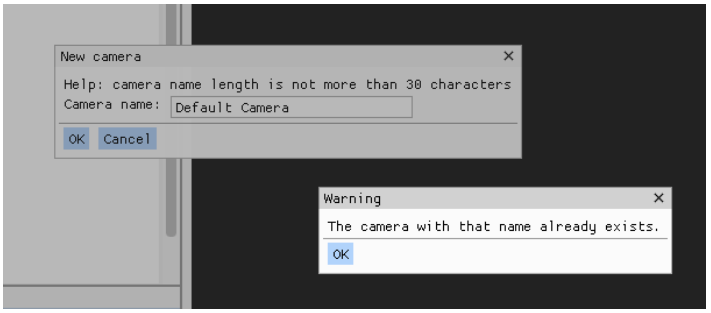

Рисунок 5.4 – Результат тестування вводу імені яке вже існує

Як і проєктувалось дозволяється вводити тільки латинські букви верхнього та нижнього регістрів та цифри.

Аналогічні дії та результати були при тестуванні перейменування камери.

5.2 Тестування роботи інших частин додатку

Тестування роботи додатку проводиться звичайним використанням додатку, як це робив би користувач.

На рисунку 5.5 зображено результат тестування операцій з сутностями. Сутності переміщуються, маштабуються, змінюють орієнтацію та колір. Всі компоненти та їх частини були протестовані по всіх координатах тривимірного простору та моделі RGB.

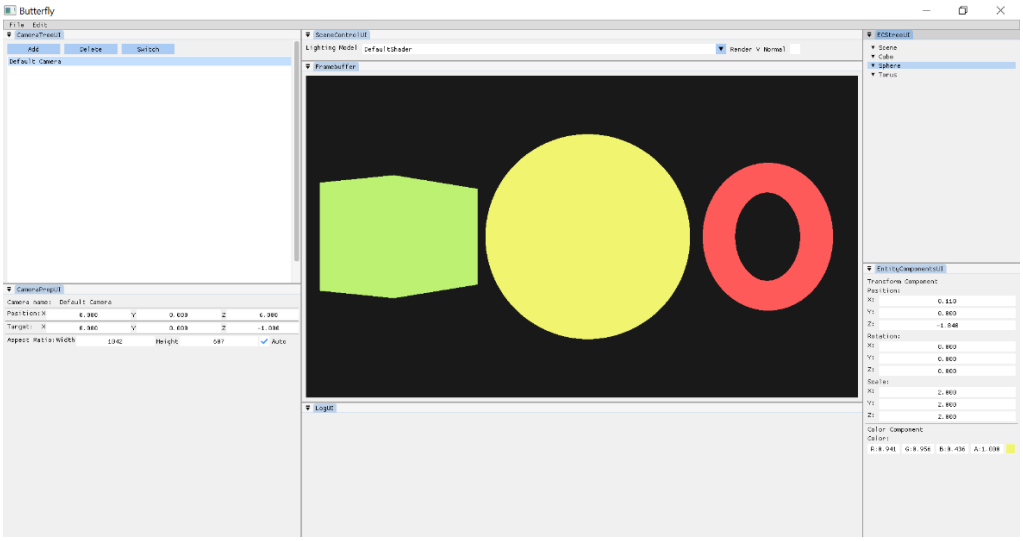

Рисунок 5.5 – Результат тестування операцій з сутностями

На рисунку 5.6 зображено результат тестування рендарингу нормалей. Нормалі до кожної вершини рендаряться. Через велику кількість вершин Сферу та Тор не можливо побачити серед нормалей.

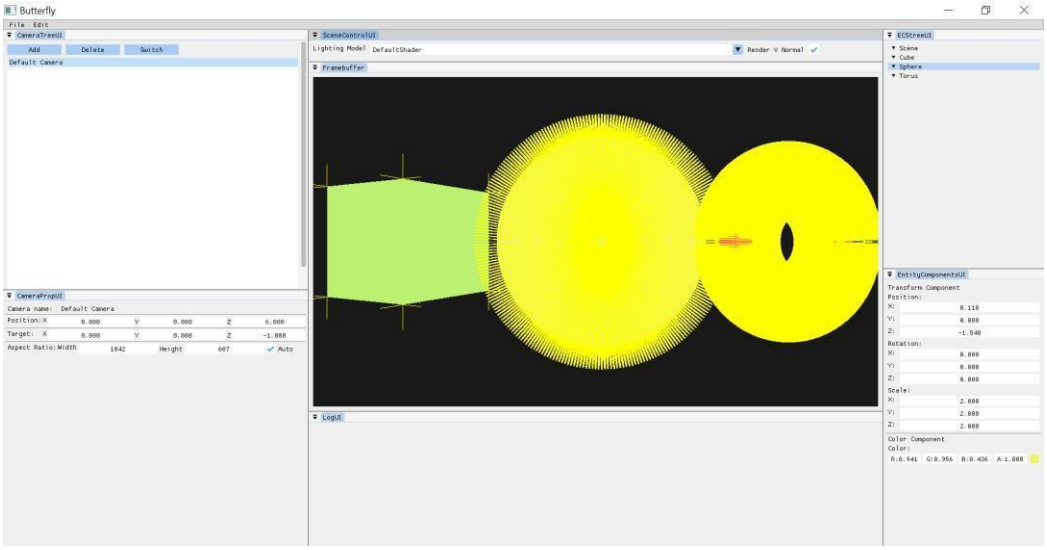

Рисунок 5.6 – Результат тестування рендарингу нормалей

На рисунку 5.7 та 5.8 зображено результат тестування зміни положення та цілі камери. Положення та ціль камери змінюються. Зміна положення та цілі були протестовані по всіх координатах тривимірного простору.

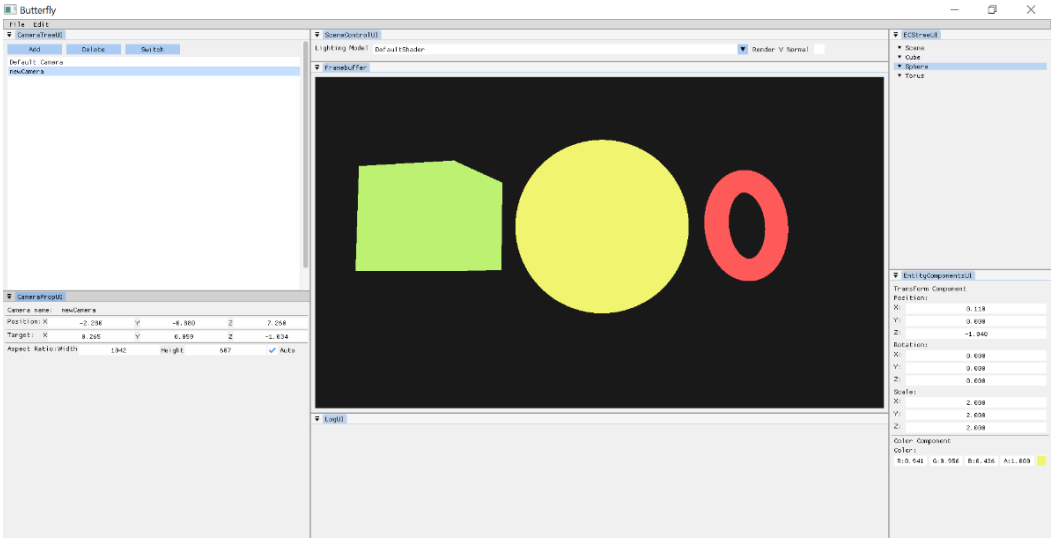

Рисунок 5.7 – Результат тестування зміни положення та цілі камери, newCamera

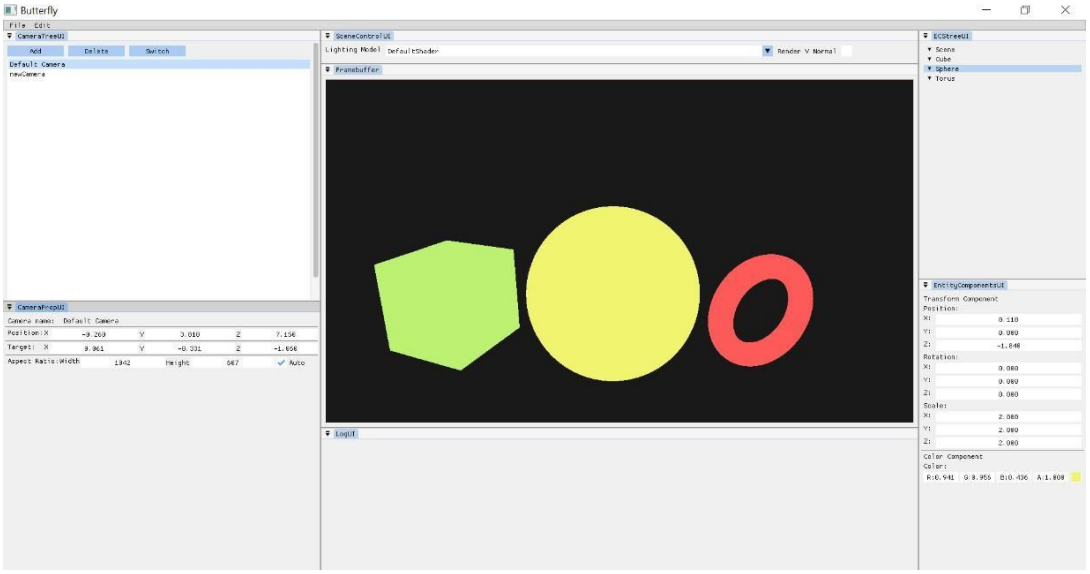

Рисунок 5.8 – Результат тестування зміни положення та цілі камери, Default Camera

На рисунку 5.9 зображено результат тестування зміни aspect ratio. Режим авто, висота та ширина aspect ratio працюють коректно.

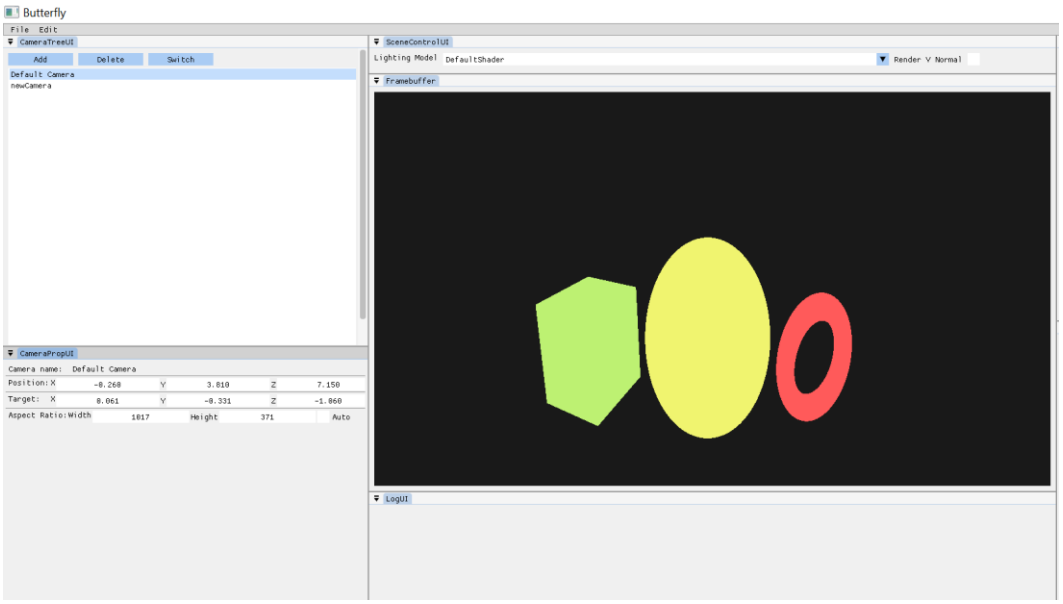

Рисунок 5.9 – Результат тестування зміни aspect ratio

На рисунку 5.10 зображено результат тестування моделі освітлення Фонга. Всі характеристики працюють коректно. Для позначення місцеположення джерела світла використовується текстура лампочки.

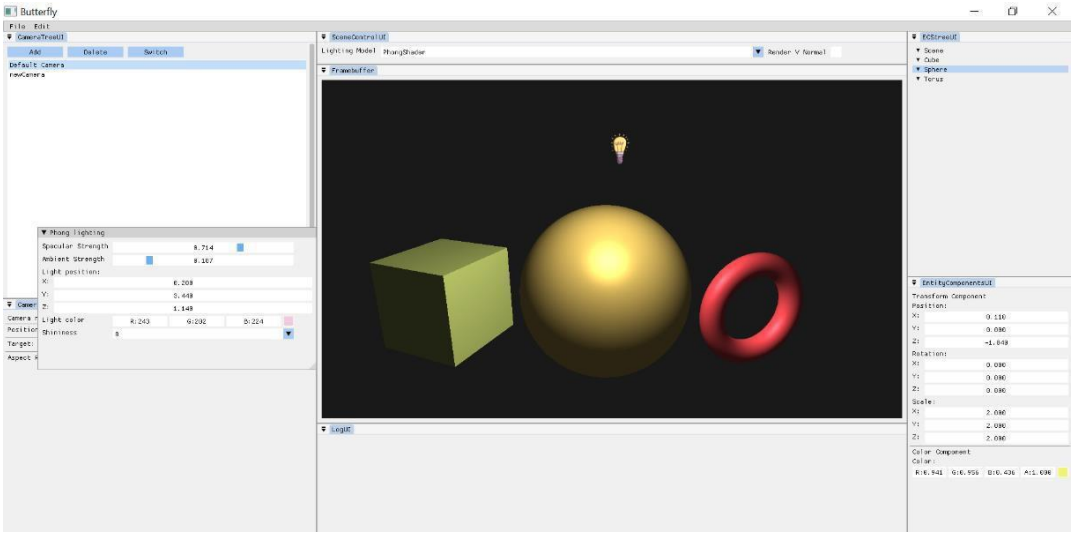

Рисунок 5.10 – Результат тестування моделі освітлення Фонга

На рисунку 5.11 зображено результат тестування моделі освітлення Кука-Торранса. Всі характеристики працюють коректно.

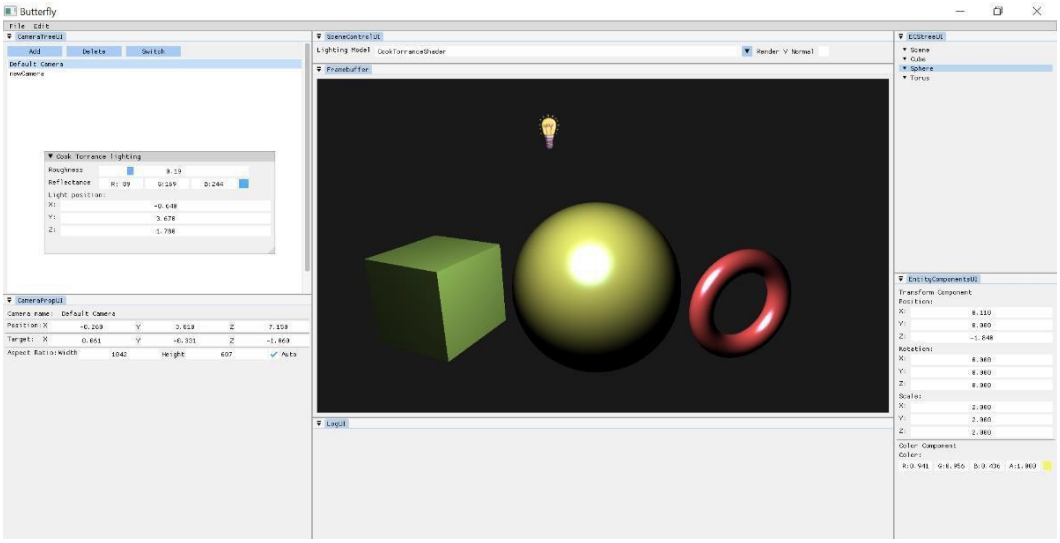

Рисунок 5.11 – Результат тестування моделі освітлення Кука-Торранса

ВИСНОВКИ

В результаті виконання проєкту було успішно розроблено і реалізовано додаток, що базується на мові програмування C++ та використовує такі бібліотеки, як OpenGL, GLFW, Glad, ImGui, GLM, Assimp, stb_image. Головною метою роботи було створення моделей освітлення та користувацького інтерфейсу для взаємодії з ними в учбових чи комерційних цілях.

У процесі розробки додатку було ретельно досліджено та використано можливості наведених бібліотек. OpenGL була використана для рендерингу графічних об'єктів та створення реалістичного зображення. GLFW та Glad були використані для кросплатформенності та роботи з вікнами та контекстами OpenGL. ImGui дозволила створити зручний та інтуїтивно зрозумілий користувацький інтерфейс для взаємодії з додатком. GLM надавала потужні математичні функції для роботи з графікою. Assimp дозволяла завантажувати та обробляти 3D-моделі. Бібліотека stb_image використовувалась для завантаження текстур.

Також було розроблено шейдери на мові програмування GLSL, які відповідають за реалістичне моделювання освітлення в додатку. Це дозволило досягти високої якості графіки та створити реалістичні ефекти світла.

Завдяки розробленому додатку, користувач має можливість взаємодіяти з 3D-сценою та налаштовувати параметри освітлення. Інтуїтивний інтерфейс дозволяє легко маніпулювати об'єктами та налаштовувати їх властивості. Моделі освітлення, реалізовані за допомогою шейдерів GLSL, додають реалізму та візуальної привабливості до сцени.

Також були покращені навички в роботі з C++ та ін. бібліотеками використаними для створення додатку.

Але були допущені і помилки при проєктуванні та конструюванні додатку які нажаль стали помітні наприкінці розробки. Зокрема неправильно було реалізовано патерн DI. Також деякі частини додатку могли бути спроєктовані

краще а деякі структури даних STL можна оптимізувати написавши власні реалізації, наприклад: замість типу std::string написати власні рядки які базуються на концепції та типі Copy on Write, теж саме можно зробити для масива (чи вектора), також можна оптимізувати хеш таблиці написавши власний тип словник. Але всі ці покращення потребують значних людських ресурсів, як з точки зору опиту так і часу. Також можна перепроєктувати систему камер прибравши обмеження на унікальність імен скориставшись концепцією патерну ECS та можна покращити графічний інтерфейс вікна завантаження моделей.

Є ще декілька частин додатку, які можна було б розробити краще але загалом, розробка цього додатку на основі мови програмування C++ та бібліотек OpenGL, GLFW, Glad, ImGui, GLM, Assimp, stb_image, а також розробка шейдерів на мові програмування GLSL, була успішно завершена. Додаток забезпечує зручну взаємодію з 3D-сценою, реалістичну модель освітлення та високу якість графіки. Результати дипломної роботи відповідають поставленим цілям та можуть бути використані в подальших дослідженнях та розробках у галузі комп'ютерної графіки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Effective Modern C++ (1st edition) (2014) Scott Meyers
2. C++ Programming Language (4st edition) (2013) Bjarne Stroustrup
3. Tour of C++ (3st edition) (2022) Bjarne Stroustrup
4. C++ Crash Course: A Fast-Paced Introduction (1st edition) (2019) Josh Lospinoso
5. Optimized C++: Proven Techniques for Heightened Performance (1st edition) (2016) Kurt Guntheroth
6. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V (9st edition) (2016) John Kessenich more
7. OpenGL SuperBible: Comprehensive Tutorial and Reference (7th edition) (2015) Graham Sellers more
8. Computer Graphics: Principles and Practice (3rd edition) (2013) John Hughes more
9. Real-Time Rendering (4th edition) (2018) Tomas Akenine-Mo¨ller more
10. Physically Based Rendering: From Theory to Implementation (3rd edition) (2016) Matt Pharr more
11. Fundamentals of Computer Graphics (4th edition) (2015) Steve Marschner
12. Computer Graphics from Scratch: A Programmer's Introduction to 3D Rendering (1th edition) (2021) Gabriel Gambetta
13. Foundations of Game Engine Development, Volume 2: Rendering (1th edition) (2021) Eric Lengyel
14. Foundations of Game Engine Development, Volume 1: Mathematics (1th edition) (2016) Eric Lengyel
15. GPU Pro 4: Advanced Rendering Techniques (1th edition) (2013) Wolfgang Engel

16. GPU Pro 5: Advanced Rendering Techniques (1th edition) (2014) Wolfgang Engel

17. GPU Pro 6: Advanced Rendering Techniques (1th edition) (2015) Wolfgang Engel

18. GPU Pro 7: Advanced Rendering Techniques (1th edition) (2016) Wolfgang Engel

19. GPU Zen: Advanced Rendering Techniques (1th edition) (2017) Wolfgang Engel

20. Game Engine Gems 3 (1th edition) (2016) Eric Lengyel

ДОДАТОК А

КОД

Функція main

```
int main()
{
	Application appInstance;
	appInstance.run();
	return 0;
}
```

Декларація класа Application

```
class Application
{
public:
	Application();
	~Application();

	void run();

private:

	class OpenGL
	{
	public:
		OpenGL(uint32_t verMajor, uint32_t verMinor, uint32_t profile, const char *ptrGlslVersion);

		std::string mGlslVersion;
		uint32_t mVerMajor;
		uint32_t mVerMinor;
		uint32_t mProfile;
	};

	void  initOpenGL();
	void initWindow();
	void initUI();

	void onEvent(GLFWevent &event);
	void onWindowEventClose(GLFWevent &event);
	void onWindowEventResize(GLFWevent &event);

	bool mRunning;
	std::unique_ptr<OpenGL> mOpenGLinfo;
	std::shared_ptr<Window::WindowProperties> mWinProp;

	std::shared_ptr<Window> mWindow;
	std::unique_ptr<UI> mUI;   // Main UI

	std::vector<std::unique_ptr<UIwindow>> UIs;

	// Scene
	std::shared_ptr<CameraManager> mCamManager;
	std::shared_ptr<Framebuffer> mViewport;
	std::shared_ptr<ShaderManager> mShaderManager;
	std::shared_ptr<LightingModelManager> mLightingModelManager;
	std::shared_ptr<Scene> mScene;
	std::shared_ptr<EntityManager> mEntityManager;
```

```
        std::shared_ptr<SceneSettings> mSettings;
};
```

Декларація класа Window

```
class Window
{
public:
        class WindowProperties
        {
        public:
                winSize   mWidth;
                winSize  mHeight;
                std::string mTitle;
        };

        class WindowHints
        {
        public:
                WindowHints();
                void setVerMajor(uint32_t versionMajor);
                void setVerMinor(uint32_t versionMinor);
                void setOpenGLprofile(uint32_t openGLprofile);

                uint32_t getVerMajor() const;
                uint32_t getVerMinor() const;
                uint32_t getOpenGLprofile() const;
        private:
                uint32_t   mVersionMajor;
                uint32_t   mVersionMinor;
                uint32_t mOpenGLprofile;
        };



        using callback = std::function<void(GLFWevent&)>;

        Window(std::shared_ptr<WindowProperties> wp, std::unique_ptr<WindowHints> winHints);
        ~Window();

        void setCallback(callback&& cbFunc);
        void initCallbacks();
        void update();
        void pollEvents();

        void maximizeWindow();

        void updateSize();

        void hide();
        void show();

        friend class UI;
private:
        void initWindowHints(std::unique_ptr<WindowHints> winHints);
        void initWindow(winSize width, winSize height, const char* ptrTitle);
        void initUserData(std::shared_ptr<WindowProperties> wp);

        class UserData
```

```
	{
	public:
		callback mCbFunc;
		std::shared_ptr<WindowProperties> mWP;
	};



	GLFWwindow* mPtrWindow;
	UserData* mPtrWinUserData;
};
```

Декларація класа Framebuffer

```
class Framebuffer {
public:
	Framebuffer(uint32_t width, uint32_t height);

	~Framebuffer();

	void bind();

	void unbind();

	GLuint   getTexture();

	uint32_t  getWidth();

	uint32_t getHeight();

	void resize(int newWidth, int newHeight);


private:
	void initialize();

	uint32_t mWidth;
	uint32_t mHeight;
	GLuint mFramebuffer;
	GLuint mTexture;
	GLuint mDepthBuffer;
};
```

Декларація класа GLFWevent

```
class GLFWevent
{
public:
	enum class Object
	{
		Window = 1
		// ...
	};
```

```
enum class Action
{
```

```
        // Window
        Close = 1, Resize
        // ...
    };

    virtual ~GLFWevent() = default;
    virtual Object getObject() = 0;
    virtual Action getAction() = 0;
};
```

Декларація класа WindowEventClose

```
class WindowEventClose : public GLFWevent
{
public:
    virtual ~WindowEventClose() override;
    virtual Object getObject() override;
    virtual Action getAction() override;
    static Action getStaticAction();
};
```

Декларація класа WindowEventResize

```
class WindowEventResize : public GLFWevent
{
public:
    WindowEventResize(winSize newWidth, winSize newHeight);
    virtual ~WindowEventResize() override;
    virtual Object getObject() override;
    virtual Action getAction() override;
    static Action getStaticAction();

    winSize getWidth() const;
    winSize getHeight() const;

private:
    winSize mWidth;
    winSize mHeight;
};
```

Декларація класа EventDispatcher

```
class EventDispatcher
{
public:
    EventDispatcher(GLFWevent& event);

    template<typename T, typename F>
    bool dispatch(const F& func);
```

```
private:
        GLFWevent &mEvent;
};
```

Декларація класа LightingModel

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

        virtual void setModelMatrix(const std::string& name, const glm::mat4& model) = 0;
        virtual void setViewMatrix(const std::string& name, const glm::mat4& view) = 0;
        virtual void setProjectionMatrix(const std::string& name, const glm::mat4& projection) = 0;
        virtual void setObjectColor(const std::string& name, const glm::vec4& color) = 0;

        virtual const std::string &getShaderName() = 0;

        virtual void renderMesh() = 0;

        virtual void setAllChanged() = 0;
        virtual void use() = 0;
};
```

Декларація класа DefaultLightingModel

```
class DefaultLightingModel : public LightingModel
{
public:
        DefaultLightingModel(std::shared_ptr<Shader> shader);

        void setModelMatrix(const std::string& name, const glm::mat4& model) override;
        void setViewMatrix(const std::string& name, const glm::mat4& view) override;
        void setProjectionMatrix(const std::string& name, const glm::mat4& projection) override;
        void setObjectColor(const std::string& name, const glm::vec4& color) override;

        const std::string &getShaderName() override;

        void renderMesh() override;

        void setAllChanged() override;
        void use() override;
private:
        std::shared_ptr<Shader> mShader;
};
```

Декларація класа PhongLightingModel

```
class PhongLightingModel : public LightingModel
{
public:
```

```
        PhongLightingModel(std::shared_ptr<Shader> shader);

        void setModelMatrix(const std::string& name, const glm::mat4& model) override;
        void setViewMatrix(const std::string& name, const glm::mat4& view) override;
        void setProjectionMatrix(const std::string& name, const glm::mat4& projection) override;
        void setObjectColor(const std::string& name, const glm::vec4& color) override;

        const std::string& getShaderName() override;

        void setAllChanged() override;
        void use() override;

        void setCameraPosition(const glm::vec3 &cameraPosition);
        void setLightPosition(const glm::vec3 &lightPosition);
        void setLightColor(const glm::vec3 &lightColor);
        void setAmbientStrength(float ambientStrength);
        void setSpecularStrength(float specularStrength);
        void setShininess(float shininess);

        void setupMesh(std::shared_ptr<Shader> shader, std::shared_ptr<ShaderLayout> layout, bool show);
        void setTexture2D(const char* ptrName, int32_t v0);
        void createTexture2D(const char* ptrTexturePath);
        void renderMesh() override;

        glm::vec3 getCameraPosition();
        glm::vec3 getLightPosition();
        glm::vec3 getLightColor();
        float  getAmbientStrength();
        float  getSpecularStrength();
        float getShininess();
private:
        std::shared_ptr<Shader> mShader;

        // Settings
        glm::vec3 mCameraPosition;
        glm::vec3 mLightPosition;
        glm::vec3 mLightColor;
        float  mAmbientStrength;
        float  mSpecularStrength;
        float mShininess;

        std::unique_ptr<LightMesh>  mMesh;

        glm::mat4 mModel;
        glm::mat4 mView;
        glm::mat4 mProjection;
        bool mMeshRender;
};
```

Декларація класа CookTorranceLightingModel

```
class CookTorranceLightingModel : public LightingModel
{
public:
        CookTorranceLightingModel(std::shared_ptr<Shader> shader);

        void setModelMatrix(const std::string& name, const glm::mat4& model) override;
        void setViewMatrix(const std::string& name, const glm::mat4& view) override;
```

```
        void setProjectionMatrix(const std::string& name, const glm::mat4& projection) override;
        void setObjectColor(const std::string& name, const glm::vec4& color) override;

        const std::string& getShaderName() override;

        void renderMesh() override;

        void setTexture2D(const char* ptrName, int32_t v0);
        void createTexture2D(const char* ptrTexturePath);
        void setupMesh(std::shared_ptr<Shader> shader, std::shared_ptr<ShaderLayout> layout, bool show);

        void setAllChanged() override;
        void use() override;

        void setCameraPosition(const glm::vec3& cameraPosition);
        void setLightPosition(const glm::vec3& lightPosition);
        void setRoughness(float roughness);
        void setReflectance(const glm::vec4 &reflectance);

        glm::vec3 getCameraPosition();
        glm::vec3 getLightPosition();
        float getRoughness();
        glm::vec4 getReflectance();
private:
        std::shared_ptr<Shader>  mShader;

        // Settings
        glm::vec3 mCameraPosition;
        glm::vec3 mLightPosition;

        float mRoughness;
        glm::vec4 mReflectance;

        std::unique_ptr<LightMesh> mMesh;

        glm::mat4 mModel;
        glm::mat4 mView;
        glm::mat4 mProjection;
        bool mMeshRender;
};
```

Декларація класа Scene

```
class Scene
{
public:
        Scene(std::shared_ptr<EntityManager> entityManager, std::shared_ptr<LightingModelManager>
lightModelManager, std::shared_ptr<CameraManager> cameraManager, std::shared_ptr<ShaderManager> shaderManager,
std::shared_ptr<SceneSettings> settings);

        void createEntity(std::shared_ptr<Entity> entity);

        void deleteEntity(std::shared_ptr<Entity> entity);

        void render();

private:
        std::shared_ptr<EntityManager> mEntityManager;
```

```
        std::shared_ptr<LightingModelManager> mLightModelManager;
        std::shared_ptr<CameraManager> mCameraManager;
        std::shared_ptr<ShaderManager> mShaderManager;
        std::shared_ptr<SceneSettings> mSettings;
};
```

Декларація класа SceneSettings

```
class SceneSettings
{
public:
        SceneSettings();
        void setStatusRenderVertexNormal(bool status);
        bool getStatusRenderVertexNormal();
private:
        bool mRenderVertexNormals;
};
```

Декларація класа UI

```
class UI
{
public:
        UI(std::shared_ptr<Window> window);
        ~UI();
        void frameBeg();
        void frameEnd();

        void setForOpenGL(const std::string &glslVersion);

private:
        void initUI();
        void initStyle();

        std::shared_ptr<Window> mWindow;
};
```

Декларація класа UIwindow

```
class UIwindow
{
public:
        virtual ~UIwindow() = default;
        virtual void render() = 0;
};
```

Декларація класа CameraTreeUI

```
class CameraTreeUI : public UIwindow
{
public:
	CameraTreeUI(const std::shared_ptr<CameraManager>& camManager, const std::shared_ptr<Framebuffer>&
frameBuffProp);  // Default
	~CameraTreeUI() override;

	void render() override;
private:
	enum class WarningType
	{
		No = 1,
		WarningCamMax,
		WarningCamMin,
		WarningEmptyString,
		WarningOnlySpaces,
		WarningCameraNameExists
	};

	enum class DialogType
	{
		No = 1,
		DialogCameraNewName,
		DialogNewCamera
	};

	void renderMenu();
	void renderList();
	void renderContextMenu(bool isSelected);
	void renderWarning();
	void renderDialogWarning();
	void renderDialog();
	void renderPopupMsg(const char *ptrType, const char *ptrMsg, bool &show);
	void renderCameraNewNameDialog(bool &show);
	void renderNewCameraDialog(bool &show);

	void createBuffer(uint32_t size);
	void destroyBuffer();

	bool checkBufferNotEmpty();
	bool containsOnlySpaces(const char *ptrString);

	std::shared_ptr<CameraManager> mCamManager;
	std::shared_ptr<Framebuffer> mFrameBuffProp;
	bool isWarning;
	bool isDialogWarning;
	WarningType warnType;
	bool isDialog;
	DialogType dialType;
	char *mPtrBuffer;
	uint32_t mBufferSize;
};
```

Декларація класа CameraPropUI

```
class CameraPropUI : public UIwindow
{
public:
```

```
        CameraPropUI(const std::shared_ptr<CameraManager> &camManager, std::shared_ptr<Framebuffer> viewport);
        // Default camera
        void render();
private:
        void renderCameraName();
        void renderThreeDrag(const char *ptrName, glm::vec3 &component, const char *ptrCompLabelX, const char
*ptrCompLabelY, const char *ptrCompLabelZ, float step);
        void renderAspectRatio();

        std::shared_ptr<CameraManager> mCamManager;
        std::shared_ptr<Framebuffer> mViewport;

        float mPositionStep;
        float mTargetStep;

        bool mIsDisabled;
};
```

Декларація класа MainMenuUI

```
class MainMenuUI : public UIwindow
{
public:
        MainMenuUI(std::shared_ptr<EntityManager> entityManager, std::shared_ptr<ShaderManager> shaderManager);
        ~MainMenuUI() override;
        void render() override;

private:
        std::shared_ptr<EntityManager> mEntityManager;
        std::shared_ptr<ShaderManager> mShaderManager;
};
```

Декларація класа EntityComponentsUI

```
class EntityComponentsUI : public UIwindow
{
public:
        EntityComponentsUI(std::shared_ptr<EntityManager> entityManager);
        ~EntityComponentsUI() override;
        void render() override;
private:
        std::shared_ptr<EntityManager> mEntityManager;
};
```

Декларація класа ECStreeUI

```
class ECStreeUI : public UIwindow
{
public:
        ECStreeUI(std::shared_ptr<EntityManager> entityManager);
```

```
        ~ECStreeUI() override;
        void render() override;
private:
        std::shared_ptr<EntityManager> mEntityManager;
        std::shared_ptr<Entity> root;
};
```

Декларація класа LogUI

```
class LogUI : public UIwindow
{
public:
        ~LogUI() override;
        void render() override;
};
```

Декларація класа SceneControlUI

```
class SceneControlUI : public UIwindow
{
public:
        SceneControlUI(std::shared_ptr<LightingModelManager> lightingManager, std::shared_ptr<SceneSettings>
settings, std::shared_ptr<CameraManager> cameraManager);
        ~SceneControlUI() override;
        void render() override;

        void renderSettingsUI();
        void renderPhongUI();
        void renderCookTorranceUI();
private:
        std::shared_ptr<LightingModelManager> mLightingManager;
        std::shared_ptr<SceneSettings> mSettings;
        std::shared_ptr<CameraManager> mCameraManager;

        bool mRenderModel;
        LightingModelManager::LightingModelName mModelType;
};
```

Декларація класа ViewportUI

```
class ViewportUI : public UIwindow
{
public:
        ViewportUI(std::shared_ptr<Framebuffer> viewBuffer, std::shared_ptr<CameraManager> cameraManager,
std::shared_ptr<ShaderManager> shader, std::shared_ptr<Scene> scene);
        ~ViewportUI() override;
        void render() override;

private:
        std::shared_ptr<Framebuffer> mViewBuffer;
```

```
std::shared_ptr<CameraManager> mCameraManager; ImVec2
mPreviousClientAreaSize;
std::shared_ptr<ShaderManager> mShaderManager;
std::shared_ptr<Scene> mScene;
}
```

Декларація класа CameraManager

```
class CameraManager
{
public:
        using count = decltype(std::declval<std::unordered_map<std::string, std::shared_ptr<Camera>>>().size());
        CameraManager(const std::shared_ptr<Camera> camera);          // Default
        bool addCamera(const std::shared_ptr<Camera> camera);
        bool deleteSelectedCamera();
        bool renameSelectedCamera(const char *ptrNewName);
        std::shared_ptr<Camera> &getCamera(const std::string &key);
        count getCount();
        std::unordered_map <std::string, std::shared_ptr<Camera>>::iterator getIterator(const std::string &key);
        bool findByCameraName(const char *ptrName);
        bool checkMax();
        bool checkMin();
        std::shared_ptr<Camera> &getCurrentCamera();
        std::shared_ptr<Camera> &getSelectedCamera();
        std::unordered_map <std::string, std::shared_ptr<Camera>>::iterator getSelectedCameraIter();
        void setCurrentCamera(std::shared_ptr<Camera> current);
        void setSelectedCamera(std::shared_ptr<Camera> selected);
        void changeCurrentSelected();
        std::unordered_map <std::string, std::shared_ptr<Camera>>::iterator begin();
        std::unordered_map <std::string, std::shared_ptr<Camera>>::iterator end();
private:
        static std::unordered_map<std::string, std::shared_ptr<Camera>> mCameras;
        std::shared_ptr<Camera> mCurrentCamera;
        std::shared_ptr<Camera> mSelectedCamera;
        std::unordered_map<std::string,  std::shared_ptr<Camera>>::iterator  mSelectedIter;
}
```

Декларація класа EntityManager

```
class EntityManager
{
private:
        std::vector<std::shared_ptr<Entity>> mEntities;
        std::shared_ptr <Entity> mSelectedEntity;
public:
        void addEntities(const std::vector<std::shared_ptr<Entity>>& entities);
        void setSelected(std::shared_ptr <Entity> selectedEntity);
        std::shared_ptr<Entity> getSelected();
        void addEntity(std::shared_ptr<Entity> entities);
        void deleteEntity(std::shared_ptr<Entity> entity);
        std::vector<std::shared_ptr<Entity>>::iterator begin();
        std::vector<std::shared_ptr<Entity>>::iterator end();
        std::reverse_iterator<std::vector<std::shared_ptr<Entity>>::iterator> rbegin();
        std::reverse_iterator<std::vector<std::shared_ptr<Entity>>::iterator> rend();
}
```

Декларація класа LightingModelManager

```
class LightingModelManager
{
public:
	enum class LightingModelName
	{
		Default = 0,
		Phong,
		CookTorrance
		// ...
	};
	void addModel(LightingModelName name, std::shared_ptr<LightingModel> model);
	// removeModel(...
	void changeCurrentModel(LightingModelName name);
	std::shared_ptr<LightingModel>& getCurrentModel();
	std::unordered_map<LightingModelName, std::shared_ptr<LightingModel>>::iterator begin();
	std::unordered_map<LightingModelName, std::shared_ptr<LightingModel>>::iterator end();
private:
	std::unordered_map<LightingModelName, std::shared_ptr<LightingModel>> mModels;
	std::shared_ptr<LightingModel> mCurrentModel;
};
```

Декларація класа ElementBuffer

```
class ElementBuffer
{
public:
	ElementBuffer();
	~ElementBuffer();
	ElementBuffer(const ElementBuffer& obj) = delete;
	ElementBuffer& operator=(const ElementBuffer& obj) = delete;
	void create(const std::vector<uint32_t>& indices);
	void bind();
	void unbind();
private:
	uint32_t mID;
};
```

Декларація класа VertexBuffer

```
class VertexBuffer
{
public:
	VertexBuffer();
	~VertexBuffer();
	VertexBuffer(const VertexBuffer& obj) = delete;
	VertexBuffer& operator=(const VertexBuffer& obj) = delete;
	void create(const std::vector<float> &vertices, std::shared_ptr<Layout> layout);
	std::shared_ptr<Layout> getLayout();
	void bind();
	void unbind();
private:
	std::shared_ptr<Layout> mLayout;
	uint32_t mID;
};
```

Декларація класа VertexArray

```
class VertexArray
{
public:
	VertexArray();
	~VertexArray();
	VertexArray(const VertexArray& obj) = delete;
	VertexArray& operator=(const VertexArray& obj) = delete;

	void create(std::initializer_list<std::shared_ptr<VertexBuffer>> vertexBuffs,
		std::shared_ptr<ElementBuffer> ptrElementBuff);
	void bind();
	void unbind();
private:
	std::vector<std::shared_ptr<VertexBuffer>> mVertexBuffs;
	std::shared_ptr<ElementBuffer> mElementBuff;
	uint32_t mID;
};
```

Декларація класа Texture2D

```
class Texture2D
{
public:
	Texture2D();
	~Texture2D();
	Texture2D(const Texture2D &obj) = delete;
	Texture2D& operator=(const Texture2D &obj) = delete;
	void setTextureUniform(uint32_t shaderProgram, const char* ptrName, int32_t v0);
	void create(const uint8_t *ptrData, uint32_t width, uint32_t height, uint32_t channels);
	void bind();
	void unbind();
private:
	uint32_t mID;
};
```

Декларація класа Layout

```
class Layout
{
public:
	Layout(uint32_t index, uint32_t size, uint32_t type, bool normalized, uint32_t stride, uint32_t offset);
	~Layout() = default;
	Layout(const Layout& obj);
	Layout& operator=(const Layout& obj);
	uint32_t mIndex;
	uint32_t mSize;
	uint32_t mType;
	bool mNormalized;
	uint32_t mStride;
	uint32_t mOffset;
};
```

Декларація класа ShaderLayout

```
class ShaderLayout
{
public:
	enum class LayoutName
```

```
	{
		Position = 0,
		Color,
		Normal,
		Texture
	};
	void addLayout(ShaderLayout::LayoutName name, std::shared_ptr<Layout> layout);
	std::shared_ptr<Layout> getLayout(ShaderLayout::LayoutName name);
private:
	std::unordered_map<ShaderLayout::LayoutName,  std::shared_ptr<Layout>>  mShaderLayout;
};
```

Декларація класа Shader

```
class Shader
{
public:
	Shader();
	~Shader();
	Shader(const Shader &obj) = delete;
	Shader &operator=(const Shader &obj) = delete;
	void create(const char *ptrVertShCode, const char *ptrGeomShCode, const char *ptrFragShCode);
	const std::string& getName();
	uint32_t getProgram();
	void setName(const std::string &name);
	void use();
	void setMat4(const std::string &varName, const glm::mat4 &value);
	void setVec4(const std::string &varName, const glm::vec4 &value);
	void setVec3(const std::string& varName, const glm::vec3& value);
	void setFloat(const std::string& varName, float value);
private:
	enum class Process
	{
		COMPILING,
		LINKING
	};
	std::uint32_t createShader(const char *ptrCode, std::uint32_t type);
	std::uint32_t createProgram(std::uint32_t vertSh, std::uint32_t geomSh, std::uint32_t fragSh);
	void checkStatus(std::uint32_t obj, Process type);
	std::uint32_t mID;
	std::string mName;
};
```

Декларація класа Camera

```
class Camera
{
public:
	Camera(const std::string  &name, const glm::vec3 &position, const glm::vec3 &target, float fov, uint32_t
aspectRatioWidth, uint32_t aspectRatioHeight, float nearClip, float farClip, const glm::vec3 &upDown = glm::vec3(0.0f, 1.0f,
0.0f));
	~Camera() = default;
	Camera(const Camera &obj) = delete;
	Camera &operator=(const Camera &obj) = delete;
	const glm::mat4 &getProjectionMatrix() const;
	const glm::mat4 &getViewMatrix() const;
	const std::string &getName() const;
	glm::vec3 getPosition() const;
	glm::vec3 getTarget() const;
	void setName(const std::string &name);
	void setPosition(const glm::vec3 &position);
	void setTarget(const glm::vec3 &target);
```

```
        void setAspectRatio(uint32_t aspectRatioWidth, uint32_t aspectRatioHeight);
        uint32_t getAspectRatioWidth();
        uint32_t getAspectRatioHeight();
        static std::shared_ptr<Camera> makeSharedCamera(const std::string &cameraName, uint32_t aspectRatioWidth, uint32_t
aspectRatioHeight);
        static uint32_t getCameraNameMaxLength();
        void setAutoAspectRatio(bool status);
        bool getAutoAspectRatio();
private:
        void updateViewMatrix();
        void updateProjectionMatrix();
        std::string mName;
        glm::vec3 mPosition;
        glm::vec3 mTarget;
        glm::vec3 mUpDown;
        float mFov;
        uint32_t  mAspectRatioWidth;
        uint32_t mAspectRatioHeight;
        float mAspectRatio;
        float mNearClip;
        float mFarClip;
        glm::mat4 mViewMatrix;
        glm::mat4 mProjectionMatrix;
        bool mAutoAspectRatio;
};
```

Декларація класа FileReader

```
class FileReader
{
public:
        FileReader() = default;
        ~FileReader() = default;
        FileReader(const FileReader &obj) = default;
        FileReader& operator=(const FileReader &obj) = default;
        std::string read(const char *ptrFilePath);
        Texture2D *readTexture(const char *ptrFilePath, bool flip);
        std::vector<std::shared_ptr<Entity>> loadModel(const char *ptrFilePath);
private:
};
```

Декларація класа Timer

```
class Timer
{
private:
        typedef std::chrono::high_resolution_clock                clock;
        typedef std::chrono::duration<float, std::ratio<1, 1>> second;
public:
        Timer();
        ~Timer() = default;
        Timer(const Timer& obj) = default;
        Timer& operator=(const Timer& obj) = default;
        void reset();
        float elapsed() const;
private:
        std::chrono::time_point<clock>  mBeg;
};
```

Декларація класа LightMesh

```
class LightMesh
{
public:
	LightMesh();
	~LightMesh();
	void setMesh(std::shared_ptr<ShaderLayout> shaderLayout);
	void setTexture2D(const char* ptrName, int32_t v0);
	void createTexture2D(const char* ptrFilePath);
	void setShader(std::shared_ptr<Shader> shader);
	void useShader();
	void render();
	void setModelMatrix(const std::string& name, const glm::mat4& modelMatrix);
	void setViewMatrix(const std::string& name, const glm::mat4& modelMatrix);
	void setProjectionMatrix(const std::string& name, const glm::mat4& modelMatrix);
private:
	VertexArray mVertArray;
	std::shared_ptr<Shader> mShader;
	Texture2D* mPtrTexture2D;
	std::vector<uint32_t> mIndices;
	std::vector<float> mVertices;
	std::vector<float> mTexCoord;
};
```

Декларація класа ShaderManager

```
class ShaderManager
{
public:
	void addShader(const std::string& name, std::shared_ptr<Shader> shader, std::shared_ptr <ShaderLayout> layout);
	std::shared_ptr<Shader> getShader(const std::string &name);
	std::shared_ptr<ShaderLayout> getShaderLayout(const std::string &name);
	void setCurrentShader(const std::string &name);
	std::shared_ptr<Shader> getCurrentShader();
	std::shared_ptr<ShaderLayout> getCurrentShaderLayout();
	std::unordered_map<std::string, std::pair<std::shared_ptr<Shader>, std::shared_ptr<ShaderLayout>>>::iterator begin();
	std::unordered_map<std::string, std::pair<std::shared_ptr<Shader>, std::shared_ptr<ShaderLayout>>>::iterator end();
private:
	std::unordered_map<std::string, std::pair<std::shared_ptr<Shader>, std::shared_ptr<ShaderLayout>>> mShaders;
	std::pair<std::shared_ptr<Shader>, std::shared_ptr<ShaderLayout>> currentShader;
};
```

Декларація класа EntityOperations

```
class EntityOperations
{
public:
	void setup(std::vector<std::shared_ptr<Entity>>& entities, std::shared_ptr<ShaderLayout> shaderLayout);
};
```

ДОДАТОК Б ШЕЙДЕРИ

Default Shader

Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNorm;

uniform vec4 objectColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out vec4 vertColor;
out vec3 vertNormal;

void main()
{
	gl_Position = projection * view * model * vec4(aPos.x, aPos.y, aPos.z, 1.0f);
	mat3 normalMatrix = mat3(transpose(inverse(view * model)));
	vertNormal = normalize(normalMatrix * aNorm);
	vertColor = objectColor;
}
```

Geometry Shader

```
#version 330 core

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

in vec4 vertColor[];
in vec3 vertNormal[];
out vec4 geomColor;
out vec3 geomNormal;

void main()
{
   for(int i = 0; i < gl_in.length(); ++i)
   {
      gl_Position = gl_in[i].gl_Position;
      geomColor = vertColor[i];
      geomNormal = vertNormal[i];
      EmitVertex();
   }
   EndPrimitive();
}
```

Fragment Shader

```
#version 330 core

in vec4 geomColor;
in vec3 geomNormal;
out vec4 fragColor;

void main()
{
        fragColor = geomColor;
}
```

Light Mesh Shader

Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 vertTexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
   gl_Position = projection * view * model * vec4(aPos.x, aPos.y, aPos.z, 1.0f);
   vertTexCoord = aTexCoord;
}
```

Geometry Shader

```
#version 330 core

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

in vec2 vertTexCoord[];

out vec2 geomTexCoord;

void main()
{
   for(int i = 0; i < gl_in.length(); ++i)
   {
      gl_Position = gl_in[i].gl_Position;
      geomTexCoord = vertTexCoord[i];
      EmitVertex();
```

```
    }
    EndPrimitive();
}
```

Fragment Shader

```
#version 330 core
out vec4 FragColor;

in vec2 geomTexCoord;

uniform sampler2D texture1;

void main()
{
    FragColor = texture(texture1, geomTexCoord);
}
```

Normal Shader

Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out VS_OUT {
    vec3 normal;
} vs_out;

uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = view * model * vec4(aPos, 1.0);
    mat3 normalMatrix = mat3(transpose(inverse(view * model)));
    vs_out.normal = normalize(vec3(vec4(normalMatrix * aNormal, 0.0)));
}
```

Geometry Shader

```
#version 330 core
layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

in VS_OUT {
```

```
    vec3 normal;
} gs_in[];

const float MAGNITUDE = 0.4;

uniform mat4 projection;

void GenerateLine(int index)
{
    gl_Position = projection * gl_in[index].gl_Position;
    EmitVertex();
    gl_Position = projection * (gl_in[index].gl_Position + vec4(gs_in[index].normal, 0.0) * MAGNITUDE);
    EmitVertex();
    EndPrimitive();
}

void main()
{
    GenerateLine(0); // нормаль первой вершины
    GenerateLine(1); // нормаль второй вершины
    GenerateLine(2); // нормаль третьей вершины
}
```

Fragment Shader

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0, 1.0, 0.0, 1.0);
}
```

Phong Shader

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 vertexPosition;
layout (location = 1) in vec3 vertexNormal;

out vec3 vertFragPos;
out vec3 vertNormal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
```

```
{
   vertFragPos = vec3(model * vec4(vertexPosition, 1.0));
   vertNormal = normalize(mat3(transpose(inverse(model))) * vertexNormal);

   gl_Position = projection * view * vec4(vertFragPos, 1.0);
}
```

Geometry Shader

```
#version 330 core

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

in vec3 vertFragPos[];
in vec3 vertNormal[];

out vec3 geomFragPos;
out vec3 geomNormal;

void main()
{
   for(int i = 0; i < gl_in.length(); ++i)
   {
      gl_Position = gl_in[i].gl_Position;
      geomFragPos = vertFragPos[i];
      geomNormal = vertNormal[i];
      EmitVertex();
   }
   EndPrimitive();
}
```

Fragment Shader

```
#version 330 core

in vec3 geomFragPos;
in vec3 geomNormal;

out vec4 fragColor;

uniform vec4 objectColor;      // Цвет объекта

uniform vec3 lightPosition;   // Позиция источника света
uniform vec3 lightColor;      // Цвет источника света
uniform float ambientStrength; // Коэффициент окружающего освещения
uniform float specularStrength; // Коэффициент силы бликов
uniform float shininess;

uniform vec3 cameraPosition;

void main()
{
```

```
    vec3 viewDirection = normalize(cameraPosition - geomFragPos);
    vec3 lightDirection = normalize(lightPosition - geomFragPos);

    // ambient
    vec3 ambient = ambientStrength * lightColor;

    // diffuse
    float diff = max(dot(geomNormal, lightDirection), 0.0);
    vec3 diffuse = diff * lightColor;

    // specular
    vec3 reflectionDirection = reflect(-lightDirection, geomNormal);
    vec3 specular = specularStrength  * pow(max(dot(viewDirection, reflectionDirection), 0.0), shininess) * lightColor;

    // color
    vec3 finalColor = (ambient + diffuse + specular) * objectColor.xyz;

    fragColor = vec4(finalColor, 1.0);
}
```

Cook-Torrance Shader

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 vertexPosition;
layout (location = 1) in vec3 vertexNormal;

out vec3 vertFragPos;
out vec3 vertNormal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    vertFragPos = vec3(model * vec4(vertexPosition, 1.0));
    vertNormal = mat3(transpose(inverse(model))) * vertexNormal;

    gl_Position = projection * view * vec4(vertFragPos, 1.0);
}
```

Geometry Shader

```
#version 330 core

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
```

```
in vec3 vertFragPos[];
in vec3 vertNormal[];

out vec3 geomFragPos;
out vec3 geomNormal;

void main()
{
   for(int i = 0; i < gl_in.length(); ++i)
   {
      gl_Position = gl_in[i].gl_Position;
      geomFragPos = vertFragPos[i];
      geomNormal = vertNormal[i];
      EmitVertex();
   }
   EndPrimitive();
}
```

Fragment Shader

```
#version 330 core

in vec3 geomFragPos;
in vec3 geomNormal;

out vec4 fragColor;

uniform vec4 objectColor;
uniform vec3 lightPosition;
uniform vec3 cameraPosition;

uniform float roughness;
uniform vec4 reflectance;

void main( void )
{
            vec3 v = normalize(cameraPosition - geomFragPos);   // view dir
            vec3 l = normalize(lightPosition - geomFragPos);          // light dir
            vec3 n = normalize(geomNormal);   // normal

            vec4  r0   = reflectance; // vec4 (1.0, 0.92, 0.23, 1.0);
            vec3  l2   = normalize ( l );
            vec3  n2   = normalize ( n );
            vec3  v2   = normalize ( v );
            vec3  h    = normalize ( l2 + v2 );
            float nh   = dot (n2, h);
            float nv   = dot (n2, v2);
            float nl   = dot (n2, l2);

                                                // Compute Beckman
            float r2   = roughness * roughness;
            float nh2  = nh * nh;
            float ex   = -(1.0 - nh2)/(nh2 * r2);
            float d    = pow(2.7182818284, ex ) / (r2*nh2*nh2);

            vec4  f    = mix(vec4(pow(1.0 - nv, 5.0)), vec4(1.0), r0);
```

```
                                        // Default G
        float x    = 2.0 * nh / dot(v2, h);
        float g    = min(1.0, min (x * nl, x * nv));

                                        // Resulting color
        vec4  ct   = f*(0.25 * d * g / nv);
        float diff = max(nl, 0.0);
        float ks   = 0.5;

        fragColor = diff * objectColor + ks * ct;
}
```

ДОДАТОК В
ЗАУВАЖЕННЯ НОРМОКОНТРОЛЕРА

Таблиця В.1 – Зауваження нормоконтролера

Сторінка	Зміст	Зауваження
Зміст, розділ 1		Зайві крапки в нумерації підрозділів, пунктів…
ПЗ		Нещільне заповнення листів
С.66-67		Невірні переліки
ПЗ		Не усі формули зроблено в редакторі формул, змінні в тексті та формулі різні
Розділ 4		Зовсім погано оформлений текст ПЗ – багато помилок
Перелік використаних джерел		Перелік джерел оформлено не за ДСТУ, посилання на усі джерела

8242305268840808

На електронний документ накладено: 1 (Один) підписи чи печатки:
На момент друку копії, підписи чи печатки перевірено:
Програмний комплекс: eSign v. 2.3.0;
Засіб кваліфікованого електронного підпису чи печатки: ІІТ Користувач ЦСК-1
Експертний висновок: №05/02/02-1424 від 05.04.2016;
Цілісність даних: не порушена;

Підпис № 1 (реквізити підписувача та дані сертифіката)
Підписувач: КОЩЕЙ РУСЛАН РОМАНОВИЧ 3583800092;
Належність до Юридичної особи: ФІЗИЧНА ОСОБА;
Код юридичної особи в ЄДР: 3583800092;
Серійний номер кваліфікованого сертифіката: 248197DDFAB977E504000000C10DF70067D02F04;
Видавник кваліфікованого сертифіката: АЦСК АТ КБ «ПРИВАТБАНК»;
Тип носія особистого ключа: Незахищений;
Тип підпису: Удосконалений;
Сертифікат: Кваліфікований;
Час та дата підпису: 21:16 08.06.2023;

Чинний на момент підпису. Підтверджено позначкою часу для підпису від АЦСК (кваліфікованого надавача електронних довірчих послуг)