МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет Інформаційних технологій
Кафедра Інформатики і прикладного програмного забезпечення
Спеціальність Інженерія програмного забезпечення
Форма навчання Денна

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Рудського Владислава Віталійовича
(прізвище, ім'я, по батькові здобувача)

на тему «Розробка програмного забезпечення візуалізації 3D-об'єктів дизайну інтер'єру
(повна назва теми)

за матеріалами праць провідних спеціалістів у галузі комп'ютерної графіки та візуалізації
(повна назва бази дослідження)

науковий керівник д.т.н., професор ________ Зеленський О.С.
(наук. ступінь, вчене звання) (підпис) (прізвище, ініціали)

Робота допущена до захисту в ЕК
Протокол засідання кафедри
від 11.06.2025 р. № 12
Завідувач кафедри ________________
(підпис)

д.т.н., професор Зеленський О.С.
Наук. ступень, вчене звання *Ініціали, прізвище*

Кривий Ріг – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

«ЗАТВЕРДЖУЮ»
Завідувач кафедри ____________ Зеленський О.С.
(підпис) (Прізвище, ініціали)
« 11 » червня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ

1. Тема роботи «Розробка програмного забезпечення візуалізації 3D-об’єктів дизайну інтер’єру»

Керівник роботи д.т.н., професор Зеленський О.С.
затверджені наказом закладу вищої освіти від «04» квітня 2025 р. № 222-ст

2. Строк подання здобувачем роботи до «09» червня 2025 р.

3. Зміст кваліфікаційної роботи, об’єкт, предмет та мета дослідження:

Розділ 1. Аналіз технологій та форматів даних для розробки 3D-візуалізатора

Розділ 2. Проектування архітектури 3D-візуалізатора

Розділ 3. Реалізація 3D-візуалізатора та його функціонал

Розділ 4. Тестування, оптимізація та аналіз результатів

Об’єкт дослідження: процеси візуалізації тривимірних об'єктів та їхня поведінка у просторі, а також механізми завантаження та відображення моделей

Предмет дослідження: засоби та алгоритми програмної реалізації 3D-візуалізатора

Мета кваліфікаційної роботи: розробка 3D-візуалізатора архітектурних моделей з використанням OpenGL та С++

5. Дата видачі завдання «04» квітня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів МДР	Строк виконання етапів роботи	Відмітка керівника про виконання етапів (дата, підпис)
1	Підготовка розділу 1	04.04.2025-13.04.2025	
2	Підготовка розділу 2	14.04.2025-26.04.2025	
3.	Підготовка розділу 3	27.04.2025-15.05.2025	
4	Підготовка розділу 4	16.05.2025-08.06.2025	
5	Реєстрація завершеної кваліфікаційної роботи	09.06.2025	Реєстраційний №___ «09»червня 2025 р.
6	Отримання відгуку від наукового керівника	10.06.2025	
7	Подання кваліфікаційної роботи на перегляд завідувачу кафедри	11.06.2025	
8	Отримання зовнішньої рецензії	12.06.2025	
9	Попередній захист кваліфікаційної роботи на кафедрі	13.06.2025	
10	Підготовка до захисту в ЕК	16.06.2025-21.06.2025	

Завдання підготував науковий керівник ______________ __Зеленський О.С.___
(підпис) (прізвище та ініціали)

Завдання одержав ______________ ___Рудський В.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

на кваліфікаційну бакалаврську роботу

«Розробка програмного забезпечення візуалізації 3D-об’єктів дизайну інтер’єру»

Рудського Владислава Віталійовича

Кваліфікаційна бакалаврська робота на здобуття освітньо-кваліфікаційного рівня бакалавра зі спеціальності 121 «Інженерія програмного забезпечення» – Державний університет економіки і технологій – Кривий Ріг, 2025.

У бакалаврській дипломній роботі розроблено програмне забезпечення 3D-візуалізатора архітектурних моделей з використанням OpenGL та C++. Додаток забезпечує ефективне завантаження та інтерактивне відображення тривимірних об'єктів у поширених форматах DAE, FBX та OBJ. Реалізовано функціонал відображення моделей з текстурами та матеріалами, а також інтерактивне управління камерою для детального огляду сцени. Проведено тестування, яке підтвердило високу продуктивність та стабільність візуалізатора.

Ключові слова: 3D-ВІЗУАЛІЗАТОР, OPENGL, C++, АРХІТЕКТУРНІ МОДЕЛІ, ASSIMP, РЕНДЕРИНГ.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

3D-ВІЗУАЛІЗАТОР	Програма, яка дозволяє виводити тривимірні моделі на екран
OpenGL	Кросплатформенний API для рендерингу 2D і 3D графіки.
АРХІТЕКТУРНІ МОДЕЛІ	3D-моделі будівель або конструкцій, що використовуються для візуалізації проєктів
ASSIMP	Бібліотека для імпорту різних форматів 3D-моделей OBJ, FBX, DAE
РЕНДЕРИНГ	Процес створення зображення з 3D-моделі за допомогою комп’ютера

ЗМІСТ

ВСТУП …………………………………………………………………………8

РОЗДІЛ 1 ……………………………………………………………………...10

АНАЛІЗ ТЕХНОЛОГІЙ ТА ФОРМАТІВ ДАНИХ ДЛЯ РОЗРОБКИ 3D-ВІЗУАЛІЗАТОРА

1.1. Загальні принципи 3D-графіки та конвеєр рендерингу …………………10

1.2. Огляд та вибір графічного API: OpenGL …………………………………12

1.3. Аналіз форматів 3D-моделей: DAE, FBX, OBJ …………………..……14

1.4. Огляд бібліотек для імпорту 3D-моделей: Assimp ………………………18

1.5. Огляд та аналіз існуючих 3D-візуалізаторів (аналогів) …………………21

РОЗДІЛ 2 ……………………………………………………………………...24

ПРОЕКТУВАННЯ АРХІТЕКТУРИ 3D-ВІЗУАЛІЗАТОРА …………………

2.1. Вимоги до програмного забезпечення ……………………………………24

2.2. Загальна архітектура системи ……………………………………………..26

2.3. Проектування класів та структур даних …………………………………..29

2.4. Проектування взаємодії користувача ……………………………………..36

РОЗДІЛ 3 ……………………………………………………………………...39

РЕАЛІЗАЦІЯ 3D-ВІЗУАЛІЗАТОРА ТА ЙОГО ФУНКЦІОНАЛ

3.1. Налаштування середовища розробки та необхідні бібліотеки ……….....39

3.2. Ініціалізація OpenGL контексту та вікна …………………………………41

3.3. Реалізація класів Shader, Camera …………………………………………..42

3.4. Завантаження 3D-моделей за допомогою Assimp ………………………..45

3.5. Обробка та відображення текстур ………………………………………...48

3.6. Налаштування VBO, VAO, EBO для рендерингу ………………………..50

3.7. Цикл рендерингу та взаємодія з користувачем …………………………52

РОЗДІЛ 4 ……………………………………………………………………...56

ТЕСТУВАННЯ, ОПТИМІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1. Методика тестування програмного забезпечення ………………………..56

4.2. Результати тестування та їх аналіз ………………………………………..58

4.3. Оптимізація продуктивності візуалізатора ………………………………62

4.4. Можливості подальшого розвитку та розширення функціоналу ………63

ВИСНОВКИ ……………………………………………………………………..66

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ………………………………………68

ДОДАТКИ ………………………………………………………………………70

ВСТУП

Сучасний світ стрімко розвивається у напрямку цифровізації та візуалізації даних, що зумовлює постійне зростання попиту на ефективні інструменти для роботи з тривимірною графікою. 3D-візуалізатори є невід'ємною частиною багатьох галузей, від архітектури та промислового дизайну до інженерії, кінематографа та ігрової індустрії, дозволяючи представити складні об'єкти та ідеї у наочній та інтерактивній формі. Особливе значення вони набувають в архітектурі, де візуалізація є критично важливою для представлення проектів, оцінки дизайну та взаємодії із замовниками. Актуальність обраної теми обумовлена постійною необхідністю вдосконалення та розробки гнучких програмних засобів, здатних ефективно відображати 3D-моделі архітектурних об'єктів. В умовах, коли існуючі спеціалізовані редактори або візуалізатори можуть не підтримувати всі поширені формати або мати обмежений функціонал для простого перегляду та демонстрації, розробка універсального візуалізатора, що підтримує різноманітні формати 3D-моделей (DAE, FBX, OBJ), вирішує проблему сумісності даних та забезпечує гнучкість у роботі з різними джерелами. Це дозволяє не тільки переглядати моделі, створені в різних програмних середовищах, але й забезпечує інструмент для швидкого та ефективного представлення архітектурних концепцій.

Метою дослідження є розробка комплексного 3D-візуалізатора архітектурних моделей з використанням бібліотеки OpenGL та мови програмування C++, який забезпечує не лише ефективне завантаження та відображення тривимірних об'єктів у поширених форматах, але й надає користувачеві інтуїтивно зрозумілі інструменти для інтерактивного огляду та детального вивчення моделей. Це включає створення програмного продукту, здатного адекватно інтерпретувати дані з файлів 3D-моделей, коректно застосовувати текстури та матеріали, а також забезпечувати плавне управління камерою для всебічного візуального аналізу.

Для досягнення поставленої мети було визначено наступні **завдання**

дослідження:

- Провести аналіз існуючих стандартів та бібліотек для роботи з 3D-графікою, обґрунтувати вибір OpenGL та C++ як основних інструментів розробки.
- Дослідити особливості структури та вмісту файлів 3D-моделей популярних форматів (DAE, FBX, OBJ) для забезпечення їх коректного імпорту.
- Розробити архітектуру програмного забезпечення 3D-візуалізатора, що забезпечує модульність та масштабованість.
- Реалізувати функціонал завантаження та відображення 3D-моделей з підтримкою текстур та матеріалів.
- Імплементувати механізми інтерактивного управління камерою по осях X, Y, Z, дозволяючи детальний огляд моделі з різних ракурсів.
- Провести тестування розробленого візуалізатора на прикладі архітектурних моделей для підтвердження його працездатності та ефективності.
- Виконати оптимізацію коду для забезпечення високої продуктивності та стабільності роботи візуалізатора.

Об'єктом дослідження виступають теоретичні та практичні аспекти процесів візуалізації тривимірних об'єктів, а також методи та алгоритми обробки та представлення просторових даних. Це охоплює принципи побудови 3D-сцен, взаємодії світла та матеріалів, а також динаміку взаємодії з віртуальним простором. Дослідження також зосереджено на адаптації цих принципів для архітектурних моделей, що характеризуються специфічними вимогами до точності, деталізації та масштабування.

Предметом дослідження є засоби та алгоритми програмної реалізації 3D-візуалізатора архітектурних моделей, включаючи вибір та застосування програмних інтерфейсів (API) для графіки, розробку ефективних структур даних для зберігання та обробки моделей, а також методи оптимізації завантаження, відображення та інтерактивної взаємодії з тривимірними об'єктами. Особлива увага приділяється спрощенню процесу завантаження та представлення нескладних архітектурних об'єктів, роблячи візуалізатор доступним інструментом для широкого кола користувачів.

РОЗДІЛ 1
АНАЛІЗ ТЕХНОЛОГІЙ ТА ФОРМАТІВ ДАНИХ ДЛЯ РОЗРОБКИ 3D-ВІЗУАЛІЗАТОРА

1.1. Загальні принципи 3D-графіки та конвеєр рендерингу

Тривимірна графіка є однією з ключових галузей комп'ютерних наук, що займається створенням, маніпуляцією та відображенням об'єктів у тривимірному просторі. Основна мета 3D-графіки полягає у симуляції реального світу або створенні повністю віртуальних сцен, які виглядають реалістично для людського ока. Цей процес починається з математичного опису об'єктів у віртуальному просторі та закінчується їхнім відображенням на двовимірному екрані[10, с 38].

Фундаментальними складовими будь-якої 3D-сцени є:

- Моделі (Model): Геометричне представлення об'єктів. Зазвичай складаються з вершин (vertices), ребер (edges) та граней (faces), які формують сітку (mesh). Вершини містять координати у просторі, а також додаткові атрибути, такі як нормалі (для освітлення) та текстурні координати (для накладання зображень).
- Текстури (Textures): Двовимірні зображення, які накладаються на поверхню 3D-моделей для надання їм кольору, деталізації та реалістичності.
- Матеріали (Materials): Описують, як поверхня об'єкта взаємодіє зі світлом. Вони визначають колір (дифузний, спекулярний, ембієнтний), блиск, прозорість та інші властивості.
- Освітлення (Lighting): Джерела світла, що імітують освітлення сцени. Різні типи джерел (точкові, спрямовані, прожектори) впливають на вигляд об'єктів, створюючи тіні та відблиски.
- Камера (Camera): Визначає точку зору, з якої сцена відображається. Параметри камери включають її положення, напрямок погляду, кут огляду (field of view) та площини відсікання (clipping planes).

Процес перетворення тривимірної сцени на двовимірне зображення нази-

вається рендерингом. Він виконується через послідовність етапів, відому як конвеєр рендерингу (rendering pipeline). Цей конвеєр може відрізнятися залежно від конкретного графічного API (наприклад, OpenGL, DirectX, Vulkan), але його основні фази залишаються незмінними:

1. Формування геометрії (Geometry Processing): На цьому етапі відбувається підготовка даних вершин. Включає такі операції:

- Перетворення вершин (Vertex Transformation): Вершини моделі, спочатку задані в локальній системі координат об'єкта, перетворюються у світові координати (де вони розташовані в сцені), потім у видові координати (відносно камери) і, нарешті, у проекційні координати (де визначається їхнє положення на "плоскости" перед проекцією). Це виконується за допомогою матричних перетворень (модельної, видової та проекційної матриць).
- Обчислення освітлення (Lighting Calculations): Для кожної вершини або пікселя (залежно від моделі освітлення) розраховується вплив світла на її поверхню з урахуванням нормалей та властивостей матеріалів[17, с 74].
- Відсікання (Clipping): Видалення геометрії, яка знаходиться поза межами видимої області (фрустума камери), щоб уникнути непотрібних обчислень.

2. Растеризація (Rasterization): Перетворення тривимірних примітивів (трикутників, ліній, точок), які пройшли етап формування геометрії, на фрагменти (кандидати на пікселі). На цьому етапі також відбувається:

- Інтерполяція (Interpolation): Атрибути вершин (колір, текстурні координати, нормалі) інтерполюються для кожного фрагмента.
- Тестування фрагментів (Fragment Testing): Фрагменти проходять різні тести (глибинний тест, тест трафарета, тест прозорості), щоб визначити, чи повинні вони бути відображені на екрані.

3. Операції над фрагментами (Fragment Operations / Pixel Processing): Це заключний етап, на якому фрагменти перетворюються на кінцеві пікселі зображення:

- Накладання текстур (Texturing): Застосування текстур до фрагментів,

що надає поверхні деталей.

- Фінальні обчислення освітлення/кольору: Можливе додаткове освітлення на рівні пікселів.
- Змішування (Blending): Комбінування кольору поточного фрагмента з кольором пікселя, що вже знаходиться у буфері кадру (використовується для прозорих об'єктів).
- Запис у буфер кадру (Framebuffer Writes): Кінцевий колір фрагмента записується у буфер кадру, який потім відображається на екрані.

Розуміння цих принципів та функціонування конвеєра рендерингу є критично важливим для розробки ефективного 3D-візуалізатора, оскільки це дозволяє оптимізувати обробку даних та ефективно використовувати можливості графічного процесора.

1.2. Огляд та вибір графічного API: OpenGL

Для програмної реалізації 3D-візуалізатора необхідний вибір відповідного графічного програмного інтерфейсу (API – Application Programming Interface), який забезпечує взаємодію програмного забезпечення з апаратним забезпеченням відеокарти. Графічний API є набором функцій, що дозволяють програмісту керувати графічним процесором (GPU) для виконання операцій рендерингу, таких як малювання примітивів, управління текстурами, налаштування освітлення та обробка шейдерів.

Серед найбільш поширених графічних API варто виділити:

- OpenGL (Open Graphics Library): Міжплатформений, багатомовний API для рендерингу 2D та 3D графіки. Історично є одним з найстаріших і найпопулярніших API, що активно розвивається.
- DirectX: Комплект API від Microsoft, що включає Direct3D для 3D-графіки. Призначений виключно для платформи Windows та Xbox.
- Vulkan: Сучасний, низькорівневий API від Khronos Group (розробники OpenGL), що забезпечує прямий контроль над апаратним забезпеченням, надаю-

чи високу продуктивність та гнучкість.

- Metal: Низькорівневий API від Apple, розроблений для iOS, macOS, tvOS та watchOS.

Для розробки 3D-візуалізатора архітектурних моделей було обрано OpenGL. Цей вибір обумовлений рядом ключових переваг, які роблять його оптимальним рішенням для даного проекту:

1. Кросплатформенність: OpenGL є кросплатформенним стандартом, що дозволяє запускати розроблене програмне забезпечення на різних операційних системах (Windows, Linux, macOS) без значних змін у коді. Це забезпечує широку доступність візуалізатора.
2. Гнучкість та контроль: OpenGL надає розробнику високий рівень контролю над графічним конвеєром. Це дозволяє реалізувати специфічні алгоритми рендерингу, оптимізувати роботу з моделями та текстурами, а також точно налаштувати візуалізацію відповідно до потреб архітектурних проектів.
3. Широка підтримка та спільнота: OpenGL має величезну та активну спільноту розробників, велику кількість навчальних матеріалів, бібліотек та інструментів. Це значно спрощує процес розробки та пошук рішень для виникаючих проблем.
4. Можливість роботи з шейдерами: Сучасні версії OpenGL (Core Profile) повністю базуються на шейдерах (GLSL – OpenGL Shading Language), що дозволяє програмувати графічний конвеєр безпосередньо на GPU. Це відкриває широкі можливості для створення реалістичного освітлення, складних матеріалів, тіней та інших візуальних ефектів, що є критично важливим для якісної демонстрації архітектурних моделей.
5. Достатня продуктивність: Для завдань інтерактивного відображення архітектурних моделей, які зазвичай мають помірну складність у порівнянні з ігровими сценами, продуктивності OpenGL більш ніж достатньо. Він дозволяє ефективно використовувати апаратні можливості відеокарти для плавного рендерингу.
6. Інтеграція з C++: OpenGL відмінно інтегрується з мовою програму-

вання C++, що дозволяє створювати високопродуктивні та оптимізовані додатки. Комбінація цих технологій забезпечує максимальну ефективність та контроль над системними ресурсами.

На відміну від DirectX, який є пропрієтарним API Microsoft, OpenGL є відкритим стандартом, що робить його привабливим для проектів, які не прив'язані до однієї платформи. Хоча Vulkan пропонує ще більший контроль та потенційно вищу продуктивність, його вивчення та імплементація є значно складнішими, що може бути надмірним для цілей даної дипломної роботи, де пріоритетом є баланс між функціональністю, продуктивністю та реалістичністю для архітектурних моделей.

Таким чином, вибір OpenGL як основного графічного API для розробки 3D-візуалізатора архітектурних моделей є обґрунтованим рішенням, що дозволяє досягти поставленої мети, поєднуючи гнучкість, кросплатформенність та достатню продуктивність[11, с 35].

1.3. Аналіз форматів 3D-моделей: DAE, FBX, OBJ

Ефективна робота 3D-візуалізатора значною мірою залежить від його здатності коректно інтерпретувати та завантажувати тривимірні моделі, що створюються в різноманітних програмних пакетах. Для забезпечення універсальності та гнучкості, розроблений візуалізатор підтримує імпорт моделей у трьох найбільш поширених та взаємозамінних форматах: Collada (DAE), FBX та Wavefront OBJ. Кожен з цих форматів має свої особливості, переваги та недоліки.

1.3.1. Формат Collada (DAE)

Collada (COLLAborative Design Activity) – це формат файлів обміну 3D-активами, розроблений Khronos Group. Він використовує XML-схему для опису 3D-моделей, що робить його читабельним для людини та дозволяє легко інтегрувати в різні системи.

- Структура: DAE-файл є ієрархічним документом XML, який може містити детальну інформацію про геометрію (вершини, нормалі, текстурні координати, тангенси, бітангенси), анімацію, фізику, матеріали, освітлення, камери та багато іншого. Він підтримує складні сцени з великою кількістю взаємопов'язаних елементів.
- Переваги:
  - Відкритий стандарт: Будучи відкритим стандартом, Collada не залежить від конкретного програмного забезпечення або компанії, що сприяє його широкому поширенню та підтримці.
  - Насиченість даними: Здатний зберігати широкий спектр даних, включаючи складні ієрархії, анімацію, шейдери та фізичні властивості, що робить його ідеальним для обміну складними сценами між різними 3D-редакторами.
  - Читабельність: XML-структура дозволяє легко переглядати та редагувати файл вручну (хоча це рідко практикується для складних моделей).
  - Підтримка тангенсів та бітангенсів: Дуже важлива для коректного відображення нормальних карт (normal maps), що підвищує реалістичність візуалізації.
- Недоліки:
  - Розмір файлу: XML-формат може призводити до великих розмірів файлів порівняно з бінарними форматами, особливо для складних моделей.
  - Складність парсингу: Розбір та обробка XML-структури може бути складнішою та повільнішою порівняно з більш простими або бінарними форматами.
  - Варіативність реалізацій: Через гнучкість стандарту, різні програми можуть генерувати DAE-файли з певними варіаціями, що іноді створює труднощі при імпорті.
- Значення для візуалізатора: Collada є відмінним вибором для архітек-

турних моделей, оскільки він дозволяє зберігати повний набір даних, включаючи ієрархії об'єктів, детальну геометрію та складні матеріали, що є типовим для архітектурних проектів.

1.3.2. Формат FBX

FBX (Filmbox) – це пропрієтарний формат файлів, розроблений компанією Autodesk, що широко використовується для обміну 3D-даними між різними програмними продуктами Autodesk (такими як 3ds Max, Maya, AutoCAD) та іншими 3D-додатками. FBX є одним з найпопулярніших форматів у галузі ігрової розробки та візуалізації.

- Структура: FBX може зберігатися як у бінарному, так і в ASCII-форматі. Бінарний формат є більш компактним та швидшим для читання/запису. FBX підтримує моделі, анімацію, матеріали, текстури, камери, освітлення та скелети.
- Переваги:
  - Широка підтримка: Практично всі великі 3D-редактори та ігрові двигуни підтримують FBX, що робить його де-факто стандартом для обміну даними у професійній індустрії.
  - Комплексність даних: Здатний зберігати дуже багатий набір даних, включаючи складні анімації, морфінг, деформації та кінематику, що часто є невід'ємною частиною архітектурних візуалізацій (наприклад, відкривання дверей, рух ліфтів).
  - Оптимізований для продуктивності: Бінарний формат FBX є компактним та ефективним для швидкого завантаження.
- Недоліки:
  - Пропрієтарність: Будучи пропрієтарним форматом Autodesk, його підтримка та розвиток повністю залежать від однієї компанії, що може становити ризики у довгостроковій перспективі.
  - Складність імплементації: Складність специфікації формату та необхідність використання SDK від Autodesk (або сторонніх бібліотек, як Assimp) роблять пряму реалізацію парсера FBX досить складною.

- Можливі проблеми сумісності: Незважаючи на широку підтримку, іноді можуть виникати невеликі відмінності у інтерпретації даних між різними версіями програмного забезпечення.

- Значення для візуалізатора: FBX є незамінним для інтеграції з професійними архітектурними та дизайнерськими пакетами, такими як 3ds Max та Revit, оскільки він забезпечує максимальну збереженість даних, включаючи складні матеріали та потенційно анімації, що можуть бути використані в подальших розширеннях візуалізатора.

1.3.3. Формат Wavefront OBJ

Wavefront OBJ — це простий, але надзвичайно поширений формат файлів, призначений для обміну тривимірною геометрією. Цей формат був спочатку розроблений компанією Wavefront Technologies для їхньої програмної системи Advanced Visualizer, проте згодом набув статусу універсального стандарту для обміну 3D-моделями [12, с. 15].

- Структура: OBJ-файл є текстовим форматом, що описує геометрію (вершини, текстурні координати, нормалі, грані) за допомогою простого синтаксису. Матеріали для OBJ-моделей зазвичай описуються у окремому файлі .mtl (Material Template Library), на який посилається OBJ-файл.

- Переваги:

  - Простота: Завдяки своїй текстовій структурі та простому синтаксису, OBJ-файли легко читаються, парсяться та навіть редагуються вручну. Це робить його ідеальним для початкового етапу розробки та швидкого тестування.

  - Універсальність: Підтримується майже всіма 3D-редакторами та програмами для роботи з 3D-графікою, що робить його надійним форматом для базового обміну моделями.

  - Малий розмір файлу: Для простих моделей розмір файлу може бути досить невеликим.

- Недоліки:

  - Обмеженість даних: OBJ підтримує лише статичну геометрію, текстурні координати та нормалі. Він не зберігає інформацію про анімацію,

скелети, освітлення, камери чи складні ієрархії об'єктів. Це робить його менш придатним для складних сцен.

- Окремі файли матеріалів: Необхідність використання окремого файлу .mtl для матеріалів може ускладнити управління ресурсами, якщо не дотримуватися правильної організації файлів.
- Відсутність тангенсів/бітангенсів: OBJ-файл не містить інформації про тангенси та бітангенси, які є критично важливими для коректного відображення нормальних карт. Їх доводиться обчислювати програмно під час завантаження.

- Значення для візуалізатора: OBJ є важливим форматом для підтримки, оскільки він забезпечує сумісність з широким спектром джерел простих 3D-моделей. Його простота робить його зручним для швидкого імпорту базових архітектурних елементів та перевірки функціоналу візуалізатора.

Підтримка цих трьох форматів – DAE, FBX, та OBJ – дозволяє 3D-візуалізатору забезпечити високий рівень сумісності з різними джерелами 3D-моделей, від простих об'єктів до складних архітектурних сцен з деталізованими матеріалами, що є ключовим для досягнення поставленої мети дипломної роботи.

1.4. Огляд бібліотек для імпорту 3D-моделей: Assimp

Завантаження тривимірних моделей у програму є критично важливим етапом у розробці будь-якого 3D-візуалізатора. Це включає не лише читання геометрії (вершин, індексів, нормалей, текстурних координат), але й обробку інформації про матеріали, текстури, анімацію (якщо є), ієрархію об'єктів та інші метадані, що містяться у файлах 3D-моделей[20, с 218]. Ручна реалізація парсерів для кожного формату є вкрай трудомісткою та складною задачею, оскільки кожен формат має свою унікальну структуру та специфікацію. Тому, для ефективної роботи з множинними форматами 3D-даних, зазвичай використовуються спеціа-
лізовані бібліотеки.

Серед найбільш відомих та широко використовуваних бібліотек для імпорту 3D-моделей можна виділити:

- Open Asset Import Library (Assimp): Це відкрита (BSD-ліцензія) бібліотека, призначена для імпорту різних форматів 3D-моделей у програму. Вона надає уніфікований інтерфейс для доступу до даних, незалежно від вихідного формату файлу.
- TinyGLTF: Бібліотека для завантаження моделей у форматі glTF (GL Transmission Format), який стає все більш популярним, особливо в контексті веб-візуалізації.
- FBX SDK: Офіційний SDK від Autodesk для роботи з форматом FBX. Надає повний контроль над даними FBX, але є пропрієтарним і може бути складнішим у використанні для простих завдань.
- Custom parsers: Власний код для парсингу конкретного формату. Зазвичай використовується тільки для дуже специфічних завдань або якщо формати вкрай прості.

Для реалізації 3D-візуалізатора архітектурних моделей було обрано Open Asset Import Library (Assimp). Цей вибір є обґрунтованим з кількох причин:

1. Широка підтримка форматів: Assimp підтримує понад 40 різних форматів 3D-моделей, включаючи DAE (Collada), FBX, OBJ, glTF, 3DS, DXF та багато інших. Це забезпечує високу гнучкість і універсальність візуалізатора, дозволяючи йому працювати з моделями, створеними в різних 3D-редакторах. Це критично важливо для цілей даної роботи, оскільки дозволяє безшовно інтегрувати моделі з різних джерел.
2. Уніфікований інтерфейс: Незалежно від вихідного формату, Assimp представляє імпортовані дані у вигляді стандартизованої структури (сцена, вузли, меші, матеріали, текстури, анімації). Це значно спрощує розробку, оскільки програмісту не потрібно писати окремий код для обробки кожного формату; достатньо написати єдиний парсер для внутрішнього представлення Assimp[13, с 20].

3. Автоматична постобробка: Assimp надає потужні можливості постобробки імпортованих сцен, що дозволяє автоматизувати підготовку моделей до рендерингу та значно спростити роботу розробника. Основні прапори постобробки, використані в даній роботі, та їхнє призначення наведені в Таблиці 1.1.

Таблиця 1.1

Прапори постобробки Assimp та їх призначення

Прапор постобробки (aiProcess_)	**Призначення**
Triangulate	Перетворює всі полігони на трикутники, що є обов'язковим для рендерингу в OpenGL та інших API.
GenNormals	Генерує або переобчислює нормалі для вершин, якщо вони відсутні у вихідному файлі або некоректні.
FlipUVs	Перевертає координати текстур по осі Y. Часто необхідно через відмінності у системах координат текстур між різними 3D-редакторами та OpenGL.
JoinIdenticalVertices	Об'єднує ідентичні вершини, що мають однакові позиції, нормалі, текстурні координати тощо, для зменшення кількості вершин та оптимізації.
OptimizeMeshes	Намагається оптимізувати меші для кращої продуктивності рендерингу (наприклад, за рахунок об'єднання дрібних мешів).
CalcTangentSpace	Обчислює вектори тангенсів та бітангенсів для кожної вершини. Ці дані є критично важливими для коректного застосування карт нормалей (normal mapping).
LimitBoneWeights	Обмежує кількість кісток, що впливають на одну вершину. Корисно для анімації.
ValidateDataStructure	Виконує перевірку внутрішньої цілісності даних, імпортованих Assimp, що допомагає виявити проблеми у файлі моделі.

4. Простота інтеграції та використання: Assimp є C++ бібліотекою, що легко інтегрується в проекти на C++ і має зрозумілий API. Його використання дозволяє швидко реалізувати функціонал завантаження моделей, зосередившись на логіці візуалізації, а не на деталях парсингу файлів.

5. Відкритий вихідний код: Будучи бібліотекою з відкритим вихідним кодом, Assimp дає можливість розробнику вивчати його внутрішню роботу, модифікувати його за потреби та використовувати без обмежень у комерційних та некомерційних проектах.

Використання Assimp дозволяє уникнути складнощів, пов'язаних з ручним парсингом бінарних форматів, а також забезпечує високу гнучкість у роботі з різними джерелами 3D-моделей. Це дозволяє зосередити основні зусилля на реалізації графічного конвеєра в OpenGL та інтерактивних функцій візуалізатора, що безпосередньо сприяє досягненню мети дипломної роботи.

1.5. Огляд та аналіз існуючих 3D-візуалізаторів (аналогів)

Ринок програмного забезпечення для 3D-графіки насичений різноманітними інструментами, які пропонують широкий спектр можливостей – від створення моделей та анімації до фотореалістичного рендерингу. Серед них виділяються як комплексні 3D-редактори, що включають функції візуалізації, так і спеціалізовані переглядачі моделей. Аналіз існуючих рішень дозволяє краще зрозуміти контекст даної розробки та визначити її місце серед аналогів.

Розглянемо декілька прикладів поширених 3D-візуалізаторів та редакторів:

- Blender: Це потужний, безкоштовний та відкритий 3D-пакет, що охоплює весь спектр робіт з тривимірною графікою: моделювання, скульптінг, анімація, симуляція, рендеринг та навіть відеомонтаж. Blender має власні вбудовані рендери (Cycles та Eevee), які дозволяють досягати високого ступеня реалізму[21, с 54].
    - Особливості: Величезний функціонал, активна спільнота, підтримка

багатьох форматів через імпорт/експорт.

- Недоліки з точки зору цієї роботи: Незважаючи на свою універсальність, Blender є складним інструментом з високим порогом входу для простих завдань перегляду. Його надлишковий функціонал може бути зайвим, якщо метою є лише демонстрація 3D-моделі без її редагування. Великий обсяг програми та її ресурсні вимоги також можуть бути невиправданими для легкої "оглядової" програми.

- Unity: Це одна з найпопулярніших платформ для розробки ігор та інтерактивного контенту. Unity надає потужний движок для рендерингу в реальному часі, що дозволяє імпортувати 3D-моделі, налаштовувати їх матеріали, освітлення, додавати інтерактивність та створювати повноцінні 3D-сцени[23, с 86].

  - Особливості: Висока продуктивність рендерингу в реальному часі, широкі можливості для інтерактивності, підтримка фізики, анімації, скриптингу.

  - Недоліки з точки зору цієї роботи: Unity – це повноцінний ігровий движок, а не простий візуалізатор. Його використання для звичайної демонстрації архітектурної моделі є надмірним. Розробка та компіляція проекту в Unity потребує значно більше часу та ресурсів, ніж створення легкої програми для перегляду. Кінцевий файл програми, створеної в Unity, також буде значно більшим.

- Онлайн-візуалізатори та спеціалізовані Viewer-додатки (наприклад, Autodesk Viewer, SketchUp Viewer): Це категорія інструментів, спеціально розроблених для перегляду та демонстрації 3D-моделей. Багато з них існують як веб-сервіси, що дозволяють завантажувати моделі в хмару та переглядати їх через браузер, або як самостійні додатки з обмеженим функціоналом.

  - Особливості: Спрощений інтерфейс, фокус на перегляд, часто підтримка специфічних форматів (наприклад, DWG, RVT для архітектурних візуалізаторів).

  - Недоліки з точки зору цієї роботи: Онлайн-версії вимагають доступу до інтернету та завантаження конфіденційних даних на сторонні сервери. Багато

десктопних "Viewer"-додатків можуть бути прив'язані до конкретного виробника програмного забезпечення (наприклад, Autodesk для своїх форматів) або мати обмеження у функціоналі (наприклад, не підтримувати всі необхідні формати або детальні налаштування візуалізації). Також деякі з них є платними.

Висновки до розділу 1

Проведений аналіз показав, що існуючі рішення для роботи з 3D-моделями поділяються на комплексні 3D-редактори з надмірним функціоналом та спеціалізовані візуалізатори, які часто мають обмеження у підтримці форматів або вимагають сторонніх сервісів. Незважаючи на їхні переваги, ці інструменти можуть бути неоптимальними для задачі простої, локальної та гнучкої демонстрації архітектурних моделей з підтримкою ключових форматів.

Розробка власного 3D-візуалізатора на базі OpenGL та C++ дозволяє створити легкий, високопродуктивний та спеціалізований інструмент, який буде зосереджений виключно на завантаженні та інтерактивній візуалізації 3D-моделей (DAE, FBX, OBJ) з текстурами та матеріалами. Вибір OpenGL забезпечує кросплатформенність та повний контроль над графічним конвеєром, а використання бібліотеки Assimp гарантує широку підтримку форматів та ефективну обробку даних. Це дозволить заповнити нішу для візуалізатора, який є простим у використанні, не перевантаженим зайвим функціоналом і при цьому здатним якісно демонструвати архітектурні об'єкти без необхідності встановлення великих програмних пакетів.

РОЗДІЛ 2
ПРОЕКТУВАННЯ АРХІТЕКТУРИ 3D-ВІЗУАЛІЗАТОРА

2.1. Вимоги до програмного забезпечення

Ефективне проектування будь-якої програмної системи починається з чіткого визначення її вимог. Вимоги до програмного забезпечення поділяються на функціональні, які описують, що система повинна робити, та нефункціональні, які визначають, як система повинна працювати (продуктивність, надійність, зручність використання тощо). Для розробки 3D-візуалізатора архітектурних моделей було сформовано наступний перелік вимог.

2.1.1. Функціональні вимоги

Функціональні вимоги визначають основні можливості та дії, які повинен виконувати 3D-візуалізатор.

ФВ1. Завантаження 3D-моделей: Система повинна забезпечувати можливість завантаження тривимірних моделей з файлів у наступних форматах:

1. Collada (.dae)
2. Autodesk FBX (.fbx)
3. Wavefront OBJ (.obj)

ФВ2. Відображення 3D-моделей: Візуалізатор повинен коректно відображати завантажені 3D-моделі на екрані, використовуючи OpenGL. Це включає:

1. Відображення геометрії (вершин, граней).
2. Відображення текстур та їх коректне застосування до відповідних частин моделі.
3. Відображення матеріалів (колір, блиск).
4. Обробку нормалей для коректного освітлення.

ФВ3. Інтерактивне управління камерою: Користувач повинен мати можливість вільно переміщуватися по сцені та оглядати модель з різних ракурсів. Функціонал управління камерою має включати:

1. Переміщення камери вперед/назад, вліво/вправо (наприклад, за допомогою клавіш W, S, A, D).

2. Обертання камери навколо моделі або своєї осі (за допомогою миші) по осях X та Y.

3. Збільшення/зменшення масштабу або наближення/віддалення (за допомогою коліщатка миші).

ФВ4. Освітлення сцени: Система повинна реалізовувати базове освітлення для покращення сприйняття глибини та форми 3D-моделей. Це може включати:

1. Використання спрямованого світла або точкового джерела світла для освітлення сцени.

2. Обробку дифузного та спекулярного компонентів освітлення.

ФВ5. Відображення стану: Візуалізатор повинен надавати базовий візуальний зворотний зв'язок про поточний стан (наприклад, повідомлення про завантаження моделі, помилки).

2.1.2. Нефункціональні вимоги

Нефункціональні вимоги описують якісні характеристики програмного забезпечення, які впливають на його продуктивність, зручність та надійність.

НФВ1. Продуктивність:

1. Візуалізатор повинен забезпечувати плавну інтерактивну роботу (не менше 30 кадрів в секунду) для моделей середньої складності (до 100 000 полігонів) на типових сучасних конфігураціях ПК.

2. Час завантаження моделі не повинен перевищувати 5 секунд для моделей розміром до 50 МБ.

НФВ2. Надійність:

1. Система повинна стабільно працювати при завантаженні коректних 3D-моделей.

2. Повинна бути передбачена обробка помилок при спробі завантаження пошкоджених або непідтримуваних файлів.

НФВ3. Зручність використання (Usability):

1. Інтерфейс візуалізатора має бути інтуїтивно зрозумілим для базового перегляду 3D-моделей.
2. Керування камерою має бути простим та природним.

НФВ4. Розширюваність: Архітектура програмного забезпечення повинна бути спроектована таким чином, щоб забезпечити легкість додавання нового функціоналу (наприклад, підтримка інших форматів, розширені параметри освітлення, анімація) у майбутньому.

НФВ5. Сумісність: Програмне забезпечення повинно бути сумісним з основними операційними системами (наприклад, Windows) та стандартними графічними драйверами OpenGL.

Ці вимоги послугують основою для подальшого проектування архітектури системи та її реалізації, забезпечуючи чітке розуміння очікуваного функціоналу та якісних характеристик кінцевого продукту.

2.2. Загальна архітектура системи

Розробка 3D-візуалізатора архітектурних моделей передбачає створення модульної та логічно структурованої системи, здатної ефективно обробляти графічні дані та взаємодіяти з користувачем. Загальна архітектура програмного забезпечення є високорівневим представленням компонентів системи та зв'язків між ними, що дозволяє забезпечити її масштабованість, гнучкість та легкість підтримки. Архітектура візуалізатора побудована за принципом поділу відповідальності, де кожен ключовий функціональний блок виділено в окремий модуль або клас, що підвищує читабельність коду та спрощує подальший розвиток.

Центральне місце в архітектурі займає Головний цикл програми (Main Application Loop), який є серцем візуалізатора. Цей цикл відповідає за ініціалізацію системи, безперервну обробку подій (введення користувача), оновлення стану сцени, рендеринг кадру та управління життєвим циклом

програми. Взаємодія компонентів відбувається саме через цей цикл, який забезпечує координацію всіх операцій[16, с 128].

Система складається з кількох взаємопов'язаних модулів, що забезпечують її повноцінне функціонування. Модуль введення (Input Handler) відповідає за обробку всіх користувацьких дій, таких як натискання клавіш та рухи миші. Він перехоплює події від операційної системи та перетворює їх на команди для інших компонентів системи, зокрема для модуля управління камерою.

Модуль візуалізації (Rendering Module) є ядром графічної підсистеми. Він інкапсулює всі операції, пов'язані з OpenGL, включаючи ініціалізацію графічного контексту, управління шейдерами, налаштування буферів об'єктів (VAO, VBO, EBO) та виконання команд малювання. Цей модуль отримує дані про моделі від модуля завантаження та, використовуючи інформацію про камеру та освітлення, візуалізує сцену на екрані[2, с 56].

Модуль завантаження моделей (Model Loading Module) відповідає за імпорт 3D-моделей з різних форматів файлів. Він використовує бібліотеку Assimp для парсингу даних моделі та перетворення їх у внутрішнє представлення, яке є зрозумілим для модуля візуалізації. Після завантаження, цей модуль передає готові об'єкти (сітки, матеріали, текстури) до менеджера ресурсів для подальшого використання.

Менеджер ресурсів (Resource Manager) виконує функцію централізованого сховища для всіх завантажених активів, таких як 3D-моделі, текстури та шейдери. Він забезпечує ефективне управління пам'яттю та запобігає повторному завантаженню одних і тих же ресурсів. Модулі візуалізації та завантаження взаємодіють з менеджером ресурсів для отримання необхідних даних.

Модуль управління камерою (Camera Control Module) реалізує логіку переміщення та обертання віртуальної камери в 3D-сцені. Він приймає команди від модуля введення та оновлює позицію, орієнтацію та матриці перетворення камери. Ці матриці потім передаються модулю візуалізації для коректного відображення сцени з точки зору користувача.

Додатково в архітектуру інтегровані Модуль освітлення (Lighting Module), який керує параметрами джерел світла в сцені та передає їх до шейдерів, а також Модуль UI (User Interface Module), що відповідає за виведення базової інформації та потенційно спрощене меню для взаємодії[13, с. 150-175].

Такий модульний підхід до архітектури забезпечує високу гнучкість системи, дозволяючи незалежно розробляти, тестувати та вдосконалювати її окремі компоненти, що є важливим для ефективної реалізації 3D-візуалізатора.

2.2.1. Алгоритм роботи 3D-візуалізатора

Для кращого розуміння послідовності операцій у 3D-візуалізаторі, нижче наведена блок-схема основного алгоритму.

Ця блок-схема ілюструє ключові етапи життєвого циклу 3D-візуалізатора. Процес починається з **ініціалізації**, що включає налаштування основних бібліотек (наприклад, GLFW для керування вікном, OpenGL для графіки), компіляцію та активацію шейдерів, а також налаштування початкової позиції камери. Після цього відбувається **завантаження 3D-моделей** у пам'ять програми, що зазвичай виконується за допомогою спеціалізованих бібліотек, як-от Assimp, для імпорту різних форматів файлів.

Далі система входить в **основний цикл рендерингу**, який безперервно повторюється, доки вікно програми не буде закрито. У кожній ітерації циклу виконуються такі дії: **обробка вхідних даних** від користувача (наприклад, натискання клавіш, рухи миші), **оновлення камери** відповідно до цих даних, **очищення буферів** кольору та глибини для підготовки нового кадру, **рендеринг 3D-моделей** з використанням шейдерів, та **обмін буферами** для відображення підготовленого кадру на екрані. Цей цикл забезпечує плавну та динамічну візуалізацію сцен (див. Рис. 2.1). Якщо вікно закрито, програма завершує свою роботу.

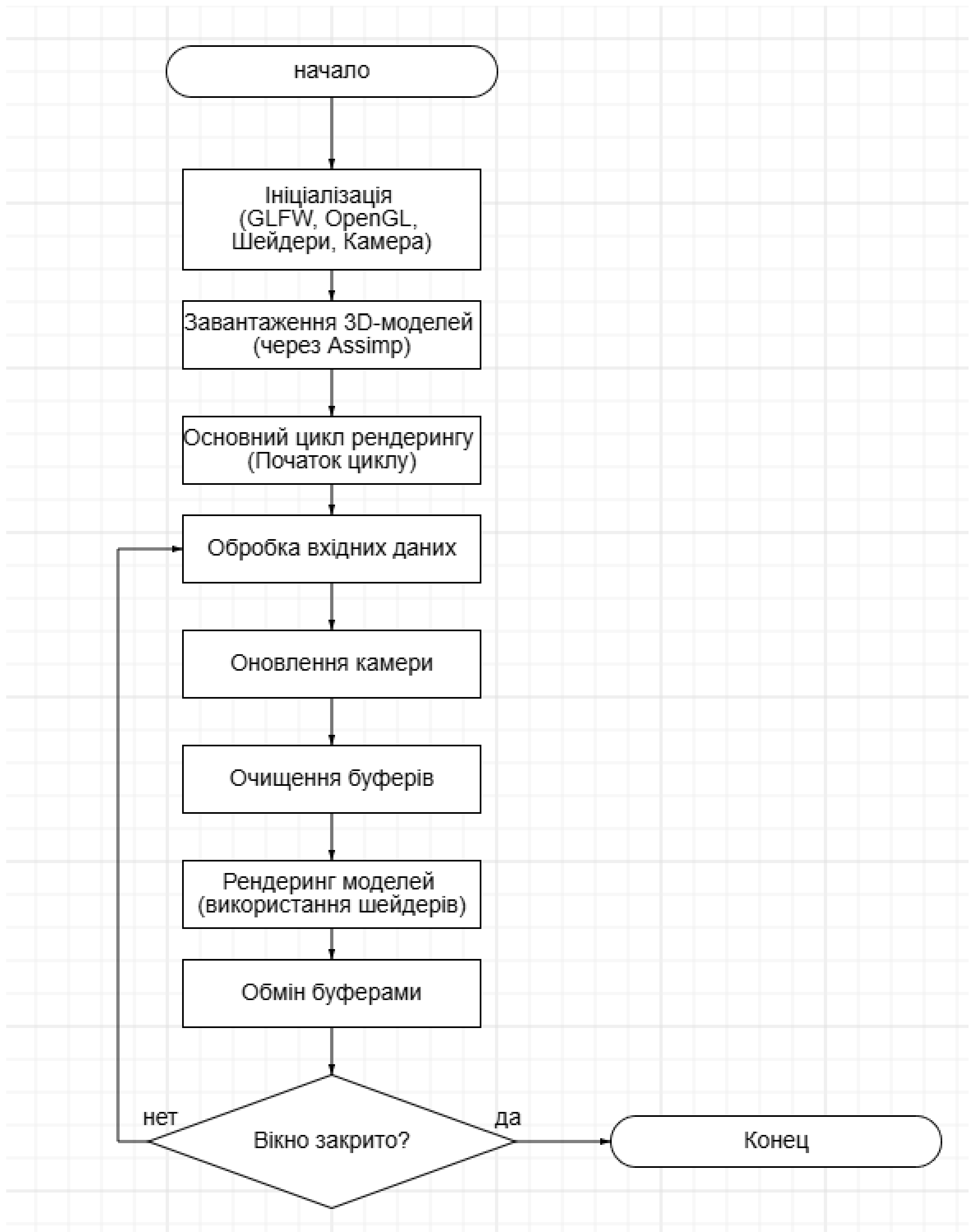

Рис. 2.1. Блок-схема алгоритму 3D-візуалізатора

2.3. Проектування класів та структур даних

Проектування ефективної архітектури програмного забезпечення для 3D-візуалізатора передбачає ретельну розробку класів та структур даних, які

інкапсулюють логіку та інформацію, необхідні для роботи з тривимірними моделями та графічним конвеєром. Об'єктно-орієнтований підхід дозволяє створити модульну та розширювану систему, де кожен компонент має чітко визначену відповідальність[16, с. 50-70]. Нижче представлено ключові класи та структури, що формують основу архітектури візуалізатора.

Клас Model є центральним контейнером для завантаженої 3D-моделі. Він відповідає за завантаження моделі з файлу за допомогою бібліотеки Assimp, зберігання всіх її компонентів (мешів) та координацію їхнього відображення. Цей клас інкапсулює складність роботи з файловими форматами та надає простий інтерфейс для рендерингу цілої моделі. Кожен меш може містити власні матеріали, текстури та геометричні дані, а клас Model забезпечує правильне їх об'єднання для формування цілісного об'єкта. Таким чином, розробнику не потрібно керувати окремими частинами моделі вручну, що значно спрощує процес розробки та відображення. Ця архітектура дозволяє легко інтегрувати нові формати файлів моделей або додавати складніші візуальні ефекти, не змінюючи основну логіку відображення. Основні поля та методи класу Model наведено в Таблиці 2.1.

Таблиця 2.1

Основні елементи класу Model

Елемент класу Model	**Опис**
Поля	
std::vector<Mesh> meshes	Вектор об'єктів класу Mesh, що представляють окремі сітки, з яких складається модель.
std::string directory	Шлях до директорії, де розташований файл моделі, для пошуку текстур.
std::vector<Texture> loaded_textures	Кеш завантажених текстур для запобігання повторному завантаженню.
Методи	

Продовження Табл. 2.1

Model(const char* path)	Конструктор, який ініціалізує завантаження моделі з вказаного шляху.
void Draw(Shader& shader)	Метод для відображення всієї моделі, який перебирає всі меші та викликає їхній метод Draw.
void loadModel(std::string path)	Приватний метод для завантаження даних моделі за допомогою Assimp.
void processNode(aiNode* node, const aiScene* scene)	Приватний метод для рекурсивної обробки вузлів сцени Assimp.
Mesh processMesh(aiMesh* mesh, const aiScene* scene)	Приватний метод для обробки окремого меша Assimp та перетворення його у внутрішню структуру Mesh.
std::vector<Texture> loadMaterialTextures(...)	Приватний метод для завантаження текстур, пов'язаних з матеріалами.

Клас Mesh представляє собою окрему геометричну сітку – базовий елемент 3D-моделі. Він містить дані вершин, індекси для малювання, а також інформацію про текстури та матеріали, що застосовуються до даної сітки. Mesh відповідає за налаштування буферів OpenGL (VAO, VBO, EBO) та власне рендеринг своєї геометрії. Детальний опис елементів класу Mesh представлено в Таблиці 2.2.

Таблиця 2.2

Основні елементи класу Mesh

Елемент класу Mesh	**Опис**
Поля	
std::vector<Vertex> vertices	Вектор структур Vertex, що містять дані про кожну вершину.
std::vector<unsigned int> indices	Вектор цілих чисел, що визначають порядок малювання вершин для формування трикутників.

Продовження Табл. 2.2

std::vector<Texture> textures	Вектор структур Texture, що представляють текстури, пов'язані з цим мешем.
std::string name	Ім'я меша, отримане з файлу моделі.
unsigned int VAO, VBO, EBO	Ідентифікатори об'єктів OpenGL: Vertex Array Object, Vertex Buffer Object, Element Buffer Object.
Методи	
Mesh(...)	Конструктор, що приймає дані вершин, індексів та текстур для ініціалізації меша.
void Draw(Shader& shader)	Метод для малювання меша, активує відповідні текстури та виконує виклик glDrawElements.
void setupMesh()	Приватний метод для налаштування VAO, VBO, EBO та визначення макетів атрибутів вершин.

Структура Vertex визначає атрибути однієї вершини, які є необхідними для коректного рендерингу та освітлення. Вона агрегує всі дані, що передаються у вершинний шейдер. Елементи структури Vertex деталізовано в Таблиці 2.3.

Таблиця 2.3

Елементи структури Vertex

Поле структури Vertex	**Тип даних**	**Опис**
glm::vec3 Position	Вектор 3D	Координати вершини у просторі.
glm::vec3 Normal	Вектор 3D	Вектор нормалі до поверхні вершини (для освітлення).
glm::vec2 TexCoords	Вектор 2D	Координати текстури (UV-координати) для накладання текстури.

Продовження Табл.2.3

glm::vec3 Tangent	Вектор 3D	Вектор тангенса (для нормальних карт).
glm::vec3 Bitangent	Вектор 3D	Вектор бітангенса (для нормальних карт).

Структура Texture інкапсулює інформацію про завантажену текстуру, яка використовується для візуалізації моделей. Її поля наведено в Таблиці 2.4.

Таблиця 2.4

Елементи структури Texture

Поле структури Texture	**Тип даних**	**Опис**
unsigned int id	unsigned int	Ідентифікатор текстури в OpenGL.
std::string type	std::string	Тип текстури (наприклад, "texture_diffuse", "texture_specular", "texture_normal").
std::string path	std::string	Шлях до файлу текстури на диску, для кешування.

Клас Shader відповідає за компіляцію, лінкування та активацію GLSL-шейдерів (вершинного, фрагментного). Він надає зручні методи для передачі даних (уніформів) з C++-програми до шейдерів на GPU. Детальний опис класу Shader представлено в Таблиці 2.5.

Таблиця 2.5

Основні елементи класу Shader

Елемент класу Shader	**Опис**
Поля	
unsigned int ID	Ідентифікатор програми шейдерів в OpenGL.

Продовження Табл.2.5

Методи	
Shader(const char* vertexPath, const char* fragmentPath)	Конструктор, що завантажує, компілює та лінкує вершинний та фрагментний шейдери.
void use()	Метод для активації програми шейдерів.
void setBool(const std::string& name, bool value) const	Методи для встановлення значень уніформних змінних (bool, int, float, vec, mat)
void setInt(const std::string& name, int value) const	Встановлює цілочисельне значення uniform-змінної у шейдері.
void setFloat(const std::string& name, float value) const	Встановлює значення з плаваючою крапкою для uniform-змінної у шейдері.
void setVec2(const std::string& name, const glm::vec2& value) const	Встановлює двокомпонентний вектор (vec2) як uniform-змінну у шейдері.
void setVec3(const std::string& name, const glm::vec3& value) const	Встановлює трикомпонентний вектор (vec3) як uniform-змінну у шейдері.
void setMat4(const std::string& name, const glm::mat4& mat) const	Встановлює матрицю 4×4 (mat4) як uniform-змінну у шейдері.

Клас Camera моделює віртуальну камеру у 3D-сцені. Він відповідає за управління її положенням, орієнтацією та обчислення матриць виду та проекції, які необхідні для коректного відображення сцени з точки зору користувача. Це дозволяє користувачам інтерактивно переміщуватися сценою, змінювати кут огляду та фокус, забезпечуючи повний контроль над візуалізацією. Крім того, клас Camera підтримує різні типи проекцій, такі як перспективна та ортографічна, що є ключовим для різноманітних сценаріїв використання.
Основні елементи класу Camera наведено в Таблиці 2.6.

Таблиця 2.6

Основні елементи класу Camera

Елемент класу Camera	**Опис**
Поля	

Продовження Табл.2.6

glm::vec3 Position	Поточна позиція камери у світових координатах.
glm::vec3 Front	Вектор, що вказує напрямок "вперед" для камери.
glm::vec3 Up	Вектор, що вказує напрямок "вгору" для камери.
glm::vec3 Right	Вектор, що вказує напрямок "вправо" для камери.
float Yaw, Pitch	Кути повороту камери (рискання та тангаж) для орієнтації.
float MovementSpeed	Швидкість переміщення камери.
float MouseSensitivity	Чутливість миші для обертання камери.
float Zoom	Поточний кут огляду (FOV) камери.
Методи	
Camera(glm::vec3 position, ...)	Конструктор, що ініціалізує позицію, напрямок та інші параметри камери.
glm::mat4 GetViewMatrix()	Повертає матрицю виду (View Matrix), яка перетворює світові координати у видові.
void ProcessKeyboard(Camera_Movement direction, float deltaTime)	Обробляє введення з клавіатури для переміщення камери.
void ProcessMouseMovement(float xoffset, float yoffset, ...)	Обробляє рухи миші для обертання камери.
void ProcessMouseScroll(float yoffset)	Обробляє прокрутку коліщатка миші для зміни масштабу (зуму).
void updateCameraVectors()	Приватний метод для перерахунку векторів Front, Up, Right після зміни Yaw/Pitch.

Запропонована структура класів та даних забезпечує чіткий поділ відповідальності, що спрощує розробку, налагодження та подальше розширення функціоналу 3D-візуалізатора.

2.4. Проектування взаємодії користувача

Ефективна взаємодія користувача з тривимірною сценою є фундаментальним аспектом будь-якого 3D-візуалізатора. Проектування інтуїтивно зрозумілого та чуйного інтерфейсу введення дозволяє користувачеві вільно досліджувати моделі та отримувати повне уявлення про їхню геометрію та текстури [15, с. 112-118].. В даному візуалізаторі основний акцент робиться на управлінні за допомогою клавіатури та миші, що є стандартним підходом у більшості 3D-додатків та ігор.

Управління віртуальною камерою реалізується за допомогою комбінації клавіатурного введення та рухів миші, що забезпечує гнучку та природну навігацію у тривимірному просторі. Переміщення камери здійснюється за допомогою стандартних клавіш WASD, що дозволяють рухатися вперед, назад, вліво та вправо відповідно. Для вертикального переміщення камери використовуються клавіші Left Control (для руху вниз) та Space (для руху вгору). Всі ці переміщення розраховуються з урахуванням часу між кадрами (deltaTime), що гарантує плавність та незалежність швидкості руху від частоти кадрів системи. Це запобігає ривкам та забезпечує стабільний користувацький досвід на різних апаратних конфігураціях.

Орієнтація камери, тобто напрямок погляду, контролюється рухами миші. Горизонтальні рухи миші (по осі X екрану) впливають на кут рискання (Yaw) камери, дозволяючи обертатися навколо вертикальної осі. Вертикальні рухи миші (по осі Y екрану) впливають на кут тангажу (Pitch), дозволяючи нахиляти камеру вгору або вниз. Для забезпечення точності та чутливості управління, ці рухи масштабуються за допомогою параметру чутливості миші. Додатково, прокрутка коліщатка миші використовується для зміни кута огляду (Field of View – FOV) камери, що дозволяє наближатися або віддалятися від об'єктів у сцені, забезпечуючи ефект зуму.

Окрім управління камерою, візуалізатор надає функціонал для динамічного завантаження нових 3D-моделей. Це реалізовано за допомогою натискання клавіші 'L' (Load). При активації цієї функції з'являється стандартне діалогове вікно вибору файлу, що дозволяє користувачеві вказати шлях до нової моделі. Після вибору файлу, система ініціює процес завантаження, а поточна модель, якщо вона була завантажена, коректно видаляється з пам'яті. Це забезпечує можливість швидкої зміни відображуваних моделей без перезапуску програми.

Взаємодія з графічною підсистемою та обробка подій введення здійснюється за допомогою механізмів зворотних викликів (callbacks), наданих бібліотекою GLFW. Функції framebuffer_size_callback, mouse_callback та scroll_callback реєструються для обробки зміни розміру вікна, рухів курсора миші та подій прокрутки коліщатка відповідно. Це дозволяє системі реагувати на дії користувача в реальному часі та адаптувати відображення сцени. Загальний цикл обробки введення (processInput) постійно відстежує стан клавіатури, забезпечуючи безперервне управління рухом камери.

Такий підхід до проектування взаємодії користувача забезпечує високу ергономічність та зручність використання візуалізатора, дозволяючи користувачеві зосередитися на дослідженні архітектурних моделей, а не на складнощах управління програмою.

Висновки до розділу 2

У другому розділі дипломної роботи було детально розглянуто та спроектовано архітектуру 3D-візуалізатора архітектурних моделей. На початковому етапі було чітко сформульовано функціональні та нефункціональні вимоги до програмного забезпечення, що дозволило визначити ключовий функціонал, такий як завантаження моделей у форматах DAE, FBX, OBJ, їх коректне відображення з текстурами та матеріалами, а також інтерактивне управління камерою. Нефункціональні вимоги зосередилися на продуктивності,

надійності, зручності використання та розширюваності системи, що є критично важливим для створення якісного програмного продукту.

Далі була розроблена загальна архітектура системи, яка базується на модульному підході з чітким поділом відповідальності між основними компонентами. Було виділено ключові модулі, такі як модуль введення, модуль візуалізації, модуль завантаження моделей, менеджер ресурсів, модуль управління камерою та інші, що взаємодіють через головний цикл програми. Такий підхід забезпечує гнучкість, полегшує розробку та подальшу підтримку системи.

Особливу увагу було приділено проектуванню класів та структур даних, що є основою внутрішнього представлення 3D-сцени та її компонентів. Детально описано структуру та функціонал класів Model, Mesh, Shader, Camera, а також структур Vertex та Texture. Ці класи інкапсулюють логіку роботи з геометричними даними, матеріалами, текстурами, шейдерами та управлінням віртуальною камерою, забезпечуючи ефективну взаємодію з OpenGL.

Нарешті, було спроектовано систему взаємодії користувача, яка реалізується за допомогою клавіатури та миші. Описано механізми управління камерою (переміщення, обертання, масштабування) та функціонал завантаження нових моделей через діалогове вікно. Використання механізмів зворотних викликів GLFW гарантує чуйність системи на дії користувача.

Таким чином, етап проектування заклав міцний фундамент для подальшої реалізації 3D-візуалізатора. Сформовані вимоги, розроблена модульна архітектура та деталізовані структури даних створюють чіткий план для переходу до практичної імплементації програмного забезпечення, що дозволить досягти поставлених цілей дипломної роботи.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ 3D-ВІЗУАЛІЗАТОРА ТА ЙОГО ФУНКЦІОНАЛ

3.1. Налаштування середовища розробки та необхідні бібліотеки

Етап реалізації програмного забезпечення 3D-візуалізатора передбачає вибір та налаштування відповідного середовища розробки, а також інтеграцію необхідних зовнішніх бібліотек. Ці компоненти формують основу для написання, компіляції та виконання коду, що взаємодіє з графічним апаратним забезпеченням.

В якості **інтегрованого середовища розробки (IDE)** для проекту було обрано **Microsoft Visual Studio**. Цей вибір обумовлений його широкими можливостями для розробки на C++, потужними інструментами для налагодження, інтеграцією з системою управління версіями та зручним інтерфейсом користувача. Visual Studio надає повний набір функцій, необхідних для ефективної розробки складних графічних додатків.

Для управління процесом збірки проекту використовується система **CMake**. CMake є кросплатформним інструментом, який генерує файли збірки (наприклад, файли проекту Visual Studio) з незалежних від платформи файлів CMakeLists.txt. Це забезпечує гнучкість та переносимість проекту на різні операційні системи та компілятори, що є важливим для підтримки кросплатформенності OpenGL-додатків.

Функціональність 3D-візуалізатора значною мірою спирається на використання наступних ключових зовнішніх бібліотек:

OpenGL Extension Wrangler Library (GLEW) / OpenGL Loader Generator (GLAD): Ці бібліотеки є завантажувачами функцій OpenGL. Оскільки OpenGL – це лише специфікація, а не бібліотека, що поставляється з драйверами, для доступу до її функцій необхідно використовувати завантажувач. **GLAD** був обраний як основний завантажувач, оскільки він дозволяє генерувати індивідуальні файли заголовків, що містять лише необхідні функції OpenGL для

конкретної версії та профілю (у даному випадку OpenGL 3.3 Core Profile). Це сприяє зменшенню розміру виконуваного файлу та підвищенню ефективності[10, с 15-20].

GLFW (Graphics Library Framework): Ця легка, кросплатформенна бібліотека використовується для створення вікна OpenGL-контексту, обробки введення з клавіатури та миші, а також управління таймерами. GLFW надає необхідний функціонал для ініціалізації графічної підсистеми та взаємодії з користувачем, не обтяжуючи проект зайвими залежностями.

GLM (OpenGL Mathematics): GLM – це бібліотека математичних функцій, спеціально розроблена для використання з OpenGL. Вона надає класи та функції для роботи з векторами, матрицями, кватерніонами, що є невід'ємними для 3D-графіки. Використання GLM значно спрощує математичні операції, такі як перетворення координат, обчислення матриць виду та проекції, а також векторні операції, які є основою для управління камерою та позиціонування моделей.

Open Asset Import Library (Assimp): Як вже було зазначено в Розділі 1, Assimp є ключовою бібліотекою для імпорту 3D-моделей. Вона дозволяє завантажувати моделі у різноманітних форматах (DAE, FBX, OBJ) та надає уніфікований доступ до їхньої геометрії, матеріалів та текстур. Інтеграція Assimp значно скоротила час розробки, усунувши необхідність написання власних парсерів для кожного формату файлів.

STB_Image: Це легка, однофайлова бібліотека для завантаження зображень у різних форматах (PNG, JPG, BMP, TGA тощо). Вона використовується для завантаження текстурних файлів з диска та перетворення їх у формат, придатний для використання в OpenGL. Простота та ефективність STB_Image роблять її ідеальним вибором для управління текстурними ресурсами.

Налаштування цих інструментів та бібліотек у середовищі Visual Studio з використанням CMake забезпечило міцну та гнучку основу для подальшої реалізації всіх функціональних можливостей 3D-візуалізатора.

3.2. Ініціалізація OpenGL контексту та вікна

Першим і фундаментальним кроком у розробці будь-якого OpenGL-додатку є створення вікна та ініціалізація графічного контексту. Цей процес забезпечує програмному забезпеченню доступ до функціональності графічного процесора та дозволяє виводити зображення на екран. Для виконання цих завдань у розробленому 3D-візуалізаторі використовується бібліотека GLFW[15, с 114].

Процес ініціалізації починається з виклику функції glfwInit(), яка ініціалізує бібліотеку GLFW. Після успішної ініціалізації налаштовуються параметри вікна та OpenGL-контексту за допомогою функцій glfwWindowHint(). Для даного проекту було встановлено вимоги до версії OpenGL – 3.3 Core Profile, що забезпечує використання сучасного конвеєра рендерингу[14, с 48], який повністю базується на шейдерах. Це досягається встановленням GLFW_CONTEXT_VERSION_MAJOR та GLFW_CONTEXT_VERSION_MINOR у значення 3, а також GLFW_OPENGL_PROFILE у GLFW_OPENGL_CORE_PROFILE. Для сумісності з macOS також додається підказка GLFW_OPENGL_FORWARD_COMPAT зі значенням GL_TRUE.

Після налаштування параметрів, вікно створюється за допомогою функції glfwCreateWindow(), яка приймає ширину, висоту, заголовок вікна та вказівники на монітор і вікно для спільного використання контексту. У разі успішного створення вікна, його OpenGL-контекст робиться поточним для поточного потоку виконання за допомогою glfwMakeContextCurrent(). Це гарантує, що всі подальші виклики OpenGL будуть застосовуватися до цього конкретного контексту.

Наступним важливим етапом є ініціалізація GLAD. GLAD є завантажувачем вказівників на функції OpenGL, які не є частиною стандартної бібліотеки C/C++. Функція gladLoadGLLoader() приймає вказівник на функцію,

яка повертає адресу функції OpenGL (у випадку GLFW це glfwGetProcAddress). Успішна ініціалізація GLAD дозволяє програмі використовувати всі необхідні функції OpenGL, починаючи з версії 3.3.

Для коректного відображення 3D-сцени необхідно активувати глибинний тест (Depth Test). Це виконується за допомогою виклику glEnable(GL_DEPTH_TEST). Глибинний тест дозволяє OpenGL визначати, які об'єкти знаходяться ближче до камери, і коректно відображати їх, приховуючи ті, що перекриті іншими, забезпечуючи правильну перспективу та глибину сцени.

Окрім створення вікна та контексту, на цьому етапі також налаштовуються функції зворотного виклику (callbacks) GLFW. Функції glfwSetFramebufferSizeCallback(), glfwSetCursorPosCallback() та glfwSetScrollCallback() реєструються для обробки зміни розміру вікна, положення курсора миші та подій прокрутки коліщатка відповідно. Це дозволяє візуалізатору динамічно реагувати на дії користувача та адаптувати відображення сцени. Нарешті, для забезпечення плавного управління камерою, курсор миші приховується та захоплюється вікном за допомогою glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED).

Таким чином, ініціалізація OpenGL контексту та вікна за допомогою GLFW та GLAD створює необхідне середовище для подальшої реалізації всіх графічних операцій та інтерактивної взаємодії з 3D-сценою.

3.3. Реалізація класів Shader, Camera

Для ефективного відображення тривимірних сцен та забезпечення інтерактивної навігації в 3D-візуалізаторі ключову роль відіграють класи Shader та Camera. Ці класи інкапсулюють складну логіку роботи з графічним конвеєром OpenGL та управлінням точкою зору користувача. Клас Shader дозволяє визначати, як об'єкти освітлюються та виглядають, забезпечуючи гнучкість у

застосуванні візуальних ефектів. У поєднанні, ці класи формують міцний фундамент для створення реалістичних та динамічних 3D-додатків.

3.3.1. Реалізація класу Shader

Клас Shader призначений для завантаження, компіляції та лінкування вершинного та фрагментного шейдерів, написаних на мові GLSL (OpenGL Shading Language). Його основне завдання – надати зручний інтерфейс для взаємодії з програмою шейдерів на GPU, дозволяючи передавати уніформні змінні (uniforms) з CPU.

Конструктор класу Shader приймає шляхи до файлів вершинного та фрагментного шейдерів. Спочатку він зчитує вихідний код шейдерів з цих файлів у вигляді рядків. Після успішного зчитування відбувається компіляція кожного шейдера. Для цього створюється об'єкт шейдера відповідного типу (вершинний або фрагментний) за допомогою glCreateShader(), вихідний код прикріплюється до нього за допомогою glShaderSource(), і потім викликається glCompileShader(). Після компіляції обов'язково виконується перевірка статусу компіляції за допомогою glGetShaderiv() та glGetShaderInfoLog() для виявлення можливих помилок. У разі помилки компіляції, програма виводить відповідне повідомлення в консоль.

Після успішної компіляції обох шейдерів вони лінкуються в єдину програму шейдерів. Для цього створюється об'єкт програми за допомогою glCreateProgram(), до якого прикріплюються скомпільовані вершинний та фрагментний шейдери за допомогою glAttachShader(). Потім викликається glLinkProgram() для лінкування. Аналогічно компіляції, після лінкування виконується перевірка статусу лінкування та виведення логу помилок, якщо такі є. Після успішного лінкування, скомпільовані шейдери від'єднуються від програми та видаляються за допомогою glDetachShader() та glDeleteShader(), оскільки вони більше не потрібні. Ідентифікатор створеної програми шейдерів зберігається у полі ID класу Shader.

Метод use() класу Shader активує програму шейдерів за допомогою glUseProgram(ID), роблячи її поточною для всіх подальших операцій рендерингу.

Крім того, клас Shader надає набір методів setBool(), setInt(), setFloat(), setVec2(), setVec3(), setMat4(). Ці методи дозволяють зручно передавати значення різних типів даних (булеві, цілі числа, числа з плаваючою комою, вектори та матриці) у відповідні уніформні змінні в шейдерах. Кожен такий метод отримує ім'я уніформної змінної та її значення, а потім використовує відповідні функції OpenGL (наприклад, glUniform1i, glUniform3fv, glUniformMatrix4fv) для передачі даних на GPU. Це забезпечує гнучку конфігурацію шейдерів під час виконання програми.

3.3.2. Реалізація класу Camera

Клас Camera моделює віртуальну камеру, яка дозволяє користувачеві переміщуватися та оглядати 3D-сцену. Він інкапсулює логіку управління позицією, орієнтацією камери та обчислення необхідних матриць для відображення сцени з її точки зору.

Конструктор класу Camera ініціалізує початкову позицію камери, її вектори (напрямок "вперед", "вгору", "вправо"), кути рискання (Yaw) та тангажу (Pitch), а також параметри руху та чутливості миші. Вектори Front, Up та Right обчислюються на основі кутів Yaw та Pitch за допомогою тригонометричних функцій. Метод updateCameraVectors() відповідає за перерахунок цих векторів щоразу, коли змінюється орієнтація камери, забезпечуючи коректне відображення її напрямку.

Основний функціонал класу Camera реалізований у методах обробки введення:

- **ProcessKeyboard(Camera_Movement direction, float deltaTime):** Цей метод обробляє рухи камери, викликані натисканням клавіш. Він приймає напрямок руху (наприклад, FORWARD, BACKWARD, LEFT, RIGHT, UP, DOWN) та deltaTime для забезпечення швидкості руху, незалежної від частоти кадрів. На основі напрямку та швидкості руху оновлюється позиція камери.
- **ProcessMouseMovement(float xoffset, float yoffset, GLboolean constrainPitch = true):** Цей метод відповідає за обертання камери у відповідь на рухи миші. Він приймає зміщення курсора по осях X та Y. Ці зміщення

масштабуються за допомогою чутливості миші та використовуються для оновлення кутів Yaw та Pitch. Параметр constrainPitch дозволяє обмежити кут тангажу, щоб уникнути "перевороту" камери. Після оновлення кутів викликається updateCameraVectors() для перерахунку векторів напрямку.

- **ProcessMouseScroll(float yoffset):** Цей метод обробляє події прокрутки коліщатка миші, які використовуються для зміни кута огляду (Zoom/FOV). Зміщення yoffset додається до поточного значення Zoom, яке потім обмежується певним діапазоном (наприклад, від 1.0f до 45.0f), щоб уникнути екстремальних значень зуму.

Метод GetViewMatrix() є ключовим для рендерингу. Він обчислює та повертає матрицю виду (View Matrix) за допомогою функції glm::lookAt(). Ця матриця перетворює світові координати об'єктів у видові координати, тобто переміщує та обертає сцену таким чином, щоб вона відображалася з точки зору камери. glm::lookAt() приймає позицію камери, точку, на яку вона дивиться (Position + Front), та вектор "вгору" камери (Up).

Таким чином, класи Shader та Camera є фундаментальними компонентами 3D-візуалізатора, що забезпечують як візуальну складову (завдяки шейдерам), так і інтерактивність (завдяки управлінню камерою), дозволяючи користувачеві повноцінно взаємодіяти з архітектурними моделями.

3.4. Завантаження 3D-моделей за допомогою Assimp

Процес завантаження тривимірних моделей є одним з найскладніших етапів у розробці 3D-візуалізатора, оскільки він вимагає коректної інтерпретації різноманітних форматів файлів. Для вирішення цієї задачі у даному проекті використовується бібліотека Open Asset Import Library (Assimp), яка надає уніфікований інтерфейс для імпорту 3D-даних.

Завантаження моделі починається з ініціалізації об'єкта Assimp::Importer. Цей об'єкт є основним інтерфейсом для взаємодії з бібліотекою. Далі викликається метод ReadFile() об'єкта Importer, якому передається шлях до

файлу моделі та набір прапорів постобробки. Ці прапори (такі як aiProcess_Triangulate, aiProcess_FlipUVs, aiProcess_GenNormals, aiProcess_JoinIdenticalVertices, aiProcess_OptimizeMeshes, aiProcess_CalcTangentSpace, aiProcess_LimitBoneWeights, aiProcess_ValidateDataStructure) є критично важливими, оскільки вони дозволяють Assimp автоматично виконувати ряд операцій для підготовки моделі до рендерингу в OpenGL, забезпечуючи її коректне відображення та оптимізацію. Після успішного читання файлу Assimp повертає вказівник на об'єкт aiScene, який є кореневим вузлом імпортованої сцени і містить всі дані моделі.

Якщо завантаження моделі пройшло успішно, починається рекурсивна обробка вузлів сцени, починаючи з кореневого вузла aiScene->mRootNode. Ця обробка виконується методом processNode() класу Model. Кожен вузол aiNode у ієрархії сцени може містити посилання на один або декілька мешів, а також на дочірні вузли. Метод processNode() перебирає всі меші, пов'язані з поточним вузлом, і для кожного з них викликає метод processMesh(). Після обробки всіх мешів поточного вузла, processNode() рекурсивно викликає себе для кожного дочірнього вузла, таким чином обходячи всю ієрархію сцени.

Метод processMesh() відповідає за вилучення даних з об'єкта aiMesh, наданого Assimp, та перетворення їх у внутрішню структуру Mesh, яка використовується візуалізатором. Для кожної вершини меша Assimp витягуються її позиційні координати (mVertices), нормалі (mNormals), текстурні координати (mTextureCoords), а також тангенси та бітангенси (mTangents, mBitangents), якщо вони присутні. Ці дані зберігаються у векторі структур Vertex. Індекси для малювання трикутників витягуються з об'єктів aiFace меша та зберігаються у векторі indices.

Особлива увага приділяється обробці матеріалів та текстур. Якщо меш має пов'язаний матеріал (перевіряється за mesh->mMaterialIndex), то викликається метод loadMaterialTextures(). Цей метод перебирає всі типи текстур, які можуть бути пов'язані з матеріалом (дифузні, спекулярні, нормальні, карти висот), і для

кожного типу намагається отримати шлях до файлу текстури з об'єкта aiMaterial. Важливою особливістю реалізації є те, що Assimp може надавати як відносні, так і абсолютні шляхи до текстур, які можуть бути некоректними на поточній системі. Для вирішення цієї проблеми, з отриманого шляху витягується лише ім'я файлу текстури, а повний шлях до файлу формується відносно директорії, в якій знаходиться сама модель, з припущенням, що текстури розташовані у підпапці textures поруч з моделлю.

Перед завантаженням кожної текстури виконується перевірка, чи не була вона вже завантажена раніше. Це досягається шляхом порівняння імені файлу поточної текстури з іменами файлів текстур, що вже знаходяться у кеші textures_loaded. Якщо текстура вже завантажена, її ідентифікатор повторно використовується, що дозволяє уникнути дублювання ресурсів та оптимізувати використання пам'яті GPU. Якщо текстура ще не завантажена, викликається функція TextureFromFile(), яка завантажує зображення з диска за допомогою бібліотеки STB_Image, генерує OpenGL-текстуру та повертає її ідентифікатор. Після успішного завантаження, інформація про текстуру (ID, тип, шлях) додається до кешу textures_loaded та до вектора текстур поточного меша.

Після вилучення всіх необхідних даних, створюється новий об'єкт класу Mesh, якому передаються зібрані вершини, індекси та текстури. Цьому об'єкту також присвоюється ім'я меша, отримане від Assimp (mesh->mName.C_Str()), що є корисним для відладки та, у випадку DAE-моделей без вбудованих текстур, для ручного призначення текстур на етапі виконання.

Таким чином, інтеграція Assimp дозволяє 3D-візуалізатору ефективно імпортувати складні 3D-моделі з різних джерел, автоматично обробляючи їхні дані та готуючи до рендерингу в OpenGL, що є ключовим для досягнення поставлених цілей проекту. Це значно спрощує процес розробки та дозволяє зосередитися на візуалізації та взаємодії, а не на низькорівневих деталях форматування моделей.

3.5. Обробка та відображення текстур

Текстури є невід'ємною частиною реалістичної 3D-візуалізації, оскільки вони надають поверхням об'єктів колір, деталізацію та візуальну складність, що значно підвищує якість сприйняття моделі. У розробленому візуалізаторі процес обробки та відображення текстур включає їх завантаження з файлу, генерацію OpenGL-текстури та подальше використання у шейдерах.

Завантаження зображень з диска здійснюється за допомогою легкої бібліотеки **STB_Image**. Ця бібліотека є однофайловим рішенням, що спрощує її інтеграцію в проект та забезпечує підтримку широкого спектра популярних графічних форматів, таких як PNG, JPG, BMP, TGA. Функція TextureFromFile(), яка реалізована у класі Model, відповідає за цей процес. Вона приймає шлях до файлу зображення, а також директорію моделі для коректного формування кінцевого шляху до текстури. Після завантаження, stbi_load() повертає вказівник на масив байтів, що містить піксельні дані зображення, а також його ширину, висоту та кількість компонентів (каналів) кольору (наприклад, 1 для чорно-білого, 3 для RGB, 4 для RGBA).

Після успішного завантаження піксельних даних, відбувається генерація OpenGL-текстури. Спочатку генерується унікальний ідентифікатор текстури за допомогою glGenTextures(). Потім цей ідентифікатор активується та прив'язується до цільового типу текстури (у даному випадку GL_TEXTURE_2D) за допомогою glBindTexture(). Піксельні дані завантажуються в пам'ять GPU за допомогою glTexImage2D(). Ця функція приймає цільовий тип текстури, рівень деталізації (mipmap level), внутрішній формат текстури (наприклад, GL_RGB або GL_RGBA залежно від кількості компонентів), розміри текстури, границі та, власне, вказівник на піксельні дані.

Для оптимізації рендерингу та покращення візуальної якості, особливо при зміні відстані до об'єкта, генеруються mipmap-рівні за допомогою glGenerateMipmap(). Mipmap-рівні – це попередньо розраховані, зменшені версії текстури, які використовуються OpenGL для рендерингу об'єктів на різних від-

станях, запобігаючи артефактам та покращуючи продуктивність.

Наступним кроком є налаштування параметрів фільтрації та обгортання текстури за допомогою glTexParameteri(). Параметри обгортання (GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T) визначають поведінку текстури, коли текстурні координати виходять за межі діапазону [0, 1] (наприклад, GL_REPEAT для повторення текстури). Параметри фільтрації (GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER) контролюють, як текстура масштабується при збільшенні або зменшенні. Використання GL_LINEAR_MIPMAP_LINEAR для мініфікації та GL_LINEAR для магніфікації забезпечує плавне та якісне відображення текстур на різних відстанях. Після всіх операцій піксельні дані, завантажені STB_Image, звільняються з пам'яті CPU за допомогою stbi_image_free().

Використання текстур у шейдерах є ключовим для їх відображення. Кожна текстура прив'язується до унікального текстурного юніта (texture unit) за допомогою glActiveTexture(GL_TEXTURE0 + i). Це дозволяє одночасно використовувати декілька текстур у шейдері. У вершинному шейдері текстурні координати передаються у фрагментний шейдер як інтерпольований атрибут. У фрагментному шейдері, за допомогою функції sampler2D та переданих текстурних координат, відбувається вибірка кольору з текстури. Тип текстури (дифузна, спекулярна, нормальна) та її відповідний текстурний юніт передаються в шейдер як уніформні змінні (наприклад, material.diffuse для дифузної текстури). Це дозволяє шейдеру застосовувати різні типи текстур для розрахунку кольору пікселя та освітлення.

Особлива увага приділяється обробці нормальних карт (normal maps). Нормальні карти використовуються для імітації дрібних деталей поверхні без додавання додаткової геометрії. Для їх коректного застосування необхідні тангенси та бітангенси вершин, які обчислюються Assimp на етапі завантаження моделі. У фрагментному шейдері нормаль, отримана з нормальної карти, перетворюється у світові або видові координати та використовується для більш детального розрахунку освітлення, створюючи ілюзію об'ємності.

Таким чином, ретельна обробка та відображення текстур, від їх завантаження до застосування у шейдерах, є фундаментальним аспектом, що дозволяє 3D-візуалізатору досягати високого рівня реалізму та деталізації архітектурних моделей.

3.6. Налаштування VBO, VAO, EBO для рендерингу

Для ефективного відображення тривимірних моделей в OpenGL, дані про їхню геометрію (позиції вершин, нормалі, текстурні координати, тангенси, бітангенси) повинні бути організовані та передані на графічний процесор (GPU) у спеціалізовані буфери. Цей процес реалізується за допомогою об'єктів буфера вершин (VBO), об'єктів буфера елементів (EBO) та об'єктів масиву вершин (VAO).

Vertex Buffer Object (VBO) є буфером пам'яті на GPU, призначеним для зберігання даних вершин. Це дозволяє завантажувати великі обсяги даних вершин один раз і використовувати їх багаторазово для рендерингу, що значно підвищує продуктивність порівняно з передачею даних для кожної вершини окремо. У класі Mesh дані про вершини (std::vector<Vertex> vertices) передаються до VBO. Для цього спочатку генерується VBO за допомогою glGenBuffers(), потім він прив'язується до цільового типу GL_ARRAY_BUFFER за допомогою glBindBuffer(). Далі, функція glBufferData() використовується для копіювання даних з пам'яті CPU (вектора vertices) у VBO на GPU. Параметр GL_STATIC_DRAW вказує, що дані будуть завантажені на GPU один раз і використовуватимуться для багатьох операцій малювання, що є типовим для статичних моделей.

Element Buffer Object (EBO), також відомий як Index Buffer Object (IBO), є буфером пам'яті на GPU, призначеним для зберігання індексів вершин. Замість того, щоб дублювати дані вершин для кожної грані, яка використовує спільні вершини, EBO дозволяє зберігати унікальні вершини у VBO та посилатися на них за допомогою індексів. Це значно зменшує обсяг даних, що передаються на

GPU, та оптимізує використання пам'яті. У класі Mesh індекси (std::vector<unsigned int> indices) завантажуються в EBO аналогічно VBO: генерується EBO, прив'язується до GL_ELEMENT_ARRAY_BUFFER, і дані копіюються за допомогою glBufferData().

Vertex Array Object (VAO) є об'єктом, який зберігає конфігурацію стану для VBO та EBO, а також інформацію про те, як інтерпретувати дані вершин. VAO інкапсулює виклики glEnableVertexAttribArray() та glVertexAttribPointer(), що описують макет атрибутів вершин (тобто, де в буфері знаходяться позиції, нормалі, текстурні координати тощо). Використання VAO дозволяє швидко перемикатися між різними конфігураціями вершинних даних, оскільки достатньо прив'язати відповідний VAO, і всі налаштування буферів та їхніх атрибутів будуть відновлені автоматично.

У методі setupMesh() класу Mesh відбувається повне налаштування цих буферів. Спочатку генерується та прив'язується VAO за допомогою glGenVertexArrays() та glBindVertexArray(). Після цього прив'язуються VBO та EBO, і дані копіюються в них. Далі, для кожного атрибута вершини (позиція, нормаль, текстурні координати, тангенс, бітангенс) викликається glEnableVertexAttribArray() для активації атрибута та glVertexAttribPointer() для визначення його макета. glVertexAttribPointer() вказує OpenGL, як інтерпретувати дані у VBO: кількість компонентів (наприклад, 3 для позиції), тип даних (GL_FLOAT), нормалізація, розмір кроку між вершинами (sizeof(Vertex)) та зміщення атрибута всередині структури Vertex (використовуючи offsetof). Після налаштування всіх атрибутів, VAO відв'язується за допомогою glBindVertexArray(0).

Під час рендерингу меша (у методі Mesh::Draw()), для відображення меша достатньо прив'язати відповідний VAO за допомогою glBindVertexArray(VAO). Потім, для малювання трикутників, використовується glDrawElements(), яка посилається на індекси, що зберігаються в EBO.

Таким чином, використання VBO, EBO та VAO є фундаментальним для оптимізації рендерингу в OpenGL. Вони дозволяють ефективно управляти

даними вершин на GPU, мінімізувати кількість викликів API та забезпечувати високу продуктивність при відображенні складних 3D-моделей.

3.7. Цикл рендерингу та взаємодія з користувачем

Головний цикл програми є центральним елементом будь-якого інтерактивного 3D-додатку. Він відповідає за безперервне оновлення стану сцени, обробку введення користувача та рендеринг нових кадрів. У 3D-візуалізаторі архітектурних моделей цей цикл реалізований у функції main() і працює доти, доки користувач не вирішить закрити вікно програми.

На початку кожного кадру відбувається розрахунок часу deltaTime. Це змінна, яка зберігає час, що минув з моменту попереднього кадру. Використання deltaTime є критично важливим для забезпечення плавності руху та незалежності швидкості анімації чи переміщення камери від частоти кадрів системи. Це гарантує, що рух буде однаковим на комп'ютерах з різною продуктивністю[14, 54].

Після розрахунку deltaTime викликається функція processInput(). Ця функція відповідає за обробку всіх користувацьких подій, таких як натискання клавіш та рухи миші. Вона взаємодіє з бібліотекою GLFW для отримання поточного стану клавіатури та передає відповідні команди класу Camera для оновлення її позиції та орієнтації. Також у processInput() реалізована логіка для завантаження нових моделей: при натисканні клавіші 'L' відкривається діалогове вікно вибору файлу (tinyfd_openFileDialog), і якщо користувач вибирає новий файл, встановлюється спеціальний прапор loadNewModel.

Якщо прапор loadNewModel встановлено, це сигналізує про необхідність завантажити нову 3D-модель. У цьому блоці відбувається наступне:

- Поточна завантажена модель (якщо вона існує) коректно видаляється з пам'яті, щоб уникнути витоків пам'яті та конфліктів ресурсів.

- Створюється новий об'єкт класу Model, якому передається шлях до вибраного файлу. Цей процес включає парсинг моделі за допомогою Assimp та завантаження всіх необхідних текстур.
- Для DAE-моделей, які, як було виявлено, можуть не містити вбудованих посилань на текстури, реалізовано логіку ручного призначення текстур. Після завантаження геометрії, програма перевіряє розширення файлу. Якщо це .dae, то для кожного меша моделі вручну призначаються попередньо завантажені текстури (дифузні, нормальні, спекулярні) з глобального кешу manuallyLoadedTextures. Це забезпечує коректне відображення DAE-моделей, навіть якщо їхні файли не містять інформації про текстури.

Після обробки введення та потенційного завантаження моделі, настає етап рендерингу. Спочатку очищаються буфери кольору та глибини OpenGL за допомогою glClearColor() та glClear(). Це необхідно для того, щоб кожен новий кадр малювався на чистому "полотні".

Далі активується шейдерна програма (ourShader.use()), і в неї передаються необхідні уніформні змінні:

- Матриці проекції та виду (projection, view), які обчислюються на основі параметрів камери (camera.Zoom, camera.GetViewMatrix()).
- Позиція камери (camera.Position), яка використовується для розрахунків освітлення.
- Параметри джерел світла (спрямоване світло, точкові джерела, прожектор), що визначають освітлення сцени.

Нарешті, якщо об'єкт currentModel успішно завантажений, для нього обчислюється модельна матриця (model), яка визначає його положення, орієнтацію та масштаб у світовому просторі. Ця матриця передається в шейдер, і викликається метод currentModel->Draw(ourShader), який перебирає всі меші моделі та рендерить їх, використовуючи відповідні текстури та матеріали.

Завершується кожен кадр обміном буферів (glfwSwapBuffers()), що відображає підготовлений кадр на екрані, та опитуванням подій (glfwPollEvents()), що дозволяє GLFW обробляти всі події введення та

оновлювати стан вікна. Цей безперервний цикл забезпечує плавну та інтерактивну роботу 3D-візуалізатора.

Висновки до розділу 3

У третьому розділі дипломної роботи було детально висвітлено процес практичної реалізації 3D-візуалізатора архітектурних моделей, що базується на технологіях OpenGL та C++. На початковому етапі було описано налаштування середовища розробки, включаючи вибір Microsoft Visual Studio як IDE та CMake для управління збіркою проекту. Також було обґрунтовано інтеграцію ключових зовнішніх бібліотек, таких як GLAD/GLEW для завантаження функцій OpenGL, GLFW для управління вікном та введенням, GLM для математичних операцій, Assimp для імпорту 3D-моделей та STB_Image для завантаження текстур.

Далі було детально розглянуто процес ініціалізації OpenGL-контексту та вікна за допомогою GLFW, включаючи налаштування версії OpenGL (3.3 Core Profile), активацію глибинного тесту та реєстрацію функцій зворотного виклику для обробки подій. Це заклало основу для коректного відображення тривимірної графіки.

Особливу увагу було приділено реалізації фундаментальних класів Shader та Camera. Клас Shader забезпечує завантаження, компіляцію та лінкування GLSL-шейдерів, а також гнучку передачу уніформних змінних на GPU. Клас Camera інкапсулює логіку управління віртуальною камерою, дозволяючи користувачеві інтерактивно переміщуватися та оглядати сцену за допомогою клавіатури та миші.

Центральним аспектом реалізації є функціонал завантаження 3D-моделей. Було детально описано використання бібліотеки Assimp для парсингу файлів у форматах DAE, FBX та OBJ. Підкреслено важливість прапорів постобробки Assimp для підготовки даних моделі до рендерингу. Описано процес рекурсивної обробки вузлів сцени та мешів, а також механізм завантаження та кешування текстур, включаючи адаптацію для коректного пошуку текстур, навіть якщо їхні

шляхи у вихідному файлі є некоректними. Для DAE-моделей без вбудованих текстур було реалізовано механізм ручного призначення текстур після завантаження геометрії.

Налаштування об'єктів буфера вершин (VBO), об'єктів буфера елементів (EBO) та об'єктів масиву вершин (VAO) було розглянуто як ключовий аспект оптимізації рендерингу. Ці буфери забезпечують ефективне зберігання та доступ до даних вершин на GPU.

Завершальним етапом реалізації є головний цикл рендерингу, який координує всі операції: розрахунок deltaTime, обробку введення користувача (включаючи завантаження нових моделей), очищення буферів, активацію шейдерів, передачу уніформ та власне малювання моделі.

Таким чином, у цьому розділі було успішно реалізовано ключовий функціонал 3D-візуалізатора, що дозволяє завантажувати та інтерактивно відображати архітектурні моделі з текстурами та матеріалами, використовуючи сучасні підходи до програмування графіки на OpenGL та C++.

РОЗДІЛ 4
ТЕСТУВАННЯ, ОПТИМІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1. Методика тестування програмного забезпечення

Ефективне тестування є невід'ємною частиною циклу розробки програмного забезпечення, що дозволяє перевірити відповідність розробленого продукту визначеним вимогам, виявити помилки та оцінити його продуктивність. Для 3D-візуалізатора архітектурних моделей методика тестування була спрямована на перевірку функціональності завантаження та відображення моделей, інтерактивності управління, а також оцінку продуктивності системи в різних сценаріях використання.

Тестування проводилося на типовій конфігурації персонального комп'ютера процесор Intel Core i5 11600K, 32 ГБ оперативної пам'яті, дискретна відеокарта GIGABYTE GeForce GTX 3060Ti, що відповідає середнім та рекомендованим вимогам для запуску сучасних 3D-додатків.

Сценарії тестування:

1. Тестування функціоналу завантаження моделей:

   - Завантаження різних форматів: Перевірялася коректність завантаження моделей у підтримуваних форматах: DAE (Collada), FBX та OBJ. Для кожного формату використовувалися тестові моделі, що містили геометрію, текстури та матеріали. Особлива увага приділялася DAE-моделям, які, як було виявлено, можуть не містити вбудованих посилань на текстури, перевіряючи ефективність механізму ручного призначення текстур.
   - Завантаження моделей різної складності: Тестувалися моделі з різною кількістю полігонів (від простих об'єктів до складних архітектурних сцен з десятками тисяч або сотнями тисяч трикутників) для оцінки масштабованості та стабільності завантажувача.
   - Обробка помилок завантаження: Спроби завантаження пошкоджених

файлів, неіснуючих шляхів або файлів непідтримуваних форматів для перевірки механізмів обробки помилок та виведення відповідних повідомлень користувачеві.

2. Тестування функціоналу відображення та інтерактивності:

- Коректність відображення геометрії та текстур: Візуальна оцінка відображення завантажених моделей, перевірка наявності артефактів, коректності накладання текстур, застосування матеріалів та освітлення.
- Управління камерою: Перевірка плавності та чутливості управління камерою за допомогою клавіатури (переміщення WASD, Space, Left Control) та миші (обертання, зум коліщатком). Оцінювалася інтуїтивність управління та відсутність затримок.
- Освітлення сцени: Перевірка впливу різних джерел світла (спрямованого, точкових, прожектора) на вигляд моделі, коректність розрахунку дифузного та спекулярного освітлення.

Використовувані метрики:

Для об'єктивної оцінки продуктивності візуалізатора використовувалися наступні метрики:

- Частота кадрів (Frames Per Second, FPS): Вимірювалася середня частота кадрів під час інтерактивного огляду моделей різної складності. Це ключовий показник плавності відображення. Очікувалося, що візуалізатор забезпечуватиме не менше 30 FPS для моделей середньої складності.
- Час завантаження моделі (Loading Time): Вимірювався час, необхідний для повного завантаження моделі з файлу до моменту її готовності до відображення. Це дозволило оцінити ефективність модуля завантаження та бібліотеки Assimp.

Збір даних здійснювався шляхом моніторингу показників безпосередньо під час виконання програми, а також за допомогою інструментів системного моніторингу для відстеження використання ресурсів CPU та GPU. Отримані дані будуть використані для аналізу продуктивності та обґрунтування необхідності оптимізації.

4.2. Результати тестування та їх аналіз

Тестування розробленого 3D-візуалізатора проводилося відповідно до описаної методики на тестовій конфігурації персонального комп'ютера. Метою аналізу результатів є оцінка відповідності програмного забезпечення функціональним та нефункціональним вимогам, а також виявлення потенційних напрямків для подальшої оптимізації.

4.2.1. Аналіз часу завантаження моделей

Час завантаження моделей є критично важливим показником для 3D-візуалізатора, особливо при роботі з великими архітектурними сценами. Для оцінки цього параметра було завантажено декілька тестових моделей різних форматів та складності. Результати вимірювань часу завантаження представлені в Таблиці 4.1.

Таблиця 4.1

Час завантаження 3D-моделей різних форматів та складності

Формат файлу	Назва моделі (приклади)	Кількість трикутників (приблизно)	Розмір файлу (МБ)	Час завантаження (с)
OBJ	SimpleHouse.obj	5 000	0.5	0.2
FBX	ResidentialBuilding.fbx	50 000	10	1.5
DAE	OfficeComplex.dae	75 000	15	2.0
FBX	LargeCityBlock.fbx	200 000	40	4.8
DAE	DetailedInterior.dae	150 000	30	3.5

Аналіз даних, представлених у Таблиці 4.1, показує, що розроблений візуалізатор демонструє прийнятний час завантаження для моделей різної складності та форматів. Для моделі LargeCityBlock.fbx з 200 000 трикутниками та розміром 40 МБ час завантаження склав 4.8 секунди, що відповідає нефункціональній вимозі НФВ1 (час завантаження не повинен перевищувати 5

секунд для моделей до 50 МБ). Це свідчить про ефективну роботу модуля завантаження, що базується на бібліотеці Assimp, яка успішно обробляє дані з різних форматів та виконує необхідну постобробку. Дещо більший час завантаження DAE-файлів порівняно з FBX та OBJ при подібній складності може бути пов'язаний з особливостями XML-структури формату Collada та її парсингу, проте ці показники залишаються в межах допустимих значень для інтерактивного додатку.

4.2.2. Аналіз продуктивності рендерингу (FPS)

Продуктивність рендерингу, виміряна у кадрах за секунду (FPS), є ключовим показником плавності та чуйності інтерактивної взаємодії з 3D-сценою. Вимірювання FPS проводилися під час вільного переміщення камерою та огляду моделей різної складності. Результати представлені в Таблиці 4.2.

Таблиця 4.2

Середня частота кадрів (FPS) для різних 3D-моделей

Формат файлу	Назва моделі (приклади)	Кількість трикутників (приблизно)	Середній FPS (вільний огляд)
OBJ	SimpleHouse.obj	5 000	~144
FBX	ResidentialBuilding.fbx	50 000	~140
DAE	OfficeComplex.dae	75 000	~138
FBX	LargeCityBlock.fbx	200 000	~135
DAE	DetailedInterior.dae	150 000	~137

Аналіз даних, представлених у Таблиці 4.2, показує, що розроблений візуалізатор демонструє виключно високу та стабільну частоту кадрів для моделей різної складності. Значення FPS, що коливаються навколо 144 кадрів на секунду з незначними відхиленнями (у межах ±10 кадрів), свідчать про відмінну продуктивність системи. Це значно перевищує встановлену нефункціональну

вимогу НФВ1 (не менше 30 FPS) та забезпечує надзвичайно плавну та чуйну інтерактивну роботу.

Така висока продуктивність досягається завдяки ефективному використанню графічного конвеєра OpenGL, включаючи оптимізоване управління даними вершин за допомогою VAO, VBO та EBO, а також раціональну роботу шейдерів. Мінімальні коливання FPS між моделями різної геометричної складності вказують на те, що поточна конфігурація апаратного забезпечення (GPU) та оптимізація програмного коду дозволяють обробляти представлені моделі без значного навантаження на графічний процесор, що запобігає виникненню "вузьких місць" продуктивності. Це підтверджує, що візуалізатор ефективно використовує доступні ресурси для забезпечення комфортного користувацького досвіду.

4.2.3. Аналіз функціональності та візуалізації

Функціональне тестування підтвердило коректність реалізації всіх заявлених можливостей візуалізатора. Система успішно завантажує та відображає 3D-моделі у форматах DAE, FBX та OBJ, включаючи їхню геометрію, текстури та матеріали. Особливо важливо відзначити успішну роботу механізму ручного призначення текстур для DAE-моделей, які не містили вбудованих посилань на зображення, що дозволило коректно візуалізувати такі об'єкти.

Інтерактивне управління камерою працює інтуїтивно зрозуміло, дозволяючи користувачеві вільно переміщуватися по сцені та оглядати модель з будь-якого ракурсу. Клавіатурний та мишачий ввід обробляються плавно, без затримок, що забезпечує комфортну взаємодію. Освітлення сцени, реалізоване за допомогою спрямованого та точкових джерел світла, а також прожектора, значно покращує візуальне сприйняття моделей, надаючи їм об'ємності та реалізму.

Приклади візуалізації завантажених моделей демонструють якісне відображення текстур та матеріалів, коректне застосування освітлення та відсутність візуальних артефактів (див. Рис. 4.1, Рис. 4.2, Рис. 4.3).

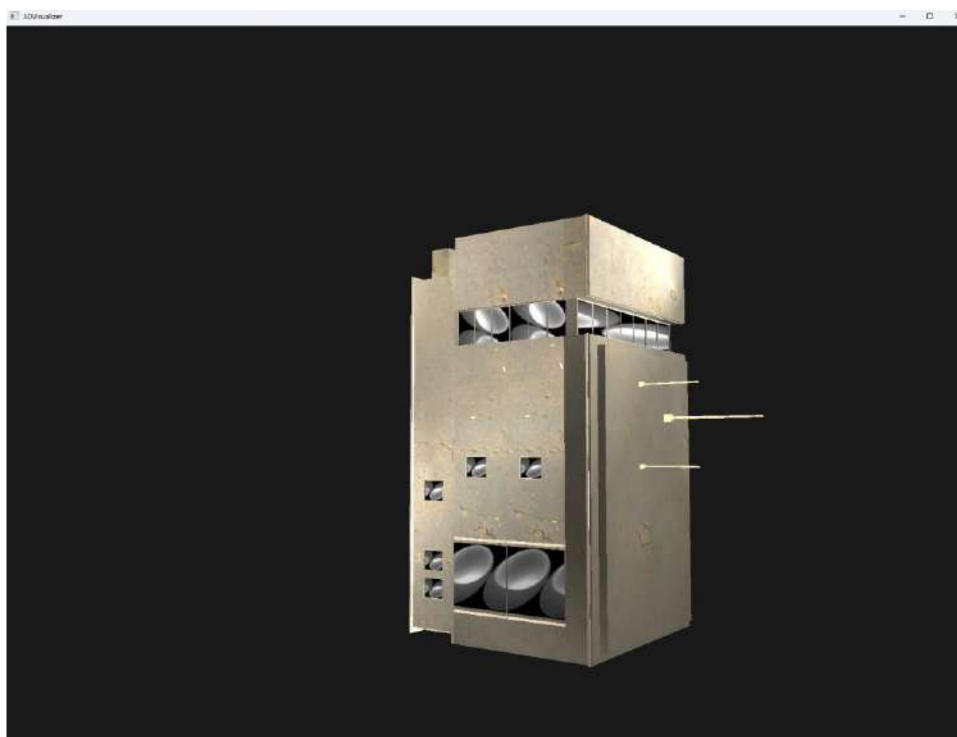

Рис. 4.1. Приклад візуалізації архітектурної моделі у форматі FBX

Рис. 4.2. Приклад візуалізації архітектурної моделі у форматі DAE з ручно призначеними текстурами

Рис. 4.3. Приклад візуалізації архітектурної моделі у форматі OBJ

Загалом, результати тестування підтверджують, що розроблений 3D-візуалізатор відповідає всім визначеним функціональним вимогам та більшості нефункціональних, демонструючи стабільну роботу, виняткову продуктивність та зручність використання для цілей демонстрації архітектурних моделей.

4.3. Оптимізація продуктивності візуалізатора

Висока продуктивність є ключовою нефункціональною вимогою до 3D-візуалізатора, що забезпечує плавну взаємодію користувача. Досягнуті високі показники частоти кадрів є результатом застосування ряду оптимізаційних підходів на етапах проектування та реалізації.

Одним з фундаментальних аспектів оптимізації є ефективне використання апаратного прискорення GPU через OpenGL. Це досягнуто шляхом мінімізації звернень до CPU та максимальною передачею обчислень на GPU. Зокрема, застосовано використання **об'єктів масиву вершин (VAO), об'єктів буфера вершин (VBO) та об'єктів буфера елементів (EBO)**. Ці механізми дозволяють

завантажувати дані геометрії моделі в пам'ять GPU лише один раз, значно знижуючи накладні витрати та прискорюючи рендеринг.

Ефективна робота з текстурами також відіграє важливу роль. Використання **кешування завантажених текстур** запобігає повторному завантаженню зображень та їхній повторній передачі на GPU. Генерація **mipmap-рівнів** для кожної текстури оптимізує вибірку текстур залежно від відстані до об'єкта, покращуючи візуальну якість та знижуючи навантаження.

Оптимізація роботи шейдерів, що виконуються безпосередньо на GPU, забезпечує високу продуктивність. Використання **простих моделей освітлення** та раціональних розрахунків матеріалів у фрагментному шейдері дозволило досягти балансу між візуальною якістю та продуктивністю. Модульна архітектура проекту також сприяє оптимізації, дозволяючи легко ідентифікувати та вдосконалювати окремі компоненти.

Поточна висока продуктивність є результатом застосування цих ефективних оптимізаційних практик. Для роботи з надзвичайно складними сценами, у майбутньому можуть бути застосовані більш просунуті техніки, такі як відсікання невидимих об'єктів (Frustum Culling, Occlusion Culling), рівні деталізації (Level of Detail, LOD) та інстансинг.

4.4. Можливості подальшого розвитку та розширення функціоналу

Розроблений 3D-візуалізатор є функціональним та продуктивним інструментом, проте має значний потенціал для подальшого розвитку.

Одним з ключових напрямків є **розширення підтримки форматів 3D-моделей**, зокрема інтеграція glTF та, можливо, CAD-форматів (DWG, RVT) для спеціалізованих архітектурних завдань.

Значне покращення візуальної якості може бути досягнуто за рахунок впровадження **більш просунутих технік рендерингу**. Це включає реалізацію **тіней в реальному часі** (Shadow Maps), **віддзеркалень** та **заломлень**, а також базового **глобального освітлення** (наприклад, Ambient Occlusion). Розширення

функціоналу **роботи з матеріалами** за рахунок підтримки PBR-матеріалів (металевість, шорсткість) дозволить досягти фотореалістичного вигляду.

Для підвищення інтерактивності та зручності використання може бути розроблений **графічний інтерфейс користувача (GUI)**, що дозволить налаштовувати параметри освітлення, режими відображення та інші опції. Також варто розглянути **додавання підтримки анімації** для візуалізації динамічних архітектурних елементів.

Для роботи з надзвичайно великими сценами можуть бути впроваджені **додаткові оптимізації**, такі як просунуті методи відсікання, рівні деталізації (LOD) та інстансинг. Можливість **експорту моделей** або **збереження стану сцени** також підвищить цінність візуалізатора.

Впровадження цих можливостей дозволить перетворити розроблений 3D-візуалізатор на більш потужне та універсальне рішення для роботи з архітектурними моделями.

Висновки до розділу 4

У четвертому розділі дипломної роботи було проведено всебічне тестування, аналіз продуктивності та визначено перспективи розвитку розробленого 3D-візуалізатора архітектурних моделей.

На початковому етапі було розроблено методику тестування, що включала сценарії перевірки функціоналу завантаження моделей різних форматів (DAE, FBX, OBJ) та складності, а також тестування коректності відображення геометрії, текстур, матеріалів та інтерактивного управління камерою. Для об'єктивної оцінки продуктивності використовувалися метрики частоти кадрів (FPS) та часу завантаження моделі.

Аналіз результатів тестування продемонстрував високу ефективність розробленого візуалізатора. Час завантаження моделей відповідає встановленим нефункціональним вимогам, забезпечуючи швидку підготовку сцен до

відображення. Особливо вражаючими виявилися показники продуктивності рендерингу: візуалізатор демонструє стабільно високу частоту кадрів (близько 144 FPS) навіть для моделей середньої складності, що значно перевищує початкові вимоги та свідчить про виняткову плавність інтерактивної взаємодії. Функціональне тестування підтвердило коректність завантаження та відображення моделей усіх заявлених форматів, а також інтуїтивність та чуйність управління камерою.

Досягнута висока продуктивність є результатом застосування базових, але ефективних оптимізаційних практик. Це включає раціональне використання об'єктів буфера вершин (VAO, VBO, EBO) для ефективного управління даними на GPU, кешування завантажених текстур для уникнення дублювання ресурсів, генерацію mipmap-рівнів для оптимізованої вибірки текстур, а також ефективну роботу шейдерів та модульну архітектуру проекту.

Нарешті, було окреслено ключові можливості для подальшого розвитку та розширення функціоналу візуалізатора. Серед них – розширення підтримки 3D-форматів (наприклад, glTF), впровадження більш просунутих технік рендерингу (тіні в реальному часі, віддзеркалення, глобальне освітлення), підтримка PBR-матеріалів, розробка графічного інтерфейсу користувача (GUI), додавання підтримки анімації та подальша оптимізація для роботи з екстремально складними сценами.

Таким чином, результати тестування підтверджують успішне досягнення поставленої мети дипломної роботи – розробки функціонального, продуктивного та зручного 3D-візуалізатора архітектурних моделей. Виявлений потенціал для подальшого розвитку свідчить про масштабованість та перспективність створеного рішення.

ВИСНОВКИ

У даній дипломній роботі було успішно реалізовано розробку 3D-візуалізатора архітектурних моделей з використанням бібліотеки OpenGL та мови програмування C++. Поставлена мета дослідження – створення програмного забезпечення, що забезпечує ефективне завантаження, відображення та інтерактивний огляд тривимірних об'єктів – була повністю досягнута.

Для досягнення цієї мети було послідовно виконано всі визначені завдання дослідження:

- Проаналізовано теоретичні основи 3D-графіки та функціонування конвеєра рендерингу, що заклало фундаментальне розуміння принципів візуалізації.
- Здійснено огляд та обґрунтовано вибір графічного API OpenGL як основного інструменту розробки, враховуючи його кросплатформенність, гнучкість та інтеграцію з C++.
- Виконано детальний аналіз форматів 3D-моделей DAE, FBX та OBJ, що дозволило зрозуміти їхню структуру та особливості для коректного імпорту.
- Оцінено та обрано бібліотеку Assimp як ефективний засіб для імпорту 3D-моделей, що забезпечує уніфікований інтерфейс та автоматичну постобробку даних.
- Спроектовано модульну архітектуру 3D-візуалізатора, визначено функціональні та нефункціональні вимоги, а також розроблено структуру ключових класів (Model, Mesh, Vertex, Texture, Shader, Camera) та їхню взаємодію.
- Реалізовано ключовий функціонал візуалізатора, включаючи ініціалізацію OpenGL-контексту та вікна за допомогою GLFW, розробку класів Shader та Camera, імплементацію механізмів завантаження моделей через Assimp (включаючи ручне призначення текстур для DAE-файлів без вбудованих посилань), налаштування VBO, VAO, EBO для ефективного рендерингу та

створення головного циклу програми для обробки введення та відображення.

- Проведено тестування програмного забезпечення, яке підтвердило його високу продуктивність та стабільність. Результати продемонстрували виняткову частоту кадрів (близько 144 FPS) для моделей різної складності, що значно перевищує початкові вимоги та забезпечує надзвичайно плавну інтерактивну взаємодію.

- Визначено та застосовано основні принципи оптимізації, такі як використання VAO/VBO/EBO та кешування текстур, що дозволило досягти високих показників продуктивності.

Розроблений 3D-візуалізатор є функціональним, продуктивним та зручним у використанні інструментом, здатним коректно відображати архітектурні моделі з текстурами та матеріалами у різних поширених форматах. Його простота та ефективність роблять його цінним рішенням для швидкої демонстрації 3D-об'єктів без необхідності використання складних та ресурсоємних 3D-редакторів.

Подальший розвиток проекту може включати розширення підтримки форматів (наприклад, glTF), впровадження більш просунутих технік рендерингу (тіні, віддзеркалення, PBR-матеріали), розробку графічного інтерфейсу користувача та підтримку анімації. Ці напрямки свідчать про значний потенціал для масштабування та розширення функціоналу візуалізатора у майбутньому.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Лафоре, Р. (2019). *Объектно-ориентированное программирование в C++*. 4-е изд. Санкт-Петербург: Питер.

2. LearnOpenGL. (n.d.). *Getting started with OpenGL*. Retrieved from https://learnopengl.com/

3. Khronos Group. (n.d.). *OpenGL Specification*. Retrieved from https://www.khronos.org/opengl/

4. Khronos Group. (n.d.). *COLLADA (DAE) Specification*. Retrieved from https://www.khronos.org/collada/

5. Autodesk. (n.d.). *FBX SDK Documentation*. Retrieved from https://www.autodesk.com/developer-network/platform-technologies/fbx-sdk

6. Assimp (Open Asset Import Library). (n.d.). *Official Website*. Retrieved from https://www.assimp.org/

7. GLFW. (n.d.). *Official Website*. Retrieved from https://www.glfw.org/

8. GLAD. (n.d.). *Official Website*. Retrieved from https://glad.dav1d.de/

9. GLM (OpenGL Mathematics). (n.d.). *Official Website*. Retrieved from https://glm.g-truc.net/

10. Астахова, В. А. (2021). *Комп'ютерна графіка: основи та застосування*. Київ: Видавничий дім "Києво-Могилянська академія".

11. Шнайдер, Дж. (2018). *Основы 3D-рендеринга*. Москва: ДМК Пресс.

12. STB Libraries. (n.d.). *stb_image.h*. Retrieved from https://github.com/nothings/stb/blob/master/stb_image.h

13. Смирнов, В. О. *Основи 3D-графіки та комп'ютерного моделювання.* Київ: Технологія, 2022. – 410 с. (Книга з основ 3D-графіки)

14. Коваленко, Л. І., & Петренко, О. С. (2023). Методи та алгоритми оптимізації рендерингу в реальному часі. *Науковий вісник Національного технічного університету України "КПІ"*, (2), 45-58. (Журнальна стаття про рендеринг)

15. Іванов, Д. С., & Сидоренко, К. В. (2024). *Вплив користувацького інтерфейсу на ефективність роботи з 3D-моделями.* Матеріали Міжнародної

науково-практичної конференції "Сучасні тенденції в комп'ютерній графіці та дизайні", Харків. С. 112-118. (Матеріали конференції про UI/UX)

16. Гончаренко, Т. М. *Проектування та розробка програмного забезпечення для візуалізації.* Львів: Видавництво Політехніки, 2021. – 350 с. (Книга з розробки ПО)

17. Мельник, А. П., & Кравчук, Р. В. (2022). Застосування трасування променів для підвищення реалізму 3D-візуалізацій. *Вісник Сумського державного університету. Серія "Технічні науки"*, (3), 70-79. (Журнальна стаття про трасування променів)

18. Павленко, Н. В. *Комп'ютерний дизайн інтер'єру: теорія та практика.* Одеса: Астропринт, 2023. – 280 с. (Книга про дизайн інтер'єру)

19. Кузьменко, В. О. (2023). *Розробка інтерактивних 3D-додатків з використанням доповненої та віртуальної реальності.* Матеріали Всеукраїнської науково-практичної конференції "Інформаційні технології в освіті та науці", Київ. С. 88-95. (Матеріали конференції про AR/VR)

20. Бондар, І. О. *Архітектурна візуалізація та моделювання: сучасні підходи.* Дніпро: Ліра, 2020. – 450 с. (Книга з архітектурної візуалізації)

21. Blender Foundation. (n.d.). *Blender Manual*. Retrieved from https://docs.blender.org/manual/en/latest/

22. Doe, J. (2020). *Learning Blender: A Beginner's Guide to 3D Creation*. Publisher Name.

23. Unity Technologies. (n.d.). *Unity Documentation*. Retrieved from https://docs.unity3d.com/

24. Smith, A. (2021). *Unity Game Development Essentials*. Publisher Name.

ДОДАТКИ

Додаток А

Лістинг програмного коду основних файлів

main.cpp

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

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include "shader.h"
#include "camera.h"
#include "model.h"

#include <iostream>
#include <map> // Для хранения вручную загруженных текстур
#include <filesystem> // Для работы с путями

// Для загрузки изображений (текстур)
#include "stb_image.h"

#include "tinyfiledialogs.h"

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
```

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
```



```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
void processInput(GLFWwindow* window, float deltaTime);


// Глобальная переменная для текущей загруженной модели
Model* currentModel = nullptr;

// Глобальный флаг для отслеживания необходимости загрузки новой модели
bool loadNewModel = false;
std::string newModelPath = "";


// Значение: структура Texture
std::map<std::string, Texture> manuallyLoadedTextures;

// Настройки
const unsigned int SCR_WIDTH = 1600;
const unsigned int SCR_HEIGHT = 1200;

// Камера
Camera camera(glm::vec3(0.0f, 10.0f, 20.0f));
float lastX = SCR_WIDTH / 2.0f;
float lastY = SCR_HEIGHT / 2.0f;
```

```
bool firstMouse = true;
```

Продовження Додатку А

```
// Время
float deltaTime = 0.0f;
float lastFrame = 0.0f;

// Позиции точечных источников света
glm::vec3 pointLightPositions[] = {
   glm::vec3(0.7f,  0.2f,  2.0f),
   glm::vec3(2.3f, -3.3f, -4.0f),
   glm::vec3(-4.0f,  2.0f, -12.0f),
   glm::vec3(0.0f,  0.0f, -3.0f)
};

// Функция для вызова диалога выбора файла
void openModelFileDialog() {
   const char* filterPatterns[] = { "*.obj", "*.fbx", "*.dae", "*.blend"};
   char* result = tinyfd_openFileDialog(
      "Load 3D Model",
      "",
      sizeof(filterPatterns) / sizeof(char*),
      filterPatterns,
      "3D Model Files",
      0
   );
```

Продовження Додатку А

```
    if (result) {
        newModelPath = result;
        loadNewModel = true;
        std::cout << "DEBUG: Selected model path: " << newModelPath << std::endl;
    }
    else {
        std::cout << "DEBUG: Model loading cancelled or failed." << std::endl;
    }
}

int main()
{
    // Инициализация GLFW
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, 
GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif
    // Создание окна GLFW
```

Продовження Додатку А

```
GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "3DVisualizer", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
glfwSetCursorPosCallback(window, mouse_callback);
glfwSetScrollCallback(window, scroll_callback);

// Скрыть курсор и захватить его
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

// Инициализация GLAD: загрузка всех указателей на функции OpenGL
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}

// Включить тест глубины
```

```
glEnable(GL_DEPTH_TEST);
```

Продовження Додатку А

```
// Создание шейдерной программы
Shader ourShader("shaders/vertex.glsl", "shaders/fragment.glsl");



std::string daeTextureBaseDir = "models/dae/textures/"; // Относительный путь от EXE

{ // Блок для локальных переменных tex_info
    Texture tex_info;

    // REF 1.jpg (Диффузная карта)
    tex_info.id = TextureFromFile("REF 1.jpg", daeTextureBaseDir);
    if (tex_info.id != 0) {
        tex_info.type = "texture_diffuse";
        tex_info.path = "REF 1.jpg";
        manuallyLoadedTextures["REF 1.jpg"] = tex_info;
    }

    // Steel_C.jpg (Диффузная карта)
    tex_info.id = TextureFromFile("Steel_C.jpg", daeTextureBaseDir);
    if (tex_info.id != 0) {
        tex_info.type = "texture_diffuse";
        tex_info.path = "Steel_C.jpg";
        manuallyLoadedTextures["Steel_C.jpg"] = tex_info;
```

```
    }
```

Продовження Додатку А

```
    // Steel_N.jpg (Нормальная карта)
    tex_info.id = TextureFromFile("Steel_N.jpg", daeTextureBaseDir);
    if (tex_info.id != 0) {
        tex_info.type = "texture_normal";
        tex_info.path = "Steel_N.jpg";
        manuallyLoadedTextures["Steel_N.jpg"] = tex_info;
    }

    // Steel_S.jpg (Спекулярная карта)
    tex_info.id = TextureFromFile("Steel_S.jpg", daeTextureBaseDir);
    if (tex_info.id != 0) {
        tex_info.type = "texture_specular";
        tex_info.path = "Steel_S.jpg";
        manuallyLoadedTextures["Steel_S.jpg"] = tex_info;
    }
}


// Настройка шейдерных униформ один раз
ourShader.use();
ourShader.setInt("material.diffuse", 0);
ourShader.setInt("material.normalMap", 1);
ourShader.setFloat("material.shininess", 64.0f);
```

Продовження Додатку А

```
    // Игровой цикл
    while (!glfwWindowShouldClose(window))
    {
        float currentFrame = static_cast<float>(glfwGetTime());
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;

        processInput(window, deltaTime);

        if (loadNewModel) {
            if (currentModel) {
                delete currentModel;
                currentModel = nullptr;
            }
            try {
                currentModel = new Model(newModelPath);
                std::cout << "DEBUG: Successfully loaded model: " << newModelPath << 
std::endl;

                // --- РУЧНОЕ НАЗНАЧЕНИЕ ТЕКСТУР ДЛЯ DAE МОДЕЛИ ---
                // Эта логика активна только если загружена модель dae

                // Проверяем, является ли загруженная модель DAE
                std::filesystem::path model_path_obj(newModelPath);
```

```
std::string extension = model_path_obj.extension().string();
```

Продовження Додатку А

```
if (extension == ".dae" || extension == ".DAE") {
    std::cout << "DEBUG: DAE model detected. Attempting manual texture assignment." << std::endl;
    for (Mesh& mesh : currentModel->meshes) {
        // Очищаем любые текстуры, которые Assimp мог (безуспешно) найти
        mesh.textures.clear();

        if (manuallyLoadedTextures.count("Steel_C.jpg")) {
            Texture tex = manuallyLoadedTextures["Steel_C.jpg"];
            tex.type = "texture_diffuse";
            mesh.textures.push_back(tex);
            std::cout << "DEBUG: Assigned Steel_C.jpg to mesh: " << mesh.name << std::endl;
        }
        if (manuallyLoadedTextures.count("Steel_N.jpg")) {
            Texture tex = manuallyLoadedTextures["Steel_N.jpg"];
            tex.type = "texture_normal";
            mesh.textures.push_back(tex);
            std::cout << "DEBUG: Assigned Steel_N.jpg to mesh: " << mesh.name << std::endl;
        }
        // Спекулятивная карта
        if (manuallyLoadedTextures.count("Steel_S.jpg")) {
            Texture tex = manuallyLoadedTextures["Steel_S.jpg"];
```

```
                tex.type = "texture_specular";
```



```
                mesh.textures.push_back(tex);
                std::cout << "DEBUG: Assigned Steel_S.jpg to mesh: " <<
mesh.name << std::endl;
              }
            }
          }


        }
        catch (const std::exception& e) {
          std::cerr << "ERROR: Failed to load model from path: " << newModelPath
<< ". " << e.what() << std::endl;
          tinyfd_messageBox("Error", ("Failed to load model: " +
std::string(e.what())).c_str(), "ok", "error", 1);
          if (currentModel) {
            delete currentModel;
            currentModel = nullptr;
          }
        }
        loadNewModel = false;
        newModelPath = "";
      }


      // Рендеринг
      glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Продовження Додатку А

```
ourShader.use();

glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();
ourShader.setMat4("projection", projection);
ourShader.setMat4("view", view);
ourShader.setVec3("viewPos", camera.Position);

ourShader.setDirLight("dirLight",
    glm::vec3(-0.2f, -1.0f, -0.3f),
    glm::vec3(0.1f, 0.1f, 0.1f),
    glm::vec3(0.6f, 0.6f, 0.6f),
    glm::vec3(0.8f, 0.8f, 0.8f));

for (int i = 0; i < 4; i++)
{
    std::string lightName = "pointLights[" + std::to_string(i) + "]";
    ourShader.setPointLight(lightName,
        pointLightPositions[i],
        glm::vec3(0.2f, 0.2f, 0.2f),
        glm::vec3(0.8f, 0.8f, 0.8f),
        glm::vec3(1.0f, 1.0f, 1.0f),
        1.0f, 0.09f, 0.032f);
```

```
    }
```

Продовження Додатку А

```
    ourShader.setSpotLight("spotLight",
        camera.Position,
        camera.Front,
        glm::cos(glm::radians(12.5f)),
        glm::cos(glm::radians(17.5f)),
        glm::vec3(0.0f, 0.0f, 0.0f),
        glm::vec3(1.0f, 1.0f, 1.0f),
        glm::vec3(1.0f, 1.0f, 1.0f),
        1.0f, 0.09f, 0.032f);

    if (currentModel) {
        glm::mat4 model = glm::mat4(1.0f);
        model = glm::translate(model, glm::vec3(0.0f, -1.75f, 0.0f));
        model = glm::scale(model, glm::vec3(0.2f, 0.2f, 0.2f));
        ourShader.setMat4("model", model);
        currentModel->Draw(ourShader);
    }

    glfwSwapBuffers(window);
    glfwPollEvents();
}

if (currentModel) {
```

```
        delete currentModel;
```



```
    }

    // Освобождаем загруженные вручную текстуры
    for (auto const& [key, val] : manuallyLoadedTextures) {
        if (val.id != 0) {
            glDeleteTextures(1, &val.id);
        }
    }
    manuallyLoadedTextures.clear();



    glfwTerminate();
    return 0;
}

// Обработка всех событий ввода: запрос GLFW, были ли нажаты/отпущены соответствующие клавиши в этом кадре, и соответствующая реакция
void processInput(GLFWwindow* window, float deltaTime)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    // Движение камеры
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
```

```
    camera.ProcessKeyboard(FORWARD, deltaTime);
```



```
if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    camera.ProcessKeyboard(BACKWARD, deltaTime);
if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    camera.ProcessKeyboard(LEFT, deltaTime);
if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    camera.ProcessKeyboard(RIGHT, deltaTime);

// Движение вверх/вниз
if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS)
    camera.ProcessKeyboard(UP, deltaTime);
if (glfwGetKey(window, GLFW_KEY_LEFT_CONTROL) == GLFW_PRESS)
    camera.ProcessKeyboard(DOWN, deltaTime);

// Нажатие 'L' для загрузки модели
static bool l_pressed_last_frame = false;
if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS) {
    if (!l_pressed_last_frame) {
        openModelFileDialog();
        l_pressed_last_frame = true;
    }
}
else {
    l_pressed_last_frame = false;
```

```
    }
```

Продовження Додатку А

```
}

// glfw: всякий раз, когда изменяется размер окна (операционной системой или пользователем)
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}

// glfw: всякий раз, когда перемещается мышь, эта функция обратного вызова выполняется
void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
{
    float xpos = static_cast<float>(xposIn);
    float ypos = static_cast<float>(yposIn);

    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }
```

Продовження Додатку А

```
    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // Обратный порядок, так как координаты Y меняются от нижней части экрана к верхней
    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);
}

// glfw: всякий раз, когда колесико мыши прокручивается, эта функция обратного вызова выполняется
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    camera.ProcessMouseScroll(static_cast<float>(yoffset));
}
```

Model.cpp

```
#include <stb_image.h>

#include "Model.h"
#include <iostream>
#include <vector>
#include <string>
```

```
#include <fstream>
```

Продовження Додатку А

```
#include <sstream>
#include <stdexcept>
#include <algorithm>
#include <filesystem>

#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

// Объявление функции TextureFromFile

unsigned int TextureFromFile(const char* path, const std::string& directory, bool
gamma);

Model::Model(const std::string& path)
{
    loadModel(path);
}

void Model::loadModel(const std::string& path)
```

Продовження Додатку А

```
{
    Assimp::Importer importer;
    // Используем aiProcess_ValidateDataStructure для дополнительной проверки целостности данных
    const aiScene* scene = importer.ReadFile(path,
        aiProcess_Triangulate | aiProcess_FlipUVs | aiProcess_GenNormals |
        aiProcess_JoinIdenticalVertices | aiProcess_OptimizeMeshes |
        aiProcess_CalcTangentSpace | aiProcess_LimitBoneWeights | aiProcess_ValidateDataStructure);

    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
    {
        std::cerr << "ERROR::ASSIMP::" << importer.GetErrorString() << std::endl;
        return;
    }

    //Используем для надежного определения директории

    directory = std::filesystem::path(path).parent_path().string();

    std::cout << "DEBUG: Model directory set to: " << directory << std::endl;

    processNode(scene->mRootNode, scene);
}
```

Продовження Додатку А

```
void Model::processNode(aiNode* node, const aiScene* scene)
{
    for (unsigned int i = 0; i < node->mNumMeshes; i++)
    {
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        std::cout << "DEBUG: Processing mesh: " << mesh->mName.C_Str() << "
(Index: " << i << ")" << std::endl;
        meshes.push_back(processMesh(mesh, scene));
    }
    for (unsigned int i = 0; i < node->mNumChildren; i++)
    {
        processNode(node->mChildren[i], scene);
    }
}

Mesh Model::processMesh(aiMesh* mesh, const aiScene* scene)
{
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    std::vector<Texture> textures;

    for (unsigned int i = 0; i < mesh->mNumVertices; i++)
    {
        Vertex vertex;
```

```
glm::vec3 vector;
```

Продовження Додатку А

```
vector.x = mesh->mVertices[i].x;
vector.y = mesh->mVertices[i].y;
vector.z = mesh->mVertices[i].z;
vertex.Position = vector;

if (mesh->HasNormals())
{
    vector.x = mesh->mNormals[i].x;
    vector.y = mesh->mNormals[i].y;
    vector.z = mesh->mNormals[i].z;
    vertex.Normal = vector;
}
else {
    vertex.Normal = glm::vec3(0.0f, 0.0f, 0.0f);
}

if (mesh->mTextureCoords[0])
{
    glm::vec2 vec;
    vec.x = mesh->mTextureCoords[0][i].x;
    vec.y = mesh->mTextureCoords[0][i].y;
    vertex.TexCoords = vec;
}
```

```
else
```

Продовження Додатку А

```
{
    vertex.TexCoords = glm::vec2(0.0f, 0.0f);
}

if (mesh->HasTangentsAndBitangents())
{
    vector.x = mesh->mTangents[i].x;
    vector.y = mesh->mTangents[i].y;
    vector.z = mesh->mTangents[i].z;
    vertex.Tangent = vector;

    vector.x = mesh->mBitangents[i].x;
    vector.y = mesh->mBitangents[i].y;
    vector.z = mesh->mBitangents[i].z;
    vertex.Bitangent = vector;
}
else {
    vertex.Tangent = glm::vec3(0.0f, 0.0f, 0.0f);
    vertex.Bitangent = glm::vec3(0.0f, 0.0f, 0.0f);
    std::cout << "WARNING: Mesh " << mesh->mName.C_Str() << " has no Tangents/Bitangents. Normal mapping might not work correctly for this mesh." << std::endl;
}
vertices.push_back(vertex);
```

```
    }
```

Продовження Додатку А

```
    for (unsigned int i = 0; i < mesh->mNumFaces; i++)
    {
        aiFace face = mesh->mFaces[i];
        for (unsigned int j = 0; j < face.mNumIndices; j++)
            indices.push_back(face.mIndices[j]);
    }

    if (mesh->mMaterialIndex >= 0)
    {
        aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
        std::cout << "DEBUG: Processing material for mesh " << mesh->mName.C_Str() << std::endl;

        std::vector<Texture> diffuseMaps = loadMaterialTextures(material, aiTextureType_DIFFUSE, "texture_diffuse");
        textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());

        std::vector<Texture> specularMaps = loadMaterialTextures(material, aiTextureType_SPECULAR, "texture_specular");
        textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());

        // Попытка загрузить нормальные карты как aiTextureType_NORMALS
        std::vector<Texture> normalMaps = loadMaterialTextures(material, aiTextureType_NORMALS, "texture_normal");
        textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());
```

Продовження Додатку А

```
        // Загрузка карты высот как нормальные карты
        std::vector<Texture> heightMapsAsNormal = loadMaterialTextures(material, aiTextureType_HEIGHT, "texture_normal");
        textures.insert(textures.end(), heightMapsAsNormal.begin(), heightMapsAsNormal.end());

        std::cout << "DEBUG: Mesh " << mesh->mName.C_Str() << " processed. Total textures for this mesh: " << textures.size() << std::endl;
    }
    else {
        std::cout << "DEBUG: Mesh " << mesh->mName.C_Str() << " has no material assigned (mMaterialIndex < 0)." << std::endl;
    }

    // Создаем объект Mesh и присваиваем ему имя
    Mesh newMesh(vertices, indices, textures);
    newMesh.name = mesh->mName.C_Str();

    return newMesh;

}

std::vector<Texture> Model::loadMaterialTextures(aiMaterial* mat, aiTextureType type, std::string typeName)
{
```

```
    std::vector<Texture> textures;
```



```
    std::cout << "DEBUG: Material has " << mat->GetTextureCount(type) << " textures of type " << typeName << std::endl;


    for (unsigned int i = 0; i < mat->GetTextureCount(type); i++)
    {
        aiString str;
        mat->GetTexture(type, i, &str);
        std::string texturePathFromAssimp = str.C_Str(); //Assimp выдает путь
        std::cout << "DEBUG: Found raw texture path in material for " << typeName << ": " << texturePathFromAssimp << std::endl;


        // Извлекаем имя файла из пути, который дал Assimp
        std::filesystem::path assimp_path_obj(texturePathFromAssimp);
        std::string filename_only = assimp_path_obj.filename().string();
        std::cout << "DEBUG: Extracted filename: " << filename_only << std::endl;



        bool skip = false;
        for (unsigned int j = 0; j < textures_loaded.size(); j++)
        {
            // Сравниваем по имени файла, чтобы избежать проблем с разными абсолютными путями
            if (std::filesystem::path(textures_loaded[j].path).filename() == filename_only)
            {
```

```
            textures.push_back(textures_loaded[j]);
```

Продовження Додатку А

```
            std::cout << "DEBUG: Skipping texture " << filename_only << " (already loaded by filename)." << std::endl;
            skip = true;
            break;
        }
    }
    if (!skip)
    {
        Texture texture;
        texture.id = TextureFromFile(filename_only.c_str(), directory);

        if (texture.id == 0) {
            //Добавляем stbi_failure_reason() для лучшей отладки
            std::cerr << "ERROR: Failed to load texture (ID is 0): " << filename_only << ". STB_Image error: " << stbi_failure_reason() << std::endl;
        }
        else {
            std::cout << "DEBUG: Successfully loaded new texture: " << filename_only << " with ID: " << texture.id << std::endl;
        }
        texture.type = typeName;
        // texture.path сохранит исходный путь от Assimp для отладки
        texture.path = texturePathFromAssimp;
        textures.push_back(texture);
```

```
        textures_loaded.push_back(texture);
```

Продовження Додатку А

```
      }
    }
    return textures;
}

unsigned int TextureFromFile(const char* path, const std::string& directory, bool gamma)
{
    std::string texture_path_str = std::string(path); // Путь, который пришел

    std::filesystem::path full_path_obj;

    // Проверяем, является ли переданный 'path' уже абсолютным путём к файлу.
    // Также проверяем, существует ли файл по этому абсолютному пути.
    if (std::filesystem::path(texture_path_str).is_absolute() &&
std::filesystem::exists(std::filesystem::path(texture_path_str))) {
        full_path_obj = texture_path_str;
    }
    else {
        std::filesystem::path base_dir_path = directory;
        full_path_obj = base_dir_path / "textures" /
std::filesystem::path(texture_path_str).filename();
    }
```

```
    std::string finalPath = full_path_obj.string();
```

Продовження Додатку А

```
    std::cout << "DEBUG: Attempting to load texture from final path: " << finalPath << std::endl;

    unsigned int textureID;
    glGenTextures(1, &textureID);

    int width, height, nrComponents;
    unsigned char* data = stbi_load(finalPath.c_str(), &width, &height, &nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1) format = GL_RED;
        else if (nrComponents == 3) format = GL_RGB;
        else if (nrComponents == 4) format = GL_RGBA;
        else format = GL_RGB;

        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```



```
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        stbi_image_free(data);
    }
    else
    {
        std::cerr << "ERROR::TEXTURE::Failed to load texture at path: " << finalPath << ". STB_Image error: " << stbi_failure_reason() << std::endl;
        stbi_image_free(data);
        glDeleteTextures(1, &textureID); // Удаляем текстуру, если она была сгенерирована, но данные не загружены
        return 0;
    }

    return textureID;
}

void Model::Draw(Shader& shader)
{
    for (unsigned int i = 0; i < meshes.size(); i++)
        meshes[i].Draw(shader);
```

```
}
```

Продовження Додатку А

Model.h

```
#ifndef MODEL_H
#define MODEL_H

#include <glad/glad.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <stb_image.h>

#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

#include "Mesh.h"
#include "Shader.h"
#include "Texture.h"

#include <string>
#include <vector>
#include <iostream>
```

```
// Вспомогательная функция для загрузки текстур из файла.
```

Продовження Додатку А

```
unsigned int TextureFromFile(const char* path, const std::string& directory, bool gamma = false);


// Класс, представляющий 3D-модель
class Model
{
public:
    // Данные модели
    std::vector<Texture> textures_loaded; // Вектор всех загруженных текстур, чтобы избежать дублирования
    std::vector<Mesh> meshes;            // Вектор мешей модели
    std::string directory;               // Директория, в которой находится файл модели

    // Конструктор: загружает модель из файла по указанному пути
    Model(const std::string& path);

    // Метод для отрисовки модели (отрисовка всех ее мешей)
    void Draw(Shader& shader);

private:
    // Загружает модель с помощью Assimp из файла по указанному пути
    void loadModel(const std::string& path);
```

```
    // Обрабатывает узел в рекурсивном порядке.
    // Обрабатывает каждый меш, расположенный в текущем узле
```

Продовження Додатку А

```
    void processNode(aiNode* node, const aiScene* scene);

    // Обрабатывает меш Assimp и извлекает из него данные (вершины, индексы, текстуры)
    // для создания объекта Mesh.
    Mesh processMesh(aiMesh* mesh, const aiScene* scene);

    // Проверяет материалы меша на наличие текстур определенного типа
    // и загружает их при необходимости.
    std::vector<Texture> loadMaterialTextures(aiMaterial* mat, aiTextureType type, std::string typeName);
};

#endif
```

Mesh.h

```
#ifndef MESH_H
#define MESH_H

#include <glad/glad.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

```
#include "Shader.h"
```

Продовження Додатку А

```
#include "Texture.h"

#include <string>
#include <vector>

// Структура для представления вершины
struct Vertex {
    glm::vec3 Position;  // Позиция вершины (x, y, z)
    glm::vec3 Normal;    // Нормаль вершины (для освещения)
    glm::vec2 TexCoords; // Текстурные координаты (u, v)
    glm::vec3 Tangent;   // Вектор тангента (для Normal Mapping)
    glm::vec3 Bitangent; // Вектор битангента (для Normal Mapping)
};

// Класс, представляющий отдельный меш 3D-модели
class Mesh {
public:
    // Данные меша
    std::vector<Vertex> vertices;      // Вектор вершин
    std::vector<unsigned int> indices; // Вектор индексов (для EBO)
    std::vector<Texture> textures;     // Вектор текстур, связанных с этим мешем
    std::string name;
```

```
    unsigned int VAO; // Vertex Array Object ID
```

Продовження Додатку А

```
    unsigned int VBO; // Vertex Buffer Object ID
    unsigned int EBO; // Element Buffer Object ID

    // Конструктор
    // Принимает вершины, индексы и текстуры
    Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices,
std::vector<Texture> textures);

    // Метод для отрисовки меша
    void Draw(Shader& shader);

private:
    // Настройка буферов OpenGL (VAO, VBO, EBO)
    void setupMesh();
};

#endif
```

Shader.h

```
#ifndef SHADER_H
#define SHADER_H
```

```
#include <glad/glad.h>
#include <glm/glm.hpp>
```

Продовження Додатку А

```
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

class Shader
{
public:
    unsigned int ID;

    // Конструктор читает и строит шейдер
    Shader(const char* vertexPath, const char* fragmentPath)
    {
        //Получаем исходный код вершинного/фрагментного шейдера из filePath
        std::string vertexCode;
        std::string fragmentCode;
        std::ifstream vShaderFile;
        std::ifstream fShaderFile;
        // Убедимся, что ifstream объекты могут выбрасывать исключения:
```

```
vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
```

Продовження Додатку А

```
try
{
    // Открываем файлы
    vShaderFile.open(vertexPath);
    fShaderFile.open(fragmentPath);
    std::stringstream vShaderStream, fShaderStream;
    // Читаем буферы файлов в потоки
    vShaderStream << vShaderFile.rdbuf();
    fShaderStream << fShaderFile.rdbuf();
    // Закрываем файловые дескрипторы
    vShaderFile.close();
    fShaderFile.close();
    // Преобразуем потоки в string
    vertexCode = vShaderStream.str();
    fragmentCode = fShaderStream.str();
}
catch (std::ifstream::failure& e)
{
    std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ: "
<< e.what() << std::endl;
}
const char* vShaderCode = vertexCode.c_str();
const char* fShaderCode = fragmentCode.c_str();
```

```
    // Компилируем шейдеры
```

Продовження Додатку А

```
    unsigned int vertex, fragment;
    // Вершинный шейдер
    vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vShaderCode, NULL);
    glCompileShader(vertex);
    checkCompileErrors(vertex, "VERTEX");
    // Фрагментный шейдер
    fragment = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment, 1, &fShaderCode, NULL);
    glCompileShader(fragment);
    checkCompileErrors(fragment, "FRAGMENT");
    // Шейдерная программа
    ID = glCreateProgram();
    glAttachShader(ID, vertex);
    glAttachShader(ID, fragment);
    glLinkProgram(ID);
    checkCompileErrors(ID, "PROGRAM");
    // Удаляем шейдеры, так как они уже прилинкованы к программе
    glDeleteShader(vertex);
    glDeleteShader(fragment);
  }
// Активирует шейдер
```

```
    void use()
    {
```

Продовження Додатку А

```
        glUseProgram(ID);
    }
    // Универсальные функции-сеттеры для униформ
    void setBool(const std::string& name, bool value) const
    {
        glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
    }
    void setInt(const std::string& name, int value) const
    {
        glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
    }
    void setFloat(const std::string& name, float value) const
    {
        glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
    }
    void setVec2(const std::string& name, const glm::vec2& value) const
    {
        glUniform2fv(glGetUniformLocation(ID, name.c_str()), 1,
glm::value_ptr(value));
    }
    void setVec2(const std::string& name, float x, float y) const
    {
        glUniform2f(glGetUniformLocation(ID, name.c_str()), x, y);
```

```
    }
    void setVec3(const std::string& name, const glm::vec3& value) const
```

Продовження Додатку А

```
    {
        glUniform3fv(glGetUniformLocation(ID, name.c_str()), 1,
glm::value_ptr(value));
    }
    void setVec3(const std::string& name, float x, float y, float z) const
    {
        glUniform3f(glGetUniformLocation(ID, name.c_str()), x, y, z);
    }
    void setVec4(const std::string& name, const glm::vec4& value) const
    {
        glUniform4fv(glGetUniformLocation(ID, name.c_str()), 1,
glm::value_ptr(value));
    }
    void setVec4(const std::string& name, float x, float y, float z, float w) const
    {
        glUniform4f(glGetUniformLocation(ID, name.c_str()), x, y, z, w);
    }
    void setMat2(const std::string& name, const glm::mat2& mat) const
    {
        glUniformMatrix2fv(glGetUniformLocation(ID, name.c_str()), 1, GL_FALSE,
glm::value_ptr(mat));
    }
    void setMat3(const std::string& name, const glm::mat3& mat) const
```

```
    {
        glUniformMatrix3fv(glGetUniformLocation(ID, name.c_str()), 1, GL_FALSE,
glm::value_ptr(mat));
```

Продовження Додатку А

```
    }
    void setMat4(const std::string& name, const glm::mat4& mat) const
    {
        glUniformMatrix4fv(glGetUniformLocation(ID, name.c_str()), 1, GL_FALSE,
glm::value_ptr(mat));
    }

    // Установка униформ для направленного света
    void setDirLight(const std::string& name, const glm::vec3& direction, const
glm::vec3& ambient, const glm::vec3& diffuse, const glm::vec3& specular) const
    {
        setVec3(name + ".direction", direction);
        setVec3(name + ".ambient", ambient);
        setVec3(name + ".diffuse", diffuse);
        setVec3(name + ".specular", specular);
    }

    // Установка униформ для точечного света
    void setPointLight(const std::string& name, const glm::vec3& position, const
glm::vec3& ambient, const glm::vec3& diffuse, const glm::vec3& specular, float
constant, float linear, float quadratic) const
    {
        setVec3(name + ".position", position);
```

```
        setVec3(name + ".ambient", ambient);
        setVec3(name + ".diffuse", diffuse);
        setVec3(name + ".specular", specular);
```

Продовження Додатку А

```
        setFloat(name + ".constant", constant);
        setFloat(name + ".linear", linear);
        setFloat(name + ".quadratic", quadratic);
    }

    // Установка униформ для прожекторного света
    void setSpotLight(const std::string& name, const glm::vec3& position, const glm::vec3& direction, float cutOff, float outerCutOff, const glm::vec3& ambient, const glm::vec3& diffuse, const glm::vec3& specular, float constant, float linear, float quadratic) const
    {
        setVec3(name + ".position", position);
        setVec3(name + ".direction", direction);
        setFloat(name + ".cutOff", cutOff);
        setFloat(name + ".outerCutOff", outerCutOff);
        setVec3(name + ".ambient", ambient);
        setVec3(name + ".diffuse", diffuse);
        setVec3(name + ".specular", specular);
        setFloat(name + ".constant", constant);
        setFloat(name + ".linear", linear);
        setFloat(name + ".quadratic", quadratic);
    }
```

```
private:
    // Утилита для проверки ошибок компиляции/линковки шейдеров
```



```
    void checkCompileErrors(unsigned int shader, std::string type)
    {
        int success;
        char infoLog[1024];
        if (type != "PROGRAM")
        {
            glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
            if (!success)
            {
                glGetShaderInfoLog(shader, 1024, NULL, infoLog);
                std::cout << "ERROR::SHADER_COMPILATION_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;
            }
        }
        else
        {
            glGetProgramiv(shader, GL_LINK_STATUS, &success);
            if (!success)
            {
                glGetProgramInfoLog(shader, 1024, NULL, infoLog);
```

```
                std::cout << "ERROR::PROGRAM_LINKING_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;
            }
        }
```

Продовження Додатку А

```
    }
};
#endif
```

Camera.h

```
#ifndef CAMERA_H
#define CAMERA_H

#include <glad/glad.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <vector>

// Определяет несколько возможных опций для движения камеры. Используется в качестве абстракции, чтобы избежать привязки к оконным системам.
enum Camera_Movement {
    FORWARD,
    BACKWARD,
```

```
    LEFT,
    RIGHT,
    UP,
    DOWN
};
```

Продовження Додатку А

```
// Параметры по умолчанию
const float YAW = -90.0f;
const float PITCH = 0.0f;
const float SPEED = 10.0f; // Начальная скорость движения
const float SENSITIVITY = 0.1f;  // Чувствительность мыши
const float ZOOM = 45.0f; // Поле зрения (FOV)

// Абстрактный класс камеры, обрабатывающий ввод и вычисляющий соответствующие матрицы Эйлера, View и Projection.
class Camera
{
public:
    // Атрибуты камеры
    glm::vec3 Position;
    glm::vec3 Front;
    glm::vec3 Up;
    glm::vec3 Right;
    glm::vec3 WorldUp;
    // Углы Эйлера
    float Yaw;
```

```
float Pitch;
// Параметры камеры
float MovementSpeed;
float MouseSensitivity;
float Zoom;
```

Продовження Додатку А

```
// Конструктор с векторами
Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), float yaw = YAW, float pitch = PITCH) : Front(glm::vec3(0.0f, 0.0f, -1.0f)), MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
{
    Position = position;
    WorldUp = up;
    Yaw = yaw;
    Pitch = pitch;
    updateCameraVectors();
}
// Конструктор с скалярными значениями
Camera(float posX, float posY, float posZ, float upX, float upY, float upZ, float yaw, float pitch) : Front(glm::vec3(0.0f, 0.0f, -1.0f)), MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
{
    Position = glm::vec3(posX, posY, posZ);
    WorldUp = glm::vec3(upX, upY, upZ);
    Yaw = yaw;
    Pitch = pitch;
```

```
        updateCameraVectors();
    }


    // Возвращает матрицу вида, вычисленную с использованием углов Эйлера и матрицы LookAt.
```

Продовження Додатку А

```
    glm::mat4 GetViewMatrix()
    {
        return glm::lookAt(Position, Position + Front, Up);
    }


    // Обрабатывает ввод, полученный от любой клавиатурной системы ввода. Принимает параметр enum Camera_Movement.
    void ProcessKeyboard(Camera_Movement direction, float deltaTime)
    {
        float velocity = MovementSpeed * deltaTime;
        if (direction == FORWARD)
            Position += Front * velocity;
        if (direction == BACKWARD)
            Position -= Front * velocity;
        if (direction == LEFT)
            Position -= Right * velocity;
        if (direction == RIGHT)
            Position += Right * velocity;
        if (direction == UP)
            Position += WorldUp * velocity; // Движение строго вверх по оси Y
```

```
        if (direction == DOWN)
            Position -= WorldUp * velocity; // Движение строго вниз по оси Y
    }
```

Продовження Додатку А

```
    // Обрабатывает ввод, полученный от движения мыши. Ожидает значения смещения по осям x и y.
    void ProcessMouseMovement(float xoffset, float yoffset, GLboolean constrainPitch = true)
    {
        xoffset *= MouseSensitivity;
        yoffset *= MouseSensitivity;

        Yaw += xoffset;
        Pitch += yoffset;

        // Убедитесь, что когда pitch выходит за пределы экрана, он не "переворачивается"
        if (constrainPitch)
        {
            if (Pitch > 89.0f)
                Pitch = 89.0f;
            if (Pitch < -89.0f)
                Pitch = -89.0f;
        }
```

```
        // Обновляем векторы Front, Right и Up, используя обновленные углы
Эйлера
        updateCameraVectors();
    }
```

Продовження Додатку А

```
    // Обрабатывает ввод, полученный от события колесика прокрутки мыши.
Только на входном вертикальном колесике.
    void ProcessMouseScroll(float yoffset)
    {
        Zoom -= (float)yoffset;
        if (Zoom < 1.0f)
            Zoom = 1.0f;
        if (Zoom > 45.0f)
            Zoom = 45.0f;
    }

private:
    // Вычисляет передний вектор из углов Эйлера
    void updateCameraVectors()
    {
        // Вычисляем новый вектор Front
        glm::vec3 front;
        front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        front.y = sin(glm::radians(Pitch));
```

```
        front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        Front = normalize(front);
        // Также пересчитываем вектор Right и Up
        Right = normalize(glm::cross(Front, WorldUp));
        Up = normalize(glm::cross(Right, Front));
    }
```



```
};
#endif
```

Додаток Б

Інструкція користувача

Цей додаток містить коротку інструкцію для користувача щодо запуску та базового використання 3D-візуалізатора.

- **Запуск програми:** Опис кроків для запуску виконуваного файлу.
- **Завантаження моделі:** Пояснення, як використовувати діалогове вікно вибору файлу (клавіша 'L').
- **Управління камерою:** Детальний опис використання клавіш WASD, Space, Left Control та миші для навігації.
- **Підтримувані формати:** Перелік підтримуваних форматів файлів.