

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3D РЕДАКТОРУ

3D EDITOR SOFTWARE DEVELOPMENT

Виконав(ла): студент(ка) 4 курсу, групи КНТ-111

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

БРОВЧЕНКО І.Д.

(ПРИЗВИЩЕ та ініціали)

Керівник СЕРДЮК С.М.

(ПРИЗВИЩЕ та ініціали)

Рецензент ТИМЕНКО К.І.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

БРОВЧЕНКО Івана Дмитровича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка програмного забезпечення 3D редактору.
3D Editor Software Development

керівник проєкту (роботи) к.т.н., доцент, СЕРДЮК Сергій Микитович,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “28” квітня 2025 року № 209

2. Строк подання студентом проєкту (роботи) 02 червня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз проблеми та постановка завдань.
2. Матеріали і методи. 3. Основні рішення з реалізації компонентів системи
4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СЕРДЮК С.М., доцент		
Нормоконтроль	ДЕЙНЕГА Л.Ю., ст. викладач		

7. Дата видачі завдання «14» квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Іван БРОВЧЕНКО
(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Сергій СЕРДЮК
(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 81с., 2 табл., 25 рис., 1 дод., 22 джерел.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, КОМП'ЮТЕРНА ГРАФІКА, 3D РЕДАКТОР, RENDER, OPENGL, SHADER, GLSL.

Об'єкт дослідження – процеси створення 3D-зображень

Предмет дослідження – програмні засоби формування 3D-сцен та методи спрощеного рендерингу для художніх задач.

Мета роботи – створити просту систему, яка дозволяє створювати бюджетне фони для манги за допомогою імпорту GLTF-моделей та основних засобів керування освітленням, а також технік скретчингу, без потреби глибоких знань графічного мистецтва чи 3D-моделювання в користувача програмного забезпечення.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, мови програмування C++, Microsoft visual studio 2022, персональний комп'ютер з процесором Ryzen 7 5700U під управлінням операційної системи Microsoft Windows 11.

Результати. Створено програмний Windows-застосунок, який дозволяє створювати бюджетне фони для манги за допомогою імпорту GLTF-моделей та їх розташування у сцені з налаштуванням освітлення.

Висновки. Розроблено програмну систему для пришвидшення й полегшення створення фонів для манги шляхом імпорту 3D-моделей, їх розташування у віртуальному просторі та налаштування освітлення.

Галузь використання – створення художніх фонових сцен у стилі манги для ілюстраторів, дизайнерів та авторів коміксів як у професійному середовищі, так і в домашніх умовах.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 81 pages, 2 tables, 25 figures, 1 appendixes, 22 sources.

SOFTWARE, COMPUTER GRAPHICS, 3D EDITOR, RENDER, OPENGL, SHADER, GLSL.

Object of the study – the processes of creating 3D images.

Subject of the study – software tools for constructing 3D scenes and simplified rendering techniques for artistic purposes.

Objective of the work – to develop a simple system that enables the creation of budget-friendly manga backgrounds through the import of GLTF models and basic lighting control tools, as well as scratch-based techniques, without requiring the user to have deep knowledge of graphic arts or 3D modeling.

Materials, methods, and technical tools: structural and object-oriented programming, C++ programming language, Microsoft Visual Studio 2022, personal computer with a Ryzen 7 5700U processor running Microsoft Windows 11.

Results: A Windows application has been developed that allows users to create cost-effective manga backgrounds by importing GLTF models and arranging them in a scene with adjustable lighting settings.

Conclusions: A software system has been developed to accelerate and simplify the process of creating manga backgrounds by importing 3D models, placing them in a virtual space, and configuring lighting.

Field of application – the creation of artistic background scenes in manga style for illustrators, designers, and comic authors, both in professional environments and for home use.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	8
Вступ.....	9
1 Аналіз проблем та постановка завдань	11
1.1 Загальні відомості про 3D-редактори	11
1.2 Концептуальні основи побудови 3D-редакторів	12
1.3 Основні графічні методи у 3D-візуалізації.....	14
1.4 Базові технології 3D-рендерингу.....	16
1.5 Аналіз програмних аналогів	19
1.6 Постановка завдань для розробки власного рішення.....	24
2 Матеріали і методи.....	26
2.1 Вибір мови програмування	26
2.2 Вибір бібліотек	27
2.3 Вибір середовища розробки.....	28
2.4 Розробка архітектури програмних модулів редактора.....	30
3 Основні рішення з реалізації компонентів системи	33
3.1 Розробка загального алгоритму роботи програмного коду редактора..	33
3.2 Налаштування середовища розробки та підключення бібліотек	36
3.3 Розробка програмних модулів редактора	39
4 Експлуатація, тестування та експериментальне дослідження програми	45
4.1 Загальні положення.....	45
4.2 Підготовка до тестування.....	46
4.3 Розгортання та запуск на цільовій системі.....	47
4.4 Перевірка функціональності	49
4.5 Аналіз продуктивності.....	52
Висновки	54
Перелік джерел посилання	55
Додаток А Слайди презентації.....	58

Додаток Б Текст програми	65
--------------------------------	----

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;

ПЗ – програмне забезпечення;

ЦП – центральний процесор;

ГП – графічний процесор;

IDE – Integrated Development Environment;

SDK – Software Development Kit;

VAO – Vertex Array Object;

VBO – Vertex Buffer Object;

EBO – Element Buffer Object;

OS – Operating System;

ВСТУП

Сучасна література манга і комікси стають дедалі більш популярними і отримують значну увагу як японської так і міжнародної аудиторії. Головною особливістю цієї літератури є розповідь за допомогою послідовних зображень, які супроводжуються письмовим текстом.

Манга в тій формі, в якій вона існує сьогодні, почала розвиватися після закінчення другої світової війни. Видавнича індустрія манги виникла в Японії в 1950-х роках і зросла до обороту понад 500 мільярдів єн у 2006 році. Міжнародна популярність манги охопила весь світ особливо зачепила Сполучені Штати, оскільки продажі на той час досягли 175-200 мільйонів доларів протягом 2006 року [1],[2].

Масова популярність манги з розвитком програмних технологій зумовила появу спеціалізованих цифрових інструментів, які дозволяють користувачам автоматизувати або полегшити створення манги[3]. На сьогодні існуючі забезпечення пропонує безліч можливостей для малювання, включно з керуванням шарами, кількома варіантами пензлів, векторними графічними інструментами і налаштованими колірними схемами.

Ці продукти в основному слугують для малювання вручну, але для досягнення найкращих результатів потрібні високі художні таланти. Користувачі мають створювати повне тло або самостійно, або поєднуючи складні маніпуляції з шарами та організацію векторних об'єктів або використовувати стилізовані готові шаблони. Більшість рішень не мають простих систем для додавання 3D-моделей до сцен і потребують додаткових рішень програмного забезпечення, як рендер у Blender або в іншому подібному програмному забезпеченні й перенесення отриманих отриманого зображення у сцену до манги.

Оскільки не існує у вільному доступі спеціального інструменту для легкого розміщення 3D-об'єктів з подальшою базовою конфігурацією сцени

для створення фону без вимог до художньої кваліфікації, тема дипломної роботи бакалавра є актуальною.

Основна мета дослідження полягає у створенні системи автоматичної генерації фону, яка вимагає від користувачів мінімальних знань як у сфері художніх навичок, так і технічних можливостей.

Для реалізації цієї мети належить вирішити такі проблеми:

- вивчити архітектурні та функціональні підходи у схожих системах візуалізації;
- розробити проєкт рушія для створення фонів і алгоритми його роботи;
- обґрунтувати вибір мови програмування та графічної бібліотеки;
- реалізувати базові функції імпорту моделей, налаштування освітлення та камери;
- оцінити працездатність рушія та проаналізувати отримані результати.

1 АНАЛІЗ ПРОБЛЕМ ТА ПОСТАНОВКА ЗАВДАНЬ

1.1 Загальні відомості про 3D-редактори

Звичайний програмне забезпечення в процесі розробки та подальшого застосування працює з використанням змінних, які керують його роботою та кінцевими результатами його роботи. Змінні можна зробити прихованими від кінцевого користувача, але все ще доступними для редагування. Для того, щоб ці змінні та їх налаштування були легко доступні користувачам, коли вони хочуть використовувати програму, створюються спеціальні графічні інтерфейси. Саме з цього інтерфейсу користувачі параметризують об'єкти та внутрішні змінні об'єктів за допомогою інтерактивних елементів такі як повзунки, кнопки та поля введення.

Реалізацію цих інтерфейсів можна забезпечити за допомогою графічних бібліотек (наприклад, OpenGL, SDL, Qt або WidgetX) шляхом додавання графічних компонентів до керування параметрами. Таким чином, користувач має можливість змінювати властивості об'єктів у програмі без необхідності маніпулювати вихідним кодом або консоллю.

3D-редактор – це програмне забезпечення, яке реалізує візуальний інтерфейс для роботи зі сценою. За допомогою елементів графічного інтерфейсу користувач може змінювати параметри об'єктів і середовища, які зберігаються у структурі даних головного комплексу програмного середовища тобто рушія. Зміни елементів інтерфейсу впливають на внутрішній стан змінних рушія. 3D-редактор як посередник між користувачем і внутрішньою логікою рушія, що дозволяє змінювати стан об'єктів без прямої роботи з кодом або підлеглими структурами структури даних.

Зазвичай редактори оснащені інтуїтивним інтерфейсом, який дозволяє взаємодіяти з об'єктами в 3D-просторі. Вони пропонують різноманітні інструменти для створення та редагування геометрії, розміщення текстур, налаштування освітлення, анімації об'єктів, що дозволяє художникам та

дизайнерам ефективно втілювати свої ідеї в 3D-просторі, надаючи сценам потрібного стилістичного вигляду.

3D-редактори можна класифікувати за основним призначенням, залежно від того, на яку задачу вони орієнтовані. До основних напрямів відносять кілька типів редакторів.

Редактори для створення 3D-моделей спрямовані на побудову геометрії об'єктів різної складності. Вони надають інструменти для моделювання форм за допомогою полігональної, NURBS- або воксельної геометрії. Прикладами таких програм є Blender, 3ds Max, Rhinoceros.

Редактори для анімації та симуляції спеціалізуються на створенні анімацій руху об'єктів, фізичних симуляцій, роботи з рідинами, частинками, тканинами та іншими динамічними системами. Серед представників цього напрямку можна відзначити Autodesk Maya і Houdini.

Редактори для візуалізації та рендерингу орієнтовані на створення фінальних фотореалістичних або стилізованих зображень шляхом налаштування освітлення, матеріалів, камер і сцен. До цієї групи належать такі системи, як KeyShot, V-Ray (плагіни для редакторів), Corona Renderer.

На сьогодні більшість сучасних 3D-редакторів є мультифункціональними та поєднують у собі всі перераховані напрями. Наприклад, Blender, Autodesk Maya, 3ds Max, дозволяють одночасно моделювати, анімувати і виконувати візуалізацію в одному середовищі.

1.2 Концептуальні основи побудови 3D-редакторів

Звичайний Основою будь-якої 3D-програми є графічний рушій, який забезпечує необхідну роботу з тривимірною сценою. Графічний рушій відповідає за візуалізацію об'єктів, обробку просторових трансформацій, освітлення, камер, тіней, матеріалів. Він також забезпечує підтримку візуалізації в реальному часі або підготовки сцен до остаточної візуалізації. Двигун виступає своєрідним ядром цілісної системи, на основі якого

побудовано візуальний інтерфейс 3D-редактора де обробляються зміни, внесені користувачем. Схема взаємодії 3D-редактора з рушієм представлена на рисунку 1.1



Рисунок 1.1 – Структурна схема взаємодії 3D-редактора і рушія

Одним з ключових моментів створення сцени в 3D-редакторі є система освітлення і матеріалів. Освітлення не тільки формує візуальне сприйняття предметів, але і їхню взаємодію зі світлом середовища через тіні, відбиття і заломлення світла. Матеріали визначають фізичні властивості поверхонь, зокрема колір, шорсткість, блиск або прозорість. Добірно виконана система освітлення і матеріалів забезпечує можливість реалізації реалістичного або стилізованого вигляду сцени відповідно до вимог користувача.

Важливе значення в роботі 3D-редактора є робота зі сценами й ассетами. Ассети містять в собі моделі, текстури, шейдери, інші компоненти, які застосовуються для будівництва сцен. Редактор повинен забезпечувати імпорт різних 3D форматів моделей, їх вірне розташування у просторі, масштабування, обертання, а також об'єднання в логічні групи. Структура сцени повинна зберігати ієрархічне розташування об'єктів, що допомагає організовувати зручно великі проєкти й зменшує навантаження при навігації серед великої кількості елементів.

1.3 Основні графічні методи у 3D-візуалізації

При створенні та обробці тривимірної графіки застосовуються різні методи візуалізації кожен зі своїм особливим характером та застосуванням. Основне з них – полігональна, векторна, растрова, об'єктно-орієнтована графіка.

Полігональна графіка – найбільш розповсюджений спосіб у 3D-моделюванні. Об'єкти складаються зі збірки полігональних, здебільшого трикутників або чотирикутників (рисунок 1.2), які з'єднують між собою вершини в тривимірному просторі [4]. Полігональні моделі мають досить легку обробку графічними рушіями і застосовуються при створенні складних форм (приклад якої приведено на рисунку 1.3) при відносно невеликому об'ємі даних. Цей метод застосовується в розробці ігор, анімації, віртуальної реальності і візуалізації об'єктів з фотореалістичним освітленням [5],[6].

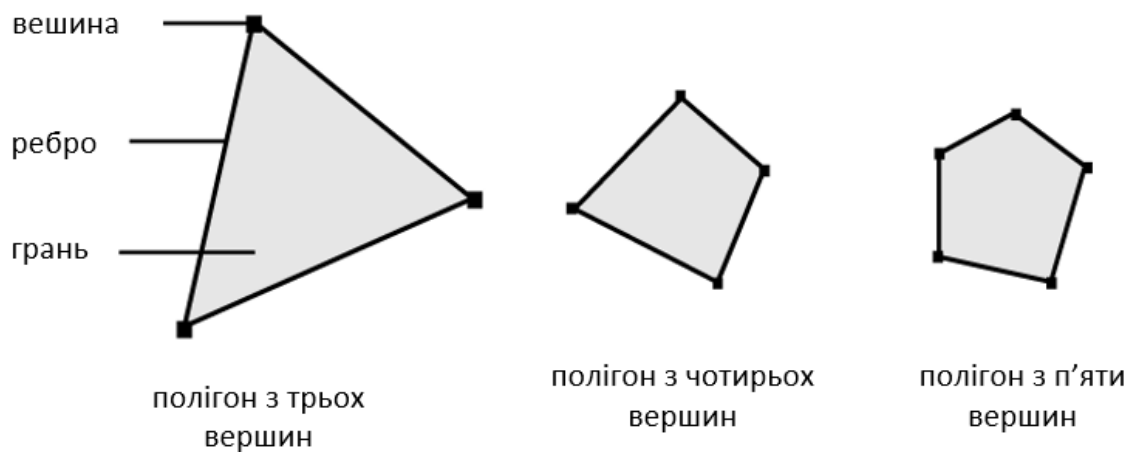


Рисунок 1.2 – Структура полігона з прикладами трьох, чотирьох та п'ятивершинних полігонів

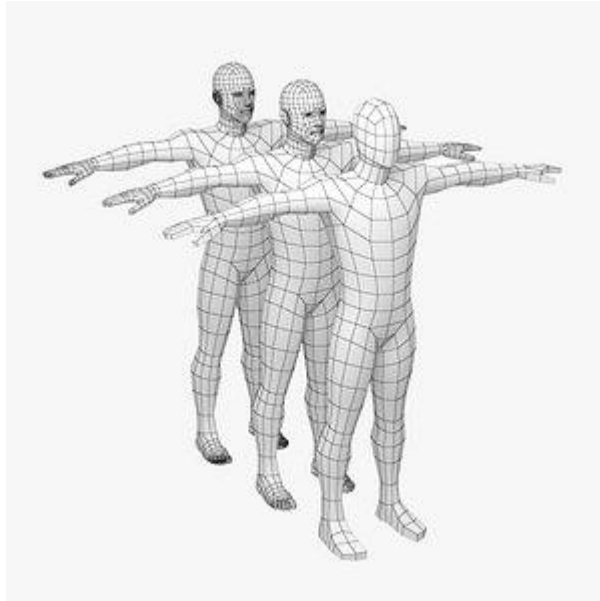


Рисунок 1.3 – Полігональна сітка тривимірної моделі людини

Векторна графіка в області 3D застосовується набагато рідше, проте її підходи можуть бути застосовані для створення кривих Безьє, сплайнів, інших складних лінійних структур. Векторні об'єкти мають математичну підметку в вигляді рівнянь, що забезпечує високий рівень якості навіть після масштабування. У 3D-візуалізації векторні елементи застосовуються досить часто для створення допоміжної геометрії, навігаційних систем або для створення анімаційних траєкторій [7].

Растрова графіка у 3D редакторах застосовується головним чином для візуалізації і текстурізації об'єктів. Вона заснована на розподілі сцени на сітку пікселів, кожен з яких має свої властивості кольору, освітлення та інших властивостей. Текстури, які застосовують до поверхонь моделей, створюються в якості растрових зображень і забезпечують надання об'єктам вигляду, що нагадує дійсність шляхом передачі мікроелементів. Також активно застосовуються в якості джерела для формування карт нормалей, карт висот та інших допоміжних матеріалів, що впливають на візуальні характеристики поверхонь. Растрова графіка є основою візуалізації в режимі реального часу, застосовується досить широко в комп'ютерних іграх.

Об'єктно-орієнтована графіка ґрунтується на ідеях об'єктно-орієнтованого моделювання, де кожен елемент сцени представлений окремим об'єктом з власними властивостями та методами. Об'єкти можуть мати власну геометрію, текстуру, матеріал, інші властивості, виконувати взаємодію між собою згідно із заданими правилами логіки сцени. Такий підхід забезпечує простоту управління великими й досить складними сценами, підтримує ієрархічні структури і допомагає легко змінювати окремі елементи під час розробки [8].

Кожен з вищеперерахованих підходів має свої переваги та недоліки й вибір певного підходу залежить від цілей проєкту, рівня необхідної деталізації та технічних можливостей програмного середовища.

1.4 Базові технології 3D-рендерингу

У процесі створення тривимірних сцен комп'ютерної графіки, перетворення координат з тривимірного простору у двовимірний екран. Одними з основних технологій, які забезпечують такий розвиток, є графічні інтерфейси, куди досить широко входить OpenGL. Ця бібліотека є кросплатформним стандартом, який дозволяє програмістам працювати з низькорівневим управлінням ресурсами графічної карти, створюючи досить складні сцени за допомогою роботи зі встановленням вершин, текстури, освітлення та шейдерів. В основі OpenGL лежить ідея реалізацій графіко-оброблюючого конвеєра, при якому послідовність дію над дані приводить до отримання кінцевого зображення на екрані.

Для візуалізації тривимірних об'єктів дані проходять кілька етапів обробки. Від початкових вершинних координат дані послідовно конвертуються у двовимірні координати, і тільки потім в пікселі зі співпадаючими кольоровими значеннями. На керування цими перетвореннями встановлені спеціальні програми – шейдери які виконуються на графічних процесорах. Вершинний шейдер обробляє просторові

координати вершин і знаходження їх місце розташування в просторі після зміщення. Перед цим примітиви (наприклад, трикутники) складаються в потрібні форми. На етапі геометричного шейдера можна додаткову обробку або генерацію нових примітивів, а шейдер тесселяції повністю ділить об'єкти на більш деталізовані частини.

Наступним етапом конвеєра є растеризація, в якій тривимірні дані перетворюються в сітку пікселів, кожен піксель обробляється фрагментним шейдером. Саме фрагментний шейдер виконує роль визначення кінцевого кольору кожного пікселя згідно з освітленням, текстур і тіней. До остаточного виводу сцена проходить стадії тестування глибини та змішування пікселів, що забезпечує коректне відображення прозорих об'єктів й порядок закріплення об'єктів у просторі.

Процес обробки даних у графічному конвеєрі представлено на рисунку 1.4 на основі основних етапів обробки від вершинних даних до готового піксельного зображення.

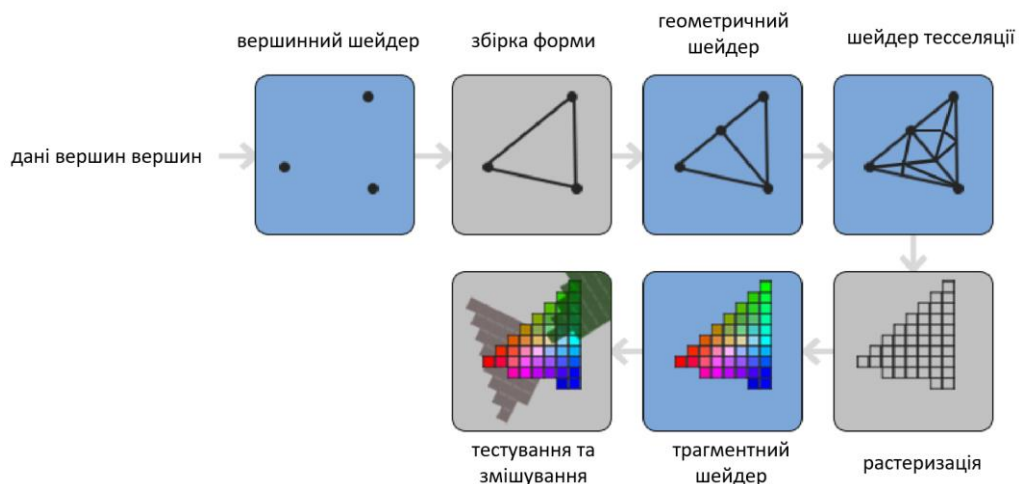


Рисунок 1.4 – Схема обробки даних у графічному конвеєрі

Основою створення об'єктів у тривимірному просторі є такі структури даних, як меші, вершини, полігони. Вершина — це звичайна одиниця простору з координатами (x, y, z) і додатковими атрибутами, наприклад нормаль або текстурні координати. Полігон складається із з'єднання декількох вершин і

складається з замкненої поверхні, найчастіше трикутника або чотирикутника. Меш – це комплекс вершин і полігонів, що складають геометрію об'єкта. Чим більше кількістю вершин і полігонів, тим детальнішим буде об'єкт, але й тим вищими будуть вимоги до об'ємів обробки [6].

Зберігання та обробка великої кількості вершинних даних в графічній програмі здійснюється за допомогою буферних об'єктів. Vertex Buffer Object (VBO) містить дані про вершини — координати їх, кольори, нормалі, текстурні координати. Element Buffer Object (EBO) містить індекси, які визначають кінцеву послідовність зростання вершин для формування полігонів, що допомагає зменшити кількість дублікатів даних. Для обслуговування рендерингу також застосовується Vertex Array Object (VAO) для об'єднання всіх необхідних налаштувань буферів і можливості швидкого доступу до даних під час відображення сцени.

Точне відображення об'єктів у тривимірному просторі передбачає визначення їх положення щодо камери. Для цього застосовується буфер глибини (Depth Buffer), який містить інформацію про відстань кожного пікселя від камери. При рендерингу сцени нові пікселі порівнюються з вже записаними значеннями глибини: якщо новий піксель ближчий до камери, він змінює попередній, інакше — ігнорується. Описаний підхід допомагає забезпечити правильний порядок перекриття об'єктів у сцені та зробити ефект просторової глибини.

Так, базові технології 3D-рендерингу стосуються усіх важливих понять від організації вершинних даних і управління буферами до управління рендерингом шляхом шейдерів і глибини сцени. Це основний базовий принцип побудови складних інтерактивних тривимірних середовищ.

1.5 Аналіз програмних аналогів

Першим на розгляд програмного забезпечення, яке дозволяє створювати тривимірну візуалізацію, є програмний продукт Blender 3D. Програма бере свій початок у 1998 року. Blender було випущено публічно в інтернеті. Як безкоштовне програмне забезпечення, що дозволяє створювати 3D-об'єкти та анімацію на тлі інших платних програмних забезпечення, які надавали той самий функціонал. Це дозволило Blender поступово набирати популярності до справжнього дня. На сьогодні програма також знаходиться у відкритому доступі з відкритим програмним кодом, Blender має велику спільноту, має значну кількість інформації у вигляді відеоматеріалів на YouTube, документації та інших ресурсів. А також, оскільки він має відкритий програмний код, то для Blender розробляються плагіни для різних завдань. На сьогоднішній день можна завантажити з офіційного сайту останню версію Blender, в якій є відповідні можливості. Крім 3D-моделювання та створення анімації, також доступні скульптинг, рендеринг, текстурування, та налаштування матеріалів.

При установці програмного забезпечення від системи потрібні такі системні вимоги: наявність вільного простору на диску об'ємом 2 гігабайта, операційна система Windows, Linux або Mac OS, а при необхідності візуалізації сцен потрібен графічний процесор, щонайменше 8 гігабайт оперативної пам'яті та 4 гігабайти відеопам'яті для виконання рендерингу. Інтерфейс програмного забезпечення представлений рисунку 1.5.

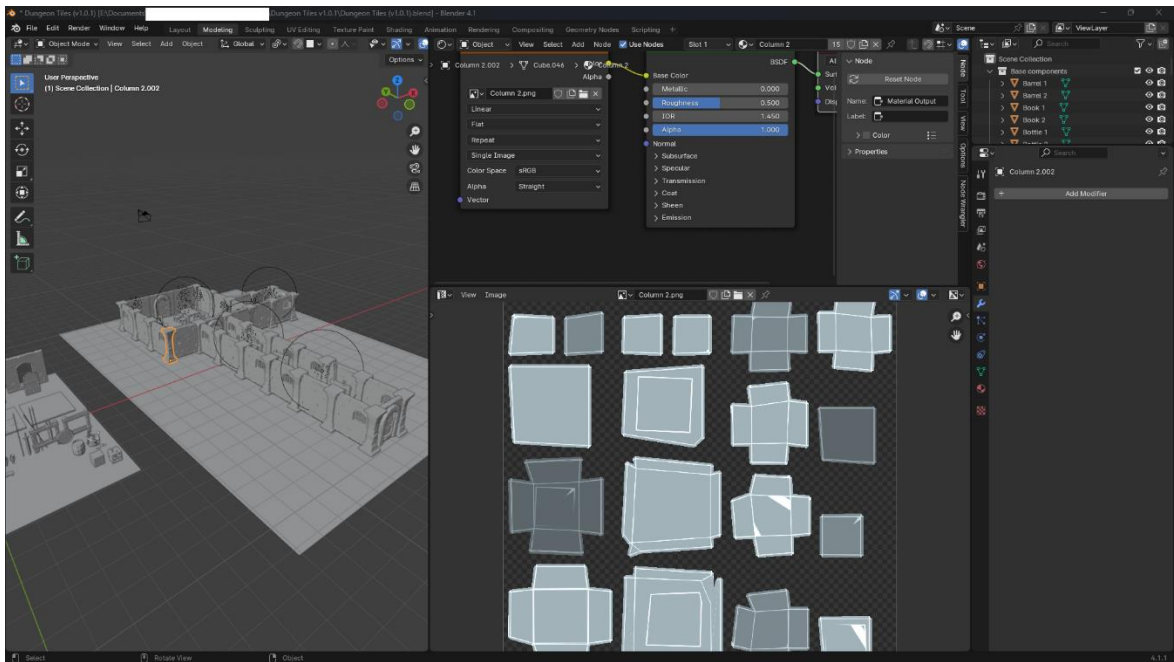


Рисунок 1.5 – Інтерфейс Blender 3D 4.1

Основним недоліком Blender є високий поріг входження. При першому запуску Blender відображається інтерфейс, що містить багато вкладок та режимів. Інтерфейс, таким чином, перевантажений і неінтуїтивний. Також при вивченні навчальних матеріалів за книгами чи відеороликами потрібне знання переліку гарячих клавіш. Слабка сторона програмного забезпечення полягає у недостатньо якісній реалізації інструментів для роботи з UV-розгорткою, менеджером сцен та анімації [9].

Друге на огляд програмного забезпечення – це ігровий рушій Unity 3D. На сьогодні це одне з найпоширеніших програмних забезпечень для кросплатформної розробки. Unity підтримує поширення для Android пристроїв, Windows, Mac OS, Linux, а також для браузерів. Підтримується також розробка для доповненої та віртуальної реальності. На відміну від Blender, Unity не призначений для моделювання та створення текстуровання. Це ігровий двигун спрямований для створення ігор і передбачає вже імпортування готових рішень: моделей, текстур тощо. Unity було створено у 2005 році компанією Unity Technologies. Unity – закрите програмне забезпечення. Однак доступна безкоштовна версія з певними обмеженнями

для користувачів з доходом нижче встановленого порога. Також передбачені платні версії: Unity Pro та Unity Plus. Ключові можливості редактора - це наявність системи фізично-коректного рендеру, компоненти для роботи з фізикою, налаштування процедурних анімацій, взаємодія з кольором, камерами; включена також мова програмування. Програмне забезпечення має інтегрований редактор шейдерів, який підтримує візуальне нод програмування, яке дозволяє налаштовувати матеріали сцени, щоб вона виглядала необхідним чином. Двигун також підтримує систему частинок, як і всі сучасні ігрові двигуни.

Під час встановлення програмного забезпечення необхідно перейти на офіційний сайт Unity, завантажити лаунчер, який важить близько одного гігабайта. У лаунчері необхідно зареєструватися, завести обліковий запис у реєстрі, а потім у лаунчері вибрати версію редактора. На сьогоднішній день остання версія - Unity 6. Кожна версія має різний розмір. Версія редактора 6 при встановленні займає на жорсткому диску від 6 до 10 гігабайт вільного простору. Під час створення проекту порожній шаблон при генерації займає близько двох гігабайт вільного простору (рисунк 1.6).

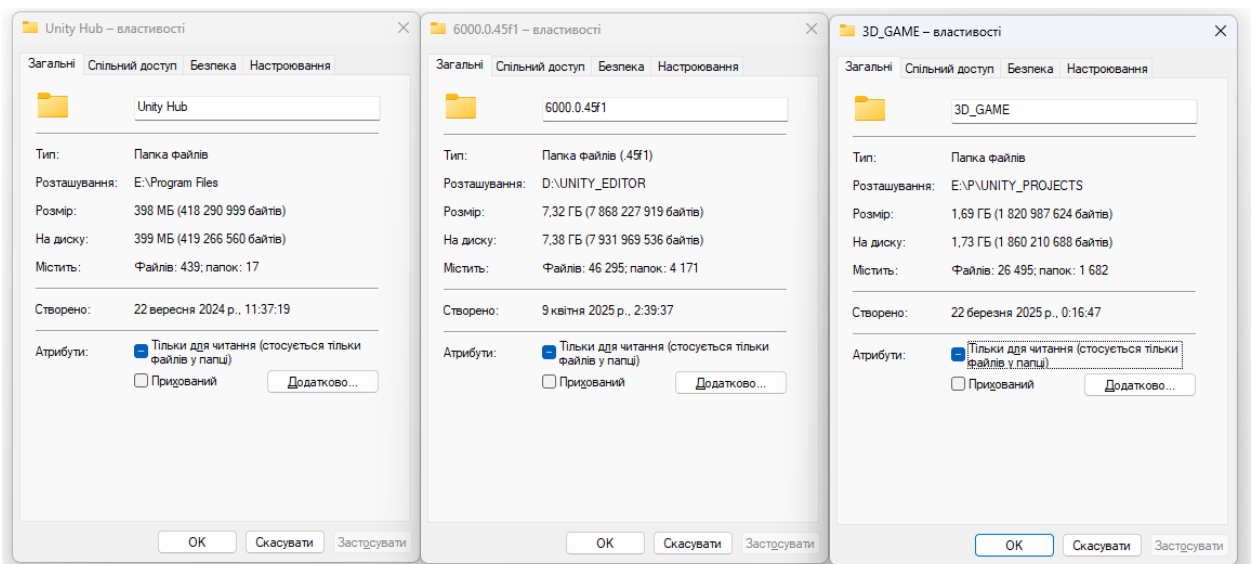


Рисунок 1.6 – Розмір встановленого редактора Unity, лаунчера та базового 3D-проєкту]

Мінімальні рекомендовані системні вимоги для редактора включають підтримку Windows 10, 11 або Mac OS, процесор із підтримкою SSE2, щонайменше 8 ГБ оперативної пам'яті, рекомендовану дискретну відеокарту, особливо якщо розробка ведеться на ноутбуці, а також наявність встановленого .NET.

Користувальницький інтерфейс редактора Unity має віконну структуру. При першому запуску редактора інтерфейс має вікна стандартного розташування: Scene, Inspector, Hierarchy об'єктів і Project (рисунк 1.7).

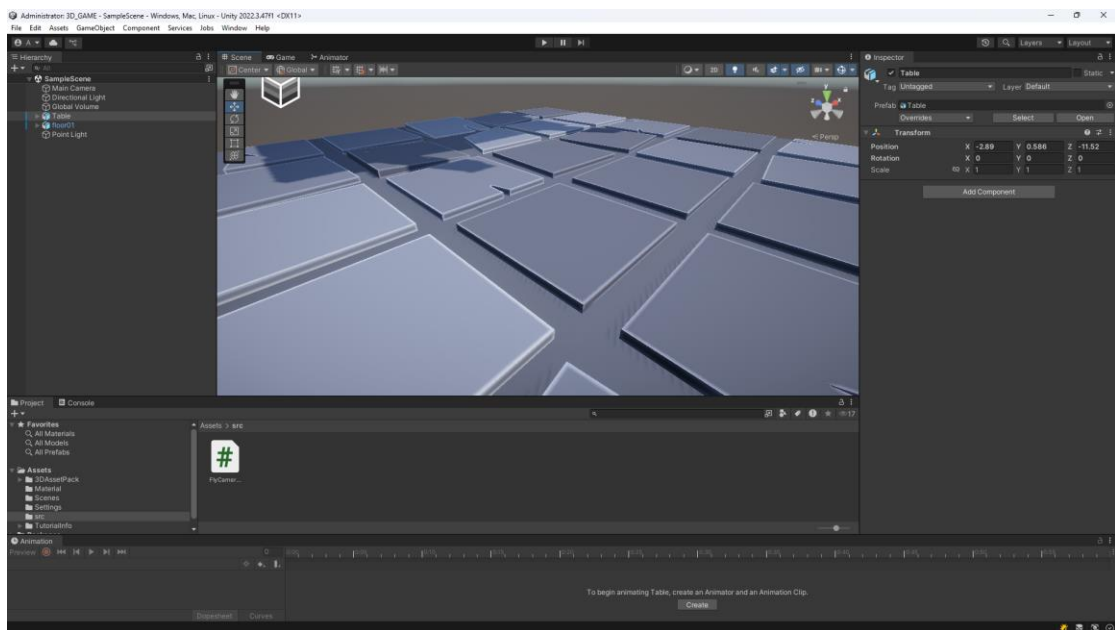


Рисунок 1.7 – Стандартне інтерфейсне середовище Unity редактора

Інтерфейс не обмежує в гнучкості налаштування: можна переміщати вікна, поєднувати їх у вкладки за допомогою перетягування, що дозволяє організувати робоче середовище згідно з моделями, безкоштовних моделей, текстур, анімацій, шейдерів, різних налаштувань, скриптів та шаблонів, які можна завантажити, трохи модифікувати під потрібний стиль, що значно прискорює розробку.

Хоча в Unity можна налаштовувати сцену і шейдери для необхідного візуального представлення моделей і текстур, у нього є обмежені можливості для складного кінематографічного графічного або стилістичного рендеру, що

залежить від зовнішніх плагінів, які вимагають додаткового місця на диску і додаткового вивчення специфіки реалізації. При створенні проєкту відсутня опція внутрішніх не потрібних для реалізації проєкту модулів, які вимагають додаткового місця на жорсткому диску при генерації мінімального проєкту.

З огляду вище можна зробити висновки, що функціонально Blender і Unity мають свої особливості та відмінності, порівняльну характеристику подано в табл. 1.1. Blender надає повний набір інструментів для моделювання, анімації та рендерингу, включаючи вбудовані рендер-руші Eevee та Cycles. Натомість Unity орієнтований переважно на розробку інтерактивних додатків та ігор, тому він не передбачає можливість створення моделей безпосередньо в редакторі й потребує імпорту готових 3D-моделей. Для виконання спеціалізованих завдань, таких як стилізований рендеринг, Unity часто потребує додаткових компонентів.

Щодо інтерфейсу та зручності використання, Blender характеризується значною складністю для нових користувачів через велику кількість налаштувань, панелей та інструментів. Це створює високий поріг входження, особливо для людей, які не мають попереднього досвіду роботи з подібними інструментами. Unity у свою чергу має більш простий і зрозумілий інтерфейс, який можна гнучко налаштовувати відповідно до потреб користувача, хоча для реалізації специфічних завдань також можуть знадобитися додаткові налаштування та компоненти.

Порівняння апаратних вимог вказує, що Blender висуває значні вимоги до графічної складової комп'ютера та оперативної пам'яті, особливо під час рендерингу складних сцен. Unity має більш помірні вимоги до апаратного забезпечення, однак займає значний об'єм дискового простору як при установці редактора, так і під час створення навіть базових проєктів.

Таблиця 1.1 – Порівняльна характеристика Blender та Unity

Критерій	Blender 3D	Unity 3D
Тип ліцензії	Відкрите ПЗ (GPL)	Умовно безкоштовне (EULA)
Призначення	3D-моделювання, рендеринг, анімація	Ігровий рушій, візуалізація сцен, XR
Робота з 3D-моделями	Повний цикл: моделювання, текстурування, анімація	Лише імпорт моделей
Стилізований рендеринг	Eevee, Cycles, налаштування у реальному часі	Можливий через плагіни або шейдери
Наявність рендер-рушія	Вбудовані Eevee, Cycles	Так, URP/HDRP
Простота інтерфейсу	Високий поріг входження	Гнучкий, простіший
Можливість розширення	Додатки, Python-плагіни	Плагіни, Unity Asset Store
Операційні системи	Windows, macOS, Linux	Windows, macOS
мінімальні системні вимоги	8 ГБ RAM, GPU 4 ГБ VRAM	8 ГБ RAM, SSE2 CPU
Розмір встановлення	Близько 2 ГБ	6–10 ГБ + проекти
Основні недоліки	Складний інтерфейс, велика кількість комбінацій клавіш	Вимагає додаткових плагінів для гнучкого, займає багато місця на диску

1.6 Постановка завдань для розробки власного рішення

Під час проектування програмного забезпечення ключовим етапом є визначення вимог до системи. Вимоги визначають основні параметри та характеристики, якими повинен володіти готовий продукт на початковому етапі, щоб виконувати першочергову функцію на яку він розрахований.

Інтерфейс редактора має бути простим та інтуїтивним, що дозволить користувачам без глибоких технічних знань швидко освоїти його функціонал та перейти до творчої роботи.

Редактор повинен забезпечувати стилізований рендеринг для чорно-білої манги та дозволяти його налаштування відповідно до технік, характерних для манги – наприклад, чорне обведення тощо.

Передбачається підтримка імпорту 3D-моделей у поширених форматах з можливістю швидкого перегляду та базового редагування без додаткової конверсії або ручної обробки.

Програмне забезпечення повинно мати низькі системні вимоги, що забезпечить його доступність для широкого кола користувачів, включаючи власників комп'ютерів середнього рівня продуктивності.

Особливу увагу слід приділити швидкому розгортанню та запуску програми. Редактор повинен працювати із мінімальними початковими налаштуваннями, забезпечуючи готовність до використання одразу після інсталяції.

Система має бути побудована за модульним принципом, що дозволить з часом розширювати функціонал, додаючи нові можливості або замінюючи окремі компоненти без потреби повної реконструкції програмного ядра.

Сформульовані вимоги визначають напрям подальших етапів розробки, включаючи побудову архітектури системи, вибір відповідних технологій та реалізацію базового функціонального прототипу.

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Вибір мови програмування

На етапі первинного проєктування першим завданням стає вибір мови програмування, яка дозволить реалізувати основні структурні компоненти розроблюваного програмного забезпечення. Мова повинна забезпечити зручне керування пам'яттю, доступ до графічного конвеєра, роботу з GPU та підключення бібліотек, необхідних поставлених завдань. Основною метою є створення інструменту з чіткою архітектурною побудовою та можливістю розширення функціональності, зокрема – стилізованого рендерингу, шейдерних ефектів і обробки подій у режимі реального часу.

Для реалізації дипломного проєкту було прийнято рішення використати мову C++. Такий вибір зумовлений кількома чинниками. Вибір мови зумовлений наявністю практичного досвіду програмування на C++, що дозволяє реалізовувати архітектурні рішення без додаткових витрат часу на адаптацію та дослідження. По-друге, C++ підтримує багаторівневу абстракцію: за необхідності можлива робота як на високому рівні (через бібліотеки обгортки), так і прямий доступ до низькорівневих ресурсів. Це дозволяє поєднати зручність проєктування з повним контролем над пам'яттю та графічним середовищем.

Технологічний стек C++ включає великий набір стабільних бібліотек, орієнтованих на графіку, таких як DirectX, OpenGL, Vulkan, які підтримують низькорівневу побудову сцени, обробку подій та інтеграцію ресурсів. Завдяки цьому досягається баланс між продуктивністю, розширюваністю та точністю в управлінні візуальними компонентами розроблювального ПЗ.

2.2 Вибір бібліотек

Для реалізації графічного рендерингу в розроблювальному ПЗ в цьому підрозділі наведено огляд на бібліотеки, які вимагають лінкування під час компоновки проєкту маючи об'єктні або динамічні модулі (*.lib, *.dll, *.so). Натомість бібліотеки, що представлені лише у вигляді заголовкових файлів, будуть розглянуті нижче в розділах дипломної роботи при описі етапів реалізації архітектурних модулів та логіки функціонування програми.

Для основи побудови графіки у проєкті обрано бібліотеку OpenGL, яка є кросплатформним інтерфейсом для створення 2D та 3D графіки з використанням апаратного прискорення. Вона надає програмісту прямий контроль над графічним конвеєром, включаючи обробку вершин, трансформацій, освітлення, текстур, тіней і шейдерів [10]. Вибір OpenGL зумовлений практичним досвідом роботи з цим API, широким розповсюдженням, доступністю навчальних ресурсів, підтримкою мови GLSL для написання шейдерів, а також повною сумісністю з мовою програмування C++.

Разом з тим, OpenGL не забезпечує засобів створення вікон, опрацювання вводу з клавіатури або миші, та загалом не включає елементи взаємодії з операційною системою [11]. Для вирішення цих завдань застосовується бібліотека GLFW. Вона є легкою у використанні, кросплатформна і спеціально розроблена для ініціалізації графічного контексту включно OpenGL. В бібліотеці реалізовані функції для створення вікон та обробки подій введення. GLFW написана мовою C, має ліцензію zlib/libpng, що дозволяє її використання в комерційних проєктах, і активно підтримується спільнотою. Її функціональність включає підтримку декількох моніторів, обробку клавіатури та миші й ігрових контроллерів, а також гнучку систему подій, що дозволяє працювати як у режимі опитування, так і через функції зворотного виклику [12].

Обрана бібліотека GLFW може бути зібрана як у вигляді динамічної, так і статичної бібліотеки, що дозволяє гнучко інтегрувати її в проєкт залежно від вимог до розгортання. GLFW взаємодіє з уже встановленим у системі OpenGL-драйвером і виконує роль посередника між застосунком та віконною підсистемою операційної системи, що дозволяє створювати графічні контексти, відстежувати події введення з клавіатури та миші, керувати вікном та моніторами без потреби у написанні низькорівневого коду під конкретну платформу.

2.3 Вибір середовища розробки

Для забезпечення повного циклу створення програмного забезпечення з використанням мови C++ та залежностями у вигляді бінарних бібліотек і заголовкових файлів було здійснено вибір IDE Microsoft Visual Studio Community 2022, програмний інтерфейс середовища розробки представлений на рисунку 2.1

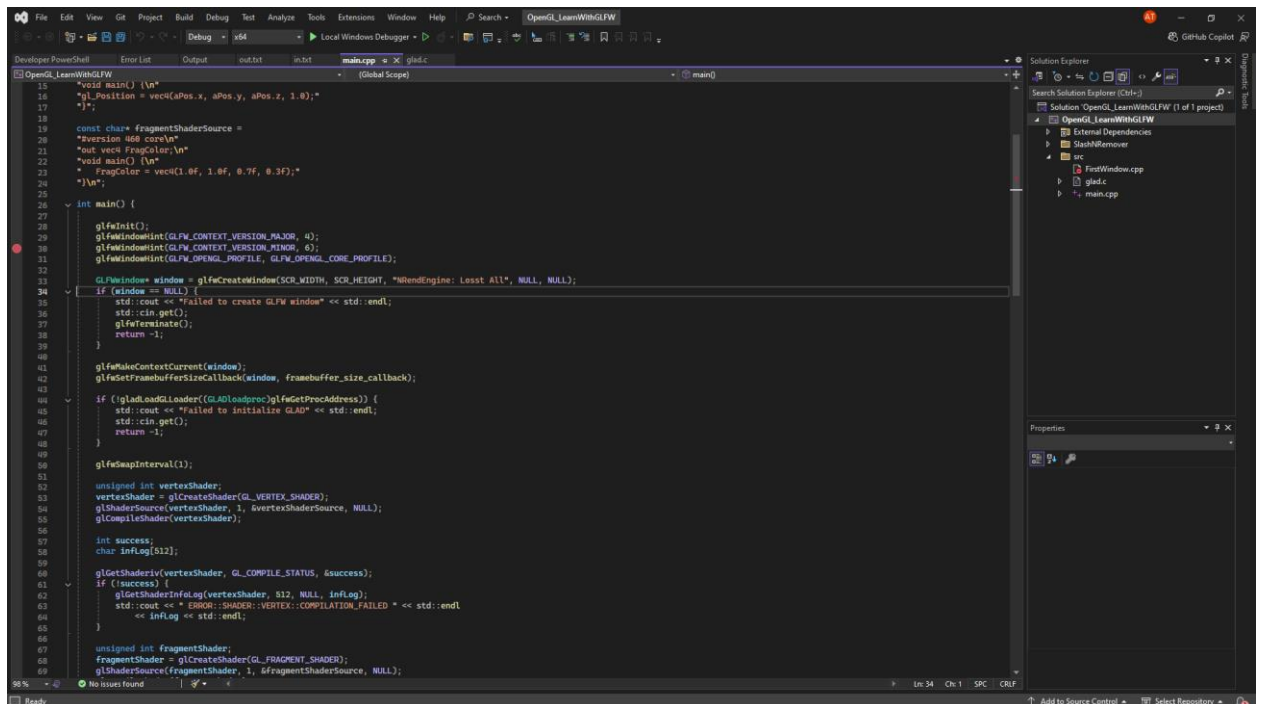


Рисунок 2.1 – Інтерфейс IDE Microsoft Visual Studio Community 2022

Таким чином, IDE Visual Studio 2022 забезпечує повноцінну інфраструктуру для розробки C++ застосунків із зовнішніми залежностями у вигляді бібліотек, включаючи засоби написання, компіляції, налагодження, спостереження за виконанням та аналізу програмного коду. Це середовище стало доцільним вибором для реалізації дипломного проєкту.

2.4 Розробка архітектури програмних модулів редактора

З метою забезпечення структурованості програмного забезпечення та відокремлення функціональних зон архітектура 3D-редактора була спроєктована на основі модульного підходу. Підхід був спрямований на підвищення гнучкості розробки, полегшення роботи з кодом, спрощення внесення змін і забезпечення подальшого масштабування системи. В якості бази для візуалізації архітектурної моделі було створено структурну блок-схему, що окреслює ключові логічні компоненти системи та взаємозв'язки між ними.

На рисунку 2.3 представлено структурну схему 3D-редактора, що демонструє логічний поділ системи на три взаємопов'язані зони: «Залежності та ресурси», «Ядро редактора» і «Час».

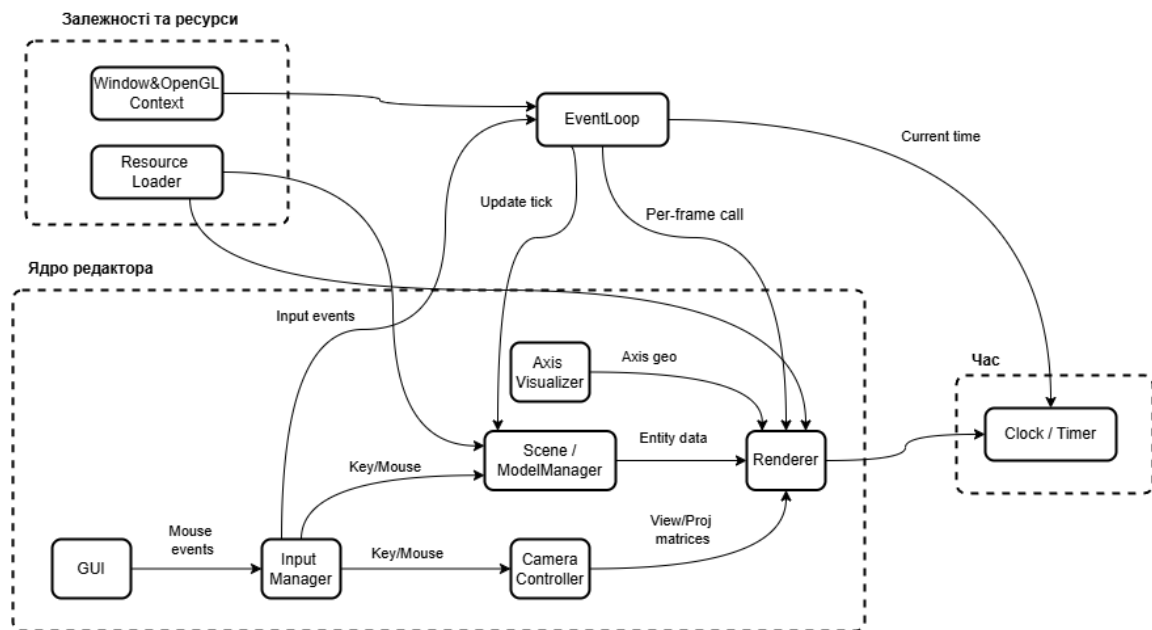


Рисунок 2.3 – Структурна схема програми 3D-редактора

На представленій схемі група «Залежності та ресурси» включає в себе низькорівневі компоненти, відповідальні за підготовку зовнішніх умов для роботи редактора. Зокрема, модуль *Window&OpenGL Context* ініціалізує графічне вікно, створює OpenGL-контекст та забезпечує потік вхідних подій у *EventLoop*. Модуль *Resource Loader* відповідає за завантаження ресурсів, таких як текстури, шейдери та моделі, й передає їх безпосередньо до *Renderer* та *Scene/ModelManager* для подальшого використання. Було використано такий підхід тому, що така розділеність дозволяє централізовано керувати ресурсами без дублювання логіки у функціональних модулях ядра.

У зоні «Ядро редактора» зосереджені модулі, що безпосередньо формують сцену, відстежують дії користувача та здійснюють її візуалізацію. GUI-модуль генерує інтерфейс та обробляє події від миші, які надходять до *Input Manager*. Останній об'єднує мишачі події з клавіатурними та створює два інформаційні потоки: події типу *Input events*, які надсилаються до *EventLoop*, та *Key/Mouse* сигнали, що одночасно надходять до *Camera Controller* і *Scene/ModelManager*.

Camera Controller розраховує поле огляду і проєкцію-матриці на основі вводу користувача, тоді як *Scene / ModelManager* модифікує

позиції, орієнтації та властивості об'єктів сцени відповідно до отриманих команд (обертання, переміщення, масштабування тощо). Допоміжний модуль Axis Visualizer використовується для візуалізації координатних осей (X, Y, Z) у разі активації відповідних режимів.

Центральною ланкою системи є EventLoop, який виконує функції диспетчера. На кожній ітерації основного циклу він опитує Clock/Timer, приймає накопичені події введення, надсилає сигнал Update tick у Scene/ModelManager для оновлення сценічного стану, а потім ініціює Per-frame call до Renderer. Модуль Renderer формує кадр на основі даних з усіх інших компонентів: геометрії сцени з Scene/ModelManager, матриць камери з Camera Controller, векторів осей із Axis Visualizer та графічних ресурсів із Resource Loader.

Група «Час» представлена окремим вузлом Clock/Timer, що функціонує як незалежне джерело поточного часу. Цей модуль не ініціює жодної активності, а лише надає значення часу на запит від EventLoop або Renderer.

Архітектури реалізує однобічний потік даних: від дій користувача через ядро до рендерера, при цьому ресурси та системні залежності забезпечуються окремим каналом. Такий підхід мінімізує зв'язність між модулями, полегшує тестування окремих блоків та дозволяє легко замінювати або масштабувати компоненти, не порушуючи логіку решти системи.

3 ОСНОВНІ РІШЕННЯ З РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Розробка загального алгоритму роботи програмного коду редактора

Робота програмного коду 3D-редактора базується на класичній структурі графічного застосунку:

- послідовна ініціалізація;
- завантаження ресурсів;
- формування початкових матриць;
- основний цикл;
- фіналізація;

Загальна логіка реалізації, яка охоплює ключові етапи життєвого циклу програми представлена блок-схемою на рисунку 3.1. Програмний код приведений у додатку А.

Після запуску система переходить до ініціалізації OpenGL/GLFW (блок 1). На цьому етапі налаштовується контекст рендерингу: задаються версії, профілі, атрибути та викликається `glfwCreateWindow()`. Якщо створення вікна не вдалося (`window == NULL`), керування переходить до гілки завершення – запис звіту у лог-файл (блок 3) і вихід «кінець».

У разі успішного створення вікна виконується завантаження ресурсів (блок 4). До цього етапу належить:

- активація OpenGL-контексту через `glfwMakeContextCurrent`;
- підключення функцій через GLAD;
- реєстрація callback-функцій (ресайз, клавіатура, миша);
- компіляція шейдерних програм (Shader, lineShader, outlineShader);
- створення об'єктів Camera, Model, Axis;
- ініціалізація GPU-ресурсів (VBO, VAO, EBO, текстури).

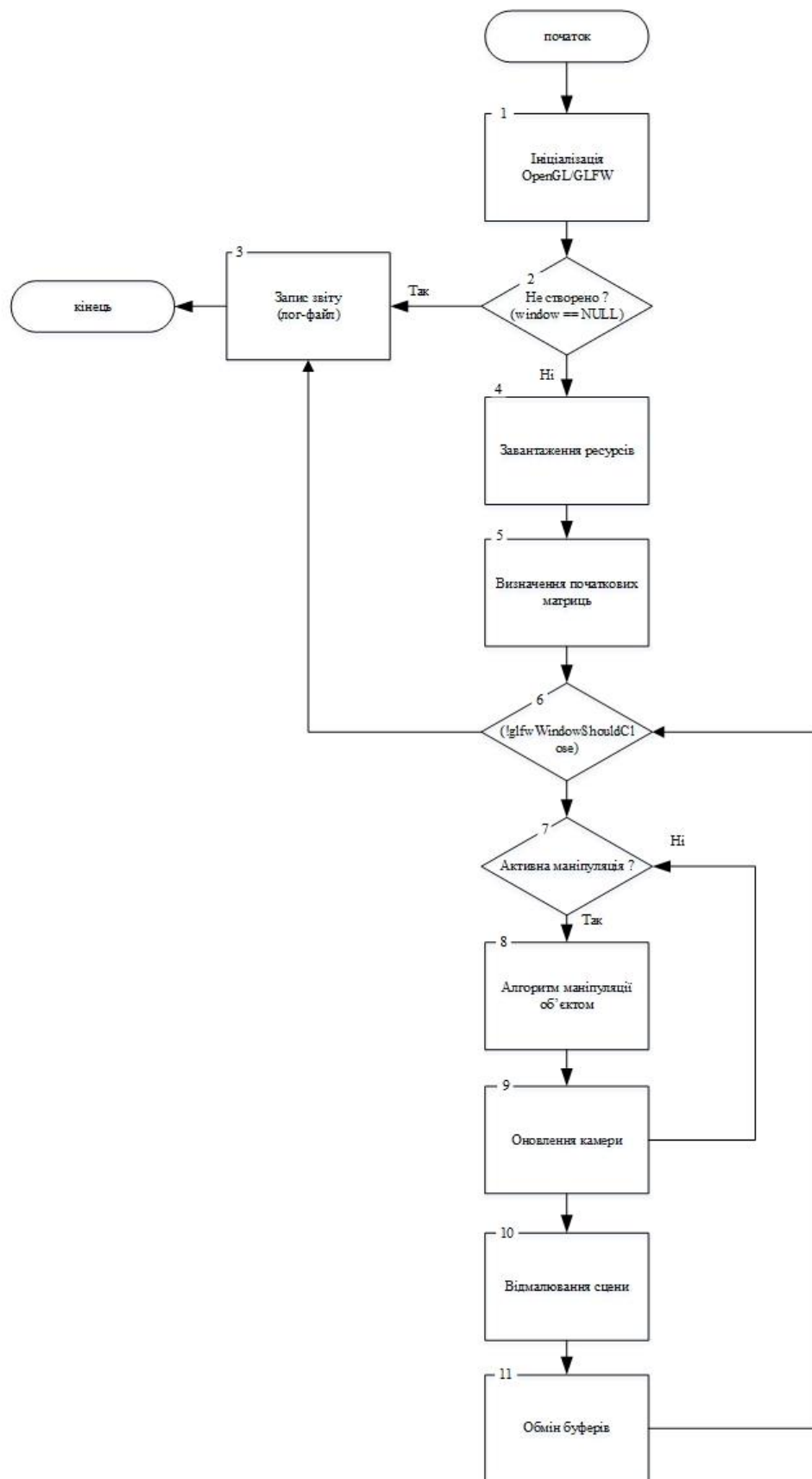


Рисунок 3.1 – Блок-схема загального алгоритму роботи 3D-редактора

Далі відбувається визначення початкових матриць (блок 5). Метод `Camera::updateMatrix()` формує view-projection матрицю з позиції огляду, тоді

як `Model::origin()` фіксує початкові перетворення кожного об'єкта сцени. Це зроблено для точного відстежувати подальші зміни під час маніпуляцій.

Основний цикл починається з перевірки умови `!glfwWindowShouldClose()` (блок 6). Якщо вікно закрито, виконується перехід до звітування та завершення. Інакше – починається обробка кадру.

У кожному циклі виконується:

- вимірювання розміру вікна;
- очищення буферів, кольору і глибини;
- перевірка, чи була активна взаємодія користувача.

Якщо активовано маніпуляцію (блок 7), викликається алгоритм `Model::Inputs()` (блок 8), який відповідає за переміщення, обертання та масштабування поточного об'єкта. Тут враховуються режими `setX/Y/Z`, `setR`, `setG`, `setQ` – кожен із яких відповідає за зміну відповідної координати чи осі трансформації. Зміни фіксуються у векторах `meshPosition`, `meshRotation`, `meshScale`, а також оновлюється матриця трансформацій через `updateMatrix()`. Після цього об'єкт `Axis` викликає `setXYZ()` для оновлення векторів координатної осі `Gizmo`.

Оновлення камери (блок 9) реалізується через `Camera::Inputs()` – залежно від подій `WASD` чи миші змінюється позиція та орієнтація камери, після чого `updateMatrix()` передає актуальні дані до `uniform`-поля `camMatrix` у шейдерах.

Відмалювання сцени (блок 10) відбувається у два проходи:

- основний шейдер виводить геометрію з освітленням і скрінтоном;
- контурний шейдер (`outlineShader`) формує чорну обводку навколо активної моделі. Якщо увімкнено вісь, `Axis::axisActive()` рендерить координатні лінії для `Gizmo`-маніпуляцій.

Після виводу сцени виконується обмін буферів (блок 11) через `glfwSwapBuffers()`, а `glfwPollEvents()` обробляє чергу подій, після чого цикл повторюється.

Якщо користувач закриває вікно, або програма отримує сигнал завершення, цикл переривається. Система викликає деструктори обгортки (Delete, glDeleteBuffers, glDeleteTextures), знищує вікно (glfwDestroyWindow) і завершує контекст (glfwTerminate). У фіналі (блок 3) записує діагностичну і службову інформацію у звіт перед остаточним завершенням.

3.2 Налаштування середовища розробки та підключення бібліотек

На першому етапі технічної реалізації було створено проєкт у середовищі Microsoft Visual Studio 2022. Обрано конфігурацію з шаблоном порожнього C++ проєкту. Структура файлів проєкту побудована відповідно до загальноприйнятих практик у Microsoft Visual Studio 2022: окремі фільтри для заголовочних файлів (Header Files), вихідних кодів (Source Files), шейдерів (Shaders) і текстур (Textures). Структура відображена на рисунку 3.2, а фізичне розташування проєкту на диску – на рисунку 3.3.

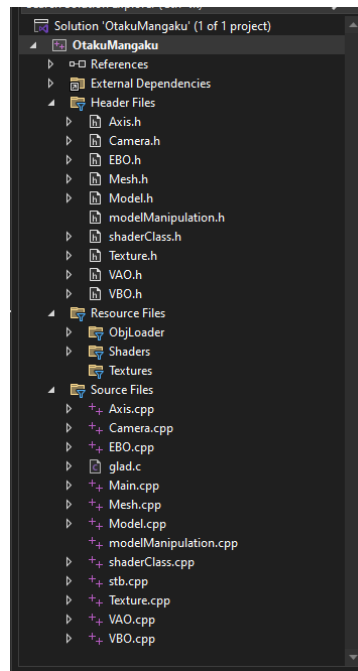
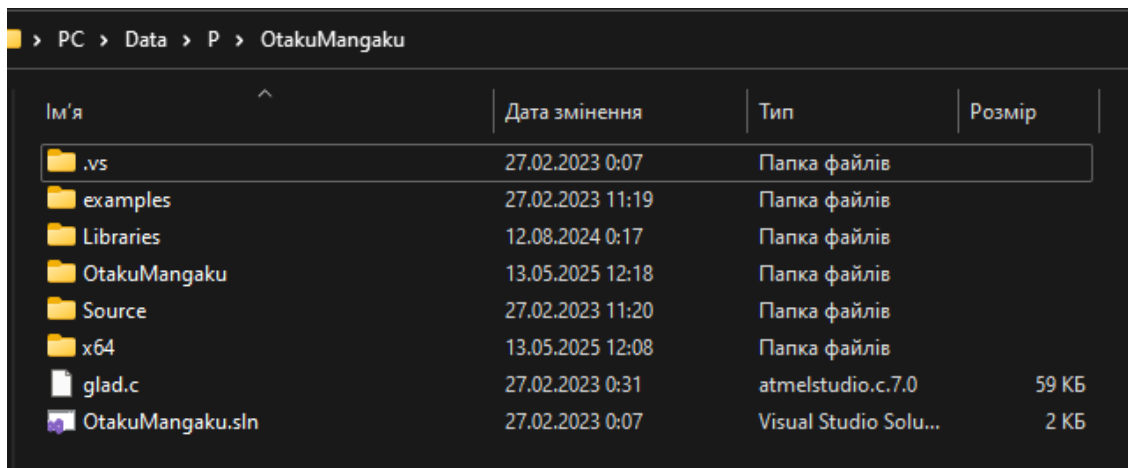


Рисунок 3.2 – Структура файлової системи проєкту



Ім'я	Дата змінення	Тип	Розмір
.vs	27.02.2023 0:07	Папка файлів	
examples	27.02.2023 11:19	Папка файлів	
Libraries	12.08.2024 0:17	Папка файлів	
OtakuMangaku	13.05.2025 12:18	Папка файлів	
Source	27.02.2023 11:20	Папка файлів	
x64	13.05.2025 12:08	Папка файлів	
glad.c	27.02.2023 0:31	atmelstudio.c.7.0	59 КБ
OtakuMangaku.sln	27.02.2023 0:07	Visual Studio Solu...	2 КБ

Рисунок 3.3 – Фізична структура файлів проєкту

До каталогу проєкту було вручну додано папку Libraries (рисунок 3.3), що містить усі необхідні зовнішні заголовочні файли та статичну бібліотеку. Основними підключеними компонентами стали:

- GLFW – бібліотека для створення OpenGL-контексту;
- Glad – OpenGL loader;
- GLM – високопродуктивна бібліотека математичних операцій, що включає векторну алгебру, матриці, кватерніони, а також допоміжні функції для геометрії та трансформацій;
- STB – бібліотека для завантаження текстур;
- nlohmann/json – бібліотека для роботи з JSON-файлами.

Конфігурація підключення включала додавання відповідних шляхів до заголовочних файлів та бібліотек у налаштуваннях проєкту. У розділі C/C++ ==> General ==> Additional Include Directories (рисунок 3.4) вказано шлях:

`$(SolutionDir)Libraries\include`

Що забезпечує доступ до всіх зовнішніх заголовочних файлів під час компіляції.

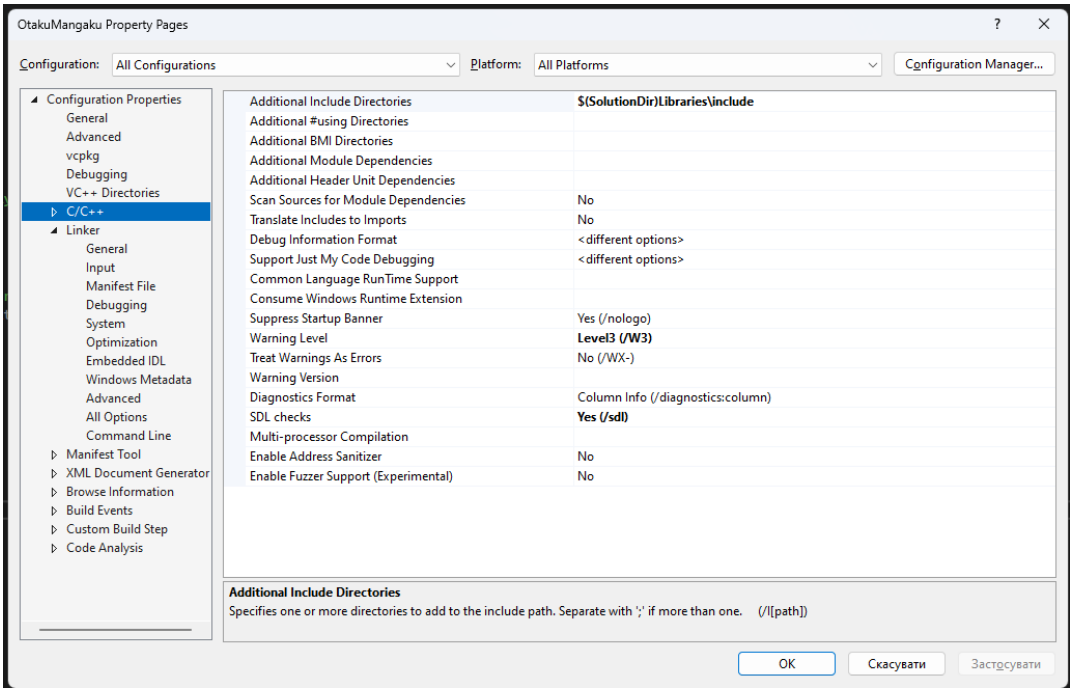


Рисунок 3.4 – Підключення зовнішніх заголовочних файлів

В розділі Linker ==> Input список бібліотек (glfw3.lib) додається до Link Library Dependencies, що дозволяє зібрати виконуваний файл із використанням статичних компонентів GLFW (рисунок 3.5).

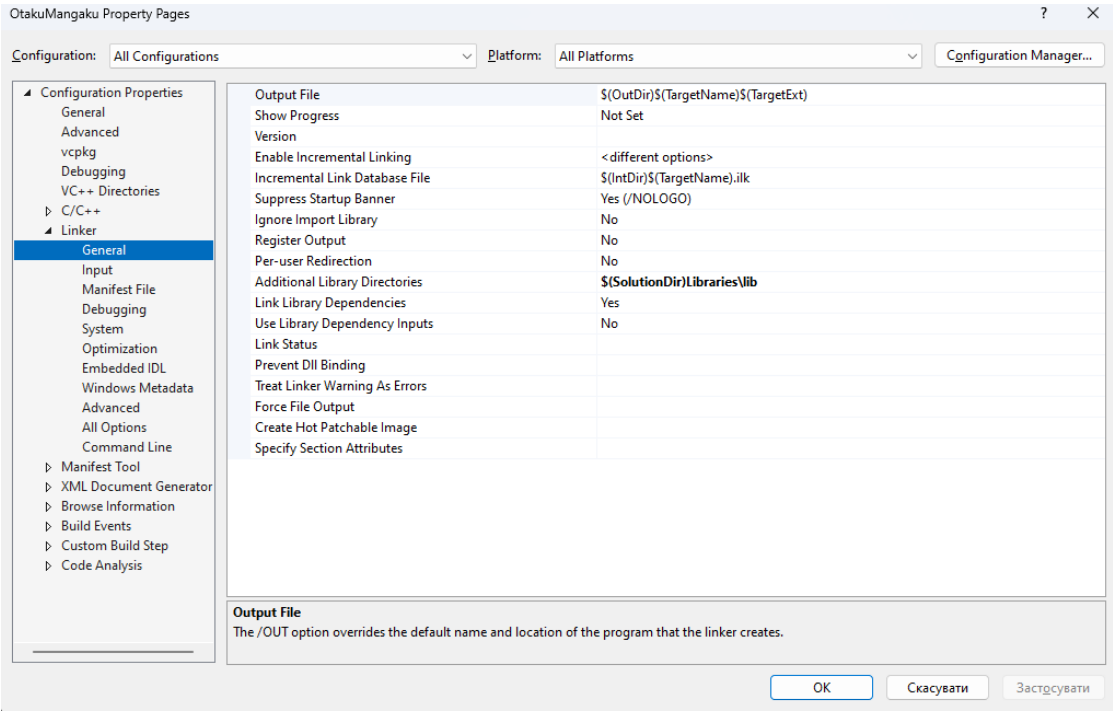


Рисунок 3.5 – Підключення зовнішньої статичної бібліотеки

Таким чином, середовище Visual Studio 2022 дозволило гнучко реалізувати підтримку зовнішніх залежностей через централізоване підключення директорій. Це значно спростило структуру проєкту, зменшило кількість ручного дублювання шляху до файлів та полегшило перенесення проєкту на інші машини.

3.3 Розробка програмних модулів редактора

На рисунку 3.6 представлено UML-діаграму класів, що реалізують об'єктно-орієнтовану архітектуру 3D-редактора. Уся система побудована на чіткій ієрархії абстракцій, де кожен рівень інкапсулює відповідну групу обов'язків: управління ресурсами GPU, обробку геометрії, візуалізацію, маніпуляцію моделями та організацію сцени.

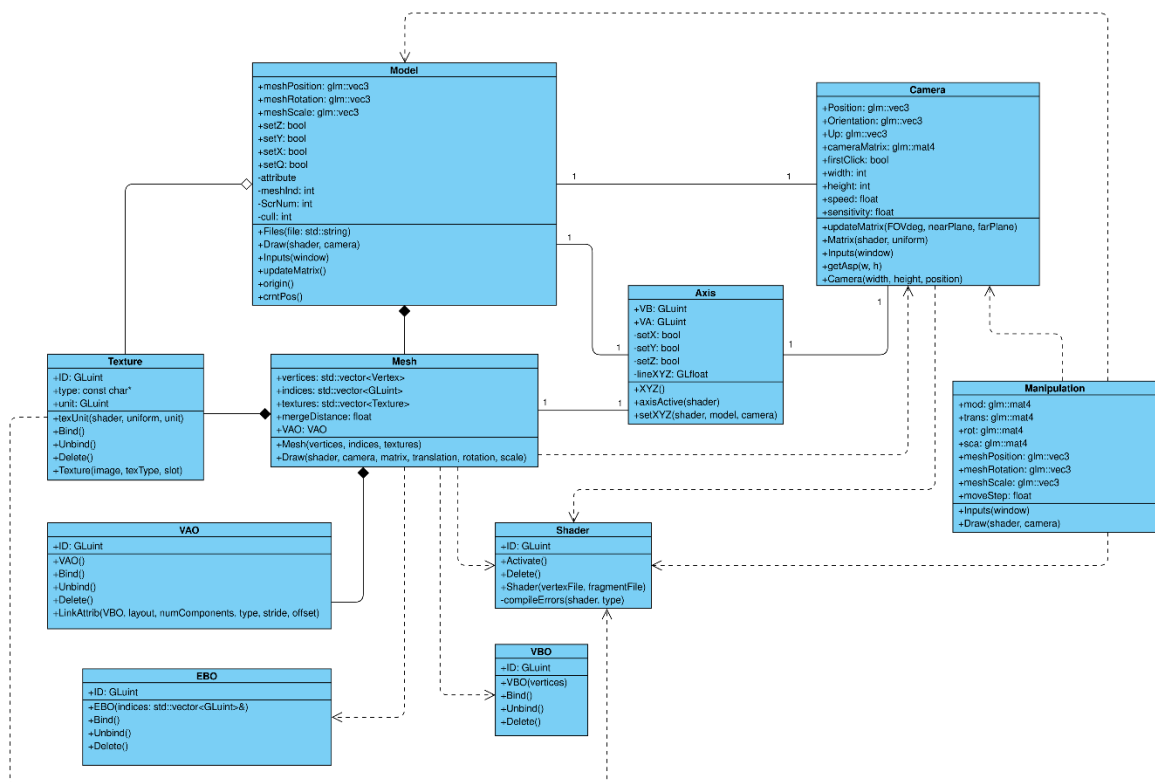


Рисунок 3.6 – UML-діаграма основних класів і зв'язків 3D-редактора

На нижньому рівні знаходяться класи обгортки над OpenGL-ресурсами: VBO, EBO, VAO та Texture. Кожен з них інкапсулює ідентифікатор GPU об'єкта (GLuint ID) і базовий набір методів для прив'язки (Bind()), відв'язки (Unbind()), та видалення (Delete()). Це реалізує патерн RAII, що гарантує контроль над життєвим циклом об'єктів без залучення зовнішнього procedural-коду [13]-[16]. Клас Shader окремо компілює та лінкує GLSL-програми, надаючи високорівневий інтерфейс активації (Activate()) та обробки помилок компіляції.

Наступний рівень представлений класом Mesh, який виступає геометричним контейнером. Об'єкт Mesh зберігає масиви вершин (vertices), індексів (indices) і текстур (textures). Під час ініціалізації він автоматично створює пов'язаний VAO і прив'язує атрибути до буфера. Метод Draw() інкапсулює повну процедуру виведення об'єкта — від активації шейдера і передачі уніформ, до виклику glDrawElements, враховуючи трансформації (translation, rotation, scale) і камеру.

Клас Model виконує роль ядра 3D-редактора, поєднуючи завантаження, зберігання, трансформацію й рендеринг геометрії, методи класу наведені у таблиці 3.1.

Таблиця 3.1 – Методи класу Model

Метод	Параметри	Опис функції
Model::Model()	std::string file	Конструктор.
Files()	const std::string& path	Завантажує glTF-файл, парсить JSON, викликає traverseNode()
Draw()	Camera& camera	Виводить усі меші, прив'язуючи камеру та матриці трансформації.
updateMatrix()	GLFWwindow* window	Оновлює модельну матрицю активного об'єкта (трансляція, ротація, масштаб)
origin()	-	Зберігає початкові transforms Mesh.

Продовження таблиці 3.1

Метод	Параметри	Опис функції
crntPos()	-	Встановлює поточні позиції/ротації/масштаб активного меша з allPos/allRot/allSca
Inputs()	GLFWwindow* window, glm::vec3& lightPos	Обробка клавіш користувача
keyUnCheck()	-	Скидає внутрішні флаги вводу (setR, setG, setQ тощо)
loadMesh()	unsigned int indMesh	Завантажує меш за індексом з JSON, парсить вершини, UV, нормалі, текстури
traverseNode()	unsigned int nodeIndex, glm::mat4 matrix	Рекурсивно обходить вузли сцени, читає трансформації, додає Mesh
getFloats()	json accessor	Повертає масив чисел float з буфера по JSON-accessor
Метод	Параметри	Опис функції
getIndices()	json accessor	Повертає індекси вершин (GLuint) з буфера
getTextures()	-	Завантажує текстури за шляхами у glTF, уникаючи дублювання
screenTones()	-	Повертає поточний screenTone як текстуру
assembleVertices()	std::vector<vec3>, std::vector<vec3>, std::vector<vec2>	
getData()	-	Завантажує байти з .bin-файлу, прив'язаного до glTF
groupFloatsVec2()	std::vector<float> floatVec	Перетворює float-масив у std::vector<vec2>

Продовження таблиці 3.1

Метод	Параметри	Опис функції
groupFloatsVec3()	std::vector<float> floatVec	Перетворює float-масив у std::vector<vec3>
groupFloatsVec4()	std::vector<float> floatVec	Перетворює float-масив у std::vector<vec4>

Після ініціалізації методом Files(const std::string& file) класу парсинг glTF-опису та читання бінарних буферів через приватну функцію getData(). Далі метод класу Model traverseNode() послідовно обходить структурний граф сцени, для кожного вузла відтворюючи локальні трансформації та накопичуючи їх у матрицю matNextNode. Коли вузол містить атрибут mesh, викликається loadMesh(), що формує об'єкт Mesh із вершин, індексів і пов'язаних текстур та додає його до вектору meshes. Одночасно вихідні трансформації записуються у паралельні контейнери translationsMeshes, rotationsMeshes, scalesMeshes, а повна модельна матриця – у matricesMeshes (фрагмент коду наведено на рсунку 3.7), що спрощує подальший доступ за індексом meshInd.

```

63
64 // Store Original Rotation
65 for (unsigned int i = 0; i < meshes.size(); i++)
66 {
67     std::cout << "Index:" << i << std::endl;
68
69     std::cout << "Rotation size:" << allRot.size() << std::endl;
70
71     glm::vec3 eulerAngles = glm::degrees(glm::eulerAngles(rotationsMeshes[i]));
72
73
74     allRot[i] = glm::degrees(eulerAngles);
75
76     allRot[i] = glm::radians(allRot[i]);
77     allPos[i] = translationsMeshes[i];
78     allSca[i] = scalesMeshes[i];
79
80     std::cout << "Original rotation:" << allRot[i].x << ", " << allRot[i].y << ", " << allRot[i].z << std::endl;
81 }
82
83
84
85 void Model::Draw(Shader& shader, Camera& camera)
86 {
87     // Go over all meshes and draw each one
88     for (unsigned int i = 0; i < meshes.size(); i++)
89     {
90
91
92         meshes[i].Mesh::Draw(shader, camera, matricesMeshes[i]);
93         glUniform1f(glGetUniformLocation(shader.ID, "outline"), outlineSize);
94         glUniform1f(glGetUniformLocation(shader.ID, "ScrTSize"), ScrTSize);
95     }
96

```

Рисунок 3.7 – Фрагмент коду функції origin() і Draw() класу Model для збереження трансформацій

Для інтерактивних маніпуляцій редактор зберігає поточні координати активного об'єкта у полях `meshPosition`, `meshRotation`, `meshScale`. Кожна зміна цих значень ініціює виклик `updateMatrix()`, у якому обчислюється оновлена матриця `matricesMeshes[meshInd]` шляхом комбінування початкових даних з новими трансляціями, обертаннями та масштабом. Така схема уможливорює незалежне редагування окремої геометричної сутності без втрати зв'язку з початковою ієрархією сцени.

Під час кадру рендерингу публічний метод `Draw(Shader& shader, Camera& camera)` ітерує вектор `meshes`, попередньо активувавши переданий шейдер та встановивши уніформи оглядової матриці (`camMatrix`), параметри контурної обводки (`outline`) і масштаб скрінтона (`ScrTSize`). Далі кожен об'єкт `Mesh` викликає власний `Mesh::Draw()`, що завершується апаратним викликом `glDrawElements`. Таким чином, логіка відображення повністю інкапсульована в межах класу, а зовнішній код оперує єдиним викликом `Model::Draw()`.

Клас реалізує композиційний патерн: `Model` володіє множиною `Mesh`, які, у свою чергу, акумулюють `Texture`, `VAO`, `VBO` та `EBO`. Подібна вкладеність забезпечує автоматизоване вивільнення GPU-ресурсів у деструкторах нижчих рівнів без додаткового коду управління пам'яттю. Взаємодія з користувачем реалізується в методі `Inputs(GLFWwindow* window)`, де обробляються події клавіатури й миші: режими `R/G/Q` перемикають обертання, переміщення та масштабування; клавіші `X/Y/Z` конкретизують вибір осі; комбінації та регулюють параметри візуалізації, включаючи масштаб скрінтона й товщину контуру. Локальні булеві прапорці `setX`, `setY`, `setZ` фіксують активну вісь, забезпечуючи безконфліктну роботу маніпуляторів навіть за швидкої зміни режимів [17]-[19].

Клас `Camera` реалізує функції позиціонування точки огляду у сцені. Він зберігає позицію, орієнтацію, напрямок вгору та обчислює `view-projection` матрицю за допомогою `updateMatrix()`. Спадкування в системі не застосовується, проте `Camera` прямо передається до інших компонентів

(наприклад, Shader, Mesh, Model) через параметри методів — це вказує на використання агрегації: камера логічно пов'язана, але не є частиною об'єктів сцени.

Допоміжний клас Axis реалізує побудову орієнтирів X/Y/Z для маніпуляцій. Він оперує внутрішніми флагами setX, setY, setZ, побудовою буфера lineXYZ і виводом ліній за допомогою axisActive() та setXYZ(). При цьому позиція осей базується на координатах активного об'єкта Model, що вказує на використання без повного володіння.

Окремий клас Manipulation – це тонка обгортка над трансформаціями. Він містить власні матриці trans, rot, sca і координати трансформації, дублюючи функціональність Model, але в ізольованому середовищі. Це дозволяє реалізовувати gizmo-набір для інтерактивної зміни позиції та орієнтації об'єкта без впливу на внутрішню матрицю моделі.

Таким чином, архітектура будується навколо агрегаційних та композиційних зв'язків, де низькорівневі класи надають функціональність через публічні інтерфейси, середній рівень (Mesh, Model, Camera) відповідає за логіку сцени, а верхній (Axis, Manipulation, Main) — за взаємодію з користувачем та управління візуалізацією. Відсутність глибокої ієрархії спадкування робить систему гнучкою до розширення без необхідності уніфікації базових типів [20]-[22].

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Загальні положення

Наведено огляд на експлуатаційне тестування та експериментальне дослідження програмного забезпечення дипломного проєкту. Зазначений етап завершує стартовий життєвий цикл програмного продукту й має на меті об'єктивно оцінити його технічну спроможність виконувати заявлений у підрозділі 1.6 функціонал.

Експлуатаційне тестування у даному контексті трактується як перевірка стабільності, надійності та відмовостійкості системи в середовищі, що моделює робочі сценарії кінцевого користувача. Воно передбачає довготривале спостереження за роботою програми при виконанні ключових операцій (завантаження 3D-моделей, зміна розміру скрінтону) та фіксацію реакції на нестандартні ситуації, зокрема пікові навантаження або відсутність зовнішніх ресурсів.

Експериментальне дослідження доповнює тестування емпіричною перевіркою продуктивності: використання ресурсів ЦП/ГП, обсяг оперативної пам'яті та поведінка під час зміни роздільної здатності чи перемикання у повноекранний режим. Такі вимірювання дають змогу кількісно оцінити правильність роботи реалізованих алгоритмів рендерингу й визначити потенційні «вузькі місця» реалізації певних компонентів системи.

Важливою особливістю досліджуваного застосунку є його здатність працювати без сторонніх динамічних бібліотек. Завдяки статичному лінкуванню залежностей GLFW для запуску достатньо базової підтримки OpenGL на цільовій системі. Це спрощує розгортання та дозволяє оцінити працездатність у мінімальному середовищі, включно з віртуальними машинами, на яких відсутні SDK або інструменти розробки.

Отримані результати слугуватимуть підґрунтям для подальшої оцінки придатності програмного забезпечення до розгортання на сторонніх платформах.

4.2 Підготовка до тестування

З урахуванням методологічних засад було сформовано тестове середовище, що дозволяє здійснити практичну перевірку працездатності програмного забезпечення за умов обмеженої системної підтримки. У якості платформи перевірки обрано віртуальну машину vmware з типовою конфігурацією кінцевого користувача, де відсутні засоби розробки, SDK та попередньо встановлені зовнішні бібліотеки.

Перед розгортанням застосунку було сформовано компактну директорію дистрибутива у конфігурації Release (Рисунок 4.1), що включає виконуваний файл OtakuMangaku.exe та повну структуру каталогу Asset. Каталог містить усі необхідні ресурси – сцени у форматі glTF, відповідні .bin-файли, текстури у форматах PNG, JPEG, а також графічні елементи скрінтонів, що використовуються у візуалізації. Усі зовнішні залежності, зокрема бібліотека GLFW, були включені до складу бінарного файлу на етапі компіляції за допомогою статичного лінкування.

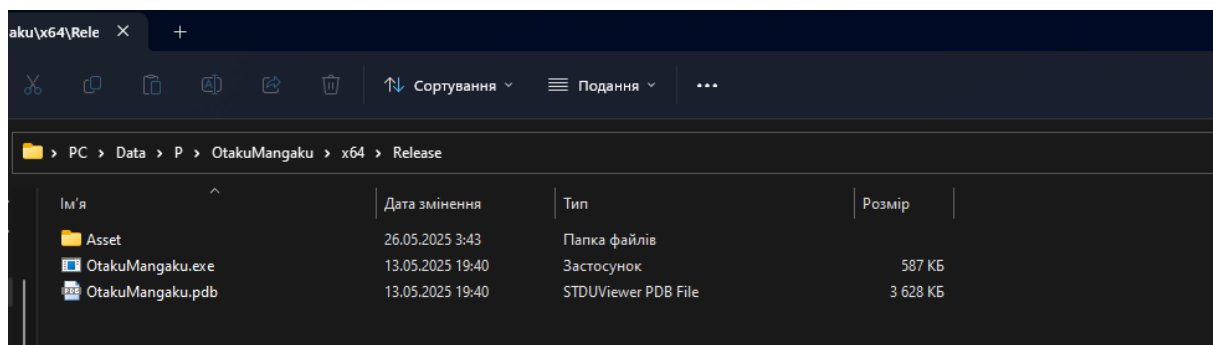


Рисунок 4.1 – Робоча папка програми OtakuMangaku

Під час перенесення застосунку на цільову систему не передбачалось встановлення жодного допоміжного програмного забезпечення. Запуск програми здійснюється без інсталяції, шляхом запуску виконуваного файлу у скопійованому середовищі. Таким чином, тестування фокусувалося на перевірці готовності до експлуатації за умови наявності базового OpenGL-драйвера, інтегрованого до складу графічної підсистеми Windows.

4.3 Розгортання та запуск на цільовій системі

Перед запуском було виконано перенесення вмісту релізної директорії, сформованої на етапі компіляції, на цільову віртуальну машину рисунок 4.2. Копіювання здійснювалось за допомогою загального мережевого ресурсу без інсталятора або інструментів пакетного розгортання. Після перенесення було перевірено цілісність структури: присутність виконуваного файлу OtakuMangaku.exe а також наявність усіх підкаталогів з ресурсами у папці Asset, включно з текстурами, сценами й допоміжними файлами.

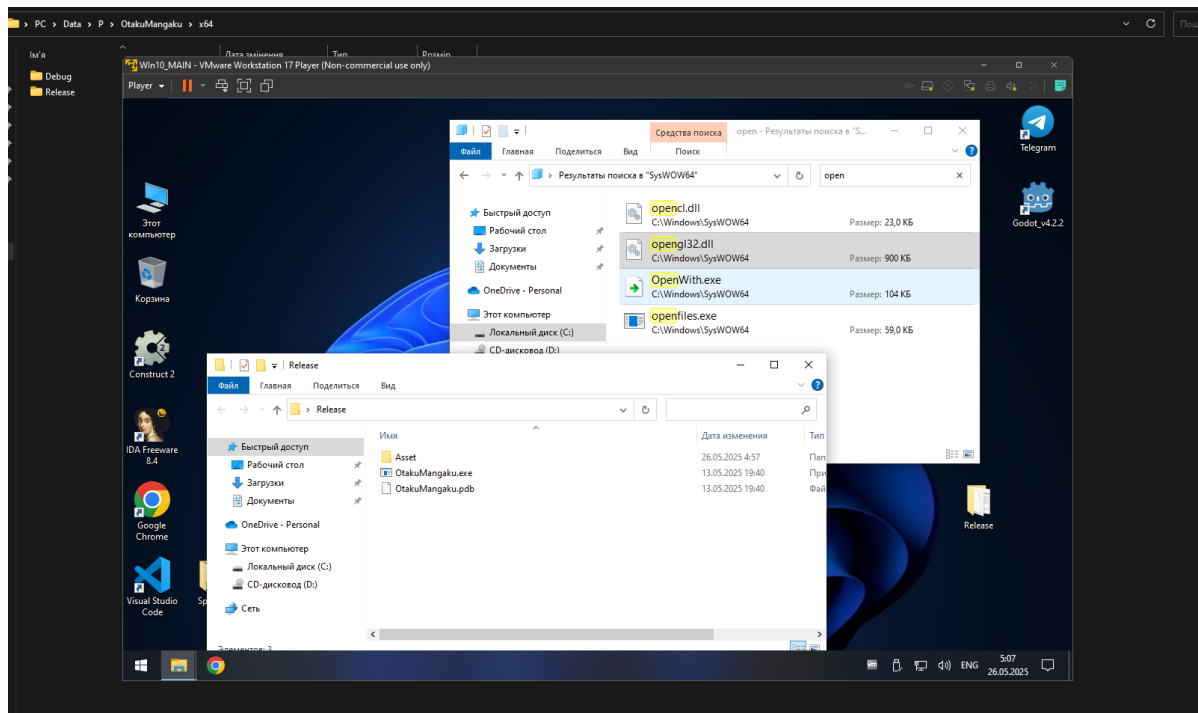


Рисунок 4.2 – Вміст директорії з програмою OtakuMangaku на віртуальній машині після копіювання релізної збірки

Після перевірки структури було здійснено запуск програми шляхом прямого виклику виконуваного файлу рисунок 4.3. Програма ініціювалась коректно: було відкрито графічне вікно з очікуваним розміром, успішно завантажено меш сітки за замовчуванням, застосовано базову систему освітлення та текстурування. Сигналів про помилки ініціалізації або повідомлень про відсутність динамічних компонентів не зафіксовано.

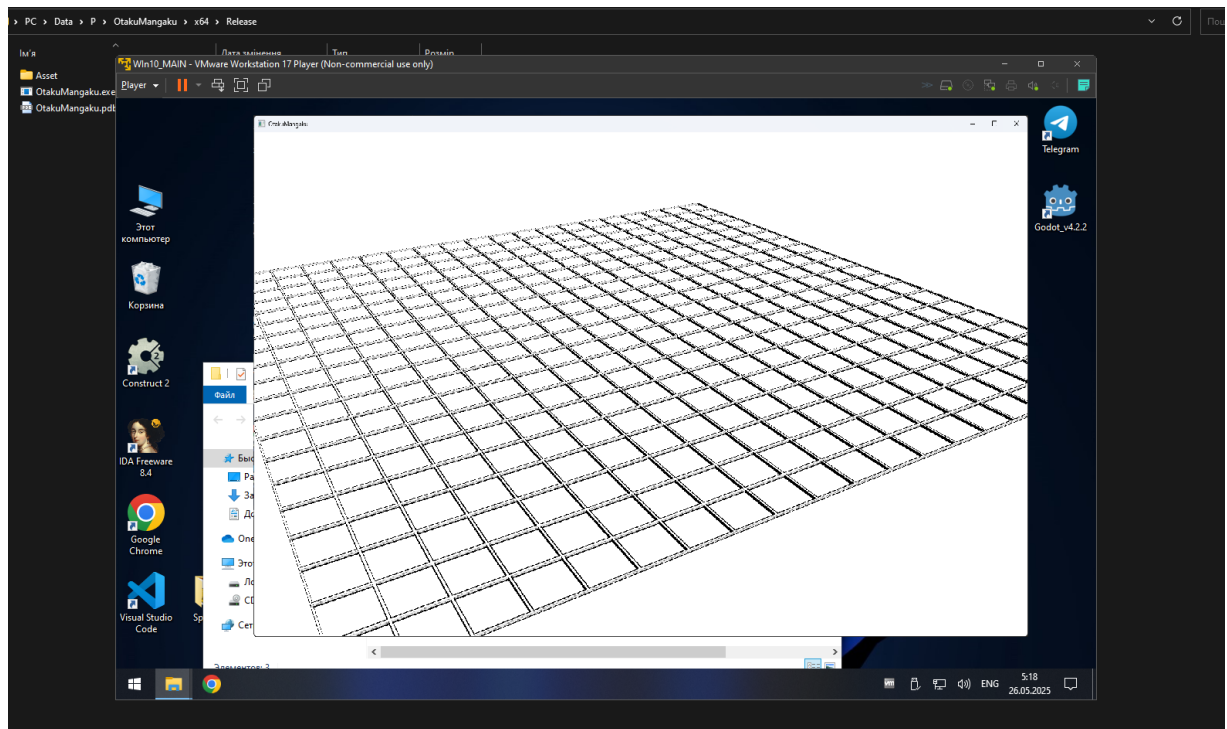


Рисунок 4.3 – Результат запуску програми OtakuMangaku на віртуальній машині

Під час запуску також перевірено реакцію програми на потенційні збої, зокрема на симуляцію сценаріїв – таких як зміна роздільної здатності, переключення до повноекранного режиму, відсутність окремих текстур або часткове порушення цілісності ресурсів. При навмисному порушенні фалів у відповідь система коректно вивела повідомлення про неможливість завантаження, без аварійного завершення процесу. Це свідчить про коректне

віпрацювання базових механізмів обробки виключних ситуацій у внутрішній логіці завантажувача ресурсів.

Запуск вважається успішним на підставі таких критеріїв: відсутність критичних помилок на етапі ініціалізації, наявність зображення вікна з очікуваним вмістом сцени, працездатність елементів взаємодії (керування камерою, реагування на натискання клавіш). Загальна поведінка системи відповідала заданим очікуванням і підтвердила її готовність до подальшого етапу функціонального тестування.

4.4 Перевірка функціональності

Після успішного запуску програми в умовах цільового середовища було проведено тестування основних функціональних компонентів застосунку з акцентом на їхню стабільність, інтерактивність та відповідність заявленим можливостям. Перевірка охоплювала ключові сценарії взаємодії, зокрема завантаження 3D-моделей, застосування ефектів скрінтонів, маніпуляції з об'єктами сцени та перемикання сцен.

Під час тестування завантаження GLTF-моделей було використано декілька варіантів сцен: `Cube.glTF`, `Scene.glTF`, `Hair.glTF`, `FigureModel.glTF`. Усі моделі були коректно завантажені у пам'ять застосунку, а відповідні текстури автоматично застосувалися до меш-сіток (рис. 4.4). Жодних помилок у логах чи візуальних артефактів виявлено не було. Час завантаження кожної сцени склав менше однієї секунди, що відповідає очікуванням для локального рендерера без потокового декодування. Програма коректно оновлює відображення сцени після кожного імпорту моделі без необхідності перезапуску або скидання контексту.

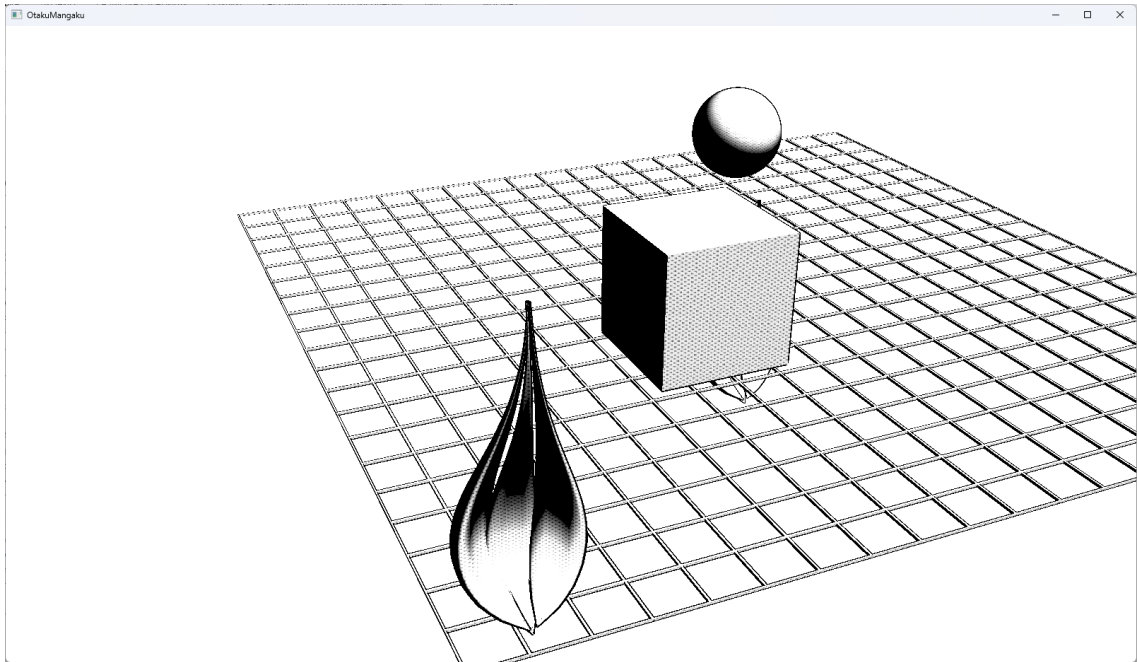


Рисунок 4.4 – Результат завантаження GLTF-моделей із попередньо включених ресурсів

Наступним етапом було перевірено завантаження GLTF-моделей безпосередньо з файлової системи користувача шляхом вибору моделі через вікно діалогу (рисунок 4.5). Після підтвердження вибраного шляху відповідна модель динамічно завантажувалася у сцену без збоїв і затримок. Стан графічного вікна при цьому автоматично оновлювався, а всі текстури застосовувались коректно (рисунок 4.6).

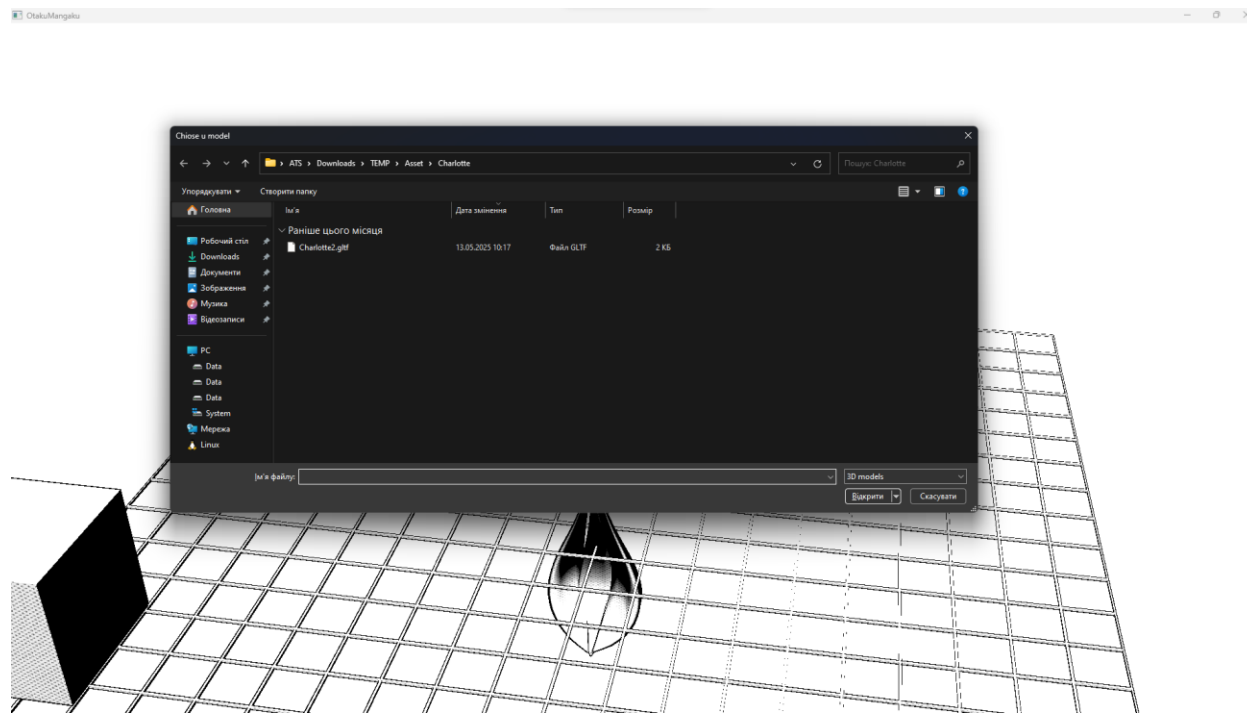


Рисунок 4.5 – Завантаження GLTF-моделі за допомогою вікна вибору файлу

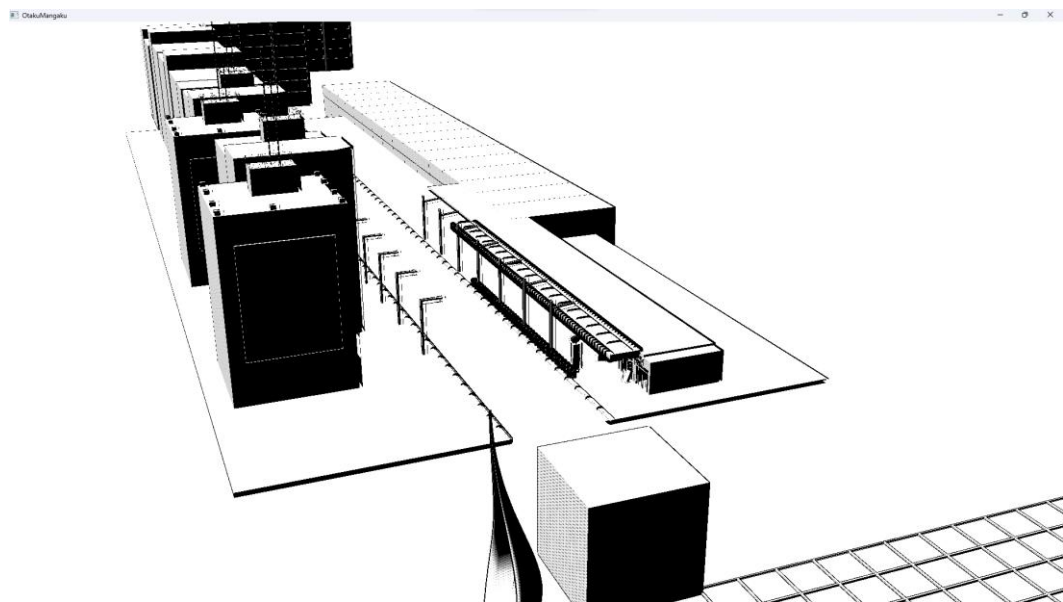


Рисунок 4.6 – Результат завантаження GLTF-моделі

Під час імпорту нової моделі в логах не було зафіксовано помилок або конфліктів при повторному завантаженні.

Окремо протестовано можливість конфігурації сцени для створення фону в стилі манги з використанням скрінтонів, текстур фону та освітлення. Результат формування сцени з активними візуальними ефектами представлено на рисунку 4.7.

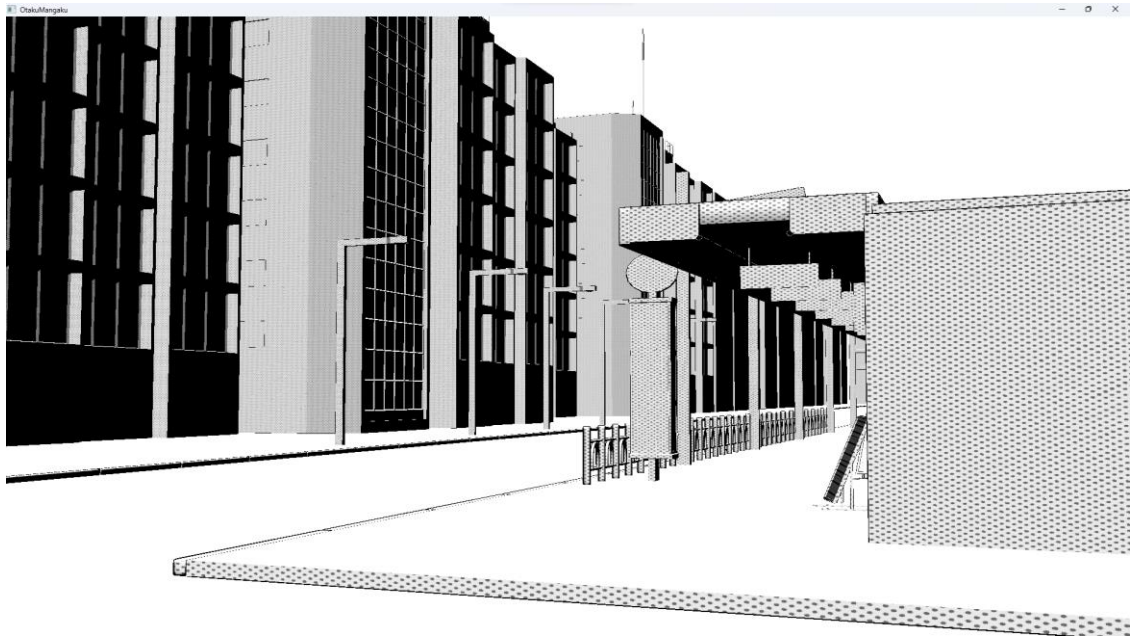


Рисунок 4.7 – Приклад сформованої сцени з фоном у для мангаки

Результати функціонального тестування демонструють відповідність реалізованих компонентів технічним вимогам, заявленим у підрозділі 1.6. Застосунок стабільно виконує основні операції з 3D-сценами, забезпечує гнучкість у взаємодії з візуальними елементами та не виявляє критичних відмов при стандартному або розширеному використанні.

4.5 Аналіз продуктивності

У межах експлуатаційного тестування було здійснено вимірювання використання системних ресурсів під час виконання ключових операцій, пов'язаних із завантаженням моделей, зміною сцен, застосуванням візуальних

ефектів і маніпуляцією об'єктами в 3D-просторі. Дослідження проводилося з використанням вбудованого диспетчера задач операційної системи Windows, що дозволило відстежити в реальному часі навантаження на центральний процесор, обсяг зайнятої оперативної пам'яті, а також активність графічного адаптера (рисунок 4.8).

Ім'я	Стан	22% ЦП	15% Пам'ять	0% Диск	0% Мережа
Razer AppEngine		1,4%	57,7 МБ	0 Мбіт/с	0 Мбіт/с
Диспетчер вікон		0,7%	71,1 МБ	0 Мбіт/с	0 Мбіт/с
OVR Redir		0,5%	1,7 МБ	0 Мбіт/с	0 Мбіт/с
Google Chrome (40)		0,3%	4 607,1 МБ	0,1 Мбіт/с	0,1 Мбіт/с
Диспетчер завдань		0,3%	79,9 МБ	0 Мбіт/с	0 Мбіт/с
RZSurroundService		0,3%	3,2 МБ	0 Мбіт/с	0 Мбіт/с
OtakuMangaku		0,2%	186,9 МБ	0 Мбіт/с	0 Мбіт/с
Docker Desktop (4)		0%	186,8 МБ	0 Мбіт/с	0 Мбіт/с
Microsoft Visual Studio 2022 (25)		0%	1 155,1 МБ	0 Мбіт/с	0 Мбіт/с

Рисунок 4.8 – Навантаження на ресурси системи під час роботи OtakuMangaku

Таким чином, продуктивність OtakuMangaku відповідає вимогам до легкої настільної графічної утиліти з фокусом на стилізоване відображення сцен у режимі реального часу. Низьке використання системних ресурсів дозволяє розглядати програму як придатну для розгортання на системах зі скромними технічними характеристиками без суттєвих втрат функціональності чи стабільності.

ВИСНОВКИ

В дипломній роботі було реалізовано програмне забезпечення для формування сцен із тривимірних моделей у стилі мангаки. Розробка виконувалася з урахуванням потреб користувачів, які не мають спеціалізованих навичок у сфері 3D-графіки, та орієнтована на спрощене створення фону для манга-композицій із мінімальним порогом входження.

У процесі роботи проаналізовано популярні інструменти для 3D-візуалізації, зокрема Blender та Unity, та виявлено їх недоліки з точки зору початкової складності, високих апаратних вимог і залежності від зовнішніх модулів. Для усунення цих недоліків було створено редактор OtakuMangaku, який не потребує інсталяції, SDK або окремих графічних рушіїв, працює на базі OpenGL із використанням GLFW, GLM і статичного лінування.

Функціональне тестування підтвердило стабільність роботи редактора: усі основні операції – завантаження glTF-моделей, застосування скрінтонів, трансформації об'єктів — виконуються без збоїв. Візуальні ефекти застосовуються у реальному часі, а використання ресурсів (CPU та RAM) залишається в межах, характерних для невимогливих застосунків.

Таким чином, створене програмне забезпечення відповідає поставленим завданням і забезпечує необхідну функціональність. Розроблене ПЗ може використовуватись як базовий інструмент для швидкого створення заднього фону мангак без потреби в налаштуванні шейдерів, вивченні складного інтерфейсу, чи спеціалізованих знань у 3D-моделюванні.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. <https://en.wikipedia.org/wiki/Manga> // Wikipedia.
2. Manga print sales statistics. animetics. URL: <https://animetics.net/us-manga-print-sales-statistics/> (date of access: 29.05.2025). – Title from screen.
3. Williams J. 12 best drawing software and apps for manga and illustration [Electronic resource] / Joseph Williams // eagle. – Mode of access: <https://en.eagle.cool/blog/post/12-best-drawing-software-and-apps-for-manga-and-illustration> (date of access: 05.06.2025). – Title from screen.
4. Pröschl V. How do polygonal models work? | Computer Graphics Essentials [Electronic resource], 2024 / Vojtěch Pröschl // youtube. – Mode of access: https://www.youtube.com/watch?v=YDXP_-UczHg (date of access: 05.06.2025).
5. Chen Y. Polygonal modeling [Електронний ресурс] : навч. посіб. / Yiqun Chen. – [Б. м. : б. в.]. – 284 с. – Режим доступу: <https://courses.cs.washington.edu/courses/cse458/05au/help/mayaguide/Complete/Polygons.pdf> (дата звернення: 06.06.2025).
6. Qin Zheng. Vector graphics for real-time 3D rendering [Electronic resource] / Qin Zheng. – [S. l.]. – Mode of access: <https://uwspace.uwaterloo.ca/items/deaf3347-337e-49f3-8ec5-f30ced872a62> (date of access: 06.06.2025). – Title from screen.
7. Schroeder W. J. The design and implementation of an object-oriented toolkit for 3D graphics and visualization [Electronic resource] / William J. Schroeder. – P. 9. – Mode of access: <https://scispace.com/pdf/the-design-and-implementation-of-an-object-oriented-toolkit-2nxu1tm1qx.pdf> (date of access: 06.06.2025). – Title from screen.
8. Vries J. D. Learn modern opengl graphics programming in a step-by-step fashion [Electronic resource] : навчальний посібник / Joey De Vries. – [S. l. : s. n.], 2020. – 552 p. – Mode of access: https://learnopengl.com/book/book_pdf.pdf (date of access: 06.06.2025).

9. OpenGL 4 shading language cookbook: build high-quality, real-time 3D graphics with opengl 4.6, GLSL 4.6 and C++17, 3rd edition. – [S. l.] : Packt Publishing, 2018. – 472 p. – Title from screen.
10. Rodríguez J. GLSL essentials / Jacobo Rodríguez. – [S. l.] : Packt Publishing, 2013. – 116 p.
11. Glfw [Electronic resource] // glfw. – Mode of access: <https://www.glfw.org/> (date of access: 06.06.2025). – Title from screen.
12. Gamma E. Design patterns elements of reusable object-oriented software. 417 p. URL: <https://www.javier8a.com/itc/bd1/articulo.pdf> (date of access: 29.05.2025). – Title from screen.
13. Gordon S. Computer Graphics Programming in OpenGL with C++ : підручник / Scott Gordon. – 2nd ed. – [S. l.] : Mercury Learning and Information, 2021. – 514 p.
14. Engel W. GPU pro: advanced rendering techniques : підручник / Wolfgang Engel. – [S. l.] : A K Peters/CRC Press, 2010. – 740 p.
15. Visual Studio documentation [Electronic resource] // microsoft. – Mode of access: <https://learn.microsoft.com/en-us/visualstudio/windows/?view=vs-2022> (date of access: 06.06.2025). – Title from screen.
16. Hussain K. OpenGL deep dive: expert techniques and performance optimization : підручник / Kameron Hussain. – [S. l.] : Sonar Publishing, 2023. – 364 p.
17. Blokdyk G. Software design pattern A complete guide - 2020 edition : підручник / Gerardus Blokdyk. – [S. l.] : 5STARCooks, 2021. – 305 p.
18. Designing components with the C++ STL: a new approach to programming (revised edition). – [S. l.] : Addison-Wesley Professional, 2000. – 320 p.
19. Iglberger K. C++ software design: design principles and patterns for high-quality software : підручник / Klaus Iglberger. – [S. l.] : O'Reilly Media, 2022. – 435 p.
20. O'Dwyer A. Mastering the C++17 STL: Make full use of the standard

library components in C++17 / Arthur O'Dwyer. – [S. l.] : Packt Publishing, 2017. – 384 p.

21. O'Dwyer A. Anton's OpenGL 4 Tutorials : підручник / Arthur O'Dwyer. – [S. l.] : Packt Publishing, 2017. – 384 p.

22. Gerdelan A. Anton's OpenGL 4 Tutorials / Anton Gerdelan. – 7th ed. – [S. l. : s. n.], 2014. – 607 p.

ДОДАТОК А
Слайди презентації

Національний університет «Запорізька політехніка»

Кафедра програмних засобів

Дипломна кваліфікаційна робота бакалавра

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
3D РЕДАКТОРУ

Виконав

ст. гр. КНТ-111

Іван БРОВЧЕНКО

Керівник

к.т.н, доцент

Сергій СЕРДЮК

1

Рисунок А.1 – Слайд вступу

Мета роботи

- Об'єкт дослідження – процеси створення 3D-зображень
- Предмет дослідження – програмні засоби формування 3D-сцен та методи спрощеного рендерингу для художніх задач.
- Мета роботи – створити просту систему, яка дозволяє створювати бюджетне фони для манги за допомогою імпорту GLTF-моделей та основних засобів керування освітленням, а також технік скретчингу, без потреби глибоких знань графічного мистецтва чи 3D-моделювання в користувача програмного забезпечення.

2

Рисунок А.2 – Слайд мети роботи

Задачі

- Аналіз предметної області та існуючих рішень.
- Проєктування архітектури 3D-редактора.
- Розробка функціональних модулів системи.
- Реалізація стилізованого рендерингу та імпорту сцен.
- Тестування, експлуатаційна перевірка та оцінка стабільності роботи програми.

3

Рисунок А.3 – Слайд задач

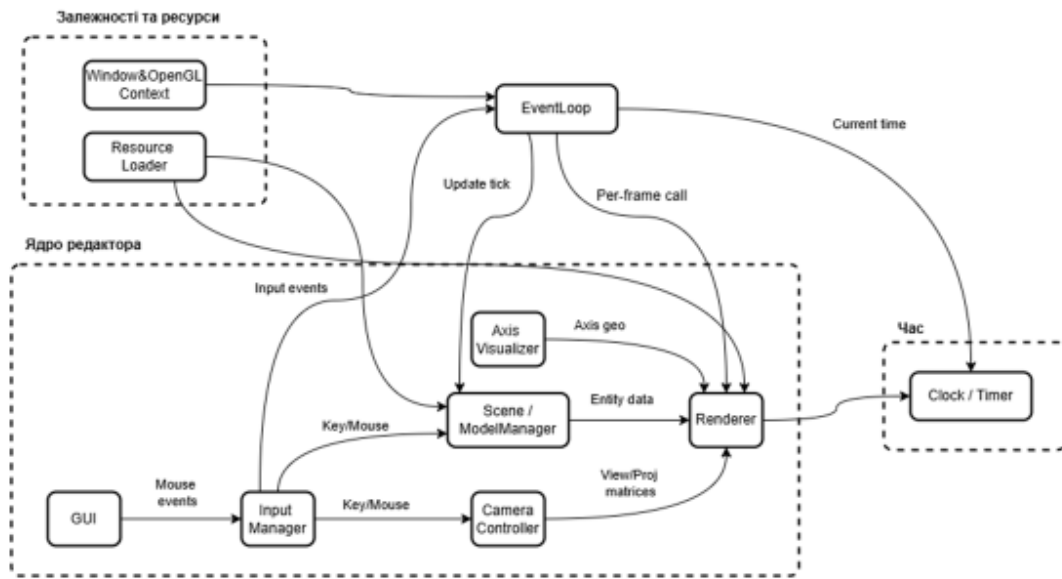
Порівняльна характеристика Blender та Unity

Критерій	Blender 3D	Unity 3D
Тип ліцензії	Відкрите ПЗ (GPL)	Умовно безкоштовне (EULA)
Призначення	3D-моделювання, рендеринг, анімація	Ігровий рушій, візуалізація сцен, XR
Робота з 3D-моделями	Повний цикл: моделювання, текстурування, анімація	Лише імпорт моделей
Стилізований рендеринг	Eevee, Cycles, налаштування у реальному часі	Можливий через плагіни або шейдери
Наявність рендер-рушія	Вбудовані Eevee, Cycles	Так, URP/HDRP
Простота інтерфейсу	Високий поріг входження	Гнучкий, простіший
Можливість розширення	Додатки, Python-плагіни	Плагіни, Unity Asset Store
Операційні системи	Windows, macOS, Linux	Windows, macOS
мінімальні системні вимоги	8 ГБ RAM, GPU 4 ГБ VRAM	8 ГБ RAM, SSE2 CPU
Розмір встановлення	Близько 2 ГБ	6–10 ГБ + проєкти
Основні недоліки	Складний інтерфейс, велика кількість комбінацій клавіш	Вимагає додаткових плагінів для гнучкого, займає багато місця на диску

4

Рисунок А.4 – Порівняльна характеристика Blender та Unity

Структурна схема програми 3D-редактора



5

Рисунок А.5 – Структурна схема програми 3D-редактора



Блок-схема загального алгоритму роботи 3D-редактора

6

Рисунок А.6 – Блок-схема загального алгоритму роботи 3D-редактора

UML-діаграма основних класів і зв'язків 3D-редактора

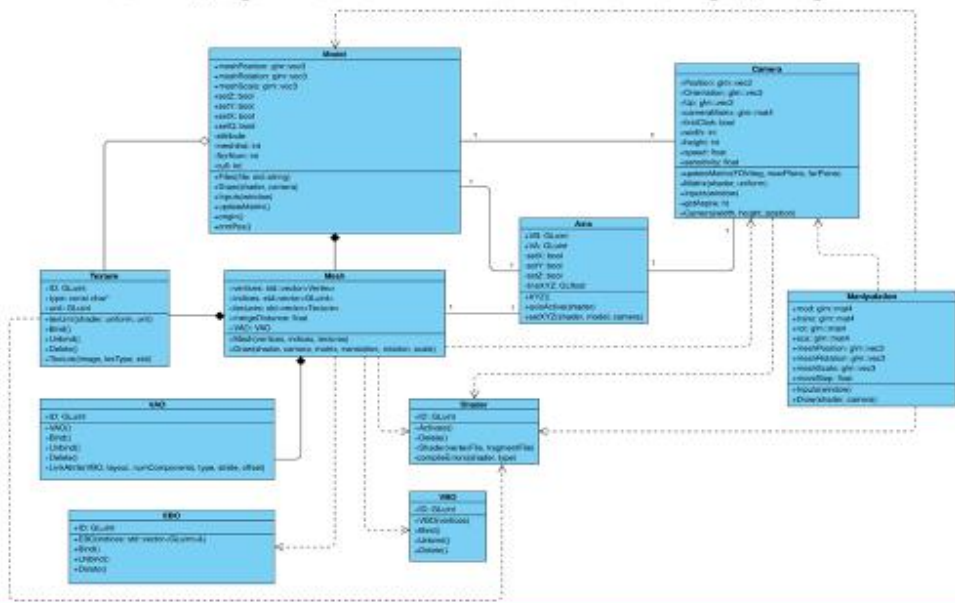


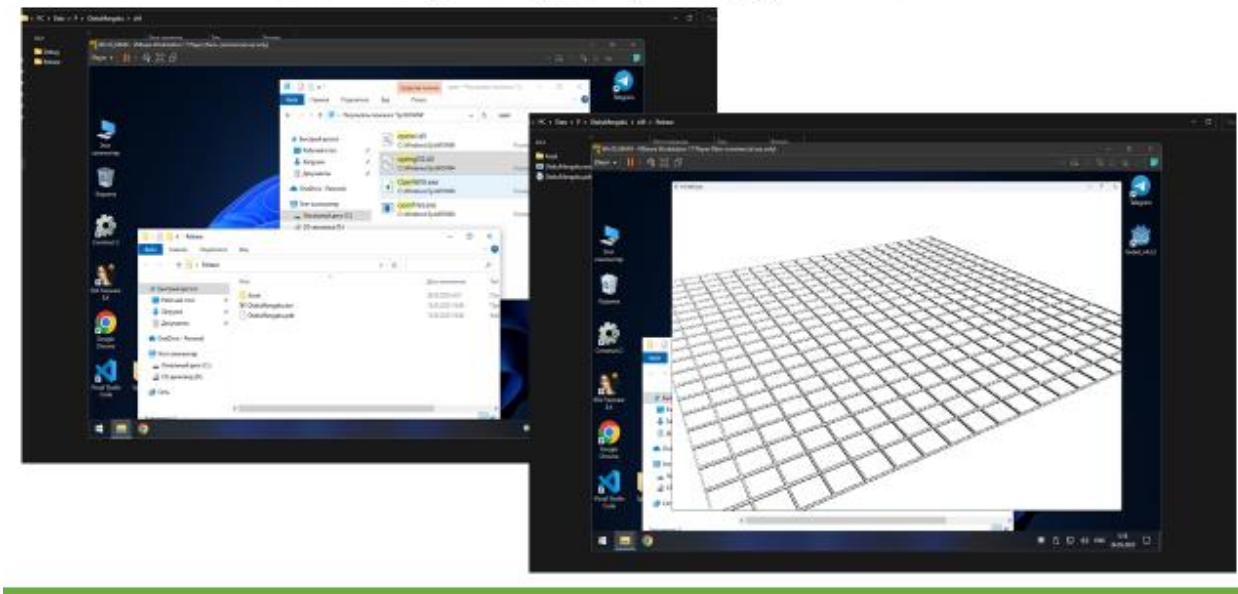
Рисунок А.7 – UML-діаграма основних класів і зв'язків 3D-редактора

Основні методи рендеру класу Model

Метод	Параметри	Опис функції
getIndices()	json accessor	Повертає індекси вершин (GLuint) з буфера
getTextures()	-	Завантажує текстури за шляхами у glTF, уникаючи дублювання
screenTones()	-	Повертає поточний screenTone як текстуру
assembleVertices()	std::vector<vec3>, std::vector<vec3>, std::vector<vec2>	
getData()	-	Завантажує байти з .bin-файлу, прив'язаного до glTF
groupFloatsVec2()	std::vector<float> floatVec	Перетворює float-масив у std::vector<vec2>
groupFloatsVec3()	std::vector<float> floatVec	Перетворює float-масив у std::vector<vec3>
groupFloatsVec4()	std::vector<float> floatVec	Перетворює float-масив у std::vector<vec4>

Рисунок А.8 – Основні методи рендеру класу Model

Тестовий запуск 3D-редактора на віртуальній машині



9

Рисунок А.9 – Тестовий запуск 3D-редактора на віртуальній машині

Використання ресурсів системи під час роботи 3D-редактора

Процеси						Запустити нове завдання	
Ім'я		Стан	22% ЦП	15% Пам'ять	0% Диск	0% Мережа	
Razer AppEngine			1,4%	57,7 МБ	0 Мб/с	0 Мб/с	
Диспетчер вікон			0,7%	71,1 МБ	0 Мб/с	0 Мб/с	
OVR Redir			0,5%	1,7 МБ	0 Мб/с	0 Мб/с	
Google Chrome (40)			0,3%	4 607,1 МБ	0,1 Мб/с	0,1 Мб/с	
Диспетчер завдань			0,3%	79,9 МБ	0 Мб/с	0 Мб/с	
RZSurroundService			0,3%	3,2 МБ	0 Мб/с	0 Мб/с	
OtakuMangaku			0,2%	186,9 МБ	0 Мб/с	0 Мб/с	
Docker Desktop (4)			0%	186,8 МБ	0 Мб/с	0 Мб/с	
Microsoft Visual Studio 2022 (25)			0%	1 155,1 МБ	0 Мб/с	0 Мб/с	

10

Рисунок А.10 – Використання ресурсів системи під час роботи 3D-редактора



11

Рисунок А.11 – Приклад сформованого фону у 3D-редакторі

Висновки

Проведено дослідження предметної області 3D-візуалізації, окреслено базові поняття тривимірних редакторів та принципи створення манга-стилізованих сцен. Визначено основну суть розробки застосунку для генерації манга-фонів із GLTF-моделей.

Спроектовано архітектуру програмного ядра на основі незалежних модулів рендерера, системи ресурсів, менеджера моделей та обробки подій.

Реалізовано метод стилізованого рендерингу у техніці скрінтона з підтримкою імпорту, трансформації та візуалізації тривимірних моделей. Розроблений програмний комплекс має практичну спрямованість, демонструє стабільну роботу та готовий до використання широким колом користувачів.

12

Рисунок А.12 – Висновки

ДОДАТОК Б
Текст програми

Б.1 Файл main.cpp

```

#include "Model.h"
#include "Axis.h"

// Var for the window size
int width = 1600;

// Var for the window size
int height = 900;

int modelInd = 0;

int swap = 0;

bool keyAct = false;
bool mouseAct = false;

// Auto Scaled Window size

void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
    // do any other resizing-related tasks here
}

void keyCallback(GLFWwindow* window, int key, int scancode, int action, int
mods)
{
    if (action == GLFW_PRESS)
    {
        keyAct = true;
        swap = 0;
    }
    else if (action == GLFW_RELEASE)
    {
        keyAct = false;
    }
}

void mouseButtonCallback(GLFWwindow* window, int button, int action, int
mods)

```



```

// Error Check (If something wrong with window)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
if (window == NULL)
{
    // msg if something wrong
    std::cout << "Failed to creat GLFW window" << std::endl;
    // Terminate before functions end
    glfwTerminate();
    // function return -1
    return -1;
}

// this call to GLFW to make the window part of curent context(context is a
object who have all of OpenGL) //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
glfwMakeContextCurrent(window);
// Glad loading configuration of OpenGL
gladLoadGL();

// Create Shader object using default.vert and defaoult.frag
Shader shaderProgram("default.vert", "default.frag");

// light intens
float lightIntens = 0.75f;
// ambient intens
float ambient = 0.0f;

// shader program Activate
shaderProgram.Activate();
glUniform1f(glGetUniformLocation(shaderProgram.ID, "ambient"),
ambient);
glUniform1f(glGetUniformLocation(shaderProgram.ID, "lightIntens"),
lightIntens);
glUniform3f(glGetUniformLocation(shaderProgram.ID, "lightPos"),
lightPos.x, lightPos.y, lightPos.z);

Shader lineShader("axis.vert", "axis.frag");

lineShader.Activate();

Shader outlineShader("outline.vert", "outline.frag");

```

```

//Create Camera Object
Camera camera(width, height, glm::vec3(-11.0f, 8.0f, 20.0f));

std::string grid = "Asset/Grid/Grid.gltf";

Model model;

model.Files(grid);

Axis xyz;

glEnable(GL_DEPTH_TEST);

//Model model("Asset/ClassRoom/scene.gltf");
//                               Main                               While                               loop
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
while (!glfwWindowShouldClose(window))
{
    // Get current Window Size
    glfwGetWindowSize(window, &width, &height);

    // Color of Background

    glfwSetKeyCallback(window, keyCallback);
    glfwSetMouseButtonCallback(window, mouseButtonCallback);

    if (keyAct == true || mouseAct == true || model.setX == true ||
model.setY == true || model.setZ == true || model.setQ == true || swap != 2)
    {
        glDisable(GL_CULL_FACE);

        glEnable(GL_CULL_FACE);
        glCullFace(GL_FRONT);
        glFrontFace(GL_CCW);

        // Clear the Back Buffer and Depth Buffer
        glClear(GL_COLOR_BUFFER_BIT
GL_DEPTH_BUFFER_BIT);

        // AXIS //-----

```

```

)
    if (model.setX == true || model.setY == true || model.setZ == true
    {
        xyz.setXYZ(lineShader, model, camera);

    }

//-----

// Handles Camera Inputs
camera.Inputs(window);

// Handles Model Inputs
model.Inputs(window, lightPos);

//TEST light ROTATE
shaderProgram.Activate();
glUniform3f(glGetUniformLocation(shaderProgram.ID,
"lightPos"), lightPos.x, lightPos.y, lightPos.z);


// model.axisTrans(shaderProgram);

// Getting Aspect for Mouse Position
camera.getAsp(width, height);
model.getAsp(width, height);


// Update and exports the Camera Matrix to the Vertex Shader
camera.updateMatrix(30.0f, 0.1f, 1000.0f);


// Draw Model
model.Draw(shaderProgram, camera);

outlineShader.Activate();
glUniform1f(glGetUniformLocation(outlineShader.ID,
"outlining"), 1.08f);

glDisable(GL_CULL_FACE);

```

[illegible]

Б.2 Файл Axis.cpp

```
#include "Axis.h"
```

```
void Axis::XYZ()
{
```

```
    // Generate the VAO and VBO with only 1 object each
```

```
    glGenVertexArrays(1, &VA);
```

```
    glGenBuffers(1, &VB);
```

```
    // Make the VAO the current Vertex Array Object by binding it
```

```
    glBindVertexArray(VA);
```

```
    // Bind the VBO specifying it's a GL_ARRAY_BUFFER
```

```
    glBindBuffer(GL_ARRAY_BUFFER, VB);
```

```
    // Introduce the vertices into the VBO
```

```
    glBufferData(GL_ARRAY_BUFFER,      sizeof(lineXYZ),      lineXYZ,
GL_STATIC_DRAW);
```

```
    // Configure the Vertex Attribute so that OpenGL knows how to read the VBO
```

```
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
```

```
    // Enable the Vertex Attribute so that OpenGL knows to use it
```

```
    glEnableVertexAttribArray(0);
```

```
    // Bind both the VBO and VAO to 0 so that we don't accidentally modify the
VAO and VBO we created
```

```
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
    glBindVertexArray(0);
```

```
}
```

```
void Axis::axisActive(Shader& shader)
```

```
{
```

```
    shader.Activate();
```

```
    glLineWidth(1);
```

```
    glBindVertexArray(VA);
```

```

        glDrawArrays(GL_LINES, 0, 2);

    }

void Axis::setXYZ(Shader& shader, Model& model, Camera camera)
{

    glm::vec3 axisPos = model.meshPosition;

    setX = model.setX;
    setY = model.setY;
    setZ = model.setZ;

    lineXYZ[0] = axisPos.x;
    lineXYZ[1] = axisPos.y;
    lineXYZ[2] = axisPos.z;
    lineXYZ[3] = axisPos.x;
    lineXYZ[4] = axisPos.y;
    lineXYZ[5] = axisPos.z;

    if (setX == true)
    {
        lineXYZ[2] = lineXYZ[2] + 100000.0f;
        lineXYZ[5] = lineXYZ[5] - 100000.0f;
        Axis::XYZ();
        Axis::axisActive(shader);
        glUniformMatrix4fv(glGetUniformLocation(shader.ID, "camMatrix"),
1, GL_FALSE, glm::value_ptr(camera.cameraMatrix));
        glUniform1f(glGetUniformLocation(shader.ID, "X"), 1.0);
        glUniform1f(glGetUniformLocation(shader.ID, "Y"), 0.0);
        glUniform1f(glGetUniformLocation(shader.ID, "Z"), 0.0);

    }
    else if (setY == true)
    {
        lineXYZ[0] = lineXYZ[0] - 100000.0f;
        lineXYZ[3] = lineXYZ[3] + 100000.0f;
        std::cout << "set Y" << std::endl;
        Axis::XYZ();
        Axis::axisActive(shader);
    }
}

```

```

        glUniformMatrix4fv(glGetUniformLocation(shader.ID, "camMatrix"),
1, GL_FALSE, glm::value_ptr(camera.cameraMatrix));
        glUniform1f(glGetUniformLocation(shader.ID, "X"), 0.0);
        glUniform1f(glGetUniformLocation(shader.ID, "Y"), 1.0);
        glUniform1f(glGetUniformLocation(shader.ID, "Z"), 0.0);
    }

    else if (setZ == true)
    {
        lineXYZ[1] = lineXYZ[1] - 100000.0f;
        lineXYZ[4] = lineXYZ[4] + 100000.0f;
        std::cout << "setZ" << std::endl;
        Axis::XYZ();
        Axis::axisActive(shader);
        glUniformMatrix4fv(glGetUniformLocation(shader.ID, "camMatrix"),
1, GL_FALSE, glm::value_ptr(camera.cameraMatrix));
        glUniform1f(glGetUniformLocation(shader.ID, "X"), 0.0);
        glUniform1f(glGetUniformLocation(shader.ID, "Y"), 0.0);
        glUniform1f(glGetUniformLocation(shader.ID, "Z"), 1.0);
    }
}

```

Б.3 Файл Camera.cpp

```

#include"Camera.h"

Camera::Camera(int width, int height, glm::vec3 position)
{
    Camera::width = width;
    Camera::height = height;
    Position = position;
}

void Camera::updateMatrix(float FOVdeg, float nearPlane, float farPlane)
{
    // Initializes matrices since otherwise they will be the NULL Matrix
    glm::mat4 view = glm::mat4(1.0f);
    glm::mat4 projection = glm::mat4(1.0f);

    // Makes Camera look in the right direction from the right position
    view = glm::lookAt(Position, Position + Orientation, Up);

    // Adds perspective to the Scene

```

```

        projection = glm::perspective(glm::radians(FOVdeg), (float)width / height,
nearPlane, farPlane);

```

```

        // Set new camera Matrix
        cameraMatrix = projection * view;
    }

```

```

void Camera::Matrix(Shader& shader, const char* uniform)
{
    // Exports the Camera Matrix to the Vertex Shader
    glUniformMatrix4fv(glGetUniformLocation(shader.ID, uniform), 1,
GL_FALSE, glm::value_ptr(cameraMatrix));
}

```

```

void Camera::Inputs(GLFWwindow* window)
{
    // Handles Key Inputs
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    {
        Position += speed * Orientation;
    }

    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    {
        Position += speed * -glm::normalize(glm::cross(Orientation, Up));
    }

    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {
        Position += speed * -Orientation;
    }

    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    {
        Position += speed * glm::normalize(glm::cross(Orientation, Up));
    }

    if (glfwGetKey(window, GLFW_KEY_LEFT_SHIFT) == GLFW_PRESS)
    {
        speed = 0.2f;
    }

    else if (glfwGetKey(window, GLFW_KEY_LEFT_SHIFT) ==
GLFW_RELEASE)

```

```

    {
        speed = 0.05f;
    }

    // Handles Mouse Inputs
    if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_MIDDLE)
    == GLFW_PRESS)
    {
        // Hides Mouse Cursor
        glfwSetInputMode(window, GLFW_CURSOR,
        GLFW_CURSOR_HIDDEN);

        // Prevents Camera from jumping on the first click
        if (firstClick)
        {
            glfwSetCursorPos(window, (width / 2), (height / 2));

            firstClick = false;
        }

        // Stores Coordinates of the Cursor
        double mouseX;
        double mouseY;

        // Fetches the Coordinates of the Cursor
        glfwGetCursorPos(window, &mouseX, &mouseY);

        // Normalize and shifts the Coordinates of the Cursor such that they
        being in the middle of the screen
        // and then "Transforms" them into Degrees
        float rotX = sensitivity * (float)(mouseY - (height / 2)) / height;
        float rotY = sensitivity * (float)(mouseX - (width / 2)) / width;

        // Calculates upcoming vertical change in the Orientation
        glm::vec3 newOrientation = glm::rotate(Orientation, glm::radians(-
        rotX), glm::normalize(glm::cross(Orientation, Up)));

        // Decides whether or not the next Vertical Orientation is legal or not
        if (!(glm::angle(newOrientation, Up) <= glm::radians(5.0f)) or
        glm::angle(newOrientation, -Up) <= glm::radians(5.0f))
        {
            Orientation = newOrientation;
        }
    }

```

```

        // Rotates the orientation left and right
        Orientation = glm::rotate(Orientation, glm::radians(-rotY), Up);

        // Sets Mouse Cursor to the middle of the screen so that it doesn't end
        up roaming around
        glfwSetCursorPos(window, (width / 2), (height / 2));

    }

    if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_MIDDLE)
    == GLFW_RELEASE)
    {
        // Unhides Cursor since Camera is not looking around anymore
        glfwSetInputMode(window, GLFW_CURSOR,
        GLFW_CURSOR_NORMAL);

        // Make sure the next time the Camera looks around it doesn't jump
        firstClick = true;
    }
}

```

Б.4 Файл EBO.cpp

```

#include "EBO.h"

// Constructor that generates a Element Buffer Object and Link it to indices
EBO::EBO(std::vector<GLuint>& indices)
{
    // Generate Buffer for Elements Buffer Object (EBO). First - Count of objects,
    Seccound - pointer to reference
    glGenBuffers(1, &ID);

    // Binding buffer of Vertex Buffer(VBO). That's need to explain what we
    modifire current object.
    // First one is Command of glBindBuffer(This type of array need for vertex
    buffer, and seccound one is Reference.
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ID);

    // Store Vertecies into VBO.
    // We need Type of the DATA in Byts, we use vertices, choice stream type
    and choice command (DRAW, READ, COPY)
    // stream type: Static - Modifier once and use many times. Dynamic - Modifier
    multiple times and use many times
    // DRAW mean vertecise will be modifire and use for draw on the screen

```

```

        // READ mean vertecise will reading
        // COPY mean vertecise will copy
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,      indices.size      )*
sizeof(GLuint), indices.data(), GL_STATIC_DRAW);
    }
    // Binds the EBO
    void EBO::Bind()
    {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ID);
    }
    // Unbinds the EBO
    void EBO::Unbind()
    {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }
    // Deletes the EBO
    void EBO::Delete()
    {
        glDeleteBuffers(1, &ID);
    }

```

Б.5 Файл Mesh.cpp

```

#include "Mesh.h"
#include <map>
Mesh::Mesh(std::vector <Vertex>& vertices, std::vector <GLuint>& indices,
std::vector <Texture>& textures)
{
    Mesh::vertices = vertices;
    Mesh::indices = indices;
    Mesh::textures = textures;

    /*
    //-----
    std::vector<GLuint> newIndex(indices.size());

    // Злиття вершин
    for (size_t i = 0; i < vertices.size(); i++) {
        bool merged = false;
        GLuint newIndex = mergedVertices.size(); // Індекс для нової
вершини

        // Перевірка, чи є схожа вершина
        for (size_t j = 0; j < mergedVertices.size(); j++) {

```

```

        if (glm::distance(vertices[i].position,
mergedVertices[j].position) < mergeDistance) {
            newIndex = j; // Використовуємо вже існуючу вершину
            merged = true;
            break;
        }
    }

    // Якщо вершина нова, додаємо її
    if (!merged) {
        mergedVertices.push_back(vertices[i]);
    }

    // Оновлюємо індекси
    for (size_t k = 0; k < indices.size(); k++) {
        if (indices[k] == i) {
            newIndices[k] = newIndex;
        }
    }
}

// Перевірка на кількість вершин для нормалей
if (mergedVertices.size() < 3) {
    std::cerr << "Not enough vertices for calculating normals!" <<
std::endl;
    return;
}

// Оновлення індексів після обробки
indices = newIndices;

// Обчислення нормалей для кожного трикутника
for (size_t i = 0; i < indices.size(); i += 3) {
    GLuint idx0 = indices[i];
    GLuint idx1 = indices[i + 1];
    GLuint idx2 = indices[i + 2];

    // Перевірка індексів на коректність
    if (idx0 >= mergedVertices.size() || idx1 >= mergedVertices.size() ||
idx2 >= mergedVertices.size()) {
        std::cerr << "Index out of range: " << idx0 << ", " << idx1 << ",
" << idx2 << std::endl;
        continue;
    }
}

```

```

// Витягування трьох вершин
glm::vec3 v0 = mergedVertices[idx0].position;
glm::vec3 v1 = mergedVertices[idx1].position;
glm::vec3 v2 = mergedVertices[idx2].position;

// Обчислення нормалі для трикутника
glm::vec3 edge1 = v1 - v0;
glm::vec3 edge2 = v2 - v0;
glm::vec3 normal = glm::normalize(glm::cross(edge1, edge2));

// Додавання нормалі до кожної з трьох вершин трикутника
mergedVertices[idx0].normal += normal;
mergedVertices[idx1].normal += normal;
mergedVertices[idx2].normal += normal;
}

// Нормалізація нормалей
for (size_t i = 0; i < mergedVertices.size(); i++) {
    mergedVertices[i].normal
glm::normalize(mergedVertices[i].normal);
}

// Оновлюємо вершини та індекси
vertices = mergedVertices;
indices = newIndices;

//-----
*/
VAO.Bind();
// Generates Vertex Buffer Object and links it to vertices
VBO VBO(vertices);
// Generates Element Buffer Object and links it to indices
EBO EBO(indices);
// Links VBO attributes such as coordinates and colors to VAO
VAO.LinkAttrib(VBO, 0, 3, GL_FLOAT, sizeof(Vertex), (void*)0);
VAO.LinkAttrib(VBO, 1, 3, GL_FLOAT, sizeof(Vertex), (void*)(3 *
sizeof(float)));
VAO.LinkAttrib(VBO, 2, 3, GL_FLOAT, sizeof(Vertex), (void*)(6 *
sizeof(float)));
VAO.LinkAttrib(VBO, 3, 2, GL_FLOAT, sizeof(Vertex), (void*)(9 *
sizeof(float)));
// Unbind all to prevent accidentally modifying them
VAO.Unbind();

```

```

        VBO.Unbind();
        EBO.Unbind();
    }

void Mesh::Draw
(
    Shader& shader,
    Camera& camera,
    glm::mat4 matrix,
    glm::vec3 translation,
    glm::quat rotation,
    glm::vec3 scale
)
{
    // Bind shader to be able to access uniforms
    shader.Activate();
    VAO.Bind();

    // Keep track of how many of each type of textures we have
    unsigned int numDiffuse = 0;
    unsigned int numSpecular = 0;
    unsigned int numScrT = 0;

    for (unsigned int i = 0; i < textures.size(); i++)
    {
        std::string num;
        std::string type = textures[i].type;
        if (type == "diffuse")
        {
            num = std::to_string(numDiffuse++);
        }
        else if (type == "specular")
        {
            num = std::to_string(numSpecular++);
        }
        else if (type == "ScrT")
        {
            num = std::to_string(numScrT++);
        }
        textures[i].texUnit(shader, (type + num).c_str(), i);
        textures[i].Bind();
        std::cout << "textureUnit " << i << std::endl;
    }
}

```

```

    }
    // Take care of the camera Matrix
    glUniform3f(glGetUniformLocation(shader.ID, "camPos"),
camera.Position.x, camera.Position.y, camera.Position.z);
    camera.Matrix(shader, "camMatrix");

    // Initialize matrices
    glm::mat4 trans = glm::mat4(1.0f);
    glm::mat4 rot = glm::mat4(1.0f);
    glm::mat4 sca = glm::mat4(1.0f);

    // Transform the matrices to their correct form
    trans = glm::translate(trans, translation);
    rot = glm::mat4_cast(rotation);
    sca = glm::scale(sca, scale);

    // Push the matrices to the vertex shader
    glUniformMatrix4fv(glGetUniformLocation(shader.ID, "translation"), 1,
GL_FALSE, glm::value_ptr(trans));
    glUniformMatrix4fv(glGetUniformLocation(shader.ID, "rotation"), 1,
GL_FALSE, glm::value_ptr(rot));
    glUniformMatrix4fv(glGetUniformLocation(shader.ID, "scale"), 1,
GL_FALSE, glm::value_ptr(sca));
    glUniformMatrix4fv(glGetUniformLocation(shader.ID, "model"), 1,
GL_FALSE, glm::value_ptr(matrix));

    // Draw the actual mesh
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0
);
}

```