Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Розробка бібліотеки 3D-візуалізації
на основі OpenGL

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-3
Микита БАРТНОВСЬКИЙ
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

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

Освітня програма
Комп'ютерна інженерія
(повна назва освітньої програми)

Керівник: ст. викл. Олександр ФОМІЧОВ
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

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

Освітня програма Комп'ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Бартновському Микиті Дмитровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка бібліотеки 3D-візуалізації на основі OpenGL

затверджена наказом по університету від " 26 " травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2025 р.

3. Вхідні дані до роботи 1) Персональний комп'ютер з підтримкою OpenGL 4.0+
2) Java Development Kit (JDK 22 та більше)
3) Програмні бібліотеки
4) Література

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз проблеми та огляд існуючих рішень;
2) вибір технології розробки та інструментальних засобів;
3) розробка алгоритмічного забезпечення;
4) розробка програмних модулів;
5) відлагодження програмних модулів;
6) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація –14 слайдів

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	18.05.25-21.05.25	
2	Вибір технології розробки та інструменталь-	22.05.25-24.05.25	
	них засобів		
3	Розробка алгоритмічного забезпечення	25.05.25-08.06.25	
4	Розробка та відлагодження програмного	09.06.25-14.06.25	
	забезпечення		
5	Оформлення матеріалів кваліфікаційної роботи	15.06.25-18.06.25	
6	Подання кваліфікаційної роботи керівникові та	18.06.25-19.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	20.06.25-21.06.25	
	вання		

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ст. викл. Олександр ФОМІЧОВ
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 66 с., 16 рис., 1 табл., 1 дод., 17 джерел.

КОМП'ЮТЕРНА ГРАФІКА, 3D-МОДЕЛЮВАННЯ, ВІЗУАЛІЗАЦІЯ, ТРАНСФОРМАЦІЯ, КООРДИНАТНА СИСТЕМА, ВЕКТОР, МАТРИЦЯ, КАМЕРА, ПЕРСПЕКТИВНА ПРОЕКЦІЯ, JAVA, ГАРБАЖ-КОЛЕКТОР, КРОСПЛАТФОРМНІСТЬ, БІБЛІОТЕКА, АРХІТЕКТУРА ECS, ШАБЛОНИ ПРОЄКТУВАННЯ, SOLID-ПРИНЦИПИ, МОДУЛЬНІСТЬ, OPENGL, JOGL, JOML, JINPUT, GLSL, ШЕЙДЕР, ТЕКСТУРА, МЕШ, UV-КООРДИНАТИ.

Метою кваліфікаційної роботи є розробка кросплатформної бібліотеки для 3D-візуалізації з використанням OpenGL, що забезпечує базовий функціонал рендерингу, обробки вводу та можливість подальшого розширення.

У ході виконання кваліфікаційної роботи було створено графічну бібліотеку на мові Java, що дозволяє виводити на екран прості тривимірні об'єкти, обробляти взаємодію з користувачем та гнучко змінювати сцену. У реалізації проєкту використовувались бібліотеки JOGL (для взаємодії з OpenGL), JOML (для математичних розрахунків) та JInput (для обробки користувацького вводу). Архітектура бібліотеки побудована на основі компонентного підходу, що забезпечує простоту масштабування та підтримку майбутніх доповнень. У роботі було враховано як функціональні, так і нефункціональні вимоги – продуктивність, кросплатформність (Windows/Linux), модульність і чистота коду. Отриманий результат є надійною основою для подальшого розвитку власного графічного рушія або як навчального інструменту для ознайомлення з принципами 3D-графіки.

ABSTRACT

Bachelor’s thesis: 66 pages, 16 figures, 1 tables, 1 appendices, 17 sources.

COMPUTER GRAPHICS, 3D MODELING, VISUALIZATION, TRANSFORMATION, COORDINATE SYSTEM, VECTOR, MATRIX, CAMERA, PERSPECTIVE PROJECTION, JAVA, GARBAGE COLLECTOR, CROSS-PLATFORM, LIBRARY, ECS ARCHITECTURE, DESIGN TEMPLATES, SOLID PRINCIPLES, MODULARITY, OPENGL, JOGL, JOML, JINPUT, GLSL, SHADER, TEXTURE, MESH, UV COORDINATES.

The major goal of this thesis is to develop a cross-platform library for 3D visualization using OpenGL, which provides basic rendering functionality, input processing and the possibility of further expansion.

In order to complete qualification work, a graphical library in Java was created that allows you to display simple three-dimensional objects, process user interaction, and flexibly change the scene. The project was implemented using the JOGL (for interaction with OpenGL), JOML (for mathematical calculations), and JInput (for processing user input) libraries. The library architecture is based on a component-based approach, which ensures easy scalability and support for future additions. The work took into account both functional and non-functional requirements - performance, cross-platform (Windows/Linux), modularity, and code cleanliness. The result is a reliable basis for further development of your own graphics engine or as a learning tool for familiarizing yourself with the principles of 3D graphics.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – Application Programming Interface (інтерфейс прикладного програмування)

AWT – Abstract Window Toolkit (абстрактний інструментарій вікон)

CPU – Central Processing Unit (центральний процесор)

EBO – Element Buffer Object (буфер елементів)

ECS – Entity-Component System (система «сутність-компонент»)

FPS – Frames Per Second (кадрів на секунду)

GC – Garbage Collector (збирач сміття)

GPU – Graphics Processing Unit (графічний процесор)

GLSL – OpenGL Shading Language (мова шейдерів OpenGL)

GUI – Graphical User Interface (графічний інтерфейс користувача)

IDE – Integrated Development Environment (інтегроване середовище розробки)

IO – Input/Output (ввід/вивід)

JOGL – Java OpenGL (Java-обгортка над OpenGL)

JOML – Java OpenGL Math Library (математична бібліотека для OpenGL на Java)

JRE – Java Runtime Environment (виконувальне середовище Java)

JVM – Java Virtual Machine (віртуальна машина Java)

OOP – Object-Oriented Programming (об'єктно-орієнтоване програмування)

VBO – Vertex Buffer Object (буфер вершин)

ВСТУП

З розвитком графічних прискорювачів світ почав наповнювати візуальний, створений на комп’ютерах контент. Ще 50 років тому всі спецефекти у фільмах створювалися за допомогою різноманітних трюків або каскадерів, і важко було уявити, що вибух чи захід сонця на іншій планеті можуть бути відтворені штучно – та ще й настільки реалістично, що їх майже неможливо відрізнити від справжніх.

Сьогодні вже складно уявити зворотне: щоб фільм не використовував комп’ютерні спецефекти. Завдяки доступності обчислювальних потужностей навіть низькобюджетні проєкти можуть дозволити собі використання 3D-графіки.

Без комп’ютерної графіки сьогодні не обходиться ані презентація продукту, ані навіть інженерне проєктування. У всіх галузях – від машинобудування до створення розважальних застосунків – усі залежать від штучної графіки завдяки її відносно низькій вартості та сильному візуальному впливу на кінцевого споживача. Адже дешевше спроєктувати кузов автомобіля у спеціалізованому програмному забезпеченні, ніж створювати фізичний макет. А купувати продукт у красивому дизайні приємніше, ніж той, в якого дизайн не виносить критики сучасності.

До появи графічних прискорювачів комп’ютерна графіка обчислювалася на центральному процесорі (CPU). Проте CPU погано підходить для завдань рендерингу через специфіку паралельних обчислень, характерних для графіки. Одночасна обробка мільйонів пікселів, складні розрахунки світла чи текстурування об'єктів можуть бути виконані на спеціальному обладнанні яке підтримує велику кількість паралельних обчислень. Для цих потреб і було створено графічні відтворювачі (GPU).

Через велику кількість різних прискорювачів створення програм окремо під кожного з них є малоефективним та непрактичним. Для

розв'язання цієї проблеми були розроблені графічні API, такі як OpenGL, Vulkan, DirectX, Metal – вони забезпечують універсальний інтерфейс між програмами та графічними пристроями. Ці API служать універсальним містком між програмним застосунком та апаратним забезпеченням.

Кожен графічний API має свої достатки та недоліки. Це може проявлятися у підтримці кросплатформності, складності, оптимізацій для різних систем чи доступу до низькорівневого контролю.

За допомогою графічних API не тільки спрощується процес створення графічних застосунків, а також з’являється можливість розширення підтримки застосунку й на інші системи. Так для прикладу можна навести ігрові рушії такі як Unity або Unreal Engine, в них є можливість обрати який саме API використовувати для рендерингу, що дозволяє розробнику більш точне налаштування програми під обрану систему з мінімальними змінами в коді.

Не дивлячись на можливості які такі API надають, з їх використанням виникають складності. Так для створення базової програми з виводу куба на екран потрібні не тільки досить глибокі знання самої мови програмування яка використовується для створення застосунку, а й базове розуміння вищої математики, комп’ютерних наук та знання таких концепцій як конвеєр рендерингу, шейдери, системи координат, матриці трансформацій, керування пам’яттю. Також API в середньому хоч і мають схожу структуру, але мають суттєві відмінності у доступних функціях та різний синтаксис мови шейдерів. Швидкий розвиток графічних прискорювачів вносить свої зміни в API так для підтримання специфічних можливостей в OpenGL було додано механізм розширень. Де програмно можна запитати систему про підтримку специфічного розширення.

Використовувати графічні API напряму дуже складно через їх низький рівень. Це можна порівняти з написанням програм на мові програмування Assembler. Так програми будуть високо оптимізовані, але вартість і складність створення такого продукту затьмарить швидкість її роботи. Тому

зазвичай формують прошарок між графічними API та кодом самої програми – графічні бібліотеки. Такий підхід знімає відповідальність за взаємодію з API з програміста кінцевого продукту на того, хто розробляє графічну бібліотеку.

Більшість графічних бібліотек не тільки надають можливість взаємодіяти з графічним API у спрощеному вигляді, а й надають додаткові інструменти розробнику, такі як: вікна, функції взаємодії з користувачем, абстракції для легкої взаємодії з бібліотекою та багато чого іншого.

Бібліотеки створюються на великій кількості мов програмування. Авжеж для такого типу проєктів зазвичай використовують мови програмування які мають велику швидкість виконання та надають можливості для взаємодії з компонентами системи напряму. Такі як C чи C++. Але функціонал для візуалізації потрібен для більшості мов програмування. Навіть для такої відносно повільної мови програмування як Python існують інтерфейси для взаємодії з графічними бібліотеками напряму. Такі інтерфейси зазвичай називають обгортками, через те, що вони майже не змінюють вигляд API до якого надають доступ.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Мови програмування

За весь час існування сфери програмування було створено безліч різних мов програмування для великої кількості різних потреб. Якісь більш підходять для створення серверної частини додатків, якісь в основному використовуються для розробки графічних інтерфейсів користувача. До основних характеристик мов програмування можна віднести: продуктивність, роботу з пам’яттю, простоту, підтримувані платформи, використовувані парадигми програмування. Для розробки бібліотеки 3D-візуалізації всі ці характеристики є важливими.

C — мова програмування, яка є однією з найстаріших, але досі активно використовуваних. Вона має високу швидкість виконання, ручний контроль над пам’яттю та надає доступ до системних компонентів. Завдяки близькості до «заліза» (hardware) ця мова не є мультиплатформною в сучасному розумінні: код для кожної платформи чи архітектури доводиться компілювати окремо. C є процедурною мовою, тому написання та структурування великих програм ускладнюється зі зростанням проєкту. Код на C стає складним для розширення, відлагодження чи підтримки.

C++ — мова програмування, похідна від C, яка надає можливості ООП, шаблонів, перевантаження операторів тощо. Завдяки підтримці ООП вона дозволяє використовувати високорівневі абстракції, зберігаючи низькорівневий контроль та ефективність. Її застосовують у системному програмуванні, розробці ігрових рушіїв, графічних застосунків, вбудованих систем і т.д. До переваг C++ належать: контроль над пам’яттю, висока швидкість виконання. Саме тому її використовують у графічних бібліотеках (OpenGL, DirectX, Vulkan), де швидкість критично важлива. До недоліків можна віднести — складний синтаксис через підтримку кількох парадигм

(ООП, процедурна, узагальнене програмування). Як і C, вона підтримує багато платформ, але код також потребує окремої компіляції.

Java — сучасна об’єктноорієнтована мова, яка компілюється в проміжний байт-код, що виконується на віртуальній машині (JVM). Це робить її кросплатформною: байт-код працює на будь-якій системі із встановленим JVM. Java використовується в корпоративній та мобільній розробці, а також для створення інтерфейсів. Хоча вона рідко застосовується для 3D-графіки, її можна використовувати для навчальних цілей. За швидкістю Java поступається C++, але має автоматичний сміттєзбірник, який керує пам’яттю. Переваги: простіший синтаксис, кросплатформність, велика стандартна бібліотека. Недоліки – нижча продуктивність і обмежений доступ до системних ресурсів порівняно з низькорівневими мовами.

C# — мова, розроблена Microsoft для платформи .NET. Вона створювалася як конкурент Java, тому має схожий простий синтаксис, але додає нові можливості. C# включає автоматичне керування ресурсами, що спрощує роботу. Найчастіше її пов’язують з ігровим рушієм Unity. Як і Java, C# є кросплатформною та має порівнянну швидкість виконання.

Python — мова високого рівня, відома простотою, що робить її популярною серед початківців. Вона підтримує ООП, процедурну та функціональну парадигми, що забезпечує гнучкість. Python використовують у веброзробці, машинному навчанні, автоматизації, тестуванні, 3D-графіці тощо. Це інтерпретована мова з величезною екосистемою бібліотек, тому вона кросплатформна. Недолік — низька швидкість виконання. Однак Python дозволяє інтегрувати модулі, написані на C++ або інших мовах.

1.2 Сфери застосування комп’ютерної графіки

Комп’ютерна графіка є частиною сфери комп’ютерних наук. Ця галузь досліджує методи маніпуляції візуальним контентом, який може мати 3D та 2D формати. До таких маніпуляцій належать методи створення, збереження

та зміни різних типів зображень і об’єктів. Далі будуть більш детально розглянуті деякі з популярних сфер застосування комп’ютерної графіки:

Комп’ютерне мистецтво: За допомогою комп’ютерної графіки можна створювати образотворче та комерційне мистецтво. До інструментів належать пакети анімацій та програми для малювання. Вони надають можливості для проєктування форм об’єктів, наприклад: створення мультфільмів, картин, логотипів компаній.

Проєктування: У сучасному світі проєктування будівель, автомобілів, літаків здійснюється за допомогою комп’ютерного креслення (CAD). Це дозволяє вносити невеликі правки за короткий час, створювати точніші креслення з покращеними технічними характеристиками.

Розваги: Найпоширенішими сферами застосування є ігрова та кіноіндустрія. У кіноіндустрії комп’ютерна графіка використовується для створення мультфільмів, музичних кліпів, телевізійних шоу та візуальних ефектів у кіно. В ігровій індустрії, де ключовим аспектом є інтерактивність, вона забезпечує реалізацію цих функцій.

Науково-освітня сфера: Використання комп’ютерних моделей значно спрощує роботу викладачів. Візуалізація допомагає представити складні концепції (наприклад: математичні графіки, структуру молекул) у зручному та наочному форматі. Це підвищує інтерес учнів і покращує їхні навички.

Комерційна візуалізація: Все більш популярним стає онлайн-шопінг. Важливу роль відіграють магазини, де користувачі можуть не лише переглянути фото товару, а й інтерактивно досліджувати тривимірні моделі. Це покращує якість вибору та досвід покупок.

Обробка моделей та зображень: Існує величезна кількість різних форматів даних, які потребують редагування для подальшого використання. Перетворення наявних зображень або моделей на більш досконалі версії – одне з ключових застосувань комп’ютерної графіки.

Інтерфейс користувача: проєктування меню, іконок, графічних об’єктів та піктограм допомагає створити зручне користувацьке середовище. Для

розробки вебсайтів використовують інструменти, які візуалізують макет у тривимірному просторі, що допомагає уникнути помилок на етапі дизайну.

1.3 Основні поняття та принципи комп’ютерної графіки

Конвеєр рендерингу — це послідовність етапів, які виконуються для перетворення об’єктів у готове зображення. Вхідні дані проходять через всі ці етапи, поки не буде сформовано фінальний результат на екрані. Кількість етапів конвеєра може відрізнятися залежно від технології. Наприклад, OpenGL використовує приблизно 9 етапів, тоді як Vulkan – близько 14. Така різниця пояснюється гнучкістю та низькорівневий підхід Vulkan. Послідовність конвеєра рендерингу можна побачити на рисунку 1.1. Візуально етапи конвеєра рендерингу OpenGL відображені на рисунку 1.2.

Першим етапом є специфікація вершин. Це той етап, при якому формується масив вершин, який надалі буде відправлений до конвеєра. Ці вершини формують примітиви. Ця частина конвеєра взаємодіє з такими об’єктами, як VAO (об'єкти вершинних масивів) та VBO (вершинні буферні об'єкти). VAO відповідають за те, який набір даних кожен VBO зберігає, у той час, як VBO зберігає самі вершинні дані у вигляді масиву. Своєю чергою, ці вершини є набором атрибутів. Кожен атрибут відповідає за певну частину даних вершини – хай то положення чи колір.

Наступним етапом є виконання вершинного шейдера (Vertex Shader). Шейдер – це програма, яка використовує спеціальну мову шейдерів, у цьому випадку це GLSL. На відміну від звичайних програм, ця виконується на стороні графічного прискорювача. На цьому етапі шейдер виконує базові операції окремо над кожною вершиною. Він отримує на вхід атрибути з попереднього етапу і перетворює вхідні дані в одиничний потік вершин. Цей етап програміст може модифікувати. Тут можуть бути створені додаткові, визначені користувачем функції та виводи на наступний етап, але є спеціальний вивід, який вказує на результуюче положення вершини –

gl_Position, який не може бути перевизначеним.

Теселяція — це доступний для програміста, але не обов’язковий етап, на якому виконується шейдер теселяції. Вона використовується для розбиття геометрії на частини. Це надає можливість створювати деталізовані поверхні без потреби їх моделювання. Це зручно для моделювання рельєфу, приміщень, персонажів, збільшуючи кількість полігонів.

Геометричний шейдер — також доступний для програміста, але не є обов’язковим до зміни етапом, на якому виконується геометричний шейдер. Він надає можливість маніпулювати геометрією під час виконання графічного конвеєра. Також геометричний шейдер може перетворювати подані в нього примітиви в інші типи – так, точкові примітиви можуть стати трикутниками й навпаки. Хоч цей етап має багато можливостей, зазвичай його використовують для додавання ефектів. Генерування мешів на стороні графічного прискорювача може бути занадто ресурсомістким процесом.

Вершинна постобробка — це етап, до якого розробник не має прямого доступу. Тут виконуються кроки збору вершин у примітиви та відкидання тих, що знаходяться за полем зору (Clipping). Також на цьому етапі виконується Face Culling – відсікання примітивів, які направлені від користувача.

Растеризація — ще один етап, до якого розробник не має прямого доступу. Тут примітиви, які дійшли до цього етапу, перетворюються на послідовність фрагментів. У свою чергу, фрагмент – це набір станів, які використовуються для розрахунку даних для кожного пікселя дисплея. Ці дані отримуються шляхом інтерполяції між даними вершин.

Фрагментний шейдер — важливий етап, до якого розробник має доступ. Тут формуються дані буфера кольору. Як результат роботи цього шейдера – список кольорів для кожного з колірних буферів, а також значення глибини та трафарету. Хоч цей етап і не є обов’язковим, при ігноруванні значень глибини та трафарету фрагменти матимуть значення за замовчуванням.

Операції для кожного зразка — це недоступний для програміста етап, на якому вирішується, як кожна вибірка в межах пікселя фіналізується перед записом у буфер кадру. Тут виконуються такі важливі процеси, як тест на приналежність пікселя, тест ножиць, тест трафарету, тест глибини, змішування, згладжування, логічні операції та маска запису.

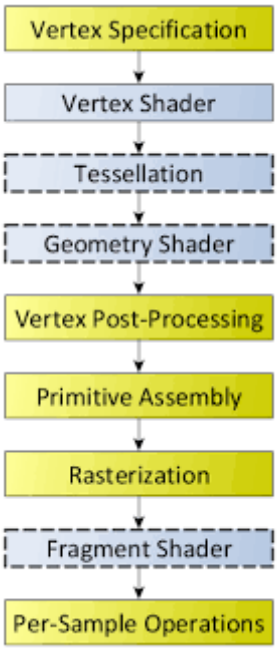

Рисунок 1.1 – Етапи конвеєру рендерингу OpenGL

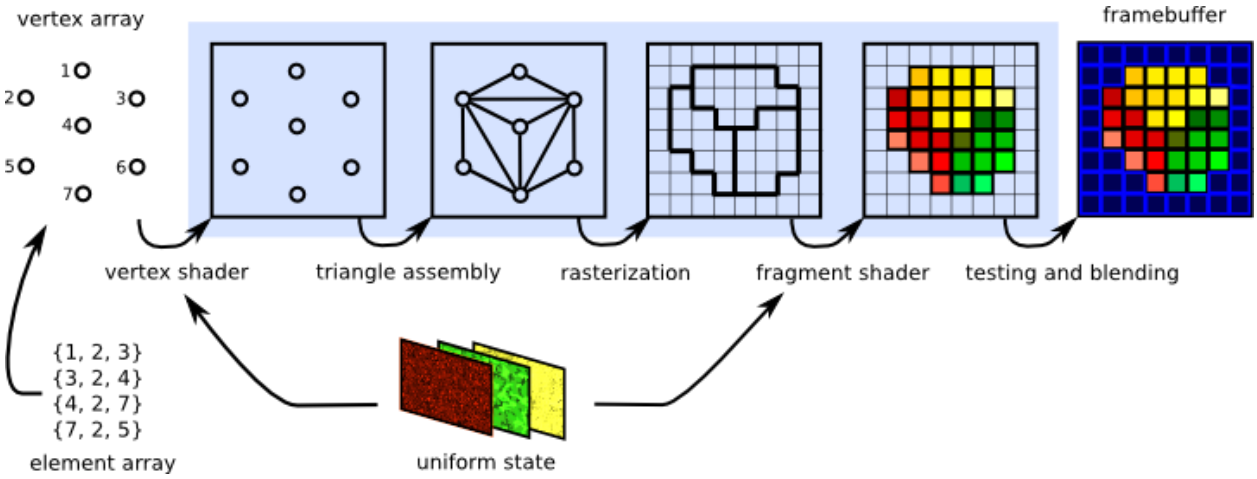

Рисунок 1.2 – Візуалізація етапів конвеєру рендерингу OpenGL

1.4 Геометричні примітиви

OpenGL підтримує декілька базових примітивних типів, таких як точки, лінії, чотирикутники та полігони. Кожен із цих примітивів задається за допомогою послідовності вершин. Примітиви та порядок вказування вершин наведено на рисунку 1.3.

Важливо, що для примітиву GL_LINES для відображення лінії потрібні пари точок, тоді як для примітиву GL_TRIANGLES кожна третя точка з масиву вершин створює новий трикутник. У випадку з примітивами GL_TRIANGLE_STRIP та GL_TRIANGLE_FAN новий полігон створюється для кожної нової вершини. Усі попередні примітиви мають заповнення, тоді як GL_LINE_LOOP відображає лише лінії, що з'єднують вершини.

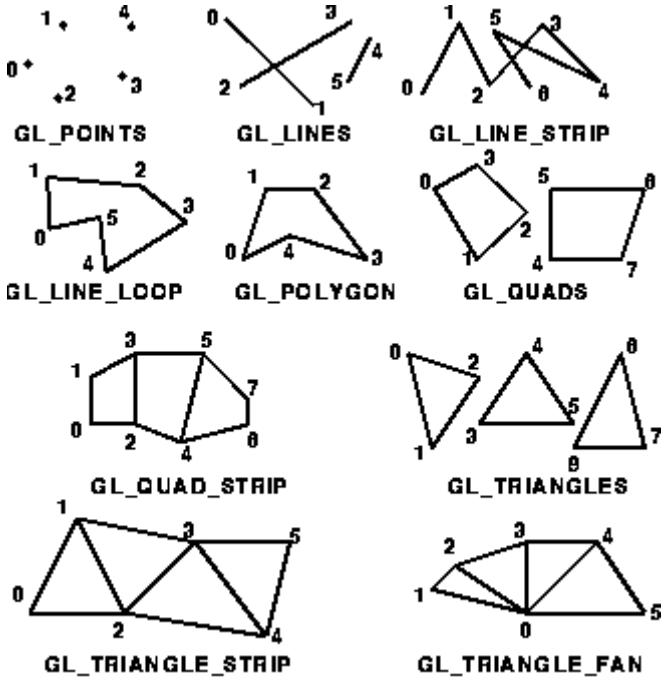

Рисунок 1.3 – Візуалізація етапів конвеєру рендерингу OpenGL

1.5 Трансформації та системи координат

Тривимірний простір зазвичай представляється за допомогою трьох осей: X, Y та Z. Ці три осі можуть бути організовані у двох конфігураціях – правобічну та лівобічну, які продемонстровано на рисунку.

Для правильного розміщення об’єктів на сцені важливо знати, яку координатну систему використовує графічне програмне середовище. Наприклад, координатні системи в OpenGL – правобічні, а в DirectX – лівобічні, рисунок 1.4.

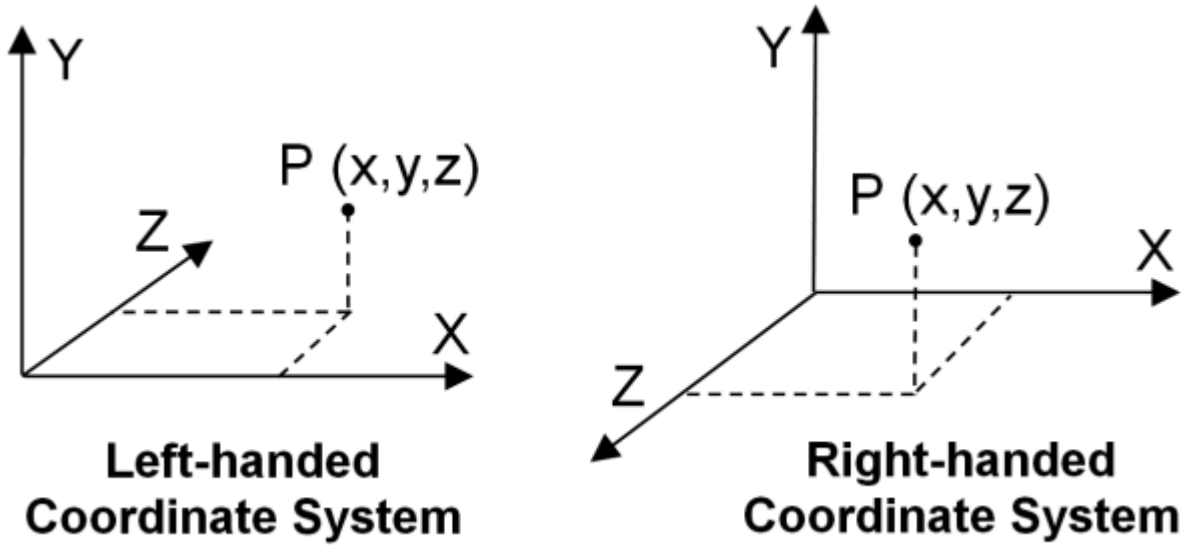

Рисунок 1.4 – Візуалізація етапів конвеєру рендерингу OpenGL

Щоб структура сцени не була змінена чи викривлена, потрібно розуміти таке поняття, як система координат. Наприклад, можна вказати сферу в позиції (0, 0, 0) відносно глобальних координат. Але якщо ми захочемо використати цю сферу, наприклад, для побудови повітряної кулі, до неї доведеться додати нові компоненти – наприклад, кошик. І тут виникає проблема: якщо ми також додамо кошик відносно глобальних координат, то вся структура розпадеться при спробі перемістити чи повернути її. Щоб

уникнути цього, доведеться або використовувати складні розрахунки, які враховують переміщення, або застосувати локальну систему координат кулі – тобто розрахунок положення корзини в просторі відбуватиметься відносно кулі. Таким чином можна сказати, що вони пов’язані між собою.

Матриця — це прямокутний масив значень. Доступ до цих значень здійснюється за допомогою індексів: спочатку вказується рядок, а потім – стовпець. Більшість матриць, які використовуються в графічних обчисленнях, мають розмірність 4×4. Найпоширенішою дією з матрицями в графіці є трансформація. Наприклад, матрицю можна використовувати для переміщення точки з одного місця в інше або для її повороту навколо певного центра. Демонстрації того, як у двовимірному просторі працюють матриці перетворення, можна побачити на рисунку 1.8.

Матриця трансформації використовується для переміщення точок з одного місця в інше на деякий вектор.

Матриця масштабування застосовується для зміни розміру групи точок. Можна змінювати утворений цією групою об’єкт як рівномірно в усіх вимірах, так і сплюснути чи витягнути його лише в обраних напрямках.

Матриця повороту є трохи складнішою за своєю побудовою, оскільки вимагає послідовного повороту об’єкта по кожній з осей окремо. Кути повороту між осями та самим об’єктом називаються кутами Ейлера.

Матриця перспективи перетворює тривимірне зображення об’єктів у двовимірне. Вона реалізує концепцію перспективи, при якій об’єкти, що знаходяться ближче до камери, здаються більшими, а ті, що далі – меншими рисунок 1.5. Щоб зрозуміти, як досягається цей ефект, достатньо уявити експеримент: взяти дві паралельно розміщені площини невеликого розміру (наприклад, квадрати), при цьому камера розміщена перпендикулярно до цих квадратів. Якщо рендерити кожну з площин почергово, буде неможливо визначити, яка з них ближча – вони виглядатимуть однаково. Для створення ефекту глибини й використовується матриця проєкції: вона перетворює паралельні лінії проєкції на такі, що сходяться в одну точку – умовну камеру.

Для побудови цієї матриці потрібно вказати: співвідношення сторін, поле зору, а також ближню та дальню площини проєкції, формула відображена на рисунку 1.6.

Ортографічна матриця зберігає паралельність ліній, тому об’єкти, що знаходяться в об’ємі перегляду, проєктуються прямо на екран, а все інше – відкидається. Така матриця використовується у двовимірній графіці.

Look-At матриця — хоч вона й не є обов’язковою (адже той самий ефект можна отримати за допомогою комбінацій уже згаданих матриць), однак зручність, яку надає такий підхід, переважає. Для побудови такої матриці потрібно вказати: положення камери, точку, куди камера дивиться, вектор, що відповідає напрямку «вгору» відносно камери, формула обчислення надана на рисунку 1.7.

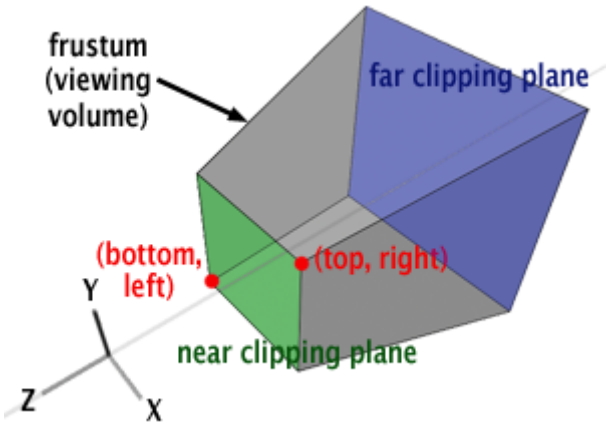

Рисунок 1.5 – Візуаьне представлення роботи матриці проекції

$$\begin{bmatrix} \frac{1}{aspect * \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 * far * near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Рисунок 1.6 – Обчислення матриці перспективи

$$\overrightarrow{fwd} = normalize(target - eye)$$

$$\overrightarrow{side} = normalize(\overrightarrow{fwd} \times \vec{Y})$$

$$\overrightarrow{up} = normalize(\overrightarrow{side} \times \overrightarrow{fwd})$$

The look-at matrix then equals:

$$\begin{bmatrix} side_X & side_Y & side_Z & -(\overrightarrow{side} \bullet \overrightarrow{eye}) \\ up_X & up_Y & up_Z & -(\overrightarrow{up} \bullet \overrightarrow{eye}) \\ -fwd_X & -fwd_Y & -fwd_Z & -(\overrightarrow{fwd} \bullet \overrightarrow{eye}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Рисунок 1.7 – Обчислення look at матриці

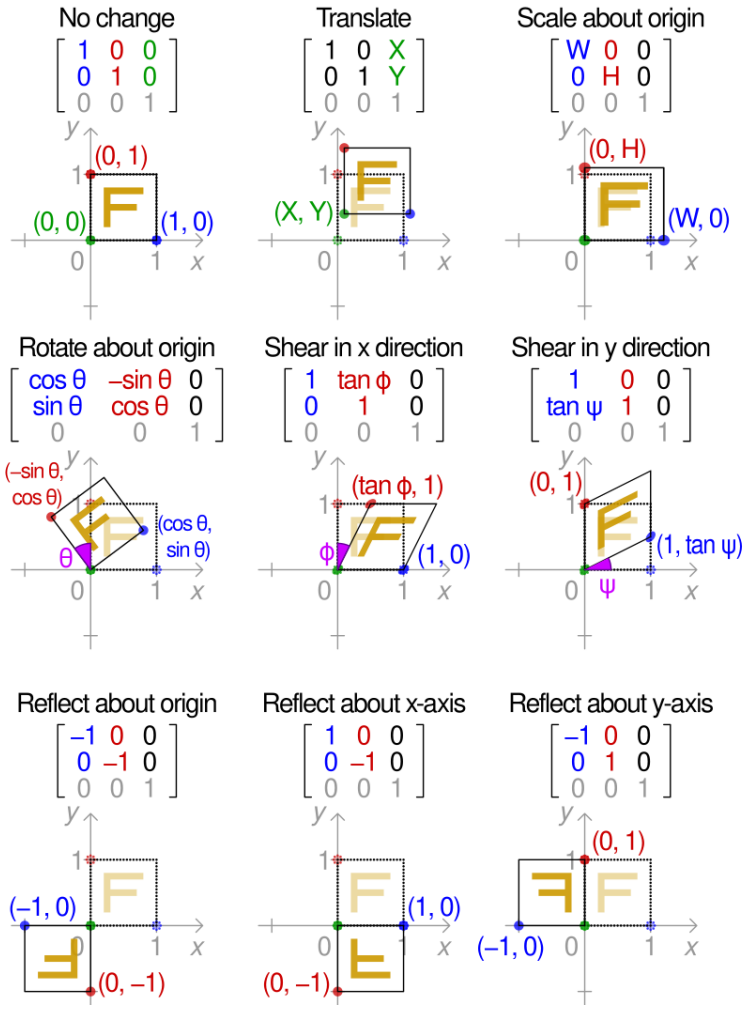

Рисунок 1.8 – Візуалізація матриць перетворень

1.6 Поняття графічної бібліотеки

Графічна бібліотека — це спеціалізований програмний компонент. Метою цього компонента є надання функціоналу та інструментів для маніпулювання, створення та рендерингу візуального контенту. Бібліотеки є абстракціями між кодом програми та більш низькорівневими конструкціями. Зазвичай вони надають спектр функціональних можливостей для малювання і редагування комп'ютерних фігур. Абстрагування дає можливість розробнику зосередитися на аспектах, важливіших для функціонування самого додатку, не заглиблюючись у низькорівневі деталі.

В основному графічні бібліотеки можна розділити на два основні типи:

низькорівневі API та високорівневі бібліотеки.

Низькорівневі API надають прямий доступ до компонентів комп’ютера, що є менш безпечним, але ефективним способом взаємодії з апаратними ресурсами. До таких зазвичай відносять OpenGL, Vulkan, Metal, DirectX.

Високорівневі бібліотеки використовують для своєї роботи низькорівневі API, слугуючи абстракцією над ними через їхню складність. Часто в них реалізовуються додаткові можливості у вигляді вікон, обробки вводу-виводу тощо. Популярними рішеннями є SFML, JavaFX, LibGDX.

1.7 Огляд та порівняння графічних API

OpenGL — у недалекому минулому був основним інструментом для програмування графіки, став популярним завдяки своїм кросплатформним можливостям та відносно простому API. Проте через застарілий дизайн він майже не підтримує багатопотоковість, що зайвий раз навантажує CPU. Попри це, OpenGL досі залишається придатним для освітніх цілей та простих додатків, хоча його актуальність у високопродуктивних сценаріях зменшилася.

Vulkan — є продовжувачем OpenGL, зробивши значний стрибок уперед, надаючи програмісту низькорівневий доступ до операцій і детальний контроль над управлінням пам'яттю та буферами команд GPU. Це забезпечує підтримку багатопотокових програм, але такий детальний контроль за ресурсами супроводжується підвищеною складністю.

DirectX 12 — має схожий із Vulkan низькорівневий підхід, але адаптований лише для платформи Windows. Надає такі функції, як трасування променів для просунутих візуальних ефектів. Хоч він і простіший за Vulkan, все ж залишається набагато складнішим за OpenGL.

Metal — власний API, розроблений Apple виключно для своєї екосистеми. Має баланс між продуктивністю та зручністю використання, надаючи доступ до функцій графічного процесора з низькими накладними

витратами.

Таблиця 1 — Порівняння графічних API

Характеристика	OpenGL	Vulkan	DirectX12	Metal
Розробник	Khronos Group	Khronos Group	Microsoft	Apple Inc
Рік випуску	1992	2016	2015	2014
Рівень абстракції	Високий	Низький	Низький	Низький
Підтримувані платформи	Крос платформений	Крос платформений	Windows, Xbox	IOS, macOS, tvOS
Продуктивність	Середня	Висока	Висока	Висока
Простота використання	Середня	Складна	Складна	Середня
Багато процесорність	Обмежена	Підтримується	Підтримується	Підтримується
Мова шейдерів	GLSL	SPIR-V, GLSL	HLSL	MSL

1.8 Огляд і порівняння графічних бібліотек та додатків

Unity — один з найрозповсюдженіших ігрових рушіїв. Є кросплатформним тому використовується для створення ігор під різні платформи, такі як мобільні пристрої, комп’ютерні системи чи VR. Підтримує як 2D, так і 3D графіку. Також має фізичний модуль, має інструменти для роботи зі звуком та мережевими технологіями. Мовою для написання ігрової логіки є C#.

Unreal Engine — ігрова бібліотека що була розроблена компанією Epic

Games. Має чи не найкращу графіку з понад усіх існуючих ігрових рушіїв. В ній використовується мова програмування C++ для написання ігрової логіки. Також логіку можна задавати за допомогою Blueprint – це така система віртуального сценарію що надає можливість впроваджувати ігрову логіку як досвідченим програмістам, так і тим хто має обмежені знання в програмуванні. Через свою складність зазвичай використовується для розробки AAA-ігор.

Godot — ігровий рушій з відкритим програмним кодом, також має підтримку 2D та 3D графіки. Для написання логіки використовує GDScript – скриптова мова подібна за синтаксисом до Python. Є простішим ігровим рушієм у порівнянні з попередніми двома. Широко використовується як в освітніх цілях, так і для розробки простих невеличких проєктів.

SFML (Проста та швидка мультимедійна бібліотека) — бібліотека написана на мові C++ та портована на велику кількість інших мов програмування. Має зручний програмний інтерфейс. Може виводити просту 2D графіку, оброблювати введення користувача, аудіо та взаємодію з мережею. Може використовуватись у навчальних цілях та для створення простих 2D ігор. Має кросплатформну підтримку.

Графічний рушій Skia — графічний рушій для 2D графіки від Google який. Розроблювався для використання у продуктах від Google таких як Chrome та Android. Його API надає можливість для рендерингу тексту , фігур та зображень з підтримкою апаратного прискорення.

bgfx — кросплатформна бібліотека яка абстрагує відразу декілька API: OpenGL, DirectX та Metal. Забезпечує інтерфейс для рендерингу, через що спрощується розробка графічних додатків. Підтримує кросплатформність.

Magnum — легкий кросплатформний графічний рушій, який використовує такі API як OpenGL та Vulkan. Сам рушій написаний мовою програмування C++. Використовуватись може як для створення ігор, так і візуалізації даних.

1.9 Опис вимог та постановка задачі

Метою даної кваліфікаційної роботи є розробка бібліотеки 3D візуалізації на основі OpenGL. Потреба в інтерактивних застосунках в яких використовується тривимірна графіка стає все більшою. Ігри, наукові дослідження, дизайн тощо. Усі ці галузі мають потребу у високопродуктивних графічних обчисленнях. Такий проєкт має поєднувати в собі як високорівневе, так і низькорівневе програмування що виставляє досить високі вимоги до розробника, а саме: знання мови програмування, шаблонів проєктування, практик написання чистого коду. Задумана бібліотека має мати функціонал для відображення простих 3D-об’єктів на сцені, можливість отримання користувацького вводу, та інтерактивність. Також має бути можливість до подальшого розширення функціональності тобто додавання модуля з фізикою, освітленням та інших графічних ефектів.

Для досягнення поставленої мети, бібліотека має підтримувати функціональні можливості: створення вікна та виводу на них графіку, додавання на сцену 3D-об’єктів з певними параметрами, наприклад: положення, поворот, колір, розмір чи текстура. Отримання користувацького вводу та змінення сцени відповідно до нього. Вивід інформації про стан програми у консоль.

До нефункціональних вимог можна віднести: продуктивність тобто розроблена бібліотека не має витрачати багато ресурсів при створенні об’єктів, Кросплатформність, програма повинна запускатися хоча б на Windows та Linux, просту архітектуру, тобто код має бути модульним та написаним згідно з принципами SOLID.

Мовою написання бібліотеки було вирішено обрати Java через її автоматичне звільнення пам’яті, відносну простоту, кросплатформність, широку базову бібліотеку. Для взаємодії з GPU потрібне низькорівневе API. Одним з найбільш розповсюджених таких API є OpenGL, він ідеально підходить для навчальних цілей, бо має велику кількість навчальних

матеріалів та підтримується майже усіма системами та мовами програмування. Для інтеграції OpenGL та Java існує бібліотека обгортка JOGL, вона надає функціонал ідентичний до нативних функцій OpenGL та інструменти для створення системних вікон. Для того, щоб мати можливість інтерактивно взаємодіяти з додатком потрібно мати спосіб отримання вводу, тут в пригоді знадобиться бібліотека Jinput, так у Java є вбудовані бібліотеки для отримання користувацького вводу, але їх функціоналу може бути не достатньо в майбутньому та можуть виникнути складності в отриманні вводу з таких типів пристроїв як джойстик чи геймпад. Для виконання математичних обчислень підійде бібліотека JOML, вона є простою та портативною.

Відповідно до вимог та обраних інструментів завдання можна сформулювати так: необхідно створити бібліотеку для 3D рендерингу з підтримкою інтерактивності, де функціонал рендерингу буде реалізований за допомогою JOGL, математичні обчислення використовуватимуть бібліотеку JOML, а інтерактивна взаємодія з користувачем буде відбуватися через бібліотеку Jinput.

2 ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

2.1 Мова програмування Java

Java — об’єктноорієнтована мова програмування високого рівня яка широко використовується під час розробки вебдодатків, корпоративних програмних систем та мобільних додатків. Лозунг цієї мови програмування це «Напиши один раз, запускай будь-де». Зараз цим навряд чи можна когось здивувати, але Java була однією з перших мов яка підтримувала кросплатформність, та виникла в той час, коли найпопулярнішими мовами програмування були такі мови як C/C++, програми написані на яких потрібно було окремо компілювати під кожну систему. Натомість Java використовує віртуальну машину Java (JVM). Тобто будь-яка система на якій встановлено JVM зможе запустити цей код. Синтаксис Java перейняла від популярних мов програмування C++ та C, відкинувши з них пряму роботу з пам’яттю, перенісши відповідальність за звільнення пам’яті на автоматичний сміттєзбірник. Тому вона досить непогано підійде для розробки графічної бібліотеки у навчальних цілях.

2.2 ООП та принципи SOLID

SOLID — це абревіатура, утворена з великих літер перших п'яти принципів ООП та дизайну.

1) Принцип єдиної відповідальності — Цей принцип стверджує, що ніколи не повинно бути більше однієї причини для зміни класу. Кожен об'єкт має один обов’язок, повністю інкапсульований у класі. Усі сервіси класу спрямовані на забезпечення цього обов’язку. Такі класи завжди буде легко змінити за потреби, оскільки зрозуміло, за що відповідає клас, а за що ні. Тобто можна вносити зміни, не боячись наслідків впливу на інші об'єкти. Також такий код набагато легше тестувати, оскільки покриваєте тестами

один функціонал, ізольовано від усіх інших.

2) Принцип відкритості/закритості — Цей принцип лаконічно описується так: програмні сутності (класи, модулі, функції тощо) повинні бути відкритими для розширення, але закритими для змін. Це означає, що має бути можливість змінювати зовнішню поведінку класу, не вносячи фізичних змін до самого класу. Дотримуючись цього принципу, класи розробляють так, щоб для адаптації під конкретні умови використання було достатньо розширити їх та перевизначити деякі функції. Завдяки цьому система є гнучкою і може працювати в змінних умовах без модифікації вихідного коду. Продовжуючи приклад із замовленням, припустимо, що нам потрібно виконати певні дії перед обробкою замовлення та після відправлення електронного листа з підтвердженням.

3) Принцип Барбари Лісков — Це певний різновид принципу відкритості/закритості. Коротко він описується як: об’єкти (класи) в програмі мають мати можливість бути заміненими класами які їх наслідують, при цьому старий код програми не має бути змінений чи модифікований, також після заміни програма має залишатися повністю функціональною. Таким чином, новий клас, який розширює (успадковує) інший клас, повинен перевизначити його методи таким чином, щоб не порушити функціональність системи. Підкласи повинні перевизначати методи базового класу, не порушуючи функціональність з погляду клієнта.

4) Принцип розділення інтерфейсів — він стверджує що класи не мають впроваджувати методи, які не будуть використовуватись. Цей принцип стверджує що занадто великі «товсті» інтерфейси мають бути розділені на менші та більш конкретні, щоб класи знали лише про ті методи, які потрібні їм у роботі. Тоді у результаті, при модифікації чи видаленні методу інтерфейсу менша кількість клієнтів буде заторкнута.

5) Принцип інверсії залежностей — цей принцип SOLID у Java описується так: залежності всередині системи будуються на основі абстракцій. Модулі верхнього рівня незалежні від модулів нижчого рівня.

Абстракції не повинні залежати від деталей – навпаки, деталі мають залежати від абстракцій. Програмне забезпечення має бути спроектоване так, щоб різні модулі були автономними та з’єднувалися один з одним через абстракції.

2.3 OpenGL

OpenGL — це специфікація що розробляється та підтримується Khronos Group, яка декларує набір стандартизованих функцій які мають виконуватися однаково на будь-яких системах, які підтримують дану специфікацію. Також часто OpenGL згадують як про низькорівневий графічний API, можна і так сказати, але якщо бути більш точним, то це специфікація кожну з яких кожен виробник реалізує сам. Тому різні графічні системи можуть мати різні реалізації цього стандарту, хоча всі вони мають відповідати стандарту OpenGL для узгодженої роботи та кросплатформності. Драйвери для OpenGL постачаються виробником компонентів, та можуть оновлюватись з часом підвищуючи продуктивність чи покращуючи сумісність. Структурно OpenGL працює як кінцевий автомат тому має проблеми з багатопотоковістю. Взаємодія з інтерфейсом OpenGL в основному відбувається через контекст, який є прив’язаним лише до одного потоку. Це йде з тих часів коли OpenGL використовував старий метод рендерингу, що зараз називається фіксованим конвеєром функцій. Зараз цей метод майже не використовується та не підтримується в сучасних специфікаціях OpenGL через його повільність та високе навантаження на CPU. У більш сучасних версіях починаючи приблизно з OpenGL 3.3 є повністю виключеним зі специфікації. Натомість впровадили більш гнучкий та ефективний метод основного профілю конвеера. Новий підхід надає більше можливостей для оптимізацій та контролю.

Також слід згадати про таку особливість OpenGL як розширення, це така функціональність яка не є частиною специфікації OpenGL, але може бути використана, для цього потрібно опитати систему про підтримку функціональності, при позитивній відповіді на запит можна використовувати додаткову функціональність. Це було зроблено для того, щоб виробники

компонентів мали можливість додавати нові функції не чекаючи їх появи у специфікації та підтримуючи конкурентну здібність компаній.

2.4 Компонентна архітектура

У сучасній розробці ігор та бібліотек для рендерингу фундаментальним підходом є компонентні системи, точніше, архітектура Entity-Component-System. Така архітектура дозволяє з легкістю доповнювати та модифікувати системи. Так будь-який об’єкт такий як гравець чи предмет інтер’єру має бути складений з окремих компонентів, таких як форма, колір, положення і т.д. До того ж компоненти зберігають або як ще кажуть “Інкапсулюють” в собі дані про об’єкт та взаємодію з ним. Це доволі зручно для людей які мають поверхневі знання в програмуванні. Ще такий підхід зручний тим що надає певний рівень абстракції спрощуючи написання коду на базі такої системи.

2.5 Додаткові бібліотеки

JOGL — це бібліотека Java, ця бібліотека надає прямий доступ до OpenGL та майже не змінює взаємодію з його API. Використовується так само як і OpenGL у 2D та 3D графіці, надаючи можливості апаратного прискорення та доступ до низькорівневих функцій OpenGL. Має підтримку всіх версій та вендорів OpenGL, версії 1.0 – 4.5, 1.0 – 3.2 OpenGL ES, 1.0 – 1.5 EGL. Є частиною проєкту JogAmp який має на меті створення високопродуктивних бібліотек Java для 3D-графіки, мультимедіа та обробки даних тому також має такі бібліотеки як JOCL (OpenCL) – використовується для математичних обчислень на GPU, JOAL (OpenAL) – робота зі звуком. JOGL є кросплатформною бібліотекою та підтримує сумісність з операційними системами Windows, Linux, MacOS та іншими. Ця бібліотека має й певні абстракції, які є необхідними для мови програмування Java: GLProfile, GLCapabilities, GLCanvas та GLAutoDrawable. Такі класи та

інтерфейси використовуються для встановлення використовуваних версій OpenGL, керування та налаштування контексту, взаємодією з базовими бібліотеками Java як, наприклад AWT, Swing, JavaFX та надає свій інструментарій для роботи з вікнами який називається NEWT (Native window toolkit). Таким чином надається доступ до всіх можливостей OpenGL не переходячи на мови C або C++. Зазвичай цю бібліотеку використовують в графічних інструментах, іграх, симуляціях, навчальних цілях.

JOML — це легка бібліотека для високопродуктивних математичних обчислень мовою програмування Java. Вона надає математичні конструкції такі як: матриці, вектори, кватерніони, їх перетворення та різні розмірності та величини. В ній використовується чиста Java без використання зовнішніх залежностей тому ця бібліотека є неймовірно портативною та зручною в використанні. Основною ціллю бібліотеки є її використання в високопродуктивних обчисленнях, тому вона є дуже оптимізованою спеціально під мову програмування Java тому майже не створює нових об’єктів під час розрахунків. Має зручні функції такі як: translate, rotate, scale, perspective, lookAt для модифікації даних. JOML використовують у наукових високопродуктивних обчисленнях, іграх, 3D редакторах, симуляціях.

JInput — це кросплатформна бібліотека для отримання вводу від периферійних пристроїв для мови програмування Java. Підтримує введення з миші, клавіатури, геймпаду, джойстику та багато інших пристроїв. В цій бібліотеці всі пристрої абстрагуються до контролера, який зберігає стан пристрою та його буфер введення. кросплатформність дозволяє їй працювати на багатьох операційних системах. Можна сказати що це низькорівнева бібліотека для взаємодії з периферійними пристроями. Оновлення даних відбувається через регулярні невеликі інтервали часу що дозволяє отримувати дані майже без затримки. Підтримує гарячу заміну пристроїв що дозволяє перепідключати пристрої під час виконання програми. Через це така бібліотека підходить для будь-яких систем реального часу, таких як ігри, чи 3D редактори.

3 СТРУКТУРА ТА РОЗРОБКА БІБЛІОТЕКИ

3.1 Віконний модуль

Вікно є одним з найголовніших модулів у бібліотеці. Воно координує цикл рендерингу та відповідає на виведення графічної інформації на дисплей роблячи її доступною користувачеві. Без цього компоненту було б не можливо отримати результати виконання програми у реальному часі та хоч якось взаємодіяти з програмою, бо вікно зберігає фокус введення, при втрачені цього фокусу буде неможливо отримати введення користувача.

UML діаграму реалізації цього модуля можна побачити на рисунку 3.1. Ця діаграма детально відображає архітектуру модуля.

Інтерфейс Window є базовим інтерфейсом, який декларує функціонал що має реалізовуватися для того, щоб вікно успішно функціонувало. Класи що реалізують даний інтерфейс можуть використовувати будь-які інструменти для реалізації цих функцій. На UML діаграмі можна побачити що зараз використовуються три бібліотеки для створення вікон: NEWT, AWT, Swing. Зберігається можливість розширення функціоналу шляхом додавання нових реалізацій вікон. Ключовим для кожного вікна є компоненти: Animator, InputHandlers, GLCanvas, Camera, Scene.

Animator – клас відповідальний за періодичне оновлення вікна. Є необхідним у додатках з рендерингом в реальному часі. Викликає оновлення контенту вікна через певні інтервали часу. Він працює незалежно в окремому потоці що є важливим для уникнення блокування потоку відповідального за рендеринг. Має функціонал для запуску, зупинки та запиту стану потоку.

InputHandlers – клас відповідальний за отримання вводу користувача з різних периферійних пристроїв, таких як миша, клавіатура, контролери різних типів. Може бути реалізований використовуючи базові Java бібліотеки, такі як AWT чи Swing. Або використовувати складніші як на

приклад Jinput для отримання вводу зі специфічних пристроїв.

Але у будь-якому разі InputHandler має мати доступ до контексту вводу для коректної роботи. Після отримання вводу він далі надає дані до інших систем бібліотеки, використовуючи патерн програмування Listener ти самим оповіщуючи інші об’єкти про події.

GLCanvas – компонент що містить в собі OpenGL контекст, саме в потоці в якому було створено цей об’єкт має виконуватись логіка відповідальна за рендеринг та доступ до усіх OpenGL функцій. Дуже важливо щоб потік з контекстом OpenGL не виконував ніяку складну логіку безпосередньо не пов’язану з рендерингом, інакше будуть зайві затримки у виконання програми.

Camera – безпосередньо пов’язана з вікном, сценою та вводом користувача. Без неї не можливо було б отримати тривимірне зображення сцени та переміщення по ній через відсутність такої функціональності в OpenGL. Цей клас використовує матричну математику для побудови проєкції даних сцени на двовимірний дисплей імітуючи трьох вимірність.

Scene – клас який відповідає за посилання об’єктів до потоку з контекстом рендерингу, або ініціалізувати текстури так через те, що користувач не має можливості напряму звертатись до OpenGL контексту.

Так модуль вікна також має певну модульність що додає гнучкість та можливість для модифікацій коду.

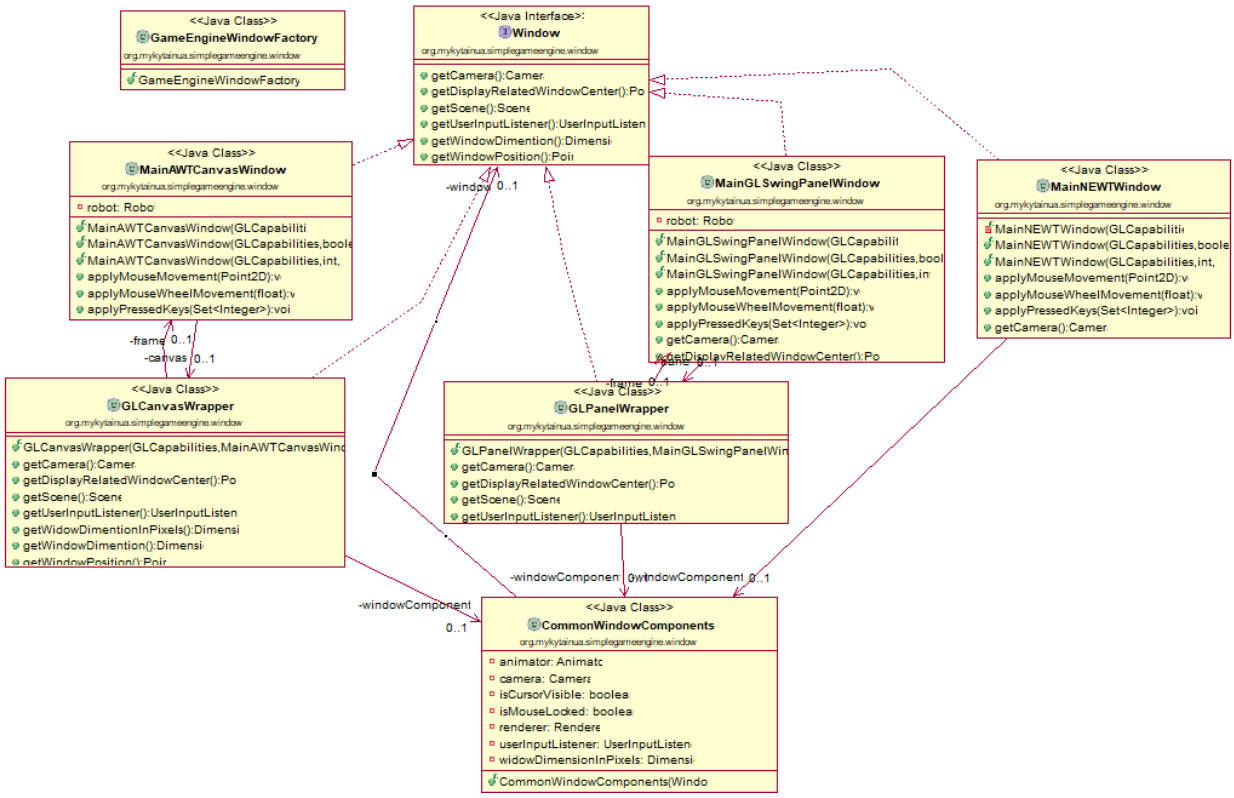

Рисунок 3.1 – UML діаграма модулю вікон

3.2 Отримання вводу користувача

Модуль для отримання вводу користувача відіграє ключову роль в реалізації інтерактивності додатка. Він відповідальний за отримання сирого вводу користувача з периферійних пристроїв, подальшої обробки цих даних та оповіщення всіх об’єктів слухачів про події. У даному контексті до периферійних пристроїв зазвичай відносять: мишу, клавіатуру, геймпад, джойстик та будь-який інший пристрій через який користувач взаємодіє з пристроєм. За відсутності цього модуля було б не можливо реалізувати переміщення по сцені, або будь-яку подальшу взаємодію з об’єктами на сцені.

В цьому модулі реалізується патерн програмування Listener. Це можна побачити на UML діаграмі рисунок 3.2, так об’єкти які виконують зчитування та первинну обробку даних введених користувачем має

реалізовувати інтерфейс UserInputListener. Цей інтерфейс декларує весь функціонал який має бути реалізовано класом, який відповідальний за зчитування вводу користувача. Об'єкти які хочуть отримувати сповіщення про введення користувача мають реалізовувати інтерфейси MouseResponser та або KeyResponser. Згідно з їх назвами отримувати оновлення про стан миші та клавіатури.

Робота з бібліотеками AWT або Swing у плані отримання користувацького вводу подібна до реалізованої в самому модулі, вона основана на системі подій, об’єкти реалізують відповідні інтерфейси, наприклад KeyListener, тоді всі об’єкти які реалізують даний інтерфейс будуть сповіщені через відповідну функцію яка задекларована в потрібному інтерфейсі як, наприклад keyPressed(KeyEvent e), об’єкт KeyEvent зберігає в собі всі дані про подію. Інша справа з Jinput, на відміну від AWT та Swing тут реалізована polling-based система. Для отримання вводу відбувається ітерація через усі периферійні пристрої. Приклад використання бібліотеки Jinput можна побачити на лістингу 3.1. При кожному кадрі йде опитування всіх пристроїв які є мишами та йде перевірка чи пристрій активний. Після отримання активного пристрою в цьому випадку миші йде отримання про стан миші. Далі дані про цей стан інтерпретуються у MouseResponser та відправляються до об’єктів слухачів. Таких як Camera.

Лістинг 3.1 – Функція для отримання вводу миші користувача

```
    public void fetchMouseData() {

        if (this.activeMouse == null) {
            Logging.message(Level.INFO, "Trying to get
mouse...");
            this.activeMouse =
this.tryToGetCurrentActiveMouse();
        }

successfully polled
        if (activeMouse != null && activeMouse.poll()) {
```

```
            Component[] components =
activeMouse.getComponents();

private Controller tryToGetCurrentActiveMouse() {

        Controller activeMouse = null;

        Logging.message(Level.FINEST, "List of found
controllers:");

        for (Controller controller : controllers) {
            try {
                Logging.message(Level.FINEST,
                        " - " + new
String(controller.getName().getBytes("ISO-8859-1"), "Windows-
1251") +
                        " (" +
                        controller.getType() +
                        ")");
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }
        }

        for (Controller controller : controllers) {
            if (controller.getType() == Controller.Type.MOUSE) {

                controller.poll();
                EventQueue queue = controller.getEventQueue();
                Event mouseEvent = new Event();

                while (queue.getNextEvent(mouseEvent)) {
                    activeMouse = controller;
                    break;
                }
                if (activeMouse != null) {
                    break;
                }
            }
        }

        return activeMouse;
    }

            for (Component component : components) {
                if (component.getIdentifier() ==
Component.Identifier.Axis.X) {
                    this.setXShift(component.getPollData());
                    Logging.message(Level.FINEST, "Mouse moved
along X by " + this.getXShift());
                } else if (component.getIdentifier() ==
Component.Identifier.Axis.Y) {
                    this.setYShift(component.getPollData());
```

```
                    Logging.message(Level.FINEST, "Mouse moved
along Y by " + this.getYShift());
                }
            }
        }
    }
```

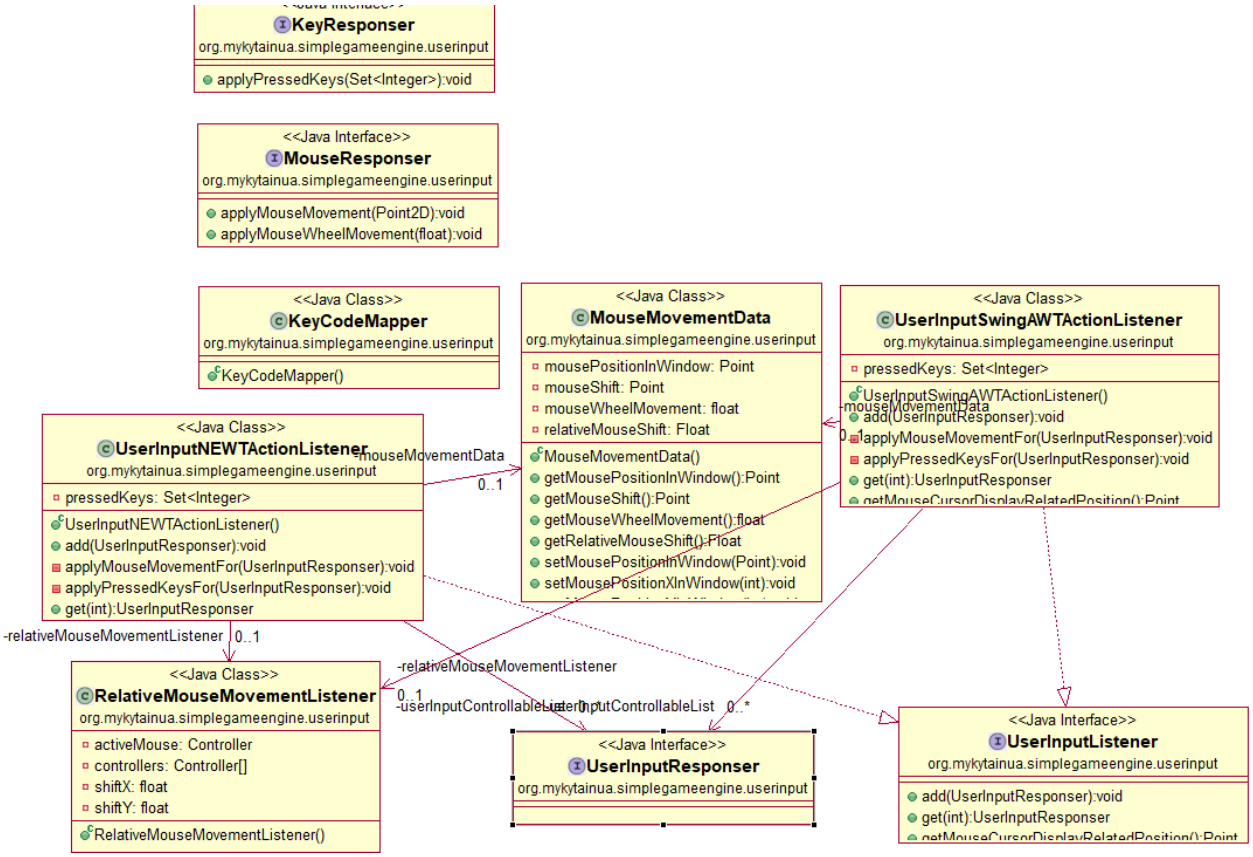

Рисунок 3.2 – UML діаграма модулю зчитування введення користувача

3.3 Об’єкти (Entity)

Є однією з базових абстракцій, яка використовується для всіх 3D об’єктів. Основною ціллю цього класу є зберігання унікальних ідентифікаторів об’єктів для того, щоб їх можна було відрізнити один від одного та керування компонентами які далі будуть розглянуті. Ідентифікатори об’єктам надаються класом GlobalIDProvider, який реалізує патерн Singelton, та гарантує унікальність ідентифікатора у системі.

3.4 Модуль компонентів

Компонентна система є розповсюдженою у багатьох сучасних ігрових рушіях чи програм для візуалізації. У такій архітектурі Entity функціонують як контейнери для зберігання компонентів, які описують стан та поведінку об’єктів. При такому підході система стає легко розширюваною, так для додавання нового функціоналу потрібно лише створити новий компонент та прописати його функціонал. Центральною абстракцією компонентної системи є інтерфейс Component. Всі компоненти мають реалізувати цей інтерфейс, але сам по собі він не несе ніяких корисних даних, для цього було створено інші інтерфейси які є більш спеціалізованими, такі як MeshComponent – використовується для опису форми об’єкта, ShaderComponent – сигналізує що компоненти використовується в шейдерах та надає додатковий функціонал для цього. VisualComponent – позначає що компонент впливає на зовнішній вигляд об’єкта, EntityComponent декларує функції для доступу до даних. Детально структуру компонентів можна побачити на UML діаграмі зображеній на рисунку 3.3.

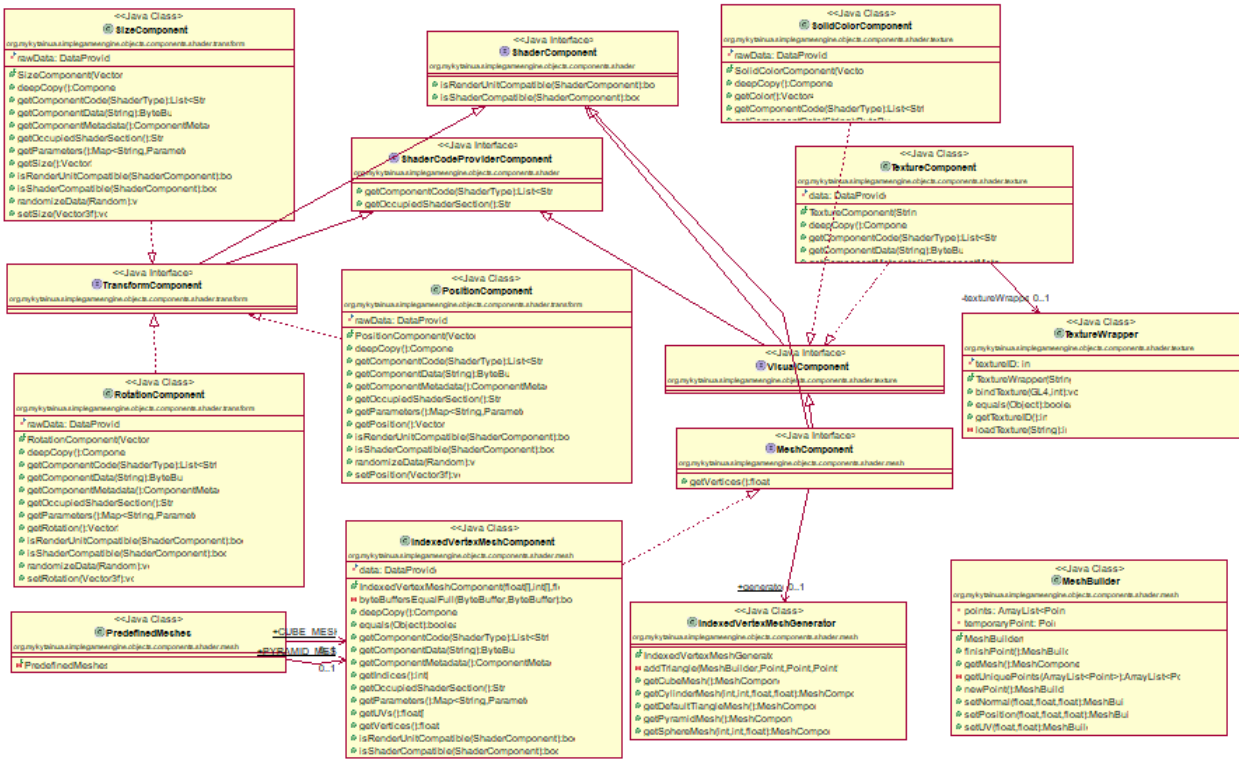

Рисунок 3.3 – UML діаграма структури компонентів

3.5 Система збірки шейдерів

В графічних бібліотеках зазвичай шейдери задаються напряму користувачем. Це надає гнучкість до зміни та оптимізації шейдерів, але натомість зобов’язує розробника слідкувати за правильністю написання шейдерів що є досить складним завданням через неможливість налагоджування коду шейдера. Відповідно до компонентної системи було реалізовано систему автоматичного збору шейдерів. Відповідальним за створення шейдерів для об’єкта є клас ShaderBuilder він відіграє надважливу роль у всьому конвеєрі рендерингу, бо створює код для шейдера вершин та фрагментів відповідно до об'єкта який був у нього переданий та Uniform об’єктів.Така система дозволяє додавати компоненти під час виконання програми та не піклуватись про правильність написання коду для шейдерів.

ShaderBuilder клас виконує збірку шейдерів відповідно до списку ShaderCodeProviderComponent компонентів які знаходяться в переданому об’єкті. Кожен ShaderCodeProviderComponent зберігає в собі код потрібний саме для його рендерингу. Цими компонентами може бути як меш, трансформування об'єкта, текстура чи світло. ShaderBuilder розбирає код кожного з компонентів на його складники, такі як layout, вихідні змінні, декларування функцій та їх реалізації. Додатково компонент може вказати код який буде вмонтований в основну функцію шейдера main() вказавши код що потрібно вмонтувати між спеціальними маркерами, такими як #{MAIN_BODY}# та #{END_MAIN_BODY}# приклад такого коду можна побачити у лістингу 3.2. Одним з головних переваг такого підходу є просте розширення. Така система є модульною тому будь-який компонент має доступ до шейдера лише імплементуючи ShaderCodeProviderComponent інтерфейс та надаючи код для шейдера вершин та фрагментному шейдеру.

Лістинг 3.2 – Приклад шейдеру компоненту rotation

```
layout (location = 0) in vec3 instanceRotation;

vec4 rotateX(vec4 position, float rad);
vec4 rotateY(vec4 position, float rad);
vec4 rotateZ(vec4 position, float rad);

//#{MAIN_BODY}#
     vertex_position = rotateZ(vertex_position,
instanceRotation.z);
     vertex_position = rotateY(vertex_position,
instanceRotation.y);
     vertex_position = rotateX(vertex_position,
instanceRotation.x);
//#{END_MAIN_BODY}#

vec4 rotateX(vec4 position, float rad)
{
     return mat4(1.0, 0.0, 0.0, 0.0,
                 0.0, cos(rad), -sin(rad), 0.0,
                 0.0, sin(rad), cos(rad), 0.0,
                 0.0, 0.0, 0.0, 1.0) * position;
}

vec4 rotateY(vec4 position, float rad)
{
     return mat4(cos(rad), 0.0, sin(rad), 0.0,
                 0.0, 1.0, 0.0, 0.0,
                 -sin(rad), 0.0, cos(rad), 0.0,
                 0.0, 0.0, 0.0, 1.0) * position;
}

vec4 rotateZ(vec4 position, float rad)
{
     return mat4(cos(rad), -sin(rad), 0.0, 0.0,
                 sin(rad), cos(rad), 0.0, 0.0,
                 0.0, 0.0, 1.0, 0.0,
                 0.0, 0.0, 0.0, 1.0) * position;
}
```

Після збірки можна отримати код шейдерів представлених на лістингах 3.3 та 3.4.

Лістинг 3.3 – Сформований шейдер вершин

```
#version 430

//#{UNIFORMS}#
uniform sampler2D samp;
uniform mat4 v_matrix;
uniform mat4 p_matrix;

//#{INPUTS}#
in vec3 instanceRotation;
in vec3 instanceSize;
in vec3 instancePosition;
in vec2 UVcoords;
in vec3 vertices;

//#{OUTPUTS}#
out vec2 texCoord;

//#{FUNCTION_DECLARATIONS}#
vec4 rotateZ(vec4 position, float rad);
vec4 rotateY(vec4 position, float rad);
vec4 rotateX(vec4 position, float rad);
vec4 scale(vec4 position, float x, float y, float z);
vec4 translate(vec4 position, float x, float y, float z);

void main(void)
{
     vec4 vertex_position = vec4(vertices, 1.0);

     //#{SIZE_TRANSFORMATION}#
     vertex_position = scale(vertex_position, instanceSize.x,
instanceSize.y, instanceSize.z);
     //#{ROTATION_TRANSFORMATION}#
     vertex_position = rotateZ(vertex_position,
instanceRotation.z);
     vertex_position = rotateY(vertex_position,
instanceRotation.y);
     vertex_position = rotateX(vertex_position,
instanceRotation.x);
     //#{POSITION_TRANSFORMATION}#
     vertex_position = translate(vertex_position,
instancePosition.x, instancePosition.y, instancePosition.z);
     //#{CAMERA_TRANSFORMATION}#
     vertex_position = v_matrix * vertex_position;
     vertex_position = p_matrix * vertex_position;

     gl_Position = vertex_position;

     //#{TEXTURE}#
     texCoord = UVcoords;
}

//#{FUNCTION_REALIZATIONS}#
```

```
vec4 rotateZ(vec4 position, float rad)
 {
     return mat4(cos(rad), -sin(rad), 0.0, 0.0,
                 sin(rad), cos(rad), 0.0, 0.0,
                 0.0, 0.0, 1.0, 0.0,
                 0.0, 0.0, 0.0, 1.0) * position;
 }


vec4 rotateY(vec4 position, float rad)
 {
     return mat4(cos(rad), 0.0, sin(rad), 0.0,
                 0.0, 1.0, 0.0, 0.0,
                 -sin(rad), 0.0, cos(rad), 0.0,
                 0.0, 0.0, 0.0, 1.0) * position;
 }


vec4 rotateX(vec4 position, float rad)
 {
     return mat4(1.0, 0.0, 0.0, 0.0,
                 0.0, cos(rad), -sin(rad), 0.0,
                 0.0, sin(rad), cos(rad), 0.0,
                 0.0, 0.0, 0.0, 1.0) * position;
 }


vec4 scale(vec4 position, float x, float y, float z)
 {
     return mat4(x, 0.0, 0.0, 0.0,
                 0.0, y, 0.0, 0.0,
                 0.0, 0.0, z, 0.0,
                 0.0, 0.0, 0.0, 1.0) * position;
 }


vec4 translate(vec4 position, float x, float y, float z)
 {
     return mat4(1.0, 0.0, 0.0, 0.0,
                 0.0, 1.0, 0.0, 0.0,
                 0.0, 0.0, 1.0, 0.0,
                 x, y, z, 1.0) * position;
 }
```

Лістинг 3.4 – Сформований шейдер фрагментів

```
#version 430
//#{INPUTS}#
in vec2 texCoord;

//#{UNIFORMS}#
```

```
uniform sampler2D samp;

//#{OUTPUTS}#
out vec4 fragColor;

//#{FUNCTION_DECLARATIONS}#
vec4 getTextureColor(vec2 texCoord, sampler2D samp);

void main(void)
{
     //#{TEXTURE}#
     fragColor = getTextureColor(texCoord, samp);
}

//#{FUNCTION_REALIZATIONS}#


vec4 getTextureColor(vec2 texCoord, sampler2D samp)
 {
     return texture2D(samp, texCoord);
 }
```

Хоча на даний час система підтримує лише два основних види шейдерів, а саме шейдер вершин та шейдер фрагментів. Є можливість за потреби розширити підтримку геометричних шейдерів, шейдерів теселяції, та розрахункових шейдерів.

У підсумку ShaderBuilder клас надає потужну абстракцію над створенням шейдерів, розглядаючи шейдери як композитні системи що динамічно створюються з компонентів.

3.6 Взаємодія бібліотеки з OpenGL

В контексті цього проєкт OpenGL слугує базовою технологією, яка обробляє всі низькорівневі взаємодії з графічним обладнанням. Хоча система рендерингу в рушії абстрагована у високорівневі концепції, такі як сутності, компоненти та блоки рендерингу, весь графічний вивід досягається шляхом прямої взаємодії з API OpenGL. Ця взаємодія є критично важливою для здатності рушія компілювати шейдери, керувати пам'яттю графічного процесора, налаштовувати стани рендерингу та здійснювати виклики малювання.

Однією з основних функцій OpenGL у цьому рушії є компіляція шейдерів та керування програмами. Рушій має систему динамічної збірки шейдерів, реалізовану за допомогою класу ShaderBuilder. Ця система генерує вихідний код GLSL для вершинних та фрагментних шейдерів, збираючи фрагменти коду з приєднаних компонентів. Після генерації вихідний код передається до класу ShaderProgram, який використовує функції OpenGL, такі як glCreateShader, glShaderSource та glCompileShader, для створення та компіляції окремих етапів шейдерів. Потім вони об'єднуються у повну програму шейдерів за допомогою функцій glCreateProgram, glAttachShader та glLinkProgram. Перевірка помилок виконується за допомогою функцій glGetShaderiv та glGetProgramiv, що дозволяє системі надавати детальні журнали у випадку помилок компіляції або компонування. Після успішного створення шейдерну програму можна активувати для використання за допомогою glUseProgram, що дозволяє їй контролювати інтерпретацію подальших команд рендерингу.

Іншою важливою взаємодією OpenGL є керування даними вершин за допомогою різних буферних об'єктів. Геометрія мешу зберігається в об'єктах Vertex Buffer Objects (VBO), а розташування цих буферів описується за допомогою об'єктів Vertex Array Objects (VAO). Крім того, індексні дані зберігаються в буферних об'єктах елементів (Element Buffer Objects, EBO), які дозволяють більш ефективно виконувати рендеринг за допомогою індексованого малювання. Ці буфери створюються та керуються за допомогою функцій glGenBuffers, glBindBuffer та glBufferData. VAO налаштовуються за допомогою glGenVertexArray, glBindVertexArray та glVertexAttribPointer для визначення способу інтерпретації даних. Після налаштування ці буфери дозволяють рушію рендерити меш за один виклик OpenGL, зменшуючи навантаження на процесор та підвищуючи продуктивність. UML діаграма класів обгортки можно побачити на рисунку 3.4.

Змінні шейдерів передаються у програми за допомогою форм та

атрибутів, які обробляються за допомогою OpenGL. Форми являють собою глобальні змінні, які є незмінними протягом одного виклику малювання і використовуються для таких речей, як матриці трансформації, параметри освітлення та прив'язки текстур. Їх можна знайти за допомогою glGetUniformLocation і встановити за допомогою таких функцій, як glUniformMatrix4fv або glUniform1f, залежно від типу форми. Атрибути, з іншого боку, є вхідними даними для кожної вершини або екземпляра у вершинному шейдері і налаштовуються за допомогою glEnableVertexAttribArray та glVertexAttribPointer. Рушій динамічно налаштовує ці прив'язки на основі того, які компоненти приєднано до сутності і які дані вони надають.

Рушій також опціонально підтримує інстанційний рендеринг. Ця техніка дозволяє малювати декілька копій однієї й тієї ж сіті з різними перетвореннями або параметрами, і все це за один виклик малювання. Для цього створюються додаткові VBO для зберігання даних для кожного екземпляра, таких як матриці моделі, і використовується glVertexAttribDivisor, щоб вказати, що певні атрибути мають оновлюватися для кожного екземпляра, а не для кожної вершини. Власне виклик малювання здійснюється за допомогою glDrawElementsInstanced, що значно покращує ефективність візуалізації у сценах з великою кількістю схожих об'єктів.

Керування текстурами - ще одна область, де OpenGL відіграє центральну роль. Рушій використовує компоненти текстур для приєднання даних зображень до сутностей. Ці зображення завантажуються до GPU за допомогою glGenTextures, glBindTexture та glTexImage2D. Параметри текстур, такі як режими фільтрації та обгортання, налаштовуються за допомогою glTexParameteri. Під час візуалізації текстури активуються за допомогою glActiveTexture та прив'язуються до шейдерів за допомогою glBindTexture, а асоціація завершується за допомогою glUniform1i, щоб вказати шейдеру, яку текстурну одиницю слід використовувати. Система

динамічної збірки шейдерів гарантує, що правильні форми sampler2D будуть присутні у шейдерах щоразу, коли залучено текстурний компонент.

Глобальна конфігурація стану OpenGL також обробляється як частина конвеєра рендерингу. Наприклад, тестування глибини вмикається за допомогою glEnable(GL_DEPTH_TEST), що дозволяє GPU коректно рендерити найближчу геометрію. Відсікання задніх граней налаштовується за допомогою glEnable(GL_CULL_FACE) і glCullFace(GL_BACK), що відкидає полігони, спрямовані від камери, для покращення продуктивності. Для ефектів прозорості та змішування рушій вмикає змішування за допомогою glEnable(GL_BLEND) та задає функцію змішування за допомогою glBlendFunc.

Сам процес рендерингу включає зв'язування усіх необхідних ресурсів та виклик власне викликів малювання. UML структуру можно побачити на рисунку 3.5. Це включає зв'язування відповідного VAO за допомогою glBindVertexArray, використання скомпільованої шейдерної програми за допомогою glUseProgram, зв'язування необхідних текстур і, нарешті, виклик glDrawElements або glDrawArrays для рендерингу геометрії. Ця послідовність оркеструється для кожного видимого об'єкта у сцені головним циклом рендерингу або системою рендерингу, що гарантує застосування правильного стану OpenGL перед рендерингом кожного кадру.

У майбутньому використання OpenGL у цьому рушії може бути розширено за кількома значущими напрямками. Система може бути адаптована для підтримки додаткових стадій шейдерів, таких як геометричні шейдери, теселяційні шейдери та обчислювальні шейдери, шляхом вдосконалення ShaderBuilder для генерування та компіляції коду для цих стадій. Можна було б підтримувати методи відкладеного рендерингу з використанням декількох цілей рендерингу, що дозволило б рушію розділити геометрію та проходження освітлення для покращення продуктивності у складних сценах. Крім того, підтримка уніфікованих буферних об'єктів (Uniform Buffer Objects, UBO) і буферних об'єктів для зберігання шейдерів

(Shader Storage Buffer Objects, SSBO) може ще більше підвищити продуктивність і гнучкість при роботі з великими наборами даних або спільними об'єктами. З впровадженням систем відсіювання та рівня деталізації (LOD) на базі GPU з використанням обчислювальних шейдерів або непрямих викликів малювання, рушій можна оптимізувати для роботи зі значно більшими сценами з мінімальним втручанням CPU. Система може бути адаптована для підтримки додаткових стадій шейдерів, таких як геометричні шейдери, теселяційні шейдери та обчислювальні шейдери, шляхом вдосконалення ShaderBuilder для генерування та компіляції коду для цих стадій. Можна було б підтримувати методи відкладеного рендерингу з використанням декількох цілей рендерингу, що дозволило б рушію розділити геометрію та проходження освітлення для покращення продуктивності у складних сценах. З впровадженням систем відсіювання та рівня деталізації (LOD) на базі GPU з використанням обчислювальних шейдерів або непрямих викликів малювання, рушій можна оптимізувати для роботи зі значно більшими сценами з мінімальним використанням CPU.

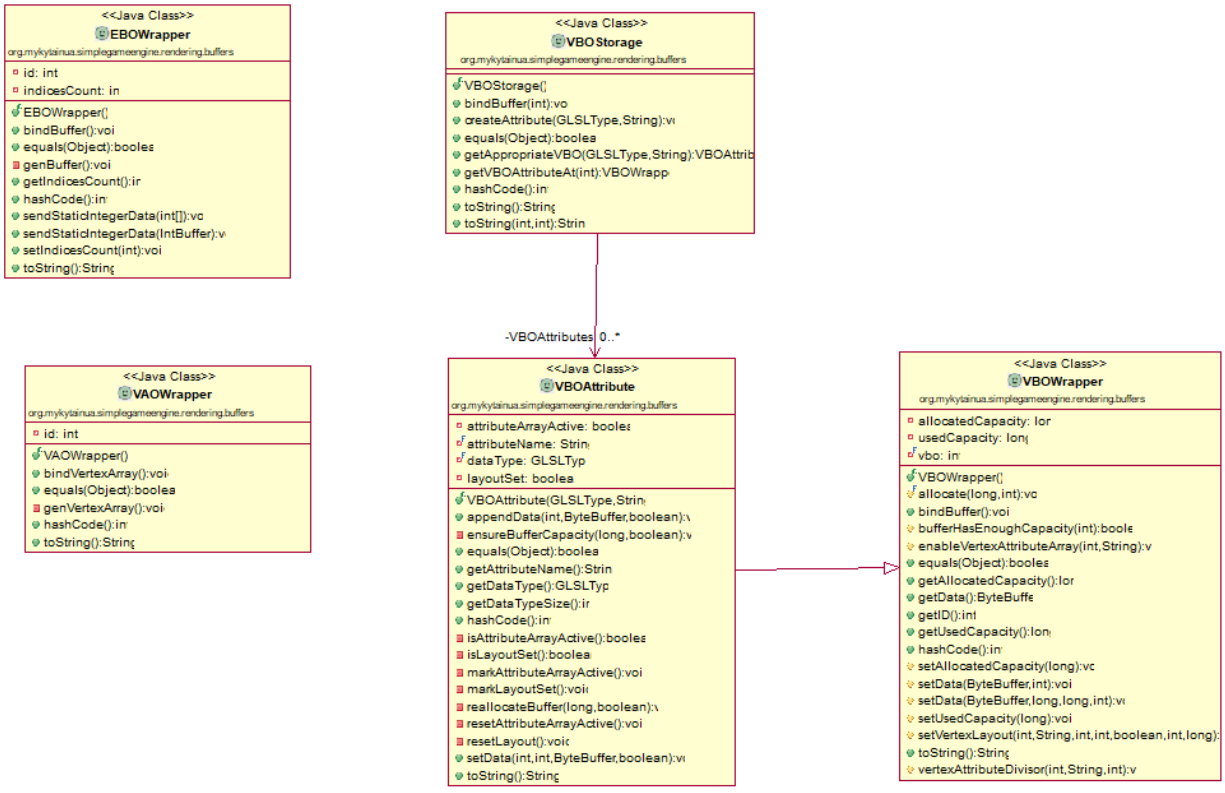

Рисунок 3.4 – Модуль для взаємодії з буферами OpenGL

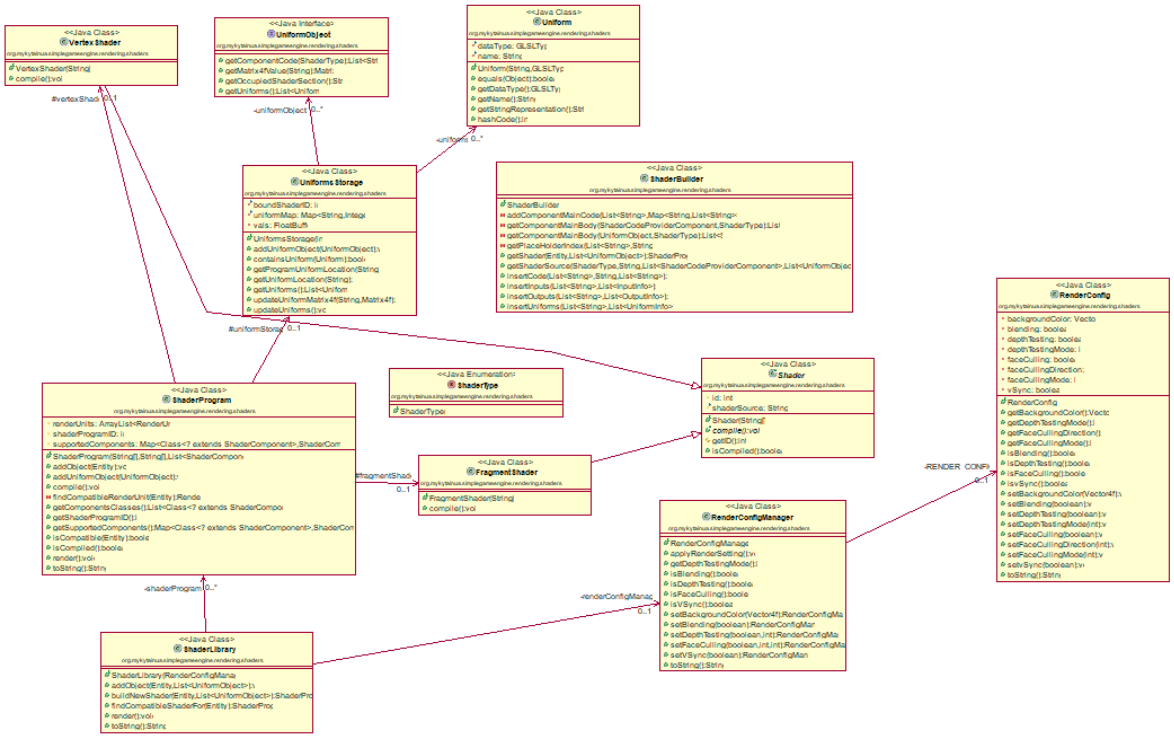

Рисунок 3.5 – Шейдерний модуль

3.7 Формування бібліотеки

Створення бібліотеки з Java-проекту за допомогою Maven передбачає низку кроків, результатом яких є багаторазовий JAR-файл, який згодом можна інтегрувати в інші проєкт. Maven спрощує весь процес, керуючи залежностями, компілюючи вихідний код та пакуючи його у формат, придатний для розгортання. Кінцевою метою є створення .jar-файлу, який містить всі скомпільовані класи та опціонально упаковані залежності, що робить його придатним для розповсюдження або зовнішнього використання.

Для початку проєкт був структурований відповідно до стандартів Maven. Вихідний код знаходиться в src/main/java, ресурси в src/main/resources, а файл pom.xml розміщується в кореневому каталозі. Цей файл pom.xml є основним конфігураційним файлом, де вказані всі деталі

збірки та залежності. Для його виконання треба виконати команду: “mvn clean package”. Важливо було вказано плагін для збірки jar файлів лістинг 3.5, щоб Maven зібрав саме бібліотеку, а не вебдодаток чи виконуваний файл:

Лістинг 3.5 – Плагін для створення jar файлу

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.4.2</version>
</plugin>
```

Це призводить до компіляції вихідного коду, запуску тестів (якщо їх не буде пропущено) і створення JAR-файлу в директорії target/. Однак цей JAR містить лише скомпільовані класи, але не зовнішні залежності. У цьому проєкті використовуються сторонні бібліотеки: JOGL, JOML й JInput. Тому їх потрібно запакувати особливим способом разом з кодом у один розповсюджуваний JAR, який ще називають "жирним".

Для цього використовують плагін Maven Shade Plugin. Він дозволяє об'єднати всі необхідні залежності у фінальний JAR-файл. Ось приклад конфігурації, яку було додано до pom файлу для реалізації цього надані в лістингу 3.6.

Лістинг 3.6 – Використання Maven Shade Plugin

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.4.1</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
implementation="org.apache.maven.plugins.shade.resource.Manifest
```

```
ResourceTransformer">

<mainClass>org.mykytainua.simplegameengine.main.Main</mainClass>
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
</plugin>
```

Тепер запуск пакунка mvn clean призводить до створення єдиного JAR-файлу, який містить весь код проекту та його залежності. Отриманий файл зазвичай знаходиться у каталозі target/.

Після того, як JAR зібрано, його можна використовувати поза проєктом.

4 ІНСТРУКЦІЯ КОРИСУВАЧА

Щоб використовувати розроблену бібліотеку спочатку потрібно імпортувати jar файл в проект та переконатися що функції бібліотеки працюють успішно.

Після того, як JAR-файл бібліотеки успішно додано до нового проекту розробник може ініціалізувати і запустити бібліотеку, використавши GameEngineWindowFactory для створення вікна. Архітектура бібліотеки має модель віконного середовища, де створюється вікно рендерингу, а сцена заповнюється сутностями. Для початку об'єкт Window отримується за допомогою методу GameEngineWindowFactory.getNEWTWindow(true), який створює вікно на основі NEWT з можливостями рендерингу OpenGL. Це вікно показується за допомогою showWindow(), а потім захоплює мишу за допомогою lockMouseCursor(true) для керування введенням. Після цього запускається рендеринг за допомогою методу startRendering().

З вікна можна отримати об'єкт Scene, доступний через window.getScene(), який є основним контейнером для всіх об'єктів, що малюються. Текстури завантажуються у сцену за допомогою scene.addTexture(String path), а пізніше витягуються за допомогою scene.getTexture(String path), щоб бути використаними як компоненти для рендерингу сутностей.

Сутності створюються за допомогою SimpleEntityBuilder, який дозволяє користувачеві вказати геометрію (Mesh) та додати компоненти, які керують візуальними атрибутами та атрибутами трансформації. До таких компонентів належать TextureComponent, SolidColorComponent, PositionComponent, RotationComponent і SizeComponent. Сутності можна створювати крок за кроком, задаючи форму (наприклад, сферу або циліндр з ShapeProvider), додаючи текстурні або кольорові компоненти та позиціонуючи їх у 3D-просторі. Після налаштування об'єкт додається до

сцени за допомогою scene.addObject().

Таким чином, бібліотека може бути інтегрована у будь-який Java-додаток, який потребує гнучкої та розширюваної системи рендерингу на основі OpenGL. Дотримуючись структури класу Main та використовуючи компонентну архітектуру, розробники можуть швидко створювати багаті 3D-сцени. Приклади розроблених сцен можна побачити на рисунках 4.1 – 4.3.

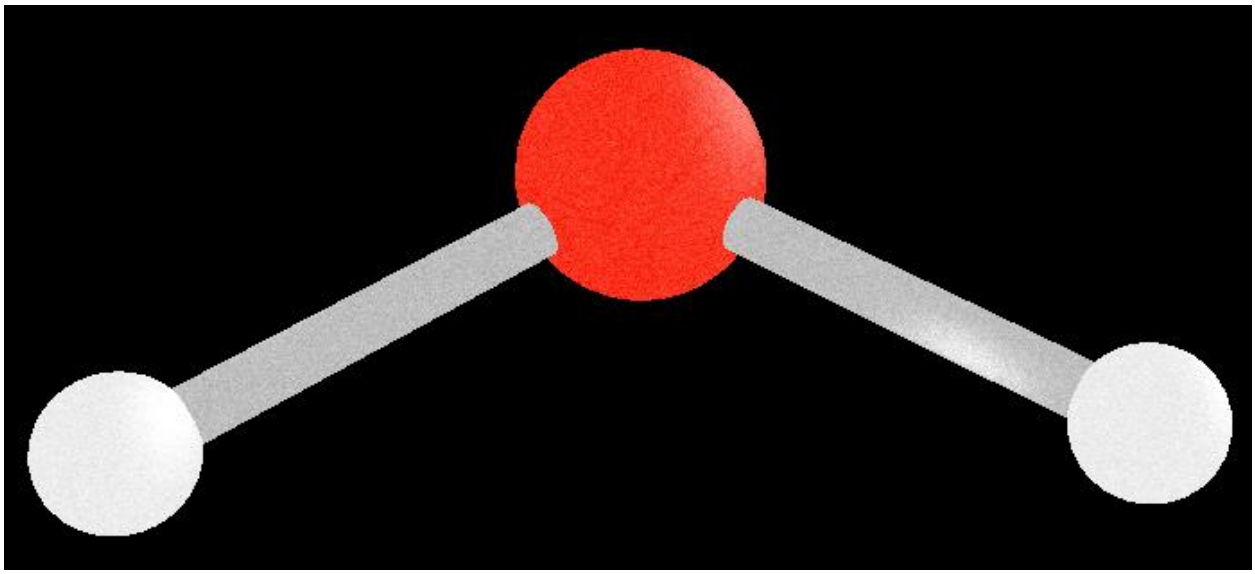

Рисунок 4.1 – Візуалізація молекули води

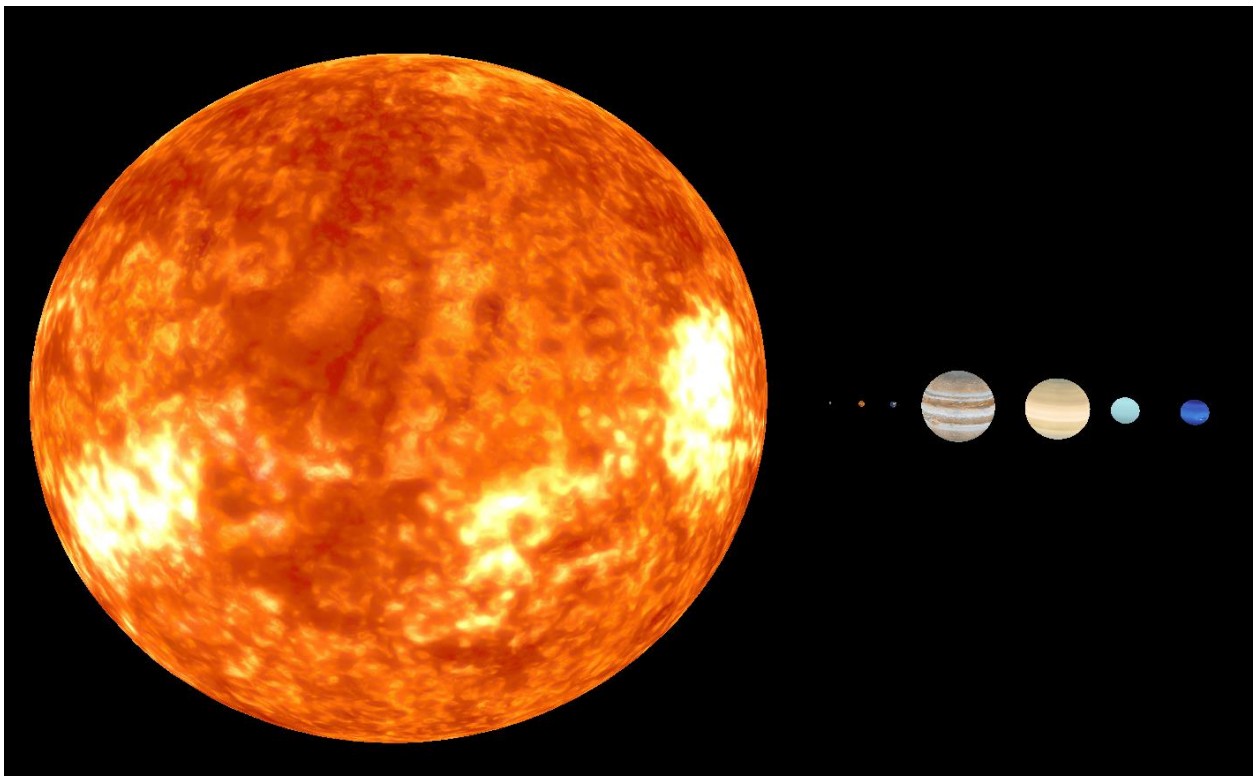

Рисунок 4.2 – Візуалізація сонячної системи

Рисунок 4.3 – Візуалізація кімнати

ВИСНОВКИ

Було розроблено бібліотеку для 3D-візуалізації на основі OpenGL. Основною мовою програмування було обрано Java, що зумовлено її широким розповсюдженням, великим набором базових бібліотек, високим рівнем абстракції, а також підтримкою автоматичного керування пам’яттю. Завдяки цим перевагам Java є привабливою для створення кросплатформених рішень у галузі графіки та візуалізації.

Для реалізації процесу рендерингу було вирішено використати OpenGL, вона є однією з найпоширеніших відносно простих бібліотек що підходить для навчальних цілей. Доступ до функціоналу OpenGL з мови програмування Java досягається через JOGL (Java OpenGL), це така Java-обгортка над нативною бібліотекою.

Щоб полегшити обчислення пов’язані із векторною та матричною математикою, було використано бібліотеку JOML (Java OpenGL Math Library), вона надає функціонал для математичних розрахунків використовуючи матриці та вектори.

Довелось окрему увагу приділити обробці введення. Через слабку підтримку постійного вводу з клавіатури та миші стандартні бібліотеки Java такі як Swing чи AWT. Для уникнення таких проблем було використано бібліотеку JInput. Ця бібліотека отримує дані про ввод корисувача напряму від системи уникаючи зайвих затримок та проблем з синхронізацією чи фіксацією подій.

В основі архітектури модуля рендерингу лежить принцип компонентної моделі (Entity-Component System). Це дозволяє користувачу будувати сцени з окремих, взаємозамінних компонентів, що суттєво підвищує гнучкість і масштабованість системи. Такий підхід позбавлений недоліків об’єктно-орієнтованого підходу з ієрархіями спадкування що заважало б в майбутньому розширювати систему. Також відкриваються нові можливості для автоматизації генерації шейдерів згідно з набором вказаних компонентів.

На відміну від інших існуючих бібліотек рендерингу для Java, такими як Java3D або LWJGL, розроблена бібліотека має меншу складність, що робить її непоганою альтернативою вже існуючим рішенням у галузях де не потрібен детальний контроль за ресурсами.
Подальший розвиток цієї бібліотеки можливий шляхом значного розширення функціональності бібліотеки. Так, наприклад можливе впровадження таких функцій, як підтримка фізики, колізій, анімацій, а також покращена система матеріалів і освітлення. Ще перспективним напрямком є розробка графічного інтерфейсу користувача (GUI) для зручнішої роботи з компонентами сцени та створення власного редактора сцен. Для оптимізації є сенс впровадити підтримку багатопотоковості, кешування ресурсів, а в далекій перспективі й підтримка новіших графічних API, таких як Vulkan, для збільшення продуктивності на сучасних GPU.
Таким чином, результатом практики стала функціональна, гнучка та розширювана бібліотека для тривимірної візуалізації з безмежним потенціалом для подальшого розвитку. Отримані знання та досвід у такій сфері як графічне програмування матимуть позитивний вплив на професійне становлення як розробника.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Bloch J. Effective java, 2nd edition. Pearson Education, Limited, 2016. 368 p.

2. Chen J. X., Chen C. Foundations of 3D graphics programming: using JOGL and java3d. Springer London, Limited, 2016.

3. Haemel N., Sellers G., Jr W. R. OpenGL superbible: comprehensive tutorial and reference. Addison-Wesley Longman, Incorporated, 2015. 880 p.

4. Horstmann C. S. Core java, volume ii--advanced features (11th edition). Prentice Hall, 2019. 960 p.

5. Index of /deployment/webstart/javadoc/jogl. JogAmp.org - Java graphics, audio, media and processing libraries exposing OpenAL, OpenGL (+FFmpeg) and OpenCL. URL: https://jogamp.org/deployment/webstart/javadoc/jogl/ (date of access: 23.05.2025).

6. Joml 1.10.8 javadoc (org.joml). Free Java Doc hosting for open source projects - javadoc.io. URL: https://javadoc.io/doc/org.joml/joml/latest/index.html (date of access: 23.05.2025).

7. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL. URL: https://learnopengl.com/ (date of access: 23.05.2025).

8. Ostermueller E. Troubleshooting java performance. Berkeley, CA : Apress, 2017. URL: https://doi.org/10.1007/978-1-4842-2979-8 (date of access: 23.05.2025).

9. Schildt H. Java: the complete reference, seventh edition (complete reference series). 7th ed. McGraw-Hill Osborne Media, 2006. 1024 p.

10. Shreiner D., Sellers G., Kessenich J. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4. 5 with SPIR-V. Pearson

Education, Limited, 2016.

11. Sierra K. Head first design patterns. O'Reilly Media, Incorporated, 2004.

12. Wayne K., Sedgewick R. Algorithms. Pearson Education, Limited, 2010. 432 p.

13. Computer graphics from scratch. No Starch Press, Incorporated, 2020. 180 p.

14. Foundations of 3D graphics programming: using JOGL and java3d. / ed. by C. Chunyang. 2nd ed. London : Springer, 2008. 386 p.

15. Gopichand M. Fundamentals of Computer Graphics - Computer Graphics: Computer Graphics Book - Computer Graphics Principles and Practice. Independently Published, 2021.

16. Gordon V. S., Clevenger J. L. Computer graphics programming in opengl with java. Mercury Learning & Information, 2021.

17. Lengyel E. Mathematics for 3D game programming and computer graphics. Course Technology, 2020.