МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім.Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ**

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Розробка редактора шейдерів у реальному часі**

Виконав: студент 4 курсу, групи 6.1210-пзс
ціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
Н. С. Кузенний
(ініціали та прізвище)
Керівник к.т.н., доцент В.І.Заяц
(посада,вчене звання,науковий ступінь,підпис,ініціали та прізвище)

Рецензент директор ТОВ «Дісітел» П.О. Лютий
(посада,вчене звання,науковий ступінь,підпис,ініціали та прізвище)

Запоріжжя
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙНАВЧАЛЬНО-НАУКОВИЙІНСТИТУТ
ім.Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення рівень вищої освіти __перший (бакалаврський)________
Спеціальність ___121 Інженерія програмного забезпечення______
(код та назва)

Освітня програма_____Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_______________Тетяна КРИТСЬКА
“_01_” __березня_____2024_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

__________Кузенному Нікіті Сергійовичу________
(прізвище,ім’я,по батькові)

1. Тема роботи Розробка редактора шейдерів у реальному часі

керівник роботи _Заяц Валерій Іванович, доцент, к.т.н._____
(прізвище, ім’я ,по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від__26.12.2023 № 2215-с

2. Строк подання студентом кваліфікаційної роботи 07.06.2024
3. Вихідні дані бакалаврської роботи
   - комплект нормативних документів ;
   - технічне завдання до роботи.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   - огляд та збір літератури стосовно теми кваліфікаційної роботи;
   - огляд та аналіз існуючих рішень та аналогів;
   - дослідження проблеми розробки редактора шейдерів для OpenGL;
   - створення програмного продукту та його опис;
   - дослідження поставленої проблеми та розробка висновків.
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
_____слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис,дата	
		Завданняприйняв	

7. Дата видачі завдання 01.03.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строквиконання етапів бакалаврської роботи	Примітка
1	Аналіз предметної області	19.02-03.03.2024	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	04.03-06.03.2024	виконано
3	Аналіз існуючих рішень	07.03-20.03.2024	виконано
4	Узгодження подальших дій з науковим керівником	21.03.2024	виконано
5	Визначення вимог	22.03-28.03.2024	виконано
6	Вибір технологій реалізації	29.03-04.04.2024	виконано
7	Створення дизайн інтерфейсу користувача за допомогою Qt Framework	05.04-11.04.2024	виконано
8	Імплемінтація попереднього перегляду у реальному часі	12.04-18.04.2024	виконано
9	Дизайн та реалізація архітектури ієрархії обє’ктів сцени	19.04-25.04.2024	виконано
10	Реалізація класів моделі редактора шейдорів	26.04-2.05.2024	виконано
11	Апробація застосунку	03.05-09.05.2024	виконано
12	Оформленя звіту	10.05-17.05.2024	виконано
13	Оформленя презентації	18.05-26.05.2024	виконано

Студент ______________________ Кузенний Н.С.
(підпис) (прізвище та ініціали)

Керівник роботи ______________________ Заяц В.І.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено
Нормоконтролер ______________________ Скрипник. І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок — 75

Рисунків — 13

Джерел — 13

Кузенний Н.С. Розробка редактора шейдерів у реальному часі: кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник В. І. Заяц. Запоріжжя : ЗНУ, 2024. 75 с.

Кваліфікаційна робота представляє розробку редактора шейдерів у реальному часі який використовує OpenGL і Qt Framework.

Шейдер – це програма одного із ступенів графічного конвеєра, що використовується в тривимірній графіці для визначення остаточних параметрів об'єкта чи зображення. Вони відповідають за генерацію візуальних ефектів.

Основною метою є створення універсального та комплексного інструменту для програмування шейдерів, який не буде прив’язаний до конкретного програмного середовища, усуваючи значну проблему існуючих інструментів.

Застосунок підтримує редагування різних типів шейдерів, у тому числі вершинних, фрагментних і геометричних шейдерів, забезпечуючи комплексне середовище розробки. А інтерфейс користувача інтуїтивно зрозумілий і доступний.

Система використовує шаблон проектування Model-View-Controller (MVC), щоб забезпечити чітке розділення завдань, підвищуючи зручність обслуговування та масштабованість.

За допомогою перелічених особливостей редактор шейдерів усуває обмеження існуючих інструментів.

Ключові слова: Шейдери, OpenGL, сцена, вершини, візуалізація, редактор шейдерів, у реальному часі.

ABSTRACT

Pages — 75

Drawings — 13

Sources — 13

Kuzennyi N.S. Development of Real-Time Shader Editor: bachelor's thesis in specialty 121 "Software Engineering" / Science. manager V.I. Zayats. Zaporizhzhia: ZNU, 2024. 75 p.

The qualification work presents the development of a real-time shader editor that uses OpenGL and the Qt Framework.

A shader is a program of one of the stages of the graphics pipeline, used in 3D graphics to determine the final parameters of an object or image. They are responsible for generating visual effects.

The main goal is to create a universal and comprehensive shader programming tool that will not be tied to a specific programming environment, eliminating a significant problem of existing tools.

The application supports the editing of different types of shaders, including vertex, fragment and geometric shaders, providing a comprehensive development environment. And the user interface is intuitive and accessible.

The system uses the Model-View-Controller (MVC) design pattern to provide a clear separation of tasks, increasing maintainability and scalability.

With the help of the listed features, the shader editor eliminates the limitations of existing tools.

Keywords: Shaders, OpenGL, scene, vertices, rendering, shader editor, real-time.

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ АНАЛОГІВ ПРЕДМЕТНОЇ ОБЛАСТІ 12
1.1 Огляд літературних джерел 12
1.2 Аналіз аналогів програмного забезпечення 13
1.2.1 SHADERed 13
1.2.2 GLSL Sandbox 15
1.2.3 ShaderToy 17
1.2.4 Висновки аналізу аналогів 18
1.3 План робіт 19
2 ТЕОРЕТИЧНІ ОСНОВИ ДЛЯ РОЗРОБКИ СИСТЕМ З ВИКОРИСТАННЯМ ШЕЙДЕРНИХ ПРОГРАМ 21
2.1 Загальні відомості про шейдерні програми 21
2.2 Аналіз сучасних технологій розробки редактора шейдерів 23
3 РОЗРОБКА РЕДАКТОРА ШЕЙДЕРІВ У РЕАЛЬНОМУ ЧАСІ 27
3.1 Опис предметної області 27
3.2 Архітектура системи 28
3.2.1 Шаблон проектування 28
3.2.2 Компонент модель 29
3.2.3 Компонент представлення 42
3.2.4 Компонент контролер 45
3.3 Функціональні вимоги до системи 48
3.3.1 Загальні вимоги 48
3.3.2 Бібліотека підпрограм (класів) 49
3.4 Функціонал системи 50

3.4.1 Функціонал представлення та контроллера 50
3.4.2 Функціонал моделі 59
3.5 Вимоги до інтерфейсу.. 67
3.6 Аналіз вимог до користувачів редактора шейдерів.............................. 68
3.7 Аналіз вимог до апаратного та програмного забезпечення.................... 69
3.8 Тестування 70
ВИСНОВКИ... 73
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 74

ВСТУП

Актуальність теми

Створення редактора шейдерів у реальному часі усуває проблеми існуючих рішень. Багато редакторів адаптовані до певних програмних платформ, таких як Unity або Godot, що обмежує їх універсальність у різних програмах на основі OpenGL. Розробляючи редактор, який не залежить від конкретної програмної екосистеми, забезпечується сумісність із широким спектром програм OpenGL, роблячи розробку шейдерів більш доступною та гнучкою.

Однією з ключових переваг редактора є його підтримка редагування вершинних та геометричних шейдерів. Ці типи шейдерів є важливими інструментами для маніпулювання геометрією та додавання динамічних ефектів до візуальних елементів, але вони часто не імплементуються або погано підтримуються в багатьох існуючих редакторах. Створюючи можливість працювати з цими типами шейдерів, редактор дає змогу розробникам ефективніше працювати з графікою на основі OpenGL.

Крім того, зворотній зв’язок і візуалізація в режимі реального часу є дуже важливою під час розробки шейдерів, дозволяючи бачити результати впроваджених змін. Це прискорює процес розробки, надаючи більш інтерактивне середовище для створення шейдерів.

Попит на складну графіку та візуальні ефекти в програмах продовжує зростати [13]. Незалежно від того, чи йдеться про ігри, моделювання, наукову візуалізацію чи інші сфери, здатність створювати та точно налаштовувати шейдери є важливою для досягнення привабливих візуальних результатів. Редактор шейдерів задовольнить важливу потребу, пропонуючи не залежний від платформи, багатофункціональний застосунок, який дає змогу розробникам удосконалити процес роботи з шейдерами для систем на основі OpenGL.

Мета дослідження

Створення додатку за допомогою якого можна проводити редагування вершинних, геометричних та фрагментних шейдерів у реальному часі.

Завдання дослідження

Аналіз існуючих рішень редагування та створення шейдерів. Вивчення архітектури OpenGL та основних можливостей цього графічного інтерфейсу приладного рівня. Розробка архітектури редактора шейдерів, яка дозволить продуктивно виконувати задачі розробки та редагування шейдерного коду, а також достатньо гнучку для підтримки зручного розширення функціоналу у майбутньому.

Об’єкт дослідження

Об’єктом дослідження є процесс розробки шейдерів на мові GLSL для графічного API OpenGL.

Предмет дослідження

Предметом дослідження є вивчення принципів роботи з OpenGL, розробки сумісного з ним застосунку з графічним інтерфейсом та дослідження мови програмування шейдерів GLSL, для встановлення вимог пов'язаних з работою з цією мовою.

Методи дослідження

Дослідження теми створення редактора шейдерів реального часу для додатків OpenGL передбачає кілька ключових методів. Один із підходів полягає у проведенні поглиблених оглядів літературних джерел та аналізу існуючих редакторів шейдерів, як загального призначення, так і тих, що стосуються певних програмних платформ. Крім того, ознайомлення з технічною документацією, форумами та спільнотами, пов’язаними з

розробкою OpenGL і шейдерів, дає цінну інформацію про найкращі практики, загальні проблеми та тенденції.

Іншим методом є створення прототипів. Це включає роботу з OpenGL для створення прототипів редакторів шейдерів, тестування різних функцій і оцінки продуктивності та аспектів зручності використання.

Практичне значення одержаних результатів

Практичне значення одержаних результатів полягає в його актуальності для спільноти розробників. Робота з графічним API OpenGL часто вимагає швидкого та ефективного способу експериментувати з різними шейдерними ефектами. Надання спеціального редактора шейдерів створює рішення для цієї поширеної проблеми, дозволяючи користувачам швидко змінювати дизайн своїх шейдерів і прискорювати процес розробки. Це не тільки підвищує продуктивність, але й допомагає створювати інновації в області комп'ютерної графіки.

Глосарій

OpenGL – це міжмовний бгатоплатформний інтерфейс прикладного програмування для візуалізації 2D і 3D векторної графіки.

OpenGL Shading Language (GLSL) — це мова шейдерного коду високого рівня з синтаксисом на основі мови програмування C.

Qt Framework – це безкоштовний набір віджетів із відкритим вихідним кодом для створення графічних інтерфейсів користувача, а також багатоплатформних програм, які працюють на різних програмних і апаратних платформах.

VBO – це вершинний буферний об’єкт, який надає методи для завантаження даних вершин (положень, кольорів, нормалей тощо) у пам’ять графічної карти для оптимізації продуктивності візуалізації.

VAO – це об’єкт масиву вершин, який інкапсулює стан, необхідний для визначення вхідних даних для вершинного шейдера, включаючи формат даних вершин і VBO, які містять дані.

EBO – це об’єкт буфера індексу, який зберігає індекси, які використовуються для визначення вершин у VBO. Це дозволяє ефективно повторно використовувати вершин для малювання складних фігур без зайвих даних.

Вершинний шейдер – це тип шейдерної програми в OpenGL, яка обробляє дані кожної вершини. Зазвичай він обробляє перетворення, такі як переклад, обертання та масштабування.

Геометричний шейдер – це додатковий шейдер, який приймає цілі примітиви (точки, лінії чи трикутники) як вхідні дані та може генерувати нові примітиви.

Фрагментний шейдер – це тип шейдерної програми в OpenGL, яка обробляє фрагменти (потенційні пікселі), створені шляхом растеризації. Він визначає остаточний колір та інші атрибути кожного пікселя.

Уніформа шейдера – це глобальна змінна GLSL, оголошена за допомогою ключового слова uniform. Вона залишається постійною для всіх оброблених вершин або фрагментів під час одного виклику малювання та використовується для передачі даних (таких як матриці перетворення, параметри освітлення тощо) від центрального процесора до графічного процесора.

Макет вершини шейдера – визначає формат і організацію даних вершин у VBO. Він включає такі специфікації, як тип, розмір і крок кожного атрибута вершини (наприклад, положення, нормаль, координата текстури) і те, як вони пов’язані з вхідними змінними вершинного шейдера.

1 АНАЛІЗ АНАЛОГІВ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд літературних джерел

Тема створення шейдерного редактора є дуже актуальною в комп'ютерній графіці, розробці ігор та інтерактивних додатках. Розуміння шейдерів і можливість створювати та редагувати їх у режимі реального часу дає можливість графічним розробникам експериментувати з різними візуальними ефектами, техніками візуалізації та художніми стилями. Це призводить до більш творчої та візуально привабливої графіки в програмах.

Наявність спеціального редактора шейдерів спрощує робочий процес для розробників у галузі комп'ютерної графіки, надаючи централізований інструмент для створення, редагування, налагодження та тестування шейдерів. Це підвищує продуктивність і скорочує час розробки.

Дослідження літературних джерел має вирішальне значення під час реалізації проекту, особливо такого, що включає складні технології, як OpenGL, і розробку шейдерів для програми редактора шейдерів. По-перше, літературні джерела забезпечують міцну основу знань і розуміння базових концепцій, принципів і найкращих практик. Такі знання допомагають розробникам приймати зважені рішення, вибирати відповідні технології та розробляти ефективні та масштабовані рішення.

Крім того, літературні джерела надає доступ до детальної документації, специфікацій, навчальних посібників, прикладів коду та довідкових матеріалів, які є важливими для впровадження певних функцій, розуміння функцій API та дотримання стандартів кодування. Ця інформація є важливою для забезпечення якості коду, сумісності та дотримання галузевих стандартів.

Однією з основних мов програмування шейдерів є GLSL, яка використовується у програмах на основі OpenGL, тому розробка редактора шейдерів, з підтримкою цієї мови, допоможе спростити роботу з кодом і покращити робочий процес розробки. Синтаксис та особливості

визначаються у книзі “The OpenGL Shading Language” [1], що полегшує розробку програмної системи.

Проблемою, відповіді на яку можна знайти у книзі “Real-Time Rendering” [2], може бути оптимізація продуктивності редактора та використання пам’яті, особливо при роботі з великими та складними шейдерами, з метою покращення швидкості візуалізації та інтеграція компіляції шейдерів у реальному часі та попереднього перегляду. Це створює технічні перешкоди, особливо під час роботи зі складними шейдерами або конвеєрами візуалізації.

Зробивши аналіз матеріалів книги “OpenGL Programming Guide” [3] можна виділити проблему налагодження та обробки помилок у шейдерах, які можуть бути складними через обмежений зворотний зв’язок від графічного процесора. Це вимагає ефективних механізмів звітування про помилки та інструментів налагодження в редакторі.

Створення універсального редактора шейдерів у реальному часі, розробленого для графічного API OpenGL. На відміну від існуючих редакторів шейдерів, адаптованих до певного програмного забезпечення, наприклад Unity або Godot. Розробляємий редактор дає змогу розробникам працювати з OpenGL у будь-якому контексті.

Така універсальність допоможе підвищенню ефективності розробників. Вони можуть легко інтегрувати свої шейдери у свої проекти, не турбуючись про проблеми сумісності.

Загалом, редактор шейдерів у реальному часі для OpenGL надає розробникам надійний, гнучкий набір інструментів для роботи з графікою.

1.2 Аналіз аналогів програмного забезпечення

1.2.1 SHADERed

SHADERed — це програма для редагування шейдерів, призначена для розробників і художників, які працюють з графікою та візуальними

ефектами. Вона забезпечує зручний інтерфейс для створення, редагування та тестування шейдерів у реальному часі. Однією з ключових переваг є багатоплатформна сумісність з такими операційними системами, як Windows і Linux, а також можливість бути скомпільованою для macOS, що робить програму доступною для широкого кола користувачів незалежно від їхніх уподобань операційної системи.

Важливою функцією SHADERed є можливість попереднього перегляду у реальному часі (див. Рисунок 1), що дозволяє користувачам негайно побачити наслідки змін коду шейдера без необхідності тривалих процесів компіляції. Цей зворотний зв’язок у реальному часі оптимізує робочий процес розробки та забезпечує швидку ітерацію, що має вирішальне значення для ефективного досягнення бажаних візуальних результатів.

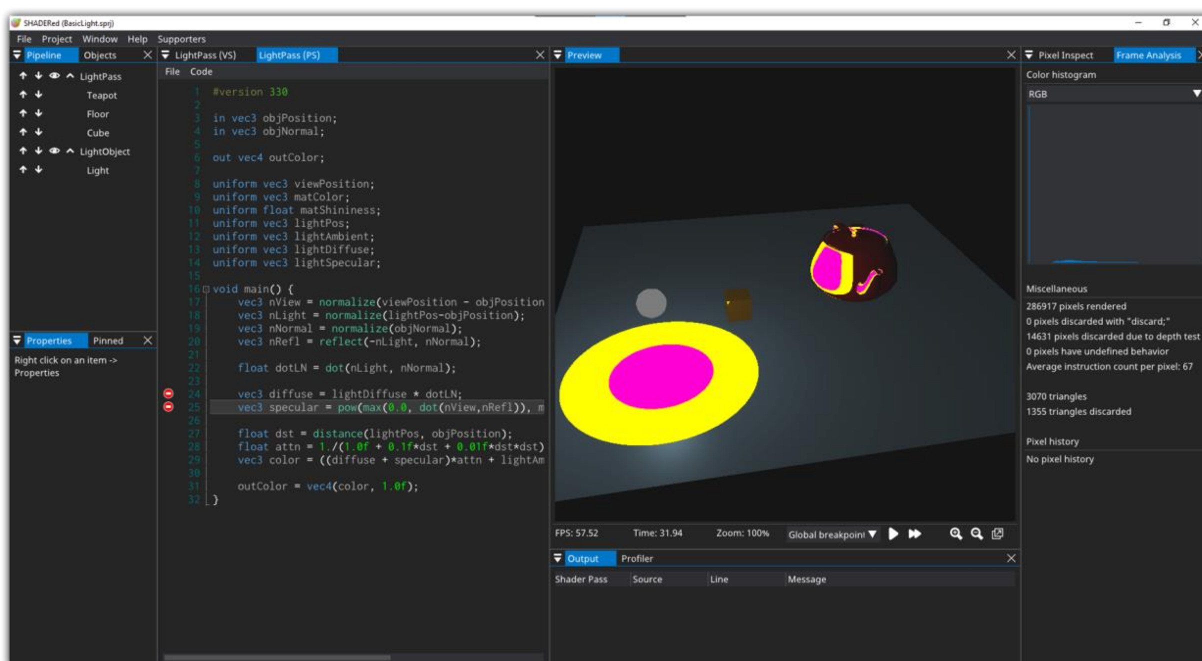

Рисунок 1 — *Інтерфейс застосунку SHADERed*

Крім того, SHADERed пропонує різноманітні вбудовані інструменти, які допомагають у розробці шейдерів, включаючи налагоджувач для виявлення та виправлення помилок, профайлер для оптимізації продуктивності та візуальний редактор для створення складних шейдерів. Ці

функції підвищують продуктивність і допомагають користувачам створювати високоякісні шейдери з більшою легкістю.

Незважаючи на те, що SHADERed має багато сильних сторін, варто зазначити, що, як і будь-яке інше програмне забезпечення, воно може складним для нових користувачів, особливо тих, хто не знайомий із концепціями програмування шейдерів.

Зробивши аналіз можна підсумувати, що SHADERed виділяється як універсальна багатоплатформна програма для редагування шейдерів із можливостями попереднього перегляду в реальному часі, набором вбудованих інструментів для ефективної розробки та широким застосуванням у різноманітних проектах у різних галузях.

1.2.2 GLSL Sandbox

GLSL Sandbox — це програма для редагування шейдерів, призначена для створення та керування графічними шейдерами за допомогою синтаксису GLSL. Ця програма також надає розробникам та художникам можливість експериментувати з шейдерами та візуальними ефектами у реальному часі, і робити це безпосередньо у браузері.

Головною перевагою програми GLSL Sandbox є її доступність. Будучи веб-платформою, користувачі можуть отримати доступ до неї з будь-якого сучасного веб-браузера без необхідності встановлення чи складного налаштування. Це робить її зручною для швидкого створення прототипів, спільного використання та співпраці між користувачами.

Як і інші аналоги GLSL Sandbox пропонує можливість редагування шейдерів у реальному часі, але інтерфейс застосунку є менш зручним порівняно зі спеціальним програмним забезпеченням (дви. Рисунок 2). Одним із недоліків є обмежена функціональність редактора коду, якому бракує розширених функцій, таких як згортання коду та кілька вкладок. Крім того, навігація між спільними шейдерами або пошук конкретних ефектів

може бути складним через відсутність надійних інструментів організації та параметрів фільтрації.

GLSL Sandbox пропонує ряд ключових функцій. Сюди входить редактор коду з підсвічуванням синтаксису та автозавершенням, настроювані уніформи для налаштування параметрів, підтримка введення текстур та можливість зберігати шейдери та ділитися ними з іншими.

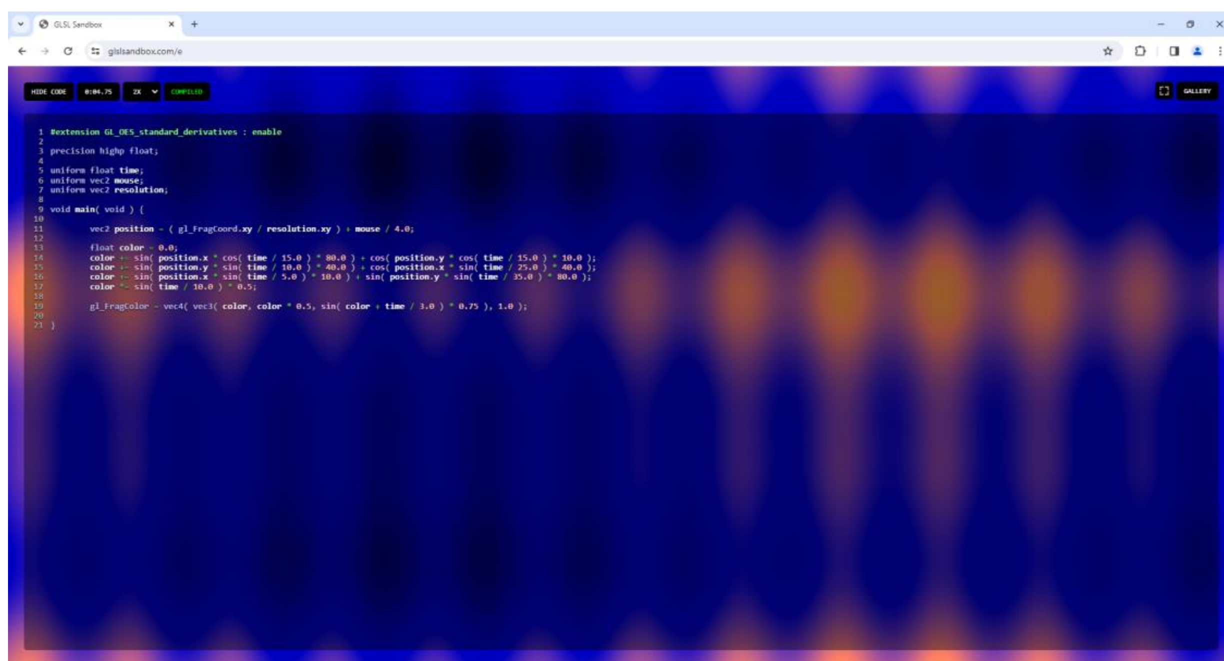

Рисунок 2 — *Інтерфейс застосунку GLSL Sandbox*

Зі зробленного аналізу витікає що, GLSL Sandbox — це зручна та універсальна програма для створення та вивчення шейдерів у реальному часі. Її доступність, попередній перегляд у реальному часі роблять її цінним інструментом для розробників, художників, викладачів та всіх, хто цікавиться графічним програмуванням. Хоча він може мати обмеження порівняно зі спеціальними середовищами розробки шейдерів, його простота робить його популярним вибором для експериментування з шейдерами та спільного використання.

1.2.3 ShaderToy

Shadertoy — це онлайн-спільнота та платформа для професіоналів та ентузіастів у галузі комп’ютерної графіки, які діляться, навчаються та експериментують з техніками візуалізації за допомогою коду GLSL.

Однією з ключових переваг ShaderToy є його доступність. Як веб-інструмент, користувачі можуть отримати доступ до нього безпосередньо зі своїх браузерів без необхідності встановлення чи додаткового програмного забезпечення. Завдяки цьому будь-кому буде легко почати створювати та ділитися шейдерами, не роблячи налаштувань складного середовища.

Також ShaderToy має велику бібліотеку шейдерів і прикладів, доступних будь-кому. Користувачі можуть досліджувати широкий спектр готових шейдерів і вчитися на них, отримуючи уявлення про різні техніки та ефекти, які використовуються в комп’ютерній графіці. Станом на середину 2021 року тисячі користувачів зробили понад 52 тисячі публічно доступних проектів.

Як і будь-який редактор шейдерів, ShaderToy підтримує можливість бачити результати у реальному часі (див.

Рисунок 3). Використання разом з цим достатньо інтуїтивного інтерфейсу, дозволяє швидко вносити коригування та експериментувати. Цей зворотний зв’язок має велике значення для розробки шейдерів, оскільки допомагає підвищити продуктивність.

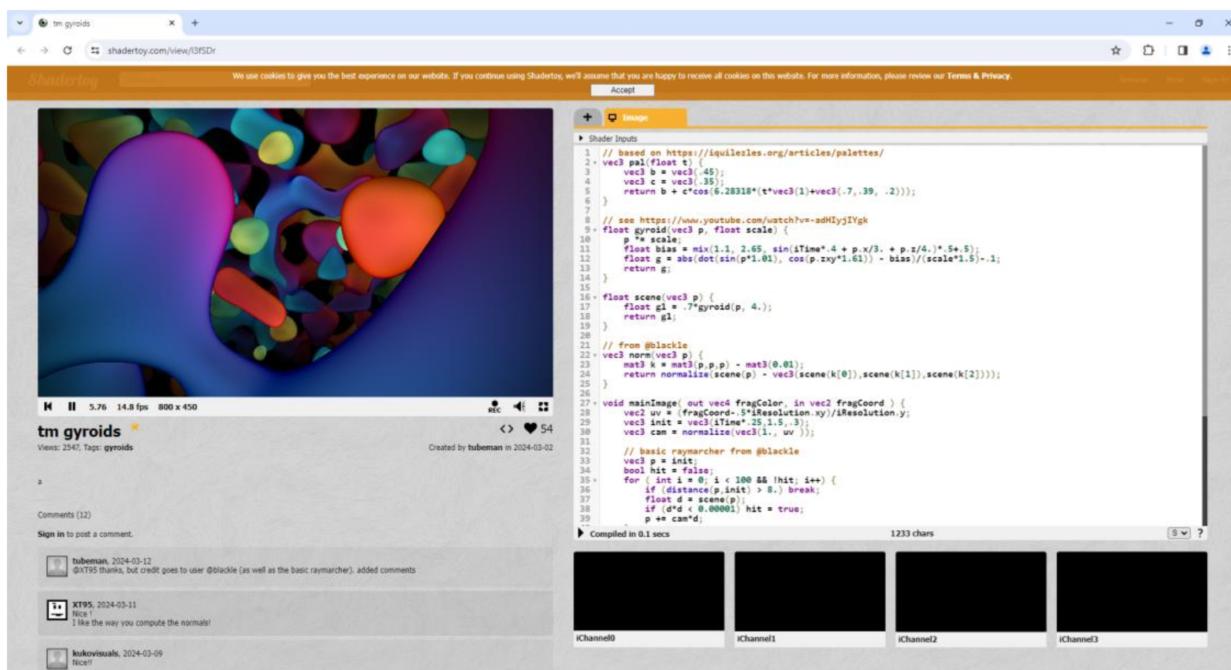

Рисунок 3 — *Інтерфейс застосунку ShaderToy*

Однак ShaderToy також має деякі обмеження. Оскільки це переважно веб-інструмент, він може не запропонувати такий самий рівень продуктивності та функціональності, як окремі програми для редагування шейдерів. Складні шейдери з важкими обчисленнями можуть мати сповільнення або обмеження через середовище браузера.

Отже можна зробити висновок, що ShaderToy — це доступний інструмент для створення шейдерів і керування ними. Його візуалізація в режимі реального часу, велика бібліотека та простота використання роблять його цінним ресурсом для розробників. Хоча він може мати деякі обмеження порівняно з окремими програмами, його веб-природа та активна спільнота роблять його популярним вибором для розробки шейдерів та експериментів.

1.2.4 Висновки аналізу аналогів

ShaderED, GLSL Sandbox і ShaderToy підтримують візуалізацію в реальному часі, фундаментально важливу функцію для розробки шейдерів.

ShaderED вирізняється великою кількістю вбудованих інструментів і зручним налагоджувачем. Однак основна зосередженість на інтеграції з ігровим рушієм Godot може обмежити його для тих, хто не входить до цієї екосистеми, представляючи серйозний недолік.

Сильними сторонами GLSL Sandbox і ShaderToy є доступність, оскільки вони є веб-додатками. Ця доступність робить їх особливо привабливими для швидкого експериментування та спільного використання. Крім того, ShaderToy має велику бібліотекою шейдерів, доданих користувачами.

Тим не менш, помітним обмеженням, яке спільне для GLSL Sandbox і ShaderToy, є той факт, що вони дозволяють працювати лише з фрагментними шейдеами, упускаючи підтримку вершинних або геометричних шейдерів. Хоча це оптимізує розробку проектів, орієнтованих на фрагментні шейдери, це може бути обмеженням для тих, кому потрібно працювати з усіма видами шейдерів.

Підсумовуючи, кожна з цих програм для редагування шейдерів пропонує певні переваги, від комплексних інструментів до доступності та внеску спільноти.

1.3 План робіт

Мета кваліфікаційної роботи — це створення додатку за допомогою якого можна проводити редагування вершинних, геометричних та фрагментних шейдерів у реальному часі. Виконання поставленої задачі можна розбити на наступні етапи:

1. Дослідження предметної області. Під час дослідження предметної області потрібно визначити технології, які допоможуть у вирішені таких питань як робота з інтерфейсом прикладного рівня OpenGL, розробка шейдерів та інтеграція графічного інтерфейсу. Зробити це

можливо за допомогою ознайомлення з відповідними літературними джерелами, онлайн ресурсами та існуючими аналогами.

2. Аналіз та вибір технологій розробки. На цьому етапі проводиться аналіз різних технологій розробки та інструментів, придатних для створення програми редактора шейдерів. Важливими факторами на цьому етапі можна вважати такі елементи, як мови програмування, бібліотеки, середовища розробки і системи контролю версій.
3. Створення архітектури програмної системи. Важливою частиною створення програмної систем є розробка її архітектури, а саме компонентів, модулів та функціональних можливостей. Також важливо враховувати, що застосунок буде використовувати класичний архітектурний шаблон Model-View-Controller (MVC), який допоможе ефективно організувати структуру проекту.
4. Реалізація запланованої архітектури. На цьому етапі проходить створення програмної системи на основі визначеного плану. Серед найважливіших модулів, які потрібно розробити, можна виділити наступні: розробка інтерфейсу користувача редактора шейдерів, інтеграція візуалізації OpenGL для попереднього перегляду та керування файлами для збереження та завантаження шейдерів.
5. Перевірка коректної роботи та наявності функціональних вимог: Перевірка функціональність і продуктивність програми редактора шейдерів виконується за допомогою апробації. Це включає в себе ретельну перевірку кожного компонента шляхом апробації: компонентів, модулів та всієї системи.

2 ТЕОРЕТИЧНІ ОСНОВИ ДЛЯ РОЗРОБКИ СИСТЕМ З ВИКОРИСТАННЯМ ШЕЙДЕРНИХ ПРОГРАМ

2.1 Загальні відомості про шейдерні програми

Шейдери – це програми, розроблені для обробки зображень використовуючи графічний процесор. Їх основною функцією є визначення візуалізації об’єктів у 3D сцені маніпулюючи вершинами та пікселями. На відміну від операцій центрального процесора, шейдери виконуються паралельно, використовуючи обчислювальну здатність сучасних графічних процесорів. Ця можливість паралельної обробки має велике значення для відображення динамічних сцен у реальному часі для будь якої інтерактивної програми.

Шейдери забезпечують можливість створювати велику кількість візуальних ефектів, які сприяють реалістичності та естетичній якості цифрових зображень. Такі ефекти, як відображення рельєфу, динамічні тіні та глибина різкості, стають можливими завдяки програмуванню шейдерів.

Програми шейдерів зазвичай пишуться мовами високого рівня, такими як GLSL для OpenGL або HLSL для DirectX. Ці мови забезпечують конструкції для виконання математичних і логічних операцій над векторами та матрицями, що є важливими для графічних обчислень. Написаний код шейдера потім компілюється та виконується на графічному процесорі, використовуючи його можливості паралельної обробки.

Термін «шейдер» охоплює різні типи програм, кожна з яких виконує певну роль у графічному конвеєрі (див. Рисунок 4 — *Приклад класичного графічного конвеєру*).

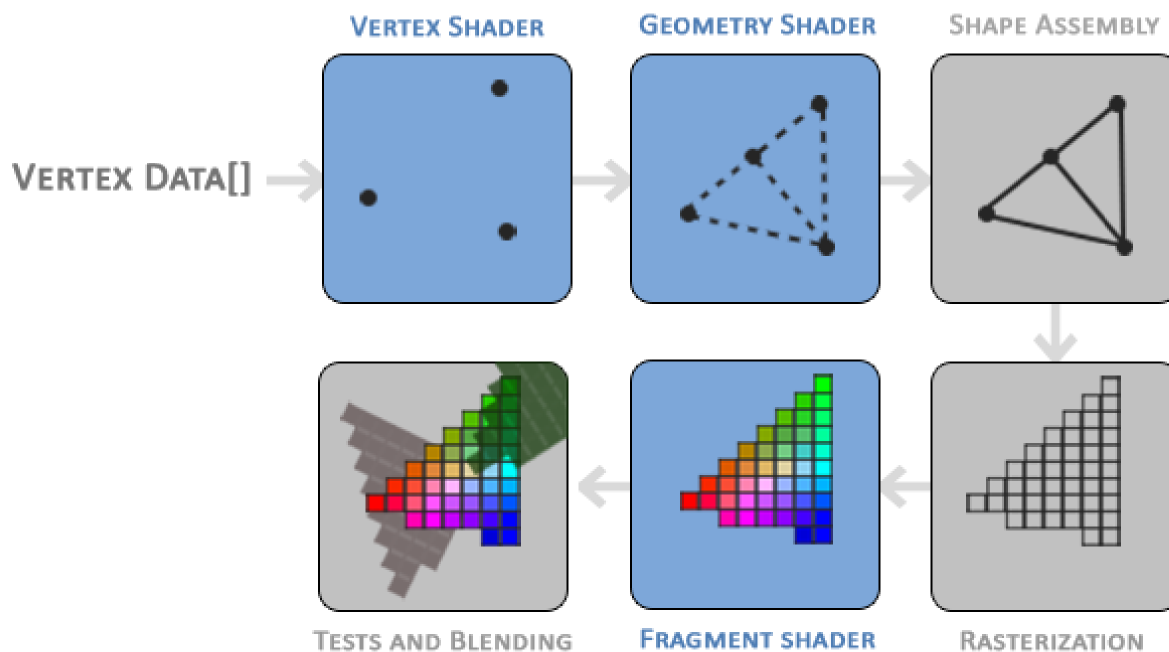

Рисунок 4 — *Приклад класичного графічного конвеєру*

Основними типами шейдерів є вершинні, геометричні та фрагментні шейдери, які є основою сучасного графічного програмування, забезпечуючи детальний контроль над процесом візуалізації.

Вершинні шейдери є першим етапом у графічному конвеєрі. Їх основна роль полягає в обробці даних вершин, які включають такі атрибути, як положення, колір, вектори нормалей і координати текстури. Частіше всього під час виконання вершинний шейдер перетворює позиції вершин із координатного простору об’єктів у простір екрану, на якому будується зображення, використовуючи серію множень матриці, зазвичай залучаючи матриці моделі, спостерігача та проекції. Ця трансформація необхідна для правильного проектування 3D об’єктів на 2D екран. Вихідні дані вершинного шейдера включають перетворені позиції вершин і потенційно додаткові дані, такі як координати текстури, які передаються на наступні етапи в конвеєрі.

Геометричні шейдери, представлені в пізніших графічних API, працюють після вершинних шейдерів і можуть обробляти цілі примітиви, такі як точки, лінії або трикутники, а не окремі вершини. Цей етап дозволяє динамічно генерувати, трансформувати або видаляти геометрію на

графічному процесорі. Наприклад, геометричні шейдери можна використовувати для створення додаткових вершин та підвищення деталізації поверхні або генерації об’ємних тіней. Можливість маніпулювати примітивами забезпечує значну гнучкість і дозволяє створювати складніші та деталізовані сцени без додаткового втручання центрального процесора.

Фрагментні шейдери, також відомі як піксельні шейдери, відповідають за визначення кольору та інших атрибутів кожного фрагмента, отриманого шляхом растеризації обробленої геометрії. Фрагмент представляє потенційний піксель у остаточному відтвореному зображенні. Фрагментні шейдери беруть інтерпольовані дані з попередніх етапів, такі як положення вершин, нормалі та координати текстури, і використовують цю інформацію для виконання операцій над фрагментами. Ці операції включають відображення текстури, коли текстури застосовуються до поверхонь, і по піксельні обчислення освітлення, які сприяють детальному затіненню та фарбуванню зображення. Фрагментні шейдери також можуть реалізовувати розширені ефекти, такі як відображення рельєфу та відображення тіні. Результатом фрагментного шейдера є кінцеве значення кольору та глибини для кожного пікселя, яке потім використовується для створення остаточного зображення.

Проаналізувавши загальні відомості можна підсумувати, що шейдери є важливим елементом сучасного графічного програмування, які дозволяють створювати динамічні та візуально привабливі зображення. Їхня здатність ефективно паралельно обробляти великі обсяги графічних даних змінила можливості цифрового рендерингу, зробивши передові графічні ефекти більш доступними для широкого спектру програм.

2.2 Аналіз сучасних технологій розробки редактора шейдерів

Редактор шейдерів — це програмний інструмент, який використовується розробниками для створення та керування шейдерами. Ці

редактори зазвичай складаються з кількох ключових компонентів, які працюють разом, щоб полегшити процес створення шейдерів.

Для редактора шейдерів була обрана комбінація наступних технологій, щоб створити надійний і ефективний інструмент для розробки шейдерів.

Мова програмування: C++ був обраний для розробки редактора шейдерів завдяки його продуктивності та підтримці низькорівневої взаємодії з системою [7]. За допомогою нього можна ефективно керувати використанням пам'яті, а також він легко інтегрується з такими графічними API, як OpenGL, що дозволяє використовувати усі можливості графічного процессора для відтворення складних шейдерів і візуальних ефектів. Загалом C++ забезпечує гнучкість і контроль, необхідні для створення високопродуктивного редактора шейдерів.

Графічний API: Це спосіб завдяки якому дві або більше комп'ютерні програми можуть комунікувати між собою. У даному випадку він служить мостом між програмними та апаратними компонентами, відповідальними за відтворення графіки на пристрої. Він надає набір функцій, протоколів та інструментів, які розробники використовують для створення та керування графічними елементами, такими як 2D-зображення, 3D-моделі, текстури, шейдери та методи візуалізації.

Для проекту редактора шейдерів було вирішено використовувати графічний API OpenGL. Це широко поширений багатоплатформний API для рендерингу 2D і 3D графіки. Універсальність, оптимізація продуктивності та широка сумісність із різними платформами роблять його популярним вибором [9] для розробки редактора шейдерів, який націлений на широкий спектр сценаріїв розробки графіки.

Фреймворк користувацького інтерфейсу: Це набір класів та інтерфейсів, які визначають елементи та поведінку віконної підсистеми інтерфейсу користувача. Інтерфейс користувача надає готові елементи, такі як кнопки, меню, вікна, діалогові вікна та віджети, які розробники можуть використовувати для проектування та налаштування зовнішнього вигляду та

функціональності своїх програм. Ці фреймворки часто включають функції для обробки подій, керування макетом і оформлення тем, що спрощує процес побудови інтуїтивно зрозумілих і зручних інтерфейсів.

Для задоволення цієї потреби був обраний Qt [4], як фреймворк інтерфейсу користувача. Qt — крос-платформовий інструментарій розробки, який широко використовується для розробки програм із графічним інтерфейсом користувача для десктопних , мобільних, вбудованих і веб-платформ. Він пропонує великий набір компонентів інтерфейсу користувача та інструментів для створення сучасних і адаптивних інтерфейсів, а також надійну підтримку рендерингу графіки.

Керування файлами шейдерів: Для обробки операцій роботи з файлами була використана комбінація вбудованої стандартної бібліотеки C++ і модулів Qt для обробки файлового введення/виведення. Стандартна бібліотека C++ надає такі функції, як ifstream і ofstream для читання з файлів і запису в них, що спрощує керування файловими операціями в логіці програми. Крім того, Qt пропонує набір модулів, таких як QFile і QIODevice, які розширюють можливості обробки файлів за допомогою таких функцій, як доступ до файлової системи, режими відкриття файлів, обробка помилок і незалежне від платформи керування шляхом до файлу. Використовуючи як стандартну бібліотеку C++, так і модулі файлового вводу-виводу Qt, було забезпечено ефективне та надійне керування файлами в редакторі шейдерів, дозволяючи користувачам легко завантажувати, зберігати та маніпулювати файлами шейдерів.

Середовище розробки: Це комплексне програмне рішення для розробки програмного забезпечення. Зазвичай, складається з редактора початкового коду, інструментів для автоматизації складання та відлагодження програм. Більшість сучасних середовищ розробки мають можливість автодоповнення коду. Середовища розробки створені для оптимізації робочого процесу, пропонуючи уніфіковану платформу, на якій розробники можуть писати, організовувати та ефективно підтримувати код.

Для розробки редактора шейдерів був обран Qt Creator. Це інтегроване середовище розробки, призначене для створення крос-платформових застосунків з використанням бібліотеки Qt. Він пропонує зручний інтерфейс із великим набором функцій, призначених для роботи з цим фреймворком, таких як Qt Designer для розробки графічного інтерфейсу користувача, розширений редактор коду, вбудований налагоджувач і підтримка контролю версій. Крім того, середовище забезпечує інтеграцію з інфраструктурою Qt, дозволяючи легко розробляти програми на основі цього фреймворку за допомогою C++ або QML (мова моделювання Qt).

3 РОЗРОБКА РЕДАКТОРА ШЕЙДЕРІВ У РЕАЛЬНОМУ ЧАСІ

3.1 Опис предметної області

Предметна область дипломної роботи охоплює перетин графічного програмування , розробки шейдерів та дизайну інтерфейсу користувача. За своєю суттю ця предметна область заглиблюється в сферу методів візуалізації OpenGL, зосереджуючись на тонкощах мов шейдерів, таких як GLSL, щоб надати розробникам можливість створювати візуально привабливі ефекти.

Ключовим моментом цієї предметної області є дослідження вершинних та геометричних шейдерів, підтримка яких часто відсутня у існуючих редакторах. Надаючи можливість редагування цих типів шейдерів, цей застосунок вирішує важливий недолік в інструментах, доступних для розробників OpenGL.

Рішення розробити програму редактора шейдерів випливає з кількох вагомих причин у сфері графічного програмування. По-перше, існуючі редактори шейдерів часто обслуговують конкретні програмні середовища, такі як Unity або Godot, обмежуючи їхню корисність для розробників, які працюють безпосередньо з OpenGL. Створюючи редактор, який зосереджується на сумісності, буде розроблен універсальний інструмент, який легко інтегрується в широкий спектр застосунків на основі OpenGL, дозволяючи їм використовувати весь потенціал можливостей цього графічного API. Дозволяючи розробникам створювати, змінювати та оптимізувати вершинні та геометричні шейдери безпосередньо в редакторі, не тільки підвищується ефективність робочого процесу, але й вирішує проблему відсутності їх підтримки у багатьох існуючих інструментах.

Підсумовуючи, предметна область охоплює всебічне дослідження графіки в реальному часі, розробки шейдерів і дизайну графічного інтерфейсу. Завдяки підтримці вершинних та геометричних шейдерів та

незалежності від інших платформ, що працюють на основі OpenGL, цей застосунок має на меті надати розробникам інструменти, необхідні для продуктивної роботи у галузі комп'ютерної графіки.

3.2 Архітектура системи

3.2.1 Шаблон проектування

Вибір правильного шаблону проектування програмного забезпечення має вирішальне значення для розробки надійної та зручної програми. Для редакторі шейдерів був обран шаблон Model-View-Controller (MVC) завдяки чіткому розподілу завдань і масштабованості [6]. На Рисунок 5 — *UML діаграма архітектури застоснка* зображено взаємодію цих трьох компонентів у системі.

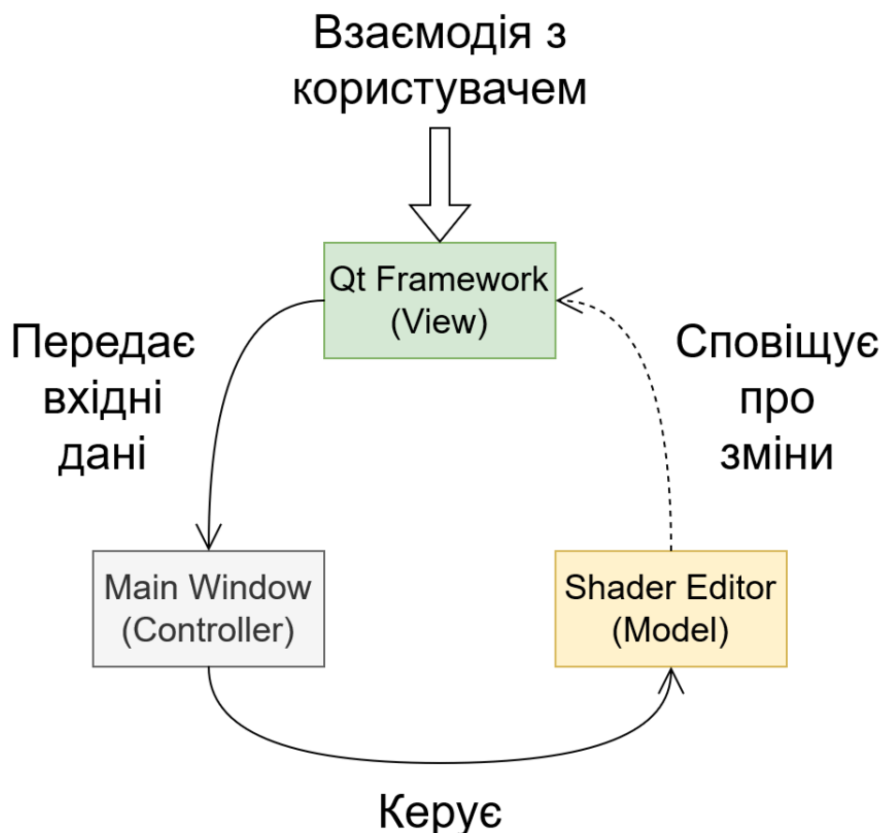

Рисунок 5 — *UML діаграма архітектури застоснка*

Компонент модель представляє дані та бізнес-логіку програми. У моєму випадку модель охоплює все, що пов'язано з шейдерами — їх завантаження, обробка, компіляція та виконання. Це розділення дозволяє легко модифікувати та розширювати функції шейдерів, не впливаючи на інші частини програми.

Компонент представлення відповідає за відображення користувачу графічного інтерфейсу. Для цього використовується Qt Framework, щоб створити візуально привабливий і зручний інтерфейс для редагування шейдерів. Віджети та інструменти Qt спрощують процес розробки та забезпечують узгоджений вигляд і відчуття на різних платформах.

Контролер діє як посередник між моделлю та представленням. Головне вікно виконує роль цього компонента, обробляючи введені користувачем дані, оновлюючи модель на основі дій користувача та оновлюючи представлення для відображення змін у даних або стані програми. Такий поділ завдань спрощує тестування, налагодження та обслуговування системи.

Завдяки застосуванню шаблону MVC редактор шейдерів отримує гнучкість, масштабованість і зручність обслуговування. Він може легко адаптуватись до майбутніх оновлень, підтримувати геометричні та вершинні типи шейдерів і інтегруватися з різними програмами на основі OpenGL, що робить його універсальним інструментом для розробки шейдерів.

3.2.2 Компонент модель

Компонент модель складається з багатьох важливих елементів, які відповідають за роботу редактора. Одним з основних є клас SceneItem, який представляє кожен елемент існуючий в 3D-середовищі. Для цього ці обє'кти повинен успадковувати цей клас, забезпечуючи стандартизований підхід до керування сценою.

У своїй основі клас SceneItem інкапсулює основні властивості, важливі для підтримки структури та організації сцени. Щоб класифікувати та розрізняти різні типи елементів сцени, використовується клас нумератор

SceneObjectType. Цей перелік допомагає визначити призначення кожного елемента в межах сцени, наприклад геометричної сітки, камери, джерела світла тощо.

Інкапсулюючи ці основні функції в класі SceneItem, створюється основа для розробки складних 3D-сцен і керування ними. Такий структурований підхід не тільки сприяє багаторазовому використанню коду, але й сприяє плавній інтеграції нових функцій і вдосконалень. На Лістинг 1 зображення структура цього класу.

Лістинг *1 клас "SceneItem"*

```
class SceneItem {
protected:
    std::vector<SceneItem*> m_childItems;
    QVariantList m_itemData;
    SceneItem* m_parentItem;
    SceneObjectType m_objType;
public:
    SceneItem(QVariantList data, SceneObjectType type,
SceneItem *parentItem = nullptr);
    SceneItem* findChildByType(SceneObjectType typeToFind);
    void appendChild(SceneItem *child);
    void insertChild(int pos, SceneItem *child);
    void deleteAtIndex(int index);
    void clearChildren();
    SceneItem* getChildByIndex(int row);
    int childCount() const;
    int columnCount() const;
    QVariant data(int column) const;
    int row() const;
    SceneItem *getParentItem();
    void setParentItem(SceneItem *parentItem = nullptr);
    virtual QList<QTreeWidgetItem*> getPropertyList();
    virtual bool setNewPropert(int row, QString newValue);
    QString getName();
    void setName(QString data);
    SceneObjectType getObjectType();
    std::vector<SceneItem*> getChildrenOfParent();
    static bool stringToVec3(QString stringValue, glm::vec3
&vectorValue);
};
```

findChildByType() – метод який повертає перший дочірній об’єкт наданого типу.

appendChild(), **insertChild()**, **deleteAtIndex()** та **clearChildren()** – методи керування дочірніми елементами, які дозволяють додавати, вставляти видаляти та очищати список об'єктів.

getChildByIndex() – повертає дочірній обє’кт за наданим індексом.

childCount() – повертає кількість дочірніх обє’ктів.

columnCount() – повертає кількість стовбців у дереві ієрархії.

data() – повертає значення у дереві ієрархії для наданого стовбця.

row() – повертає індекс поточного об’єкта.

getParentItem() – повертає батьківський об’єкт.

setParentItem() – встановлює новий батьківський об’єкт.

getPropertyList() – метод надає список властивостей об’єкта сцени.

setNewPropert() – метод встановлення нових значень властивостей в залежності від переданих аргументів.

getName() – метод повертає ім’я об’єкта.

setName() – метод встановлює нове ім’я об’єкта.

getObjectType() – метод який повертає тип об’єкту сцени.

getChildrenOfParent() – повертає список дочірніх елементв для батькіського об’єкта поточного елемента.

stringToVec3() – статичний метод, який перетворює текстовий рядок на вектор.

Далі будуть розглянуті конкретні компоненти, відповідальні за відтворення та візуалізацію 3D-сцени в програмі.

Починаючи з класу SceneViewPortObject, який є критично важливим компонентом, відповідальним за візуалізацію та керування вікном перегляду 3D сцени в програмі. Він успадковує вбудовані в бібліотеку Qt класи, для використання можливостей візуалізації та функціональності OpenGL.

SceneViewPortObject реалізує ряд методів, необхідних для ініціалізації OpenGL, оновлень зображення і обробки зміни розміру вікна. Крім того, він має методи, пов'язані з подіями миші, призначені для обробки взаємодії обертання камери. Хоча він не повертає камеру безпосередньо, для цього використовується об'єкт камери.

Клас надає методи для встановлення нових уніформ і макетів вершин, надаючи можливість динамічно редагувати та налаштування властивостей шейдерів для відтворення елементів сцени.

Таким чином, клас SceneViewPortObject пов'язує ієрархією і модель сцени за допомогою функціоналу приведеного у Лістинг 2 *клас "SceneViewPortObject"*.

Лістинг *2 клас "SceneViewPortObject"*

```
     class SceneViewPortObject : public QOpenGLWidget, public
QOpenGLFunctions_4_5_Core, public SceneItem {
         Q_OBJECT
     public:
         SceneViewPortObject(QWidget *parent = nullptr);
         ~SceneViewPortObject();
         MeshObject* addGeometry(int type);
         MeshObject* addCustomGeometry(QString filepath);
         void setShaderLayouts(MeshObject* mesh,
std::vector<LayoutConfig> newLayouts);
         void setShaderUnifroms(ShaderObject* shader,
std::vector<UniformConfig> newUniforms);
         glm::mat4 getViewProjection();
         glm::vec3 getCameraPosition();
         glm::vec2 getResolution();
     protected:
         void initializeGL() override;
         void paintGL() override;
         void resizeGL(int w, int h) override;
         void mousePressEvent(QMouseEvent* event) override;
         void mouseMoveEvent(QMouseEvent* event) override;
         void mouseReleaseEvent(QMouseEvent* event) override;
         void wheelEvent(QWheelEvent* event) override;
         CameraObject m_sceneCamera;
         QTimer* m_timer;
     };
```

addGeometry() – додає до сцени один з трьох геометричних примітивів залежно від отриманих аргументів, а саме куб, площина або сфера.

addCustomGeometry() – додає до сцени геометричну модель за отриманим шляхом до файлу.

setShaderLayouts() – отримує та встановлює новий макет геометричних вершин.

setShaderUnifroms() – отримує та встановлює новий набір уніформ шейдерів.

getViewProjection() – повертає матрицю перетворень проекції.

getCameraPosition() – повертає координати розташування камери.

getResolution() – повертає роздільну здатність вікна візуалізації.

initializeGL() – виконує ініціалізацію OpenGL.

paintGL() – оновлює зображеня у вікні візуалізації.

resizeGL() – оновлює налаштування візуалізації в залежності від роздільної здатності.

mousePressEvent() – надає камері інформацію про розташування курсора під час початку обертання.

mouseMoveEvent() – надає камері інформацію про розташування курсора під час обертання.

mouseReleaseEvent() – надає камері інформацію про розташування курсора під час закінчення обертання.

wheelEvent() – надає камері сигнал про потребу приблизити або віддалити своє розташування.

Клас ShaderObject є відповідальним за керування шейдерами. Цей клас імплиментує основні функції, пов’язані з обробкою шейдерів.

Однією з основних функцій класу, є зберігання текстових рядків шейдера та шляхів до файлів. Це забезпечує легкий доступ до вихідного коду шейдерів і полегшує компіляцію шейдерів та керування ними.

Також клас зберігає список уніформ, які повинні бути передані шейдеру під час візуалізації. Ці уніформи можуть містити такі змінні, як матриці перетворень, параметри освітлення та будь-які інші дані, необхідні для правильного відтворення об’єктів.

Крім того, клас ShaderObject надає метод для рекомпіляції шейдерів. Ця функція необхідна для динамічного оновлення та модифікації шейдерів під час виконання, дозволяючи змінювати поведінку шейдерів у реальному часі без перезапуску програми.

Таким чином, клас ShaderObject відіграє важливу роль в управлінні шейдерами, за допомогою функціоналу преведеного у Лістинг 3.

Лістинг *3 клас “SceneViewPortObject”*

```
     class ShaderObject : public QOpenGLFunctions_4_5_Core,
public SceneItem {
     private:
         GLuint m_id;
         std::string m_vertexCode;
         std::string m_fragmentCode;
         std::string m_geometryCode;
         std::vector<UniformConfig> m_uniformVector;
         std::string m_vertexFilePath;
         std::string m_fragmentFilePath;
         std::string m_geometryFilePath;
         bool isCustomPath_v;
         bool isCustomPath_f;
         bool isCustomPath_g;
         bool inGeometryMode;
         std::string readShaderFile(const char* filename);
         void createShaderProgram();
     public:
         ShaderObject(std::string vertexFilePath, std::string
fragmentFilePath);
         ShaderObject(std::string vertexFilePath, std::string
geometryFilePath, std::string fragmentFilePath);
         ~ShaderObject();
         ShaderObject(const ShaderObject& copy);
         QList<QTreeWidgetItem*> getPropertyList() override;
         void useShaderProgram();
         void deleteShaderProgram();
         void addUniform(UniformConfig uniform);
```

```
        void setUniforms(std::vector<UniformConfig> newVector);
        void activateUniforms(SceneItem* scene);
        QStringList uniformToString();
        void recompile(std::string newVertexCode, std::string
newFragmentCode);
        void recompile(std::string newVertexCode, std::string
newGeometryCode, std::string newFragmentCode);
        void bindFiles(int type, QString filePath);
        QString getVertexShaderString();
        QString getFragmentShaderString();
        QString getGeometryShaderString();
        bool getMode();
    };
```

readShaderFile() – приватний метод, який зчитує рядки коду шейдера з файлу, за отриманим шляхом.

createShaderProgram() – приватний метод, який створює шейдерну програму для роботи з OpenGL.

getPropertyList() – метод надає список властивостей об’єкта сцени.

useShaderProgram() – надає сигнал OpenGL з інструкцією, яку шейдерну програму потрібно використовувати для відображення геометрії.

deleteShaderProgram() – видаляє шейдерну програму з оперативної пам’яті.

addUniform() – додає одну уніформу до списку.

setUniforms() – встановлює новий список уніформ.

activateUniforms() – метод передає константи, за допомогою уніформ до шейдерів.

uniformToString() – виводить список уніформ у вигляді рядка.

recompile() – рекомпілює шейдери у випадку оновлення їх коду.

bindFiles() – прив’язує файли з кодом шейдерів до поточного об’єкта.

getVertexShaderString() – повертає код шейдера вершин у вигляді рядка.

getFragmentShaderString() – повертає код шейдера фрагментів у вигляді рядка.

getGeometryShaderString() – повертає код шейдера геометрії у вигляді рядка.

getMode() – повертає булєве значення чи використовує поточний об’єкт шейдер геометрії.

Для створення текстур OpenGL, і зберігання відповідної інформації про них, був розроблен клас TextureObject. Він інкапсулює важливу інформацію, пов’язану з зображеннями, включаючи розміри (висота та ширина), кількість каналів кольору, слот текстури для зв’язування з OpenGL, шлях до файлу зображення текстури та необроблені дані зображення, що зберігаються у вигляді байтів.

Клас зберігає слот текстури OpenGL, який використовується для зв’язування текстури під час операцій візуалізації. Таке керування слотами гарантує можливість використання великої кількості текстур під час їх візуалізацію на 3D-об’єктах у межах сцени.

Таким чином, цей клас служить інструментом керування текстурами в системі рендерингу. Для встановлення нового зображення та взаємодії з іншими системами, як відображення властивостей або отримання ідентифікаційного значення в системі OpenGL, використовується наступний функціонал преведений у Лістинг 4 *клас “TextureObject”*.

Лістинг *4 клас “TextureObject”*

```
class TextureObject : public QOpenGLFunctions_4_5_Core,
public SceneItem {
private:
    GLuint m_id;
    int m_width;
    int m_height;
    int m_colChanelCount;
    int m_mySlot;
    unsigned char* m_imgInBytes;
    std::string m_imgPath;
    void generateTexture();
public:
    TextureObject(int slotNum = 0);
    ~TextureObject();
```

```
    QList<QTreeWidgetItem*> getPropertyList() override;
    void assignNewTexture(QString filePath);
    GLuint getID();
};
```

getPropertyList() – метод надає список властивостей об’єкта сцени.

assignNewTexture() – призначає нове зображення, яке буде використовуватися як текстура.

getID() – повертає ідентифікаційне значення з серидовища OpenGL.

Навігації та перегляд сцени виконується за допомогою розробленого класу CameraObject. Цей клас зберігає важливу інформацію про камеру, включаючи її положення, обертання, обмеження максимального та мінімального радіусу для переміщення, а також матрицю перетворень проекції, необхідну для візуалізації.

Однією з ключових функцій класу є специфікація максимальних і мінімальних обмежень радіусу. Ці обмеження визначають допустимий діапазон відстані між камерою та об’єктами сцени, запобігаючи переміщенню камери надто близько чи надто далеко.

Клас камери полегшує навігацію в межах 3D-сцени, дозволяючи користувачам переміщуватися навколо об’єктів і дивитися на них з різних точок зору. Реалізація цього функціоналу приведена у Лістинг 5.

Лістинг *5 клас “CameraObject”*

```
class CameraObject : public SceneItem {
private:
    const float m_maxRadius = 50.0f;
    const float m_minRadius = 1.0f;
    glm::mat4 m_view;
    glm::mat4 m_projection;
    glm::vec3 m_lookAt;
    glm::vec3 m_position;
    int m_anchorMousePosX = 0;
    int m_anchorMousePosY = 0;
    float m_angleHorizontal = 0.0f;
    float m_angleVertical = 0.0f;
    float m_anchorAngleHorizontal = 0.0f;
```

```
        float m_anchorAngleVertical = 0.0f;
        float m_radius = 5.0f;
        void updateCameraPosition();
    public:
        CameraObject();
        QList<QTreeWidgetItem*> getPropertyList() override;
        bool setNewPropert(int row, QString newValue) override;
        void setProjectionAspectRation(float width, float
height);
        void setNewRotation(int x, int y);
        void setAnchor3D();
        void setAnchor2D(int mousePosX, int mousePosY);
        glm::mat4 getViewProjection();
        glm::vec3 getPosition();
        void zoomIn(float zoomValue);
        void zoomOut(float zoomValue);
    };
```

updateCameraPosition() – приватний метод для оновлення позиції камери.

getPropertyList() – метод надає список властивостей об’єкта сцени.

setNewPropert() – метод встановлення нових значень властивостей в залежності від переданих аргументів.

setProjectionAspectRation() – встановлює роздільну здатність для матриці перетворень проекції.

setNewRotation() – встановлює нове розташування камери у сферичних координатах, використовуючи отримані координати курсора.

setAnchor3D() – встновлює нові кути обертання у сферичних координатах.

setAnchor2D() – встновлює нові кординати курсора від якого буде обраховуватся обертання..

getViewProjection() – поевертає матрицю перетворень проекції.

getPosition() – повертає розташування камери у просторі.

zoomIn() – зменьшує радіус сферичних координат.

zoomOut() – збільшує радіус сферичних координат.

Також важливим елементом для роботи з шейдерами є джерело світла. За допомогою класу LightSourceObject був реалізован цей інструмент. Він служить контейнером призначеним для інкапсуляції даних, пов'язаних із джерелами світла, таких як його положення в 3D-просторі та властивості кольору.

Таким чином, спрощується інтеграція різних типів джерел світла, таких як точкове світло, спрямоване світло та прожектори. Отримання та збереження даних про джерело світла виконується за допомогою функціоналу преведеного у Лістинг 6.

Лістинг *6 клас "LightSourceObject"*

```
class LightSourceObject : public SceneItem {
private:
    glm::vec3 m_position;
    glm::vec3 m_color;
public:
    LightSourceObject();
    QList<QTreeWidgetItem*> getPropertyList() override;
    bool setNewPropert(int row, QString newValue) override;
    glm::vec3 getPosition();
    glm::vec3 getColor();
};
```

getPropertyList() – метод надає список властивостей об'єкта сцени.

setNewPropert() – метод встановлення нових значень властивостей в залежності від переданих аргументів.

getPosition() – повертає розташування джерела світла у просторі.

getColor() – повертає колір джерела світла.

Фундаментальним компонентом у системі візуалізації, призначеним для керування та відтворення 3D об'єктів, є клас MeshObject. Він служить контейнером для атрибутів і функцій, пов'язаних із рендерингом сітки, включаючи буфер вершин (VBO), об'єкт масиву вершин (VAO), об'єкт буфера індексів (EBO), макети вершин, зберігає шейдери, перетворення об'єкта (положення, обертання, масштаб) і метод візуалізації.

Клас зберігає основні структури даних OpenGL, такі як VBO, VAO та EBO. Вони визначають геометрію, атрибути вершин та порядок у якому їх потрібно візуалізувати, забезпечуючи ефективне відтворення та маніпулювання вершинами під час процесу відтворення.

Метод візуалізації об’єкта, який використовує збережені шейдери та структури даних OpenGL, послідовно застосовує шейдери для візуалізації об’єкта, використовуючи макети вершин і уніформи шейдерів зображуючи геометричну сітку у вікні попереднього перегляду.

Будучи важливим елементом усієї системи реалізація цього класу преведено у Лістинг 7 *клас “MeshObject”*.

Лістинг *7 клас “MeshObject”*

```
class MeshObject : public QOpenGLFunctions_4_5_Core, public SceneItem {
private:
    GLuint VAO;
    GLuint VBO;
    GLuint EBO;
    std::vector<LayoutConfig> m_layoutVector;
    int m_segmentLength;
    bool doDepthTest;
    glm::vec3 m_position;
    glm::vec3 m_rotation;
    glm::vec3 m_scale;
    glm::mat4 m_model;
    GeometryClass m_geometry;
    void generateBuffers();
public:
    MeshObject(int type = 0);
    MeshObject(QString filepath);
    ~MeshObject();
    MeshObject(const MeshObject &copyFrom);
    QList<QTreeWidgetItem*> getPropertyList() override;
    bool setNewPropert(int row, QString newValue) override;
    QStringList layoutToString();
    void setNewLayouts(std::vector<LayoutConfig> newVector);
    void disableLayouts();
    void deleteBuffers();
    void draw();
    void rotate(glm::vec3 rotation);
```

```
        void setPosition(glm::vec3 pos);
        void translate(glm::vec3 pos);
        void setScale(glm::vec3 scale);
        glm::mat4 getModel();
    };
```

generateBuffers() – приватний метод, який генерує масив вершин, буфери вершин та індексів для роботи з OpenGL.

getPropertyList() – метод надає список властивостей об’єкта сцени.

setNewPropert() – метод встановлення нових значень властивостей в залежності від переданих аргументів.

layoutToString() – повертає макет вершин у вигляді рядка.

setNewLayouts() – встановлює новий макет вершин геометрії.

disableLayouts() – відключає елементи макету у випадку зупинки їх використання.

deleteBuffers() – видаляє масив вершин, буфери вершин та індексів для роботи з оперативної пам’яті.

draw() – метод для відображення геометрії, поточного об’єкта, на екрані.

rotate() – обертає геометрію у матриці перетворень.

setPosition() – встановлює нове розташування геометрії у матриці перетворень.

translate() – переміщує геометрію у матриці перетворень.

setScale() – змінює масштаб геометрії.

getModel() – повертає матрицю перетворень.

Також у роботі системи важливим елементом є службові класи які виконують допоміжні функції, забезпечуючи структуроване зберігання данних про геометрію примітиві, макет вершин або уніформи шейдерів. Їх функціональні можливості підвищують ефективність робочого процесу та сприяють повторному використанню коду.

Для зберігання інформації про макети шейдерів у програмі редактора використовується клас LayoutConfig. Він служить контейнером для визначення структури макета шейдерів, включаючи такі атрибути, як положення вершин, нормалі, координати текстури та колір вершини. Інкапсулюючи інформацію про макет, клас полегшує організацію та конфігурацію геометрії 3D об'єктів у сцені.

Подібним чином клас UniformConfig відповідає за зберігання інформації про уніформу шейдерів. Це включає такі змінні, як матриці трансформації, параметри освітлення, змінну часу, роздільну здатність вікна візуалізації, параметри камери та текстури, які можуть бути необхідні шейдерам під час візуалізації. Таким чином створюється можливість динамічно налаштовувати властивості шейдерів.

Для роботи з геометрією використовується абстрактний клас GeometryClass. Він зберігає дані про вершини і дані індексів, забезпечуючи стандартизовану структуру для визначення геометричних фігур у редакторі шейдерів.

Дочірні класи, такі як CubeGeometry, PlaneGeometry та SphereGeometry, підтримують створення загальних геометричних форм, як куби, площини та сфери, відповідно. Ці підкласи надають попередньо визначені структури даних, пристосовані до кожної геометричної фігури.

Також був створен дочірній клас CustomGeometry, який підтримує завантаження геометрії із зовнішніх файлів формату obj. Це дозволяє користувачам імпортувати власні 3D-моделі та інтегрувати їх у середовище редактора шейдерів, підвищуючи універсальність застосунку.

3.2.3 Компонент представлення

Наведений функціонал класу SceneItem є невід'ємною частиною класу SceneHierarchy, який є ключовим компонентом архітектури застосунка, призначеним для візуалізації моделі сцени в інтерфейсі користувача. Для

цього він успадковує QAbstractItemModel, фреймворку Qt, використовуючи його функції для представлення даних і керування ними.

Крім того, SceneHierarchy реалізує функціональність для операцій перетягування, забезпечуючи інтуїтивну організацію елементів сцени в ієрархії.

Сам клас зберігає посилання на кореневий об’єкт типу SceneItem, який служить точкою входу в ієрархічну структуру. Це посилання дозволяє SceneHierarchy здійснювати навігацію та керувати всім графом сцени, забезпечуючи послідовність і узгодженість у представленні моделі.

Загалом, клас SceneHierarchy відіграє важливу роль у візуалізації даних в інтерфейсі користувача. Реалізація методів, приведена у Лістинг 8 *клас “SceneHierarchy”*, робить його важливою частиною архітектури програми.

Лістинг *8 клас “SceneHierarchy”*

```
class SceneHierarchy : public QabstractItemModel {
private:
    SceneItem* m_rootItem;
    static constexpr char s_treeNodeMimeType[] = 
"application/x-sceneitem";
public:
    SceneHierarchy(QObject *parent);
    QVariant data(const QModelIndex &index, int role) const 
override;
    QModelIndex index(int row, int column, const QModelIndex 
&parent = {}) const override;
    QModelIndex parent(const QModelIndex &index) const 
override;
    int rowCount(const QModelIndex &parent = {}) const 
override;
    int columnCount(const QModelIndex &parent = {}) const 
override;
    QVariant headerData(int section, Qt::Orientation 
orientation, int role = Qt::DisplayRole) const override;
    void appendChild(SceneItem* child);
    void appendRow(SceneItem* item, const QModelIndex 
&parent = QModelIndex());
    void removeRow(const QModelIndex &parent = 
QModelIndex());
    Qt::ItemFlags flags(const QModelIndex &index) const 
Q_DECL_OVERRIDE;
```

```
        Qt::DropActions supportedDropActions() const
Q_DECL_OVERRIDE;
        Qt::DropActions supportedDragActions() const
Q_DECL_OVERRIDE;
        QStringList mimeTypes() const Q_DECL_OVERRIDE;
        QMimeData *mimeData(const QModelIndexList &indexes)
const Q_DECL_OVERRIDE;
        bool dropMimeData(const QMimeData* data, Qt::DropAction
action, int row, int column, const QModelIndex & parent)
override;
    };
```

data() – віртуальний метод з класу QabstractItemModel, який повертає інформацію об’єкта, яку потрібно візуалізувати.

index() – віртуальний метод з класу QabstractItemModel, який створює індекс для обє’кта дерева.

parent() – віртуальний метод з класу QabstractItemModel, який повертає батьківський обє’кт для наданого елемента сцени.

rowCount() – віртуальний метод з класу QabstractItemModel, який повертає кількість дочірніх об’єктів для наданого елемента сцени.

columnCount() – віртуальний метод з класу QabstractItemModel, який повертає кількість столбців, які потрібно відобразити.

headerData() – віртуальний метод з класу QabstractItemModel, який повертає інформацію кореневого об’єкта, яку потрібно візуалізувати.

appendChild() – додає елемент як дочірній, для кореневого об’єкта.

appendRow() – додає новий об’єкт за індексом.

removeRow() – видаляє об’єкт за індексом.

flags() – повертає флаги об’єкта за наданим індексом.

supportedDropActions() – повертає перелік дій які підтримуються під час переміщення.

supportedDragActions() – повертає перелік дій які підтримуються під час завершеня переміщення.

mimeTypes() – повертає константу за якою можна імітувати об’єкт.

mimeData() – надає інформацію за якою можна імітувати за наданим індексом.

dropMimeData() – використовє інформацію для імітації для того щоб переместити об'єкт всередені іїрархії.

3.2.4 Компонент контролер

Контролером служить клас MainWindow в архітектурі програми, відповідальним за керування різними функціями та взаємодіями, пов'язаними з редактором шейдерів. Цей клас обробляє події і операцій, важливі для маніпулювання сценою, редагування шейдерів, керування текстурами та взаємодії з інтерфейсом користувача.

Одним із ключових обов'язків класу є обробка подій, пов'язаних із додаванням і видаленням об'єктів у сцені. Це включає надання користувачам механізмів для додавання нових об'єктів до середовища сцени та визначення їхніх властивостей. Крім того, клас полегшує взаємодію користувача для показу контекстних меню, забезпечуючи інтуїтивно зрозумілі та ефективні робочі процеси редагування сцени.

Клас головного вікна керує введенням користувача, пов'язаним із редагуванням шейдерів. Він надає користувачам можливість безпосередньо вводити код шейдера через спеціальний редактор коду. Потім клас застосовує цей код до відповідних шейдерів, полегшуючи оновлення та модифікації поведінки та зовнішнього вигляду шейдерів у реальному часі.

Окрім обробки введеного коду, підтримується відкриття файлів шейдерів і призначення коду з цих файлів шейдерам у межах сцени. Ця функція спрощує процес імпорту та інтеграції попередньо існуючого коду шейдерів у редактор, підвищуючи ефективність робочого процесу та гнучкість для розробки шейдерів. Таким самим чином проходить імпорт зображень, для того щоб використовувати їх як текстури у редакторі.

Крім того, надаються функції для редагування уніформ та макетів вершин, за допомогою діалогового вікна.

Загалом, клас служить комплексним контролером, надаючи велику кількість функцій, необхідних для редагування 3D-сцени, маніпулювання шейдерами, керування текстурами та взаємодії з інтерфейсом користувача, за допомогою методів наведених у Лістинг 9.

Лістинг *9 клас "MainWindow"*

```
class MainWindow : public QmainWindow {
    Q_OBJECT
public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
private slots:
    void onCustomContextMenu(const QPoint &point);
    void onSelectedTreeItem(const QModelIndex & index);
    void onPropertyWidgetDoubleClick(QTreeWidgetItem *item,
int column);
    void onEditorLooseFocus();
    void compareVertexText();
    void compareFragmentText();
    void compareGeometryText();
    void addGeometry(int type);
    void deleteItem();
    void addShader(int type);
    void bindFileToShader();
    void addTexture();
    void assignNewTexture();
    void addLightSource();
    void recompileShader();
    void openLayoutDialog();
    void openUniformDialog();
private:
    Ui::MainWindow* ui;
    SceneHierarchy* model;
    ShaderObject* currentShader;
    QShortcut* saveShaderVertex;
    QShortcut* saveShaderFragment;
    QShortcut* saveShaderGeometry;
    QMenu* sceneContextMenu;
    QMenu* geometryContextMenu;
    QMenu* shaderContextMenu;
    QMenu* textureContextMenu;
    QPoint lastContextMenuClick;
    QString propertyText;
```

```
    QString lastOpenedPath;
    void setUpContextMenu();
    void setUpConnections();
};
```

onCustomContextMenu() – метод оброблює подію відкриття контекстного меню в залежності від того, для якого об’єкта це меню було відкрито.

onSelectedTreeItem() – метод обробляє подію вибору об’єкту у навігаторі ієрархії сцени.

onPropertyWidgetDoubleClick() – метод оброблює подію двойного кліку для властивостей, та початку їх редагування.

onEditorLooseFocus() – метод оброблює подію втрати фокусу для властивостей, та закінчення редагування.

compareVertexText() – метод порівнює текст коду шейдера вершин у увікні вводу з текстом у файлі.

compareFragmentText() – метод порівнює текст коду шейдера фрагментів у увікні вводу з текстом у файлі.

compareGeometryText() – метод порівнює текст коду шейдера геометрії у увікні вводу з текстом у файлі.

addGeometry() – метод додає нову геометрію до редактора.

deleteItem() – метод видаляє об’єкт сцени.

addShader() – метод додає новий шейдер до редактора.

bindFileToShader() – прив’язує файл з кодом шейдера до об’єкта сцени.

addTexture() – метод додає нову текстуру до редактора.

assignNewTexture() – прив’язує файл з кодом шейдера до об’єкта сцени.

addLightSource() – метод додає нове джерело світла до редактора.

recompileShader() – метод рекомпілю шейдер який зараз редагується.

openLayoutDialog() – метод відкриває діалогове вікно для налаштування макету вершин шейдера.

openUniformDialog() – метод відкриває діалогове вікно для налаштування уніформ шейдера.

setUpContextMenu() – приватний метод, який готує контекстні меню під час ініціалізації.

setUpConnections() – приватний метод, який зв'язує події застосунку та методи контроллера, які обробляють ці події.

3.3 Функціональні вимоги до системи

3.3.1 Загальні вимоги

1. Редактор шейдерів має бути сумісний із основними операційними системами, такими як Windows, macOS і Linux, щоб забезпечити доступність і зручність використання для широкої бази користувачів.
2. Підтримка мов шейдерів, таких як GLSL, дозволяє розробникам ефективно писати, редагувати та компілювати шейдери в редакторі.
   - **a.** Можливість створювати, редагувати та компілювати вершинні шейдери.
   - **b.** Можливість створювати, редагувати та компілювати фрагментні шейдер.
   - **c.** Можливість створювати, редагувати та компілювати геометричні шейдерів, чого часто немає в редакторах шейдерів.
3. Дизайн інтерфейсу користувача повинен бути інтуїтивно зрозумілим і зручним для користувача, містити інструменти для налаштування параметрів шейдерів, попереднього перегляду в реальному часі, налагодження та візуального зворотного зв'язку.
4. Редагування уніформ шейдерів та вхідних даних таких, як джерела світла та текстури.

5. Можливість експортувати та імпортувати шейдери з файлів для спільного використання та співпраці.
6. Сумісність із стандартними робочими процесами, форматами зображень для роботи з текстурами, форматами 3D моделей для їх завантаження (наприклад .obj) і інструментами розробки шейдерів забезпечує взаємодію, безперервність робочого процесу та легкість інтеграції з існуючими конвеєрами та екосистемами.
7. Рендеринг шейдерів у реальному часі, щоб миттєво візуалізувати зміни, коли вони вносяться в редакторі.
8. Підсвічування синтаксису для коду шейдера, щоб покращити читабельність і полегшити редагування.
9. Функції виділення помилок і налагодження, щоб допомогти користувачам виявити та виправити проблеми в коді шейдера.

3.3.2 Бібліотека підпрограм (класів)

У додаток до основних технологій, в редактор шейдерів були інтегровані три бібліотеки: GLM, stb_image і QCodeEditor.

GLM — це математична бібліотека C++ [10], яка забезпечує функції для векторів, матриць, перетворень і геометричних операцій, які зазвичай використовуються в програмах OpenGL. Сумісність GLM із шейдерами GLSL робить його важливим доповненням до системи, для обробки математичних обчислень і перетворень.

stb_image — це легка бібліотека для завантаження файлів зображень, таких як PNG і JPEG, у текстури OpenGL. Простота, швидкість і легкість інтеграції зробили його зручним вибором для виконання завдань із завантаження текстур і маніпулювання ними.

QCodeEditor – це бібліотека, яка надає налаштовуємий віджет редактора коду з підсвічуванням синтаксису, доповненням коду та іншими важливими функціями для редагування коду. Заснований на CMake, який

спрощує процес інтеграції бібліотеки в систему збірки, забезпечуючи повну сумісність і легке налаштування.

3.4 Функціонал системи

3.4.1 Функціонал представлення та контроллера

Графічний інтерфейс є однією з головних частин будь-якого застосунку, тому був створен інтуїтивний інтерфейс за допомогою інструментів розробки моделі представлення. На

Рисунок 6 — *Графічний інтерфейс розробляємого редактора шейдерів* наведено графічне представлення редактора шейдерів.

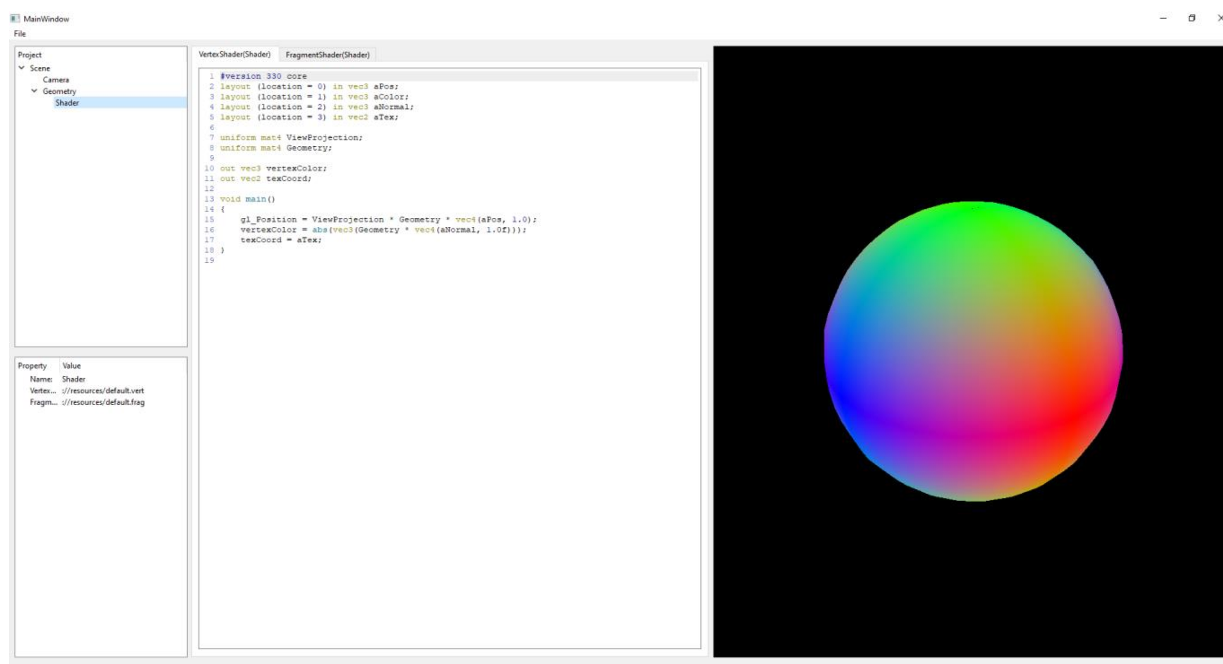

Рисунок 6 — *Графічний інтерфейс розробляємого редактора шейдерів*

Інтерфейс надає можливість додавати, видаляти та редагувати різні об’єкти 3D сцени, такі як геометрія, шейдери, текстури тощо. На Рисунок 7 — *Use Case діаграма редактора шейдерів* зображена діаграма прецедентів

редактора шейдерів, за допомогою якої можна побачити повний функціонал редактора.

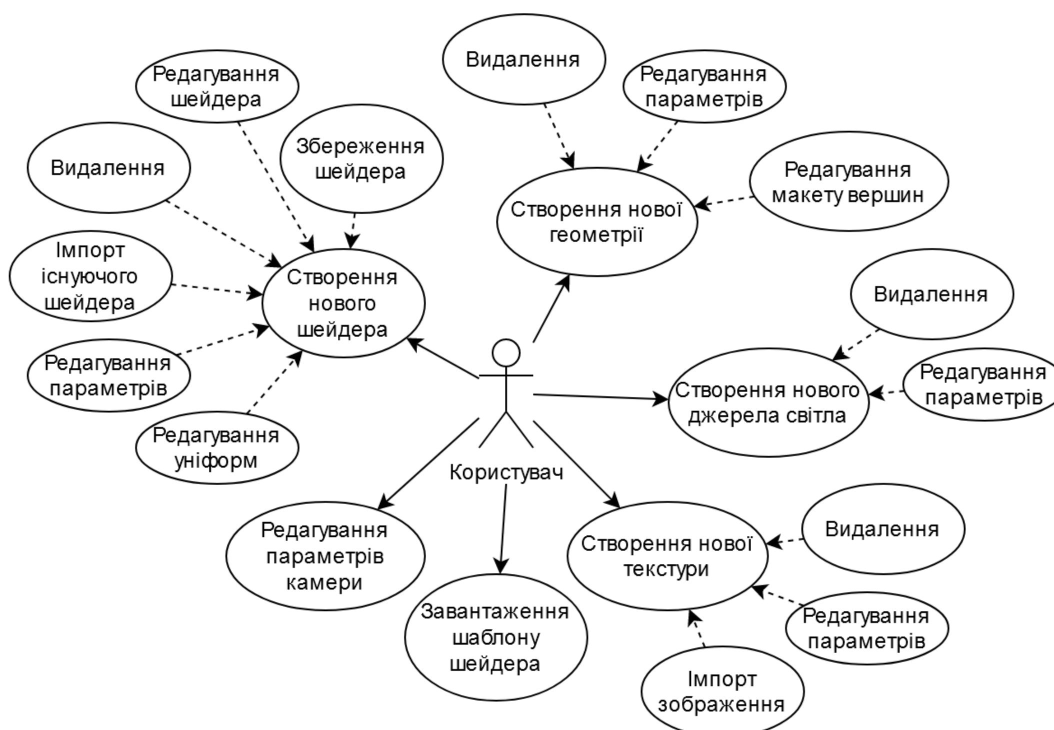

Рисунок 7 — *Use Case діаграма редактора шейдерів*

Додавання нових об’єктів до сцени

Представлення сцени у ієрархічному вигляді дозволяє користувачу додавати різні типи об’єктів до сцени редактора шейдерів. Користувачі можуть створювати об'єкти геометрії, шейдери, текстури або джерела світла, кожен з яких робить свій внесок у загальну композицію сцени. Ця функція підтримує розробку та тестування шейдерів у більш складному та реалістичному середовищі. Діаграма послідовності цього процесу, на прикладі створення об'єкта геометрії, зображена на Рисунок 8.

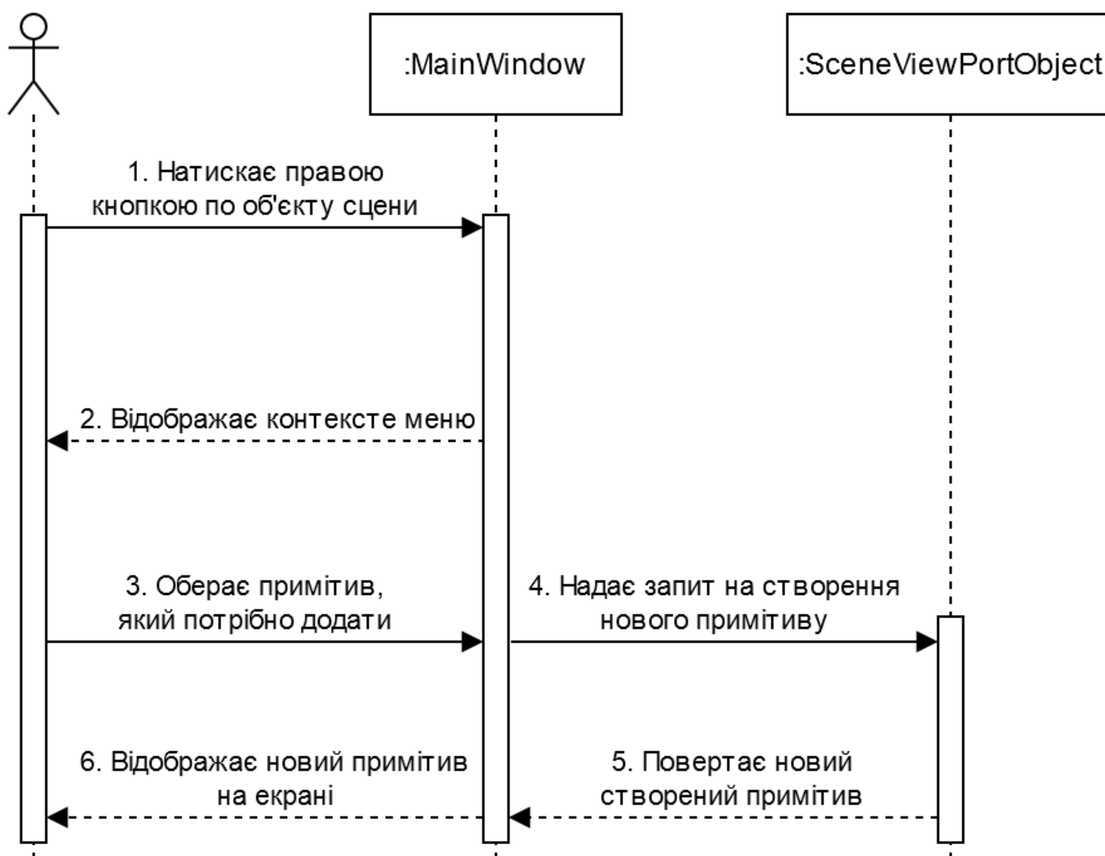

Рисунок 8 — *Sequence діаграма "Створення нового об'єкту сцени"*

Збереження шейдера

Інтерфейс дозволяє користувачу зберегти будь-які зміни, внесені до коду шейдера. Після збереження шейдер автоматично компілюється, а зміни негайно відображаються у вікні перегляду, забезпечуючи зворотній зв'язок у реальному часі. Ця дія гарантує, що робота користувача буде збережена, і він зможе побачити вплив своїх змін. На Рисунок 9 представлена діаграма послідовностей, яка демонструє цей процес.

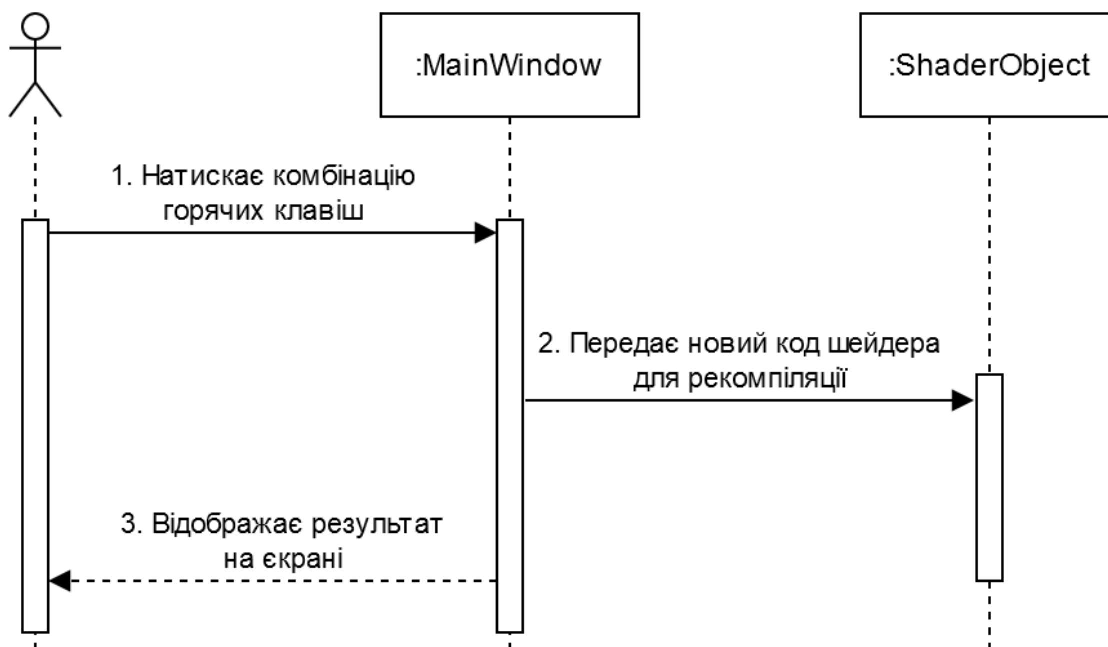

Рисунок 9 — *Sequence діаграма "Збереження програмного кода шейдера"*

Імпортування файлів до редактора

У користувачів є можливість імпортувати файли в редактор шейдерів. Це включає завантаження файлів шейдерів, з розширеннями .vert, .geom і .frag, і файлів зображень, з розширеннями .png і .jpg, які використовуватимуться як текстури в сцені. Імпортовані ресурси можна використовувати в редакторі для розробки шейдерів, тестування та покращення сцени. Діаграма послідовності цього процесу, на прикладі імпортування файлу з кодом GLSL, зображена на Рисунок 10.

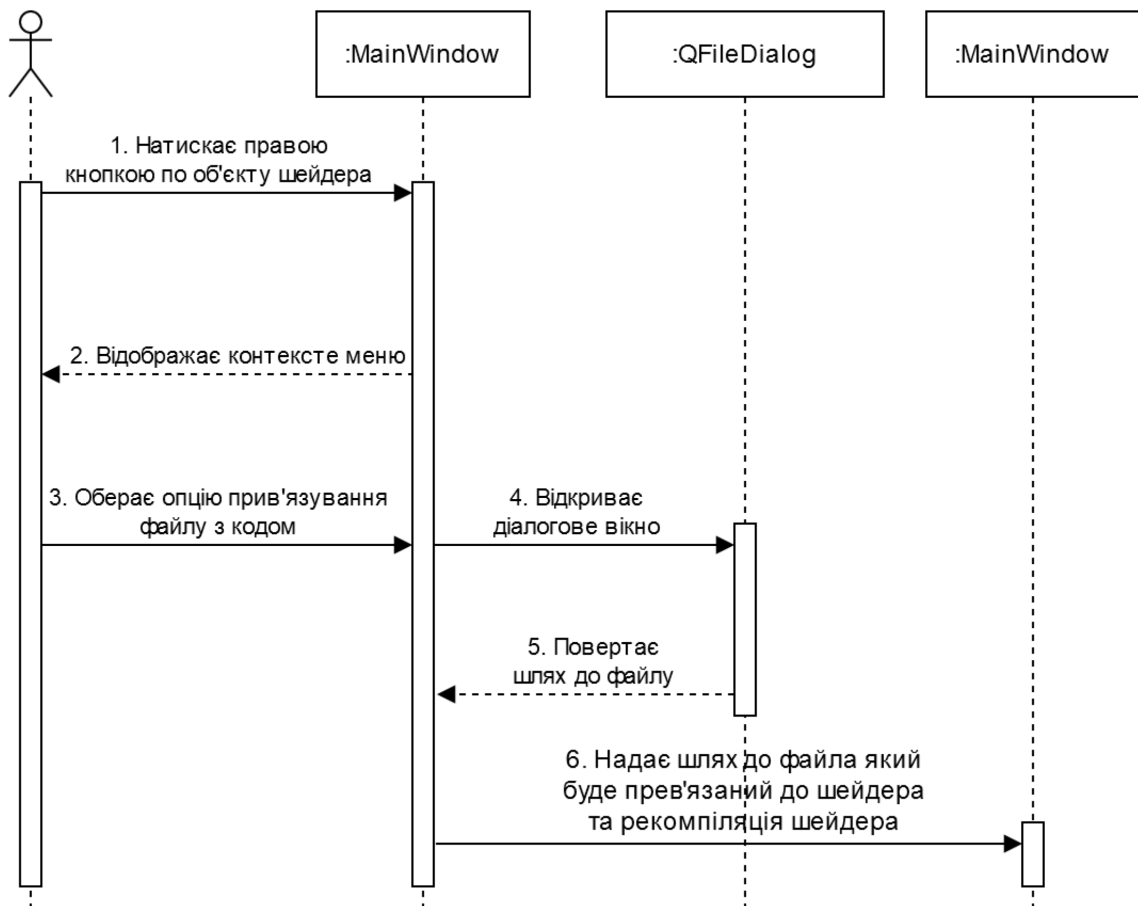

Рисунок 10 — *Sequence діаграма "Імпортування файла до редактора шейдерів"*

Редагування параметрів об'єкта

Редагування властивостей об'єкта сцени дозволяє користувачеві змінювати різні властивості об'єктів у межах сцени. Залежно від типу об'єкта можна регулювати різні властивості, такі як положення, обертання та масштаб для геометричних об'єктів або колір для джерел світла. Ця дія забезпечує гнучкість у вдосконаленні зовнішнього вигляду та поведінки об'єктів сцени, покращуючи загальну композицію сцени. Діаграма послідовності цього процесу зображена на Рисунок 11.

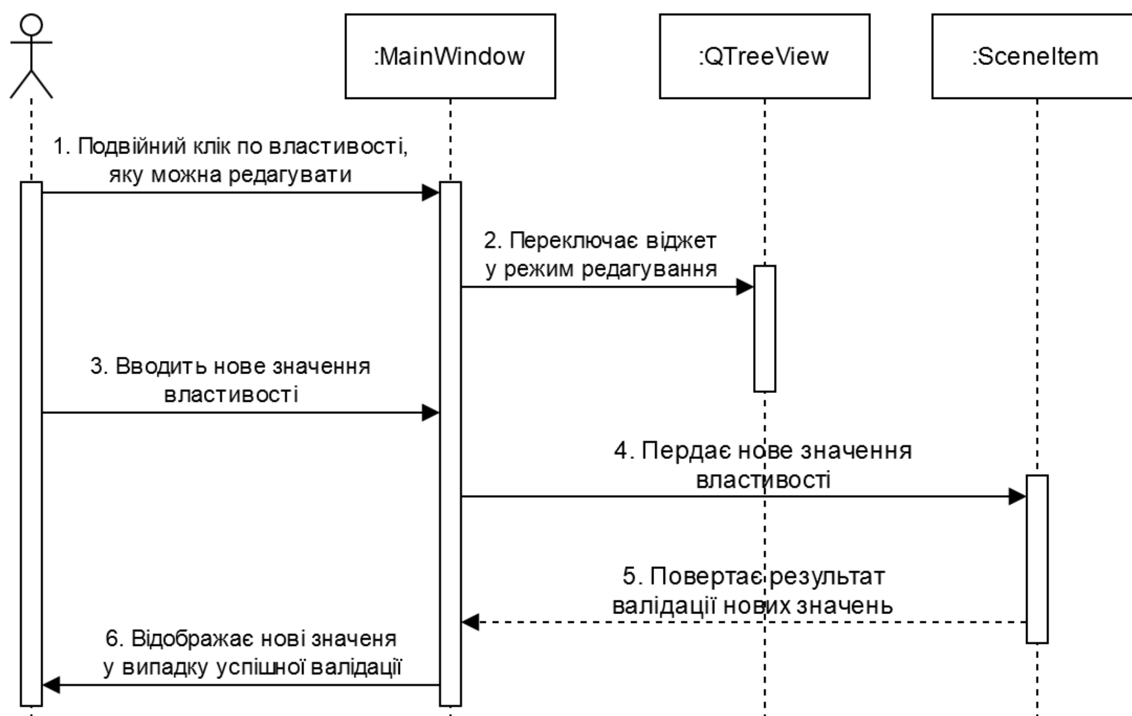

Рисунок 11 — *Sequence діаграма "Редагування параметрів об'єкта"*

Алгоритм переміщеня об'єктів

Важливим процесом у компоненті представлення є можливість користувача змінювати порядок зберігання тих чи інших об'єктів сцени, за допомогою переміщення їх використовуючи мишу.

Цей процес починається з алгоритму, приведеному у Лістинг 10, імітації даних на початку перетягування елемента у віджеті ієрархії сцени. Його мета полягає в тому, щоб захопити та зберегти дані, пов'язані з вибраним елементом, коли користувач ініціює дію перетягування, гарантуючи можливість позиціонування елемента в новому місці в ієрархії.

Лістинг *10 реалізація алгоритму зберігання даних*

```
    QMimeData *SceneHierarchy::mimeData(const QModelIndexList
&indexes) const
    {
        QMimeData *mimeData = new QMimeData;
        QByteArray byteDataArray;
```

```
        QDataStream stream(&byteDataArray,
QIODevice::WriteOnly);
        QList<SceneItem *> nodes;

        foreach (const QModelIndex &index, indexes)
        {
            SceneItem *node;
            if (!index.isValid())
            {
                node = m_rootItem;
            }
            else
            {
                node =
static_cast<SceneItem*>(index.internalPointer());
            }
            if (!nodes.contains(node))
            {
                nodes << node;
            }
        }

        int count = nodes.count();
        stream << QCoreApplication::applicationPid();
        stream << count;

        foreach (SceneItem *node, nodes)
        {
            qlonglong byteData =
reinterpret_cast<qlonglong>(node);
            stream << byteData;
        }
        mimeData->setData(s_treeNodeMimeType, byteDataArray);
        return mimeData;
    }
```

Алгоритм починається з отримання списку індексів, які використовують моделі TreeView у рамках Qt. Ці індекси зберігають інформацію, яка допомагає знайти елементи в ієрархії.

Після отримання індексів алгоритм проходить по списку, обробляючи кожен індекс послідовно. Якщо індекс дійсний, він приводиться до об’єкта типу SceneItem. У випадку коли індекс недійсний, алгоритм позначає його як

кореневий об’єкт. Таке розділення забезпечує правильну ідентифікацію та обробку всіх відповідних елементів.

Під час ітерації алгоритм перевіряє, чи поточний приведений об’єкт уже додано до списку вузлів. Якщо об’єкта ще немає в списку, він додається для того, щоб створити список унікальний елементів.

Після завершення ітерації алгоритм переходить до створення потоку байтових даних. Цей потік інкапсулює важливі дані, включаючи ідентифікатор процесу та загальну кількість елементів списку.

Продовжуючи наповнення потоку, алгоритм перебирає елементи списку вузлів, додаючи дані кожного вузла у вигляді байтових даних. Ці дані зберігають найважливіші атрибути та властивості кожного елемента.

У кінці алгоритм пов’язує потік байтових даних із типом імітованих даних «application/x-sceneitem». Цей зв’язок класифікує потік даних належним чином, полегшуючи його використання під час процесу зчитування.

Цей алгоритм відіграє ключову роль у сценаріях, де користувачі маніпулюють порядком елементів у ієрархії сцени за допомогою перетягування.

Другим етапом процесу перенесення об’єкта є зчитування імітованих даних та синхронізація зміни розташування. Наступний алгоритм виконується для завершення операції перетягування у віджеті ієрархії сцени. Він зчитує імітовані дані, підготовлені на початку операції перетягування, і розміщує вибраний елемент у новому місці, указаному користувачем.

Процес починається з перевірки наявності даних, пов’язаних із типом імітованих даних «application/x-sceneitem». Якщо таких даних немає, алгоритм припиняє роботу та повертає логічний вираз false, що вказує на невдалу операцію оскільки немає дійсних даних для обробки.

Далі алгоритм зчитує ідентифікатор процесу у якому були створені імітовані дані. Він перевіряє, чи цей ідентифікатор відповідає поточному процесу. Якщо є невідповідність, алгоритм завершує роботу та повертає

логічний вираз false, щоб запобігти неправильному тлумаченню чи застосуванню даних у різних процесах.

Після цього проходит перевірка чи дійсний наданий алгоритму індекс, якщо так, він приводиться до об’єкта типу SceneItem. У іншому випадку алгоритм призначає його як кореневий об’єкт.

Також алгоритм робить перевірку параметр рядка. Якщо значення рядка дорівнює -1, а індекс дійсний, рядок отримує значення 0, вказуючи, що елемент слід розмістити на початку. Якщо значення дорівнює -1, а індекс недійсний, встановлюється значення кількості дочірніх елементів індексу, фактично розміщуючи об’єкт у кінці. Ця перевірка потрібна для обмеження переміщення об’єкта в межах своєї поточної глибини.

Після цього алгоритм починає зчитувати кожен елемент із байтового потоку даних. Кожен елемент він приводить до типу SceneItem. Якщо будь-який елемент не має батьківського елемента, алгоритм повертає логічний вираз false, оскільки тільки кореневий об’єкт не має батьківського об’єкта, та його розташування не можливо змінити.

Для кожного дійсного об’єкта SceneItem алгоритм видаляє елемент із його поточного положення в моделі та повторно вставляє його в нове місце, указане користувачем. Під час цього процесу він надсилає сигнали до компонента представлення для оновлення відображення, візуалізуючи зміни в ієрархії.

Цей алгоритм виконується у сценаріях, коли користувачу потрібно змінити порядок елементів в ієрархії сцени. Можливість змінювати положення елементів сприяє інтуїтивно зрозумілій та гнучкій організації сцени, покращуючи загальний досвід користувача.

Після завершення цих кроків алгоритм виводить логічне значення, що вказує на успішність операції. Значення true вказує на те, що елемент успішно розміщено в новому місці, тоді як false значення вказує на помилку через такі проблеми, як недійсні батьківські елементи або невідповідні ідентифікатори процесу.

Загалом, процес зчитування імітованих даних для розміщення елемента у новій позиції є важливим компонентом, який забезпечує точне змінення положення елементів в ієрархії сцени під час операцій перетягування.

3.4.2 Функціонал моделі

Алгоритм візуалізації об'єктів

Візуалізація об'єктів сцени є основним процесом у роботі редактора шейдерів. Для цього алгоритм встановлює цикл візуалізації, викликає необхідні методи для кожного об'єкта, керує програмами шейдерів таким чином відображаючи об'єкти у вікні перегляду.

Процес починається з регулярного виклику методу paintGL(), об'єкта сцени. Цей метод служить основою циклу оновлення, забезпечуючи послідовне відтворення сцени.

У межах методу paintGL() алгоритм виконує ітерацію по кожному дочірньому об'єкту типу GEOMETRY_MESH. Для кожного з цих об'єктів він викликає метод draw(), який обробляє конкретні завдання візуалізації для об'єкта.

Всередині методу draw(), преведеному у Лістинг 11, алгоритм спочатку визначає, чи потребує об'єкт тестування глибини. Тестування глибини забезпечує те, щоб об'єкти відображались в правильному порядку з дотриманням їх просторових відносин. Якщо потрібне тестування глибини, алгоритм його вмикає; інакше він вимикається.

Лістинг *11 реалізація алгоритму візуалізації об'єкта*

```
void MeshObject::draw()
{
    if(doDepthTest)
    {
        glEnable(GL_DEPTH_TEST);
    }
    else
    {
        glDisable(GL_DEPTH_TEST);
```

```
        }
        for(int i = 0; i < childCount(); i++)
        {
            if(m_childItems[i]->getObjectType() !=
SceneObjectType::SHADER)
            {
                continue;
            }
            ShaderObject* shader =
static_cast<ShaderObject*>(m_childItems[i]);

            shader->useShaderProgram();

            glBindVertexArray(VAO);

            shader->activateUniforms(m_parentItem);

            glDrawElements(GL_TRIANGLES,
m_geometry.getIndicesSize() / sizeof(int), GL_UNSIGNED_INT, 0);
        }
        glDisable(GL_DEPTH_TEST);
    }
```

Далі метод draw() виконує ітерацію по кожному дочірньому об’єкту типу SHADER, активуючи відповідну програму шейдера. Шейдерні програми необхідні для визначення того, як графічний процесор обробляє та відображає вершини та пікселі об’єкта.

Після активації шейдерної програми алгоритм прив’язує об’єкт масиву вершин (VAO), пов’язаний із геометричною сіткою. Після прив’язки VAO алгоритм активує уніформи шейдерів.

Далі алгоритм виконує виклик методу візуалізації OpenGL, відтворюючи сітку за допомогою активованої шейдерної програми. Цей крок передбачає видачу команди графічному процесору для малювання трикутників за допомогою вершин визначених у VAO, використовуючи поточну прив’язану програму шейдера, уніформ та коректного EBO.

Цей алгоритм гарантує, що користувач отримує безперервне відтворення сцени, відображаючи всі зміни та оновлення в режимі реального часу.

Алгоритм генерації координат сфери

Такі геометричні примітиви, як куб та площина можуть задаватися константними даними у зв’язку з їх маленьким розміром, але така фігура, як сфера потребує автоматизованого підходу.

Конструктор класу SphereGeometry призначений для створення даних вершин, необхідних OpenGL для візуалізації сфери. Він обчислює вершини, нормалі, текстурні координати та індекси, необхідні для графічного представлення сфери, враховуючи задану кількість секторів і стеків.

На початку алгоритм обчислює кроки між секторами та стеками. Крок сектора визначає горизонтальну сегментацію сфери, тоді як крок стека визначає вертикальну сегментацію. Ці кроки отримуються шляхом ділення повного кола (2π радіан) на кількість секторів і півкола (π радіан) на кількість стеків відповідно.

Потім алгоритм проходить по кожному стеку, щоб обчислити його кут та відповідне положення точки на осі z. Для кожного стека він ініціює ще один цикл для обробки кожного сектора.

У внутрішньому циклі положення x і y кожної вершини обчислюються на основі поточного стека та кутів секторів. Ці координати додаються до вектора вершин. Одночасно алгоритм передає значення кольорів, цих вершин.

У додаток до позицій і даних кольору, алгоритм, приведеному у Лістинг 12, обчислює нормалі вершин, які є важливими для розрахунків освітлення в OpenGL. Ці нормалі також додаються до вектора вершин. А також координати текстур обчислюються та додаються до вектора вершин, полегшуючи застосування текстур до поверхні сфери.

Лістинг *12 реалізація алгоритму генерації вершин сфери*

```
float sectorStep = 2 * PI / sectorCount;
float stackStep = PI / stackCount;

for(int i = 0; i <= stackCount; ++i)
```

```
    {
        stackAngle = PI / 2 - i * stackStep;
        xy = radius * cosf(stackAngle);
        z = radius * sinf(stackAngle);

        for(int j = 0; j <= sectorCount; ++j)
        {
            sectorAngle = j * sectorStep;

            x = xy * cosf(sectorAngle);
            y = xy * sinf(sectorAngle);

            m_vertexArray.push_back(x);
            m_vertexArray.push_back(y);
            m_vertexArray.push_back(z);

            m_vertexArray.push_back(1.0f);
            m_vertexArray.push_back(0.5f);
            m_vertexArray.push_back(0.0f);

            nx = x * lengthInv;
            ny = y * lengthInv;
            nz = z * lengthInv;
            m_vertexArray.push_back(nx);

            m_vertexArray.push_back(nz);
            m_vertexArray.push_back(ny);

            s = (float)j / sectorCount;
            t = (float)i / stackCount;
            m_vertexArray.push_back(s);
            m_vertexArray.push_back(t);
        }
    }
```

Після цього проходить генерація індексів використовуючи цикл який проходиться по усіх стеках, під час якого алгоритм також проходить по всіх секторах.

Під час цієї фази алгоритм, приведеному у Лістинг 13, визначає початкові точки поточного та наступного стеків. Для кожного сектора він обчислює та вставляє індекси двох трикутників, за ітерацію, в список

індексів. Ці трикутники разом утворюють поверхню сфери. Однак у першому та останньому стеках потрібен лише один трикутник на ітерацію.

Лістинг *13 реалізація алгоритму генерації індексів верши сфери*

```
int k1, k2;
for(int i = 0; i < stackCount; ++i)
{
    k1 = i * (sectorCount + 1);
    k2 = k1 + sectorCount + 1;

    for(int j = 0; j < sectorCount; ++j, ++k1, ++k2)
    {
        if(i != 0)
        {
            m_vertexIndices.push_back(k1);
            m_vertexIndices.push_back(k2);
            m_vertexIndices.push_back(k1 + 1);
        }

        if(i != (stackCount-1))
        {
            m_vertexIndices.push_back(k1 + 1);
            m_vertexIndices.push_back(k2);
            m_vertexIndices.push_back(k2 + 1);
        }
    }
}
```

Алгоритм використовує математичні принципи для обчислення сферичних координат кожної вершини, які буде використовувати OpenGL. Таким чином гарантуючи, що вершини рівномірно розподіляються по поверхні сфери, створюючи плавне та точне зображення.

Коли користувачі бажають створити сферу для тестування шейдерів, цей алгоритм вирішує цю проблему, сприяючи продуктивності під час розробки шейдерів.

Генеруючи дані розташування вершин, нормалей, текстурних координат та індексів, алгоритм надає всі необхідні дані OpenGL для візуалізації візуально коректної сфери.

Алгоритм імпортування файлів 3D моделей

Важливою можливістю доступною користувачу є здатність імпортувати файли 3D моделей до редактора шейдерів, а не тільки використовувати підготовлені примітиви. Таким чином з'являється можливість редагувати шейдер в залежності від того з якою геометрією вони будуть використовуватися.

Імпортування файлів формату obj дозволяє редактору шейдерів візуалізувати будь-які 3D моделі. Це відбувається за допомогою алгоритма, який зчитує файл obj, витягує дані вершин і організовує їх у формат, придатний для відтворення в програмі.

Алгоритм [12] починається з ініціалізації векторів для збереження позицій, нормалей і координат текстури. Ці вектори тимчасово зберігають аналізовані дані з файлу obj.

Далі відкривається вказаний obj файл і зчитується його вміст рядок за рядком. Цей підхід гарантує, що файл оброблятиметься з ефективним використанням пам’яті, особливо для великих файлів.

Для кожного рядка алгоритм перевіряє тип даних, які він зберігає, аналізуючи кілька перших символів. Файли obj зазвичай містять рядки, що починаються з наступних символів: «v» для вершин, «vn» для нормалей вершин, «vt» для координат текстури та «f» для граней.

Якщо рядок починається з "v", це вказує, що він зберігає дані про положення вершини. Алгоритм аналізує значення в цьому рядку та приводить їх до чисел плаваючою комою, після чого додає їх у вектор позиції.

Якщо рядок починається з "vn", це вказує, що він зберігає дані нормалей вершини. Подібним чином алгоритм аналізує ці значення та приводить їх до чисел з плаваючою комою та додає їх у нормальний вектор.

Якщо рядок починається з "vt", це означає, що він зберігає дані координат текстури. Алгоритм аналізує значення та приводить їх до чисел з плаваючою комою та додає їх до вектора координат текстур.

Коли рядок починається з «f», він представляє грань, яка визначається серією індексів вершин. Алгоритм аналізує цей рядок, щоб отримати індекси для позицій, нормалей і координат текстури. Потім він використовує ці індекси, щоб перемістити відповідні дані з векторів позиції, нормалі та координат текстур до списку вершин. Крім того, він генерує та надсилає індекси до списку індексів, який використовуватиметься для зображення граней 3D моделі.

Таким чином, алгоритм використовується кожного разу, коли редактору потрібно прочитати та візуалізувати файл obj. Ефективно аналізуючи та впорядковуючи дані, він дозволяє користувачам завантажувати власні 3D моделі в редактор шейдерів, що дозволяє їм працювати з різною геометрією.

Алгоритм обертання камери

Процес обертання камери вікна попереднього перегляду складається з багатьох етапів, та використання декількох методів. Алгоритм ґрунтується на наступній послідовності кроків.

Спочатку він фіксує цілочисельні координати миші (x, y) після того як користувач натиснув ліву кнопку миші, позначаючи ці координати як початкову точку за допомогою методу set2Danchor(). Коли кнопка миші зажата, координати миші (x, y) як цілі числа передаються до камери за допомогою метода setNewRotation(), приведеному у Лістинг 14.

Лістинг *14 реалізація алгоритму перетворення координат миші у кути оберту*

```
    void CameraObject::setNewRotation(int x, int y)
    {
        m_angleHorizontal = m_anchorAngleHorizontal + 
(m_anchorMousePosX - x) / 100.0f;
        m_angleVertical = m_anchorAngleVertical + 
(m_anchorMousePosY - y) / 100.0f;

        if(m_angleVertical > 1.0f)
        {
            m_angleVertical = 1.0f;
```

```
        }
        else if(m_angleVertical < -1.0f)
        {
            m_angleVertical = -1.0f;
        }
        updateCameraPosition();
    }
```

Отримавши нові координати, алгоритм обчислює дельта-значення шляхом віднімання нових значень від старих. Ці дельти, що представляють кутові зсуви, потім перетворюються в градуси та додаються до поточних змінних кута x і y. Ці кути слугують основними вхідними даними для визначення позицій камери (x, y, z) у методі updateCameraPosition(), приведеному у Лістинг 15.

Лістинг *15 реалізація алгоритму перетворення кутів оберту на положення камери*

```
    void CameraObject::updateCameraPosition()
    {
        float new_y = sin(m_angleVertical) * -m_radius;

        float new_x = (cos(m_angleHorizontal) * (m_radius *
cos(m_angleVertical)));
        float new_z = (sin(m_angleHorizontal) * (m_radius *
cos(m_angleVertical)));

        m_position = glm::vec3(-new_x, -new_y, new_z);

        m_view = glm::lookAt(m_position, m_lookAt,
glm::vec3(0.0f, 1.0f, 0.0f));
    }
```

Важливо те, що у зв’язку з динамічною зміною положення камери, вона постійно переорієнтовується, щоб зберегти фокус на центрі об’єкта. Це калібрування досягається шляхом застосування коригувань положення та поворотних трансформацій для вирівнювання перспективи камери з бажаною точкою фокуса.

Коли користувач відпускає кнопку миші, використовується метод set3Danchor() для встановлення нових поточних значень кутів x і y.

З математичної точки зору суть алгоритму полягає в перетворенні 2D координат миші в 3D координати камери.

Алгоритм використовується у сценаріях, коли користувачі прагнуть спостерігати складні шейдерні ефекти з різних позицій. Дозволяючи без зусиль обертати камеру за допомогою миші, це дає їм можливість вирішувати багато проблем візуальних композицій і елементів дизайну у віртуальному середовищі.

3.5 Вимоги до інтерфейсу

Інтерфейс редактора шейдерів був розроблений, щоб розширити можливості розробників, які працюють із шейдерами OpenGL, забезпечуючи комплексний та зрозумілий застосунок. Він спрямований на спрощення процесу розробки завдяки таким вимогам до основних елементів:

1. Відображення результатів у вікні візуалізації шейдера, який редагується, у реальному часі.
2. Редактор коду шейдера:
    - **a.** Підсвічування синтаксису для мови шейдерів GLSL.
    - **b.** Автозаполнения коду та пропозиції щодо ключових слів і функцій шейдера.
    - **c.** Нумерація рядків і виділення помилок для легкого налагодження.
    - **d.** Можливість перемикатися між редакторами вершинними, геометричними та фрагментними шейдерами.
    - **e.** Зберігайте та завантажуйте код шейдера з файлів.
3. Переглядач атрибутів:
    - **a.** Показувати атрибути та властивості геометричних об’єктів, шейдерів і текстур.

   b. Дозволити редагування атрибутів, де це можливо, наприклад, змінювати назву об’єкту ієрархії.
4. Редактор уніформ та макетів:
   a. Окреме вікно редактора для редагування форми шейдерів і макета вершин.
   b. Перелічіть усі змінні уніформ та кваліфікатори макета, які використовуються в коді шейдера.
   c. Можливість редагування уніфікованих значень і специфікацій макета.
   d. Підтримка додавання та видалення уніфікованих змінних і кваліфікаторів макета.
5. Переглядач дерева ієрархії сцен:
   a. Відображення об’єктів і сутностей на сцені у ієрархічному виді.
   b. Підтримка розширення/згортання вузлів і навігації в ієрархії.
   c. Можливість змінювати послідовність ієрархії переміщення елементів за допомогою миші.
   d. Дозволити вибір об’єктів в ієрархії для легшого редагування та маніпулювання.
6. Підтримка комбінацій клавіш для типових дій і операцій.

3.6 Аналіз вимог до користувачів редактора шейдерів

Під час розробки редактора шейдерів, розуміння вимог користувачів має вирішальне значення для створення системи, яка відповідає потребам користувачів. Аналізуючи вимоги, можна перевірити, чи відповідають функції програми очікуванням, що робить її більш зручною та ефективною.

Серед користувачів програми можна виділити одну основну роль, і це розробник шейдерів, який буде виконувати наступні функції:

1. Створювати нові шейдери.

2. Редагувати існуючі шейдери, включаючи вершинні, геометричні та фрагментні шейдери.
3. Оптимізовувати код шейдера для кращої продуктивності.
4. Експортувати шейдери у сумісні формати для інтеграції в програми на основі OpenGL.

Крім того, важливо розуміти можливості системи, до яких користувачі мають доступ:

1. Візуалізація в режимі реального часу для негайного отримання відгуків про зміни шейдерів.
2. Система симуляції джерел світла.
3. Інтеграція з OpenGL для компіляції та візуалізації шейдерів.
4. Система керування константами, які передаються до шейдера.

Іншим аспектом, який слід враховувати, є навички, які користувачі повинні мати для ефективного використання програми.

1. Вміння програмувати шейдери OpenGL на мові GLSL.
2. Розуміння техніки візуалізації графіки та її етапів.
3. Знайомство зі стратегіями налагодження та оптимізації шейдерів.
4. Знання компіляції шейдерів і процесів компонування.
5. Досвід роботи з рендерингом у реальному часі.

3.7 Аналіз вимог до апаратного та програмного забезпечення

Створення редактора шейдерів вимагає повного розуміння як апаратних, так і програмних характеристик. Ці специфікації не тільки визначають продуктивність і функціональність програми, але й забезпечують сумісність із широким діапазоном систем.

Вимоги до обладнання:

1. Графічний процесор із підтримкою OpenGL 3.3 або новішої версії.
2. Багатоядерний процесор із тактовою частотою не менше 2,0 ГГц забезпечить безперебійну роботу.

3. Мінімум 4 ГБ оперативної пам’яті, більший об’єм покращує загальну продуктивність, особливо під час роботи з великими файлами шейдерів або складними сценами.
4. Жорсткий диск або SSD-накопичувач з 1 GB вільного простору.

До програмного забезпечення застосунок має лише одну вимогу, а саме використання операційної системи Windows, macOS або Linux.

3.8 Тестування

Методом тестування застосунку була обрана апробація. Для того щоб перевірити відповідність основним вимогам, які були поставлені до редактора, було розроблено два шейдери.

Перший розроблений шейдер симулює реалістичну поведінку шкіряного покриття [5], за допомогою таких поширених технік як карта нормалей, та карта віддзеркалення. Також цей шейдер використовує адаптовану під GLSL модель освітлення Блінна-Фонга [8]. Шляхом тестування (див.

Рисунок 12) було оцінено здатність шейдера точно відтворювати деталі поверхні та імітувати взаємодію освітлення.

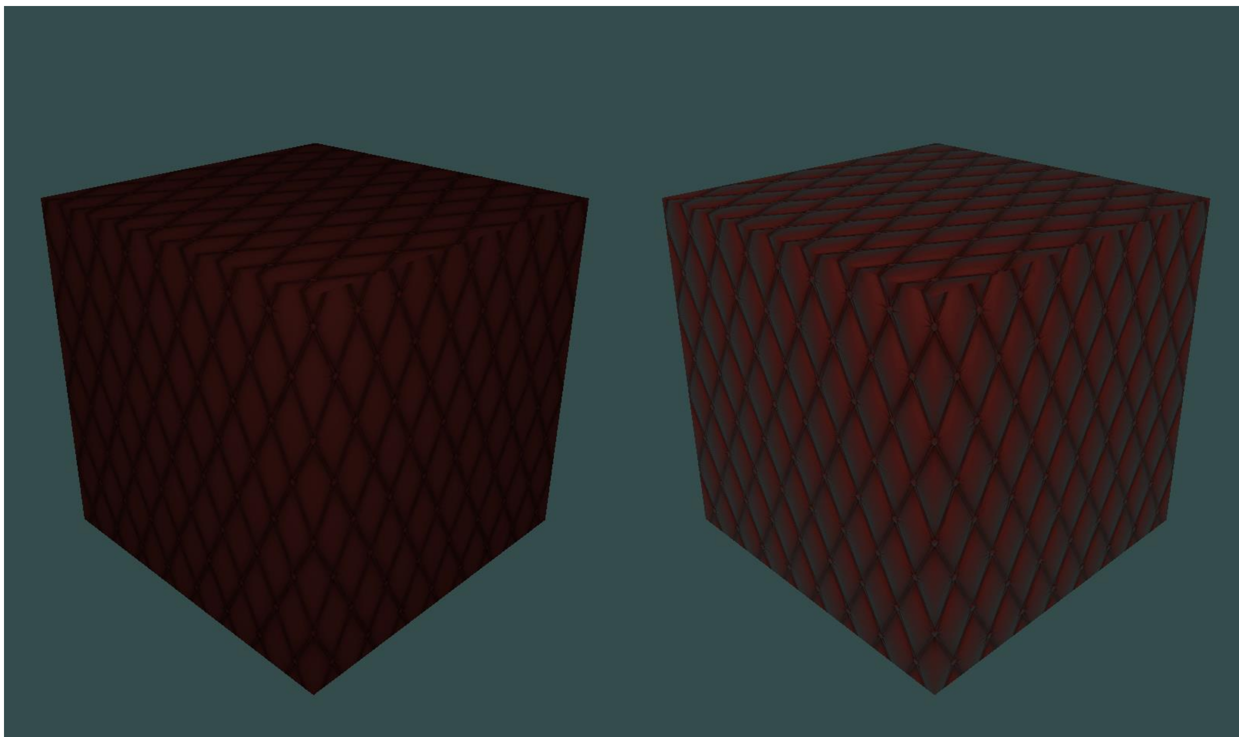

Рисунок 12 — Шейдер симуляції реалістичної поведінки матеріалу

Також апробація проводилася на шейдері контурного тонування, який дозволяє отримати більш стилізовані результати [11]. За допомогою цього була перевірена здатність створювати візуальні ефекти з кольором, контролювати інтенсивність і напрямок світла, а також інтегруватись з іншими техніками візуалізації. Приклад роботи можна побачини на

Рисунок 13

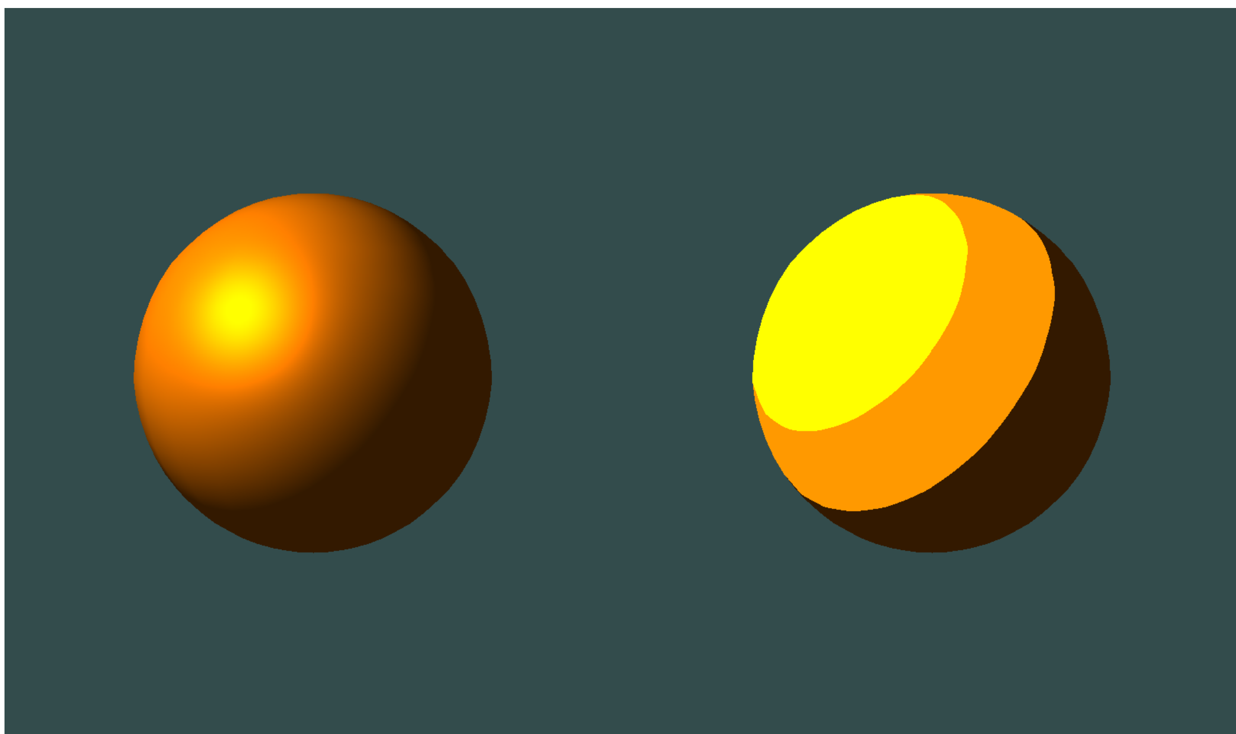

Рисунок 13 — *Стилізований шейдер контурного тонування*

За допомогою розробки цих шейдерів було проведено тестування редактора для того щоб впевнитися у його сумісність, надійність і оптимальну продуктивності з будь-яким застосунком на OpenGL. Після успішного завершення апробації, була перевірена надійності та ефективності програми для розробки шейдерів.

ВИСНОВКИ

1. Проведено аналіз існуючих рішень ShaderED, GLSL Sandbox та ShaderToy. Були визначені їх особливості, сильні та слабкі сторони. Також були опрацьовані літературні джерела, пов’язані з розробкою OpenGL застосунків та роботи з GLSL кодом.
2. Реалізовано шаблон проектування Model-View-Controller (MVC), що забезпечує чітке розділення складових і покращує зручність обслуговування та масштабованість програми. Цей вибір дизайну довів ефективність в управлінні складними взаємодіями та потоками даних, необхідних для редагування шейдерів у реальному часі.
3. Однією з властивостей є підтримка редагування не лише фрагментних шейдерів, але й вершинних і геометричних шейдерів. Ця комплексна підтримка розширює корисність програми. Включення геометричних шейдерів, зокрема, виділяє цей редактор серед багатьох існуючих інструментів, надаючи більше можливостей для творчих і технічних інновацій у графічних програмах.
4. У результаті кваліфікаційної роботи була розроблена програма редактора шейдерів у реальному часі, яка є гнучким, комплексним і ефективним інструментом для редагування шейдерів ця программа також має потенціал для подальших удосконалень і розширень, таких як включення сучасних інструментів налагодження та оптимізації. Успішна розробка такого застосунку підкреслює життєздатність і переваги створення узагальненого редактора шейдерів для програм OpenGL.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Kessenich J., Baldwin D., Rost R. The OpenGL Shading Language, Version 4.60.8 / ред. G. Leese AddisonWesley Professional, 2023. 244 с.
2. Haines E., Akenine-Möller T., Hoffman N. Real-Time Rendering, Fourth Edition. A K Peters/CRC Press, 2018. 1198 с.
3. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3 / Shreiner D., Sellers G., Kessenich J., Licea-Kane B. 8-ме вид. Boston : Addison-Wesley Professional, 2013. 986 с.
4. Qt Documentation | Home. Qt Group. URL: https://doc.qt.io/ (дата звернення: 23.03.2024 р.)
5. Vries J. D. Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion. Kendall & Welling, 2020. 522 с.
6. Shalloway A., Trott J. Design Patterns Explained: A New Perspective on Object-Oriented Design (2nd Edition) (Software Patterns Series). 2-ге вид. Addison-Wesley Professional, 2004. 480 с.
7. Stroustrup B. C++ Programming Language. Addison Wesley, 2013. 1368 с.
8. Wolff D. OpenGL 4. 0 Shading Language Cookbook. Packt Publishing, Limited, 2011. 323 с.
9. OpenGL Insights / ред.: P. Cozzi, C. Riccio. A K Peters/CRC Press, 2012. 712 с.
10. GLM. OpenGL Mathematics. URL: https://glm.g-truc.net/0.9.9/ (дата звернення: 14.04.2024 р.)
11. GPU gems: Programming techniques, tips, and tricks for real-time graphics / ред. F. Randima. Boston : Addison-Wesley, 2004. 765 с.
12. Model loading. opengl-tutorial. URL: https://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/ (дата звернення: 09.05.2024 р.)
13. The Rise In Demand For Realistic And High-Quality Graphics Games Will Drive The Technical Illustration Software Market As Per The

Business Research Company's Technical Illustration Software Global Market Report 2022. GlobeNewswire. URL: https://www.globenewswire.com/news-release/2022/12/07/2569563/0/en/the-rise-in-demand-for-realistic-and-high-quality-graphics-games-will-drive-the-technical-illustration-software-market-as-per-the-business-research-company-s-technical-illustration.html (дата звернення: 18.03.2024 р.)

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Кузенний Нікіта Сергійович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення освітньої програми “Програмне забезпечення систем”, адреса електронної пошти ipz20bd-105@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Розробка редактора шейдерів у реальному часі**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи , а також на архівування моєї роботи в базі даних цієї системи.

Дата 03.06.2024 ____________ Кузенний Нікіта Сергійович
(підпис) (прізвищета ініціали) (студент)

Дата 03.06.2024 ____________ Заяц Валерій Іванович
(підпис) (прізвищета ініціали) (керівник)